Documentation

TABLE OF CONTENTS

CCL 1.2.2

What is CCL?

CCL(Contextual C-like Language) is a programming language that resembles the C language, that can be interpreted at run-time and derives its extended functionality from a predefined context that exists within the application.

What is the Syntax?

It is very similar to C. The features and syntax guide is here

What is the Purpose of CCL?

CCL was designed to be used with Unity games and applications. CCL provides a way to run script that is defined or loaded at run-time. With CCL, additional functionality can be added to your project after it is already released. There are a few reasons that a developer may want to do this. I will expand upon this concept here.

How does it work?

The basic idea behind CCL is that it is driven by a predefined **"Context"**. Before a CCL script is run, a Context object is passed to the CCL script. The context defines the interface that is required for the script to interact with your application. CCL can act on the provided object to get data in and out of the application.

A CCL file or CCL script can be loaded while the game/application is running. The application can then compile the code into a new function and invoke that function.

The Future of CCL

Roadmap

The goals for CCL are to improve performance of compiled code. There will be compiler optimizations that make the compiled code faster at execution time. The interface between CCL and the Context object is reliant on reflection, which can be 1000 times slower than a member access operation. This can make context interactions slow. I will be implementing performance improvements in this area.

Eventually, the goal would be to push the limits of execution speed of dynamically added code and get execution times closer to native c.

Pros and Cons

Pros

- Run dynamic code
- Insert functionality into an application after it is already released
- Familiar style as C/javascript

Cons

- Compiled CCL is about 10-100 times slower than executing c code
- Reliance on reflection *(causes slowness)*

CCL Objective

CCL gives you the ability to run external code in your application after release. This kind of functionality is useful for a variety of things in applications and games. A few uses are:

- Custom event handling
- User content generation
- A user facing programming language that adds custom functionality to your app/game.
- Applications can execute code sent to them from a server. *(this is analogous to your browser executing javascript sent over from a web server.)*

The initial motivation for this idea came from a physics simulator called Algodoo. In Algodoo, you could write custom code that would run when event listeners were called. You would write them in a language called, Thyme. This functionality was so powerful that you could use the physics simulator to do all kinds of interesting things. I once even built a computer in Algodoo, complete with external storage, runnable programs and peripherals. *(In a physics simulator!)*

The point is that, applications that have the ability to execute code that is created after release have the ability to have their functionally extended far beyond what was intended.

Why Base this Language on C?

This language was designed primarily with Unity integration in mind. As a unity plugin, most Unity developers will be familiar with c which is definitely in the C family. It also borrows a few notes from javascript. *(which is also a lot like c in its own right)*

I wanted to take c and javascript, and strip out a lot of functionality. I did not want the language to be able to dynamically allocate objects/variables at run time. *(they are created when the script is "compiled")*

Also sentiment I suppose...

Features & Syntax (1.0.0)

CCL is based on the C family of languages. As it is not intended to be a complete standalone language, it is lacking a few features that come built in to most modern programming languages.

It is object-oriented, but you cannot define or create new objects with CCL alone.

You can pass methods in from the context, but you cannot create new functions with CCL.

Semi-Colons!

A CCL script is a collection of statements. In general, each statement ends with a semi-colon `;`. The semi-colon is mandatory.

```
var temp = 0; // Correct: has a semi-colon at end
var temp = 0  // Incorrect: missing semi-colon
```

Tabs, Whitespace and Newlines

Newlines and Indentation are recommended for readability, but are not mandatory. All white space is ignored by the lexer.

```
if (x == 5) {
   return x + 5;
}
```

is the same as:

```
if(x==5) {return x + 5;}
```

This is also technically valid, even though it is hideous:

```
if(x==5) {return x
+ 5
;}
```

I really don't recommend that last style though...

Comments

Comments begin with `//` and are all line comments

```
// This is a comment
bool notComment = true;
```

Comments are the only place where a newline matters, a newline character ends a comment

Variable Scope

CCL supports block scoping

```
string s = "hello";

{
   string t = s + " world"; // t = "hello world"
}
print(t);                   // t is undefined in current scope
```

That means for loops, while loops, if statements, else statements, and elseif statements have their own scope. Variables declared inside of a block `{ }` are not available outside of the brackets.

`` ` ``c for(int i=0; i < 10; i += 1) { print(i); // i is defined } print(i) // i is not defined in current scope

Variables & Types

Types

CCL is strongly typed and supports value and reference types. Most variables are allocated at compile time and by default, value types cannot be created during script execution. *(There are exceptions to this)*

There is no `new` keyword to allocate new reference variables at execution time, so you cannot create new objects in the usual c way.

The default supported primitive types are:

- bool
- int
- float
- string

Variable Declarations

Variables are defined by specifying the datatype name and then an identifier.

```
int x;
```

Variables can also be assigned to when declared to give them an initial value

```
int x = 1;
```

Variables that are not immediately assigned to when declared can have interesting behavior. They are actually set to a default value at **compile** time. Which means that a variable will be set to it's default type before the script runs the first time. If the variable is modified by the script, its new value will persist the next time it is run

For example,

```
int i;    // should default to 0 for first execution

i += 1;   // i = 1 + 0;
return i; // what will this return?
```

If you run it the first time, you will get a result of `1`

If you run it a second time, the script will return `2`

Every time this script is executed, the variable `i` is incremented, and the value is not reset before the script runs again. If you want the variable `i` to start at 0 each time the script is run, you will have to assign a 0 to the variable when it is declared.

```
int i = 0;    // make sure that i is 0

i += 1;   // i = 1 + 0;
return i; // will always return 1
```

Object

You can also declare a variable without a predefined type as `object`

Object variables can reference primitive, reference, and array types. Under the hood, they are c objects, and can point to any c object type.

`object x = 1`

Any object that does not have a typedef defined in the assembly can only be an object type, so if your context returns a custom class, the only type of variable that can reference it is an `object`.

For example, if a function in my context returns a `MyClass` type, the only variable that can reference it is an `object`

```
object ob = GetMyClass(); // assign a type MyClass to the variable ob

ob.myclassMember = 10;    // use the variable like a MyClass type
```

One caveat with using object variables is that the compiler does not know the type at compile time. Since the type is not known at compile time, the compiler does not know which operators operate on the specified variable.

```
int i = 1;

int j = i + i; // this is ok, the compiler knows that i is an integer

object ob = i; // set ob to int i

j = ob + ob;   // this is not ok, the compiler does not know the type at compile time. object + object doesn't work
```

This issue is partially rectified by type inference, which is covered in another section

Casting

You can cast objects and primitives to other primitive types

```
int x = (int)1.1;      //int
float y = (float)x;    //float
int w = (int)"123";    //int
bool z = (bool)1.1;    //bool
```

When casting array literals, they must be surrounded in parenthesis

```
Vector3 x = (Vector3)([0,0,0]); // ok
Vector3 y = (Vector3)[0,0,0];   // fails miserably
```

Literals

Boolean Booleans are lower case, like C

```
true
false
`
```

Numbers Integers and floats are supported. Doubles and longs are not supported_(yet..)_

If a number literal has a radix point than it is treated as a float, otherwise it is an int

```
0   // int
0.0 // float
1   // int
1.2 // float
```

ints can also be represented in hex

```
0x00 // int (0)
0x0F // int (15)
0xFF // int (255)
```

Strings Strings can be surrounded with double or single quotes

```
"string"
'this is a string'
```

Arrays

Arrays are declared by putting brackets after a variable declaration's type. The array bounds must be specified at "compile" time, and cannot be changed while the script is running. The array size must be defined by a numeric constant. Arrays cannot have elements of different types, all elements must be the same type. *Unless you use an object array* `object[]`

Valid Array Declaration

```
int[10] arr;
string[100] arr2;
int[4] arr3 = [0, 1, 2, 3];
```

Invalid Array Declarations

```
int arr[10];
int[] arr = new int[10];
```

Things to Keep in Mind

When declaring an array, the size must be a numeric literal. expressions and identifiers are not allowed for array size declaration.

Keep in mind! Just like the other variable types, arrays are not created when it is declared at run-time.

The array that the identifier points to is actually created at compile time. If your script modifies the value of one of the elements in the array, that change is persistent throughout multiple executions of the script. For example:

```
int[1] arr;

arr[0] += 1;
```

```
return arr[0]; // what does this return ?
```

If you run that script once, it will return `1`. That seems reasonable, because the default value for an int is 0, so you would expect int[1] to be initialized to an array that contains one 0. `[0]`. so `0 + 1 = 1` right?

If you run the script again, it will return `2`. The reason for that is that `arr[0]` was modified the first time the script ran and now `arr[0]` equals `1`. It is not re-initialized when the script is run again, it points to the same object and is not recreated each time the script is run.

If you want manually initialize the array at the start of each script, you have to assign a new array literal

```
int[1] arr = [0]; // new array literal

arr[0] += 1;

return arr[0]; // what does this return? always 1
```

Array Literals

```
[0, 1, 2, 3] // int array
[0.0, 1.1, 2.2, 3.3] // float array
["hello", "world"] // string array
```

Arrays are reference types, so while it is true that you cannot change the size of an array at run-time, you can still assign an array variable to another array of a different size.

```
int[10] arr;      // arr points to an array of int[10]

arr = [0,1,2,3];  // arr now points to a new array of int[4];
                  // The old array of int[10] is forgotten and will be cleaned up by the garbage collector
```

Numeric Arrays Mixed arrays of ints and floats are not allowed, but if any elements in an array literal are float types, the entire array will be an float array.

```
[0, 0, 0, 1]   // int array, no floats
[0.0, 1, 1, 1] // float array, at least one float
[0, 0, 0, 1.0] // float array, at least one float
```

Side Note: Why did I choose to use `[]` instead of `{}` for array literals?

This is one thing that I like about javascript syntax, so I decided to incorporate that syntax into this language. And it is also easier, as curly braces have several uses already. I did not want to have the compiler have to figure out if it was supposed to be a new block scope or the beginning of an array literal...

Type Inference

If the type of a variable, expression, or function call is not known at compile time, the engine will try to guess the type based on context. For example:

```
int a = 1;      // Type is known (int)
int b = 2;      // Type is known (int)

object ob = a;  // Type is unknown at compile time, the variable ob could point to an object of any type

return b + ob;  // Returns 3, ccl assumes that ob points to an int type because b is an int type
```

If you perform a binary operation and one of the values' types is unknown, ccl infers that they are both the same type. The variable of unknown type, assumes the type of the other variable if it is known.

If neither type is known at compile time, the engine cannot make a guess as to which type they are and this trick will not work.

```
int a = 1;            // Type is known (int)
int b = 2;            // Type is known (int)

object ob = a;        // Type is unknown at compile time, the variable ob could point to an object of any type

return ob + ob;       // Fails, because there is no + operation defined for Object and Object
                      // The engine, does not know at compile-time that these will be ints at run-time
```

```
return (int)ob + ob;  // This works, because at least one variable will have a defined int type
                      // And the engine can infer that the other one must also be an int
```

There are times when this will fail! One example is if you are performing an operation on different types

```
string str = "Hello "; // Type is known (int)
int b = 2;             // Type is known (int)

object ob = b;         // Type is unknown at compile time, the variable ob could point to an object of any type

return str + ob;       // This will fail, because it is going to assume that ob will be a string
                       // We set ob as an int
```

This will fail, even though `string + int` is valid, because the engine is going to pass in an int as if it is a string and it will crash.

Operators

Most of the supported operators should be familiar, as they are the same as most popular languages

Unary

`-x` Numeric Negate

`!x` Boolean Negate (Not)

`x++` Post Increment

`x--` Post Decrement

(pre increment operators not supported)

```
++x // Not supported
--x // Not supported
```

Arithmetic

`x + y` Addition

`x - y` Subtraction

`x * y` Multiplication

`x / y` Division

`x ** y` Exponent (x to the y power)

`x % y` Modulo (the remainder of x / y)

Logic

`x && y` And

`x || y` Or

Bitwise

`x & y` Bitwise And

`x | y` Bitwise Or

`x ^ y` Bitwise XOr

Comparison

`x == y` Equals

`x != y` Not Equals

`x > y` Greater Than

`x < y` Less Than

`x < y` Less Than

`x >= y` Greater Than or Equal To

`x <= y` Less Than or Equal To

Assignment

`x = y` Assign

`x += y` Increment and Assign *(x = x + y)*

`x -= y` Decrement and Assign *(x = x - y)*

`x *= y` Multiply and Assign *(x = x * y)*

`x /= y` Divide and Assign *(x = x / y)*

Arrays

`x[y]` Array Access

`x :: arr` In-Set (return true if array contains element) *New**

`x !:: arr` Not-In-Set (return false if array contains element) *New**

`arr1 := arr2` Array Copy (copies the contents of arr2 into arr1) *New* Coming Soon*

Member Access

`y.x` Member Access

(access the property or method 'x' on object 'y')

`y?.x` Null Conditional Member Access *Not Supported Yet*

access the property or method 'x' on object 'y' if x exists and is not null. If conditions are not met, the statement is aborted without throwing an exception.

Control Statements

If/Else

```
// if
if(x == 1) {
   //...
}

// if else
if(x == 1) {
   //...
} else {
   //...
}

// if else if else
if(x == 1) {
   //...
} else if(x == 2) {
   //...
} else {
   //...
}
```

An else statement can technically be followed by a different statement, which could allow interesting things like:

```
if(arr == null) {
   print("arr is null");
} else for(int i = 0; i < arr.length; i += 1) {
   print("array element!");
}
```

While

```
while(true) {
    //...
}
```

Do-While

Not yet supported

For

```
for(int i = 0; i < x; i += 1) {
   //...
}
```

Foreach/For-In/For-of

Not supported

You can access a data type or classes member functions using the dot operator

```
MyType.staticMethod();
```

You can access static methods of any type that is defined in the assembly. *(Any custom type that has a TypeDef defined and all of the default TypeDefs in the build in libraries)*

Security Warning!

Do not expose custom types that have static methods with unexpected security implications. Some classes have static members/methods that can have disastrous security consequences. For example, when creating the built-in libraries for the engine, I included a TypeDef for the UnityEngine.GameObject class. I quickly realized this was a bad idea because I could use the static methods in the `GameObject` class to "hack" the application in interesting ways! I used `GameObject.Find()` to find the UI elements in the application and move them around or even use `GameObject.Destroy(gameobject)` to destroy them.

It would not be good if you expose the ability to find, modify or destroy any GameObject in the app at run-time from code written at run-time!

The GameObject class is a particularly heinous example of insecure classes to expose, but there are countless others. Other examples would be:

- IO.File
- IO.Directory
- UnityEngine.Application the list goes on... there are countless horrible ways to compromise the security of your application.

Before exposing a type to the assembly, make sure to take into account the static members that will be accessible to the ccl when you do. **You should treat ccl scripts as unsafe code, you have no guarantees that what is passed in to the engine is not malicious**

AllowStaticMembers

Feature added: 1.2.2

If you want to expose a variable type that has static members that may compromise your application, you can override the `allowStaticMembers` property in the custom TypeDef. if `MyTypeDef.allowStaticMembers` returns false, the ccl script cannot access the static members on the type.

So exposing the UnityEngine.GameObject type to the ccl assembly would be safe, as long as I make sure that it's corresponding TypeDef.allowStaticMembers returns false.

Return

return stops the execution of the script, and can be used to return a value to the calling application

```
return; // exits the script
```

```
return 1; // exits the script and returns 1 to the calling application
```

Require

```
require "MyContext";
```

Specifies the accepted type of context class. If there is a `require` statement in your script, you will only be able to pass in contexts that match the specified type. Otherwise you will get a run-time exception. If no require statement is specified, you can try to pass in any context object.

Functions

You can define functions in ccl.

The syntax for declaring a new function is as follows:

```
TypeName FunctionName() {
   // function body here...
}
```

Functions can have arguments as well,

```
TypeName FunctionName(int arg0) {
   // function body here...
}
```

Here is an example of a simple function named AddInts that adds two ints and returns the sum

```
// declaring the function
int AddInts(int a, int b) {
   return a + b;
}

// Now lets use the function
int res = AddInts(1,2);

return res; // will return 3
```

Functions can have as many arguments as you could need, the limit on argument count theoretically is Int32.Max, but I dont suggest that you try that.

Function overloading is not allowed, multiple functions cannot have the same name, even if the signature is different. Every function name must be unique Functions do not allow default arguments (as of 1.2.1)

Void

if a function doesn't return anything, it is declared using the keyword `void`.

```
void SayHello() {
   Log("Hello!");
}
```

void functions actually still do return an object, but that object is null

Recursion

Functions do not support recursion, which means that a function cannot call itself. This feature will be added later, currently as of 1.2.1 variables are statically allocated.

Not always easy to debug

The engine, is not always the best at reporting where an error is and what the error is. This is a known issue, going forward, there will be a lot of effort put into making the scripts easier to debug and finding errors. The only way to make the scripts easier to debug will be by inserting more metadata into each statement that can be more easily traced at execution time. This of course makes things slower, that is why I did not do it in the first place. A solution will be found.

Not super fast

CCL is still very slow compared to c, I want to make it as fast as possible. There will be ongoing efforts to make every part of this engine as fast as possible without having to sacrifice flexibility.

No recursion support

Functions cannot call themselves at this current time, and that bothers me deeply. I will be working toward a solution in the near future.

Overloaded Functions

This is also a fact that disturbs me to my core, at this moment, the engine cannot handle an overloaded function. I know.. trust me, it kills me inside. If a method has multiple signatures, and you try to use it in a ccl script, you will most-likely get an ambiguous argument error. This is a top priority issue at the moment

Working with a Context(1.0.0)

What is a context?

The context is interface between your application and your run-time script. The context provides the functionality that you want your scripts to be able to have. It is basically an object that has properties and methods that you want to leverage in your script. The context object's properties and methods are accessed as part of the global scope alongside global variables that your script declares. The context is passed in at execution time and the script operates on it.

CCL is pretty useless without a context. CCL is designed to not really be able to do much without a context to operate on. What your script can do really is defined by what context it is operating on. This principal provides flexibility and security.

Flexibility

CCL can do what ever you want it to do based on the context you pass in.

Security

CCL can **only** do what you want it to do based on the context you pass in.

The very nature of CCL is dangerous because your application is running code that is passed in from elsewhere. The code is untrusted and could be malicious. Since that could be the case, you can limit the amount of influence that a ccl script has by only exposing functions and properties in your application that you feel are safe.

How to Pass a Context to CCL?

First, create a new class:

```
class MyContext {
   int _member1;               // private member variable
   public int member1 {        // public property
      get {
         return _member1;
      }
      set {
         _member1 = value;
      }
   }
   public string PrintData() {  // public method
      return member1.ToString();
   }
}
```

Compile the script, pass in the context, and call the compiled function

```
Compiler compiler = new Compiler(tokens);
```

```
CompiledScript compiledScript = compiler.compile();

// Create a new instance of your custom context object
MyContext context = new MyContext();

// Pass it in to the script with the SetContext method
compiledScript.SetContext(context);

// Call the compiled function
compiledScript.function();
```

How to Pass Data into a Script?

You can pass variable values in to the script by using the `SetVariable` method of the `CompiledScript` class. This is useful for passing data directly in from the application to the script as a global variable. For this to work, a global variable must exist in the script with the specified identifier.

` c // set the global variable v to 100 compiledScript.SetVariable("v", 100);

// set the global variable col to a new Color object compiledScript.SetVariable("col", new Color(1,1,1));

// set the global variable func to a new delegate compiledScript.SetVariable("func" , (Func)(() => { return "lambda"; }));

A Simple CCL Engine Demo

This is a simple Unity UI example that changes the color of the screen based on a CCL script. This demo is super basic, but it shows you how to use the engine in your application.

First, Create The Context Class

```
class ApplicationContext: MonoBehaviour {
   [SerializeField]
   private Image _bgImage;

   // Sets the background color of the bgImage object
   void SetBackgroundColor(float x, float y, float z) {
      _bgImage.color = new Color(x,y,z);
   }

   DateTime GetDateTime() {
      return System.DateTime.Now;
   }
}
```

The Application Class

```
class MyApplication : MonoBehaviour {
   CompiledScript _compiledScript;

   [SerializeField]
   ApplicationContext _context;
   //...
   private string LoadScript(string filename) {
      // Load a script from the file system at runtime
      //...
   }
   public void Start() {
      // Get the script from somewhere...
      string script = LoadScript("demo_file.ccl");

      // Load the standard libraries into the CCL Assembly
      // The libraries need to be loaded before compiling the scripts that need them
      CCL.ScriptUtility.LoadStandardLibraries();

      // Compile script, it is best to call this in a try catch, in case the script has an error
      try {
            // Compile script
            // Pass in the context class type at compile time
            _compiledScript = CCL.ScriptUtility.CompileScript(script, typeof(ApplicationContext));

            // Pass in context
            _compiledScript.SetContext(_context);
        } catch (Exception e) {
            // Compilation Error Handling code...
        }
    }

    public RunScript() {
        try {
            _compiledScript.Invoke();
        } catch (Exception e) {
            // Run Time Error Handling code...
        }

    }
}
```

The CCL File

This script can come from anywhere, it can be in the Resources folder, or saved in the persistent data folder on the end users device, it can even be fetched from a server when the application starts. The point is, it does not have to exist when you build the game/app. It can be fetched dynamically when the user starts the game/app or written on the fly by the user is you like.

Example 1

```
SetBackgroundColor(0.0, 1.0, 0.0);   // Set background color to a green
```

The script set the background color of the app to green, based on a CCL script that was loaded at run time. The CCL script sets the background color of the app using a method on the context object named `SetBackgroundColor`.

Example 2

```
int dayOfWeek = (int)GetDateTime().DayOfWeek;
if(dayOfWeek == 0) { // If today is Sunday
   SetBackgroundColor(0.0, 1.0, 0.0);   // Set background color to a green
}
else if(dayOfWeek == 1) { // If today is Monday
   SetBackgroundColor(1.0, 0.2, 0.2);   // Set background color to a red
}
else if(dayOfWeek == 6) { // If today is Saturday
   SetBackgroundColor(0.2, 1.0, 0.2);   // Set background color to a green
}
else {  // If today is Other
   SetBackgroundColor(0.6, 0.6, 0.6);   // Set background color to a grey
}
```

The script set the background color based on what day it is. The CCL script first gets the day of the week by calling a method on the context called `GetDateTime`. This method returns a `System.DateTime` object which has the property `DayOfTheWeek` The CCL script sets the background color of the app by calling `SetBackgroundColor`. based on the day of the week from the `System.DateTime` object returned by the context.

Roadmap

Near-Future

- [x] Context compatibility headers **(Done 11/08)**
  - Specify the name of the context data type at at the head of the CCL file
- [ ] Support for sbyte datatypes
- [x] Bitwise operators **(Done 10/18)**
  - Bitwise And `&`
  - Bitwise Or `|`
  - Bitwise XOr `^`
  - Bitwise Not `~`

  - Bitwise Not
  - Left Shift `<<`
  - Right Shift `>>`
- [x] Hex literals for sbyte, int **(Done 10/18)**
  - 0xFF = (sbyte)
  - 0xFFFF = (int)
- [x] Function Definitions **(Done 11/26)**
  - Define new functions in CCL scripts
- [ ] Null-Conditional member access operator
- [ ] Subroutine exports
  - export a function from ccl and call it from outside the ccl engine
- [ ] Advanced array operations
  - is subset `::`
  - union `+`
  - intersection `<>`

Future

- [ ] Dictionary Support
- [ ] Enum support
- [ ] Struct support
- [ ] Const support
- [ ] Multi-dimensional arrays

Far-Future

- [x] Customizable compilation pipeline, this would allow for custom operators and types **(Done 10/18)**
- [ ] Support for overloaded functions
- [ ] Dynamic Types, similar to objects in javascript. Kindof like a collection of key value pairs

Possible

- [ ] Support for long datatypes
- [ ] Support for char datatypes

Creating Libraries to Extend Functionality

Libraries can be added to the CCL assembly that add new datatypes and functions. Creating a library can be a bit of work but can be really rewarding if you need custom data types in your CCL scripts. This would mean that your script has the ability to declare variables of custom datatypes and maintain compile-time type with custom objects. *What does that mean?*

Why Would I Want to Make a Library?

Here is a quick example:

Vanilla (no custom types):

```
var matrix = GetMatrix4x4();     // Gets an object reference from a method on the context called GetMatrix4x4
float x = (float)matrix.a0 + (float)matrix.a0; // have to be cast to float, compiler does not know the type
```

Custom Matrix4x4 Type

```
Matrix4x4 matrix;                 // The matrix4x4 type is recognized as part of the language and the constructor is c
alled
float x = matrix.a0 + matrix.a0;  // Does not need to be cast, compiler knows type at compile-time
```

In the first example above, you can create a object by using a method on the context and then store it in a `var` reference. This is alright, but at compile-time, the compiler has no idea what type of object is stored in the reference. That means that you have to cast references to the type that you think they are supposed to be.

In the second example, the compiler knows how to directly work with the `Matrix4x4` type, including any custom operators that work on matrix types. All the datatypes of the Matrix4x4 class are known at compile-time, no casts are needed. So this is going to be faster performance-wise and easier to work with in your scripts.

Example Library

Example Library

For this example, lets say that I want to use the `IntVector2` class. An `IntVector2` is basically just 2 ints `x` and `y`

First, create the Library object

```
public class IntVector2Lib {

}
```

We are off to a good start.

Custom Type

Next, we need to a public sub-type to the `IntVector2Lib` class, this sub-type needs to inherit from the `TypeDef` class

```
public class IntVector2Lib {
   // ...

   // Custom Type Definition
   public class IntVector2TypeDef : TypeDef {

      // Return the system.type of the custom data type
      public override Type type
      {
         get
         {
            return typeof(IntVector2);
         }
      }

      // Return the name of the type
      public override string name
      {
          get
          {
              return "IntVector2";
          }
      }

      public override object Cast(object arg, CompilerData _internal)
      {
          // Create a function that lets you cast other types to this type
          //..
      }

      // specify a function that casts Func<IntVector2> to Func<object>
      public override Func<object> ToGenericFunction(object arg, CompilerData _internal)
      {
         if (arg.GetType() == typeof(Func<IntVector2>))
         {
            return () =>
            {
               return ((Func<IntVector2>)(arg))();
            };
         }
         return () =>
         {
            return arg;
         };

      }

      // Create the default constructor for the type
      public override Func<object> DefaultConstructor()
      {
         return () => { return new IntVector2(); };
      }
   }

   //...
}
```

Now we have a custom type. When the library is loaded into the CCL assembly, all Sub-Types that implement the TypeDef class are added to the assembly. The type must specify how it behaves in the script. Each custom TypeDef must specify a default constructor, a Cast to Generic Function, a name, a type, and other cast functions to cast other objects to this type.

Great, we have the custom type IntVector2 in our CCL assembly. What now?

Now, lets create custom operators that operate on that type.

Unary Operators

We are going to create a Unary Operation and a Binary Operation. To do this, we will be using c Attributes. Look at the syntax for creating a Unary Operation

```
[UnaryOperator(System.Type, Token.Type)]
public static object Function(object arg0, CompilerData cdata) {
   // ...
}
```

Lets examine this.

I created a static method in the IntVector2Lib class that has the signature `object func(object, CompilerData)` . Unary Operations must follow that format to be accepted as the correct delegate type. Above the method, there is a `[UnaryOperator()]` attribute. The UnaryOperator attribute takes 2 arguments, the first is a `System.Type` indicating the type that the operator operates on. The second, is a `Token.Type` argument that indicates which type of token invokes this operator.

So now lets actually make some functions. We will start with the negate function. This function will be added to the `IntVector2Lib` class

```
[UnaryOperator(typeof(IntVector2), Token.Type.Subract)]
public static object Negate(object arg0, CompilerData cdata) {

   // returns a function from of a CCL.Reference object if arg is a CCL.Reference
   // if arg0 is a function return that function
   // otherwise, if arg0 is a value, arg0Func is null
   Func<IntVector2> arg0Func = CompilerUtility.ForceGetFunction<IntVector2>(arg0);

   if(arg0Func == null) {
      IntVector2 temp = (IntVector2)arg0;
      return (Func<IntVector2>)(() => {
         return new IntVector2(-temp.x, -temp.y);
      });
   }
   else {
      return (Func<IntVector2>)(() => {
         IntVector2 temp = arg0Func();
         return new IntVector2(-temp.x, -temp.y);
      });
   }
}
```

Now, you will be able to use the negate operator `-` on your custom type `IntVector2`. Once this library is loaded into the assembly, this is perfectly valid CCL code:

```
IntVector2 ivec;
ivec.x = 1;
ivec.y = 1;
return -ivec;      // will return (-1,-1)
```

Binary Functions

That look pretty good, it is still a bit awkward that the components of the IntVector2 Have to be set independently. Lets change that by adding a custom assignment operator. I want to be able to assign an Int32[] array literal to an IntVector2 to assign both elements at once.

Look at the syntax for creating a Binary Operation

```
[BinaryOperator(System.Type, Token.Type, System.Type)]
```

```
public static object Function(object arg0, object arg1, CompilerData cdata) {
   // ...
}
```

It is the same as the UnaryOperator but the attribute is `BinaryOperator`. The `BinaryOperator` attribute takes an extra `System.Type` argument at the end. Again, the method must be static and the signature must be `object func(object, object, CompilerData)`.

So lets add a new `BinaryOperator` to our library:

```
[BinaryOperator(typeof(IntVector2), Token.Type.Assign, typeof(Int32[]))]
public static object Assign_Array(object arg0, object arg1, CompilerData cdata) {
   Func<IntVector2> arg0Func = CompilerUtility.ForceGetFunction<IntVector2>(arg0, cdata);
   Func<int[]> arg1Func = CompilerUtility.ForceGetFunction<int[]>(arg1, cdata);

   if(arg0Func == null && arg1Func == null) {
      IntVector2 temp0 = (IntVector2)arg0;
      int[] temp1 = (int[])arg1;
      return (Func<IntVector2>)(() => {
         temp0.x = temp1[0];
         temp0.y = temp1[1];
         return temp0;
      });
   }
   else if(arg0Func == null && arg1Func != null) {
      IntVector2 temp0 = (IntVector2)arg0;

      return (Func<IntVector2>)(() => {
         int[] temp1 = arg1Func();
         temp0.x = temp1[0];
         temp0.y = temp1[1];
         return temp0;
      });
   }
   else if(arg0Func != null && arg1Func == null) {
      int[] temp1 = (int[])arg1;
      return (Func<IntVector2>)(() => {
         IntVector2 temp0 = arg0Func();
         temp0.x = temp1[0];
         temp0.y = temp1[1];
         return temp0;
      });
   }
   else {

      return (Func<IntVector2>)(() => {
         IntVector2 temp0 = arg0Func();
         int[] temp1 = arg1Func();
         temp0.x = temp1[0];
         temp0.y = temp1[1];
         return temp0;
      });
   }
}
```

That was a bit verbose, but now we have added an assignment operator to the CCL assembly that takes a `IntVector2` type on the left-hand side and an `Int32[]` on the right hand side:

```
IntVector2 ivec = [1, 1];
return -ivec;      // will return (-1,-1)
```