MiniScript Unity Integration Guide

Extending your Unity game with a MiniScript interpreter

Joe Strout

MiniScript Maker

Version 1.4

Wednesday, June 5, 2019

0. Purpose of This Document

who are you, and why are you reading this?

This document describes how to integrate the MiniScript engine into a Unity game. There are several reasons why you might want to do this.

MiniScript can be used internally, for example, to provide a simple, lightweight scripting language for laying out level designs, or scripting AI behaviors or cut scenes. This could give level designers the power to program these things, without exposing them to all the complexities of the underlying C# code.

More typically, though, MiniScript will be used to make a game scriptable by end-users, either to support mods, or to make a programming game. MiniScript is a simple language that can be described in a single page of documentation, so players will have a fun and easy time picking it up. MiniScript is also lightweight and efficient, so you can run many MiniScripts at once, each in its own independent sandbox. Moreover, because you control the interface between MiniScript and the Unity world, you can make it completely safe: scripts can access only those objects and properties you want them to.

This document assumes that you're a Unity programmer, probably experienced in C#. It also assumes you've at least read through the MiniScript Manual, a separate document that describes the MiniScript language in detail, so you're already familiar with MiniScript's type system, and how things like lists, maps, classes, and intrinsic functions work from the user's point of view.

Now it's time to peek under the hood, and see how those things look to your C# code!

MiniScript is written in C# and was designed to work with Unity from day one, so you should find that integrating it into your game is quick and painless. But if you run into any trouble, don't panic! Join the community at http://miniscript.org or in the Unity forums, then post a clear question, and you're sure to get an answer soon.

1. Overview
know what you don't know

Integrating MiniScript with your Unity project is pretty easy:

Step 1: Install the MiniScript package (a folder containing about ten C# scripts) anywhere in your Assets folder.

Step 2: Write a MonoBehaviour script that creates and runs a Miniscript.Interpreter object. This is detailed more below.

For clarity, we're going to use the term “user scripts” to refer to scripts in the MiniScript language, which your Interpreter is executing. These user scripts might be written by actual users, or they might be written by yourself or other members on your development team — that's up to you. But we're going to call them all user scripts, to distinguish them from the C# scripts that you write for Unity.

There are three different ways that MiniScript might be used, depending on your needs. In order of most common to least common, these are:

1. Set the source of your interpreter once, and then run the script for just a little bit each frame. This basically time-slices the script and integrates nicely with your Unity game loop. The user scripts will probably need to include a long (or even infinite) loop, so they can continue to do stuff as your game runs. (Or you might add that infinite loop for them — see Chapter 4.)

2. Run each script all the way to the end, and then pull out some results. In this case the user scripts would be expected to complete very quickly. (Of course in practice you would probably combine this with something like approach 1, in case an ornery user *does* code a long loop into their script.)

3. Set up a Run-Eval-Print-Loop, where users can, at run time, type MiniScript commands and immediately see the results.

Note that a more detailed overview of the various MiniScript classes is provided in Chapter 5. For now, we’re going to proceed by example, with a focus on the steps you need to do to get MiniScript up and running in your game.

Setup

1. Include the Miniscript namespace at the top of your C# script:

```
using Miniscript;
```

2. Declare a property to hold the interpreter:

```
public Interpreter interpreter;
```

3. Instantiate that interpreter, for example, in the Awake method:

```
void Awake() {
   interpreter = new Interpreter();
}
```

This might also be a good time to set callbacks for standard, implicit, and error output, as in this example from Robo-Reindeer:

```
   interpreter.standardOutput = (string s) => reindeer.Say(s);
   interpreter.implicitOutput = (string s) => reindeer.Say(
        "<color=#66bb66>" + s + "</color>");
   interpreter.errorOutput = (string s) => {
        reindeer.Say("<color=red>" + s + "</color>");
        interpreter.Stop();
        lastError = s;
   };
```

This would also be where you add some custom intrinsic methods, if you haven't added them already; this is explained in the next chapter.

4. Stuff the user script into the interpreter by calling the Reset and Compile methods. Sometimes you will want to add some "built-in" MiniScript code, and/or set some global variables, to provide a context for the user script. Here's an example:

```
public void PrepareScript(string sourceCode) {
   string extraSource = "ship.reset = function(); self.x=0;
   self.y=0; self.rot=0; end function\n";
   interpreter.Reset(extraSource + sourceCode);
   interpreter.Compile();
   ValMap data = new ValMap();
   data["x"] = new ValNumber(transform.localPosition.x);
   data["y"] = new ValNumber(transform.localPosition.y);
   data["rot"] = new ValNumber(transform.localRotation.z);
   interpreter.SetGlobalValue(globalVarName, data);
}
```

5. Run the script for a while by calling `RunUntilDone`, passing in a small timeout value (0.01 seconds is a fairly typical). Notice that the name of this method is a bit misleading — it really should be `RunUntilDoneOrTheTimeoutTimeIsReached`, but that would be a bit too long! Typically you would call this in your `Update` or `FixedUpdate` method. Note that `RunUntilDone` will raise an exception if there is an error in the user script, so you probably want to put this in a try/catch block.

```
try {
    interpreter.RunUntilDone(0.01);
} catch (MiniscriptException err) {
    Debug.Log("Script error: " + err.Description());
}
```

Also note that if you want to check whether the script is still running, you can just call `interpreter.Running()`. If it's not running, you might want to restart it (assuming you want the user script to run continuously during your game). You can do this by calling the `Restart()` method on the interpreter.

Performance Notes

In the example above (and most of the examples included with the MiniScript package), we are allowing the script to run for up to 0.01 seconds. But the `RunUntilDone` method will bail out sooner than that if the script hits a `wait` intrinsic method call, or any other intrinsic that can't complete its work right away. (This behavior may be turned off by specifying `false` for the optional second parameter to `RunUntilDone`, in which case it will continue running the script until the full time is up or the script calls `yield`.)

But even with the bail-out behavior, if a script codes a tight loop (i.e. a loop without a `wait` or `yield`), it will take up the full time allotted to it. If you have 20 scripts each taking 0.01 seconds to complete, that is 0.2 seconds total — limiting the frame rate of your game to 5 FPS or less!

To avoid that, when you may have many scripts running, simply specify a shorter time limit when calling `RunUntilDone`. A limit of 0.001 seconds in our 20-script example above would take at most 0.02 seconds to complete, compatible with 50 FPS (worst case). 0.001 seconds is still enough time to run a fair amount of MiniScript code. If you want to get fancy, you could even dynamically throttle the scripts based on current usage and your frame rate target.

2. Bridging the Divide
making MiniScript and Unity work together

Now you've got a user script running within your Unity application. But that's not very useful unless you can somehow exchange data across this gap — for user scripts to get data from the host (Unity) app, and vice versa. There are two basic ways to do this: reading and writing global variables, and adding new custom intrinsic functions.

Accessing MiniScript Globals

First, you can exchange data via user-script global variables. For example, this Unity code sets a global variable called "energy" in the user script:

```
interpreter.SetGlobalValue("energy", new ValNumber(42));
```

...and this Unity code reads the same global variable, which may have been set by the user script:

```
Value val = interpreter.GetGlobalValue("energy");
if (val != null) Debug.Log("Energy: " + val.FloatValue());
```

You would typically do this after calling `RunUntilDone` on your Interpreter, i.e., after the user script has had a chance to modify those values.

`Value` is an abstract base class; the concrete class of any value you get will be something more specific, as described in the "MiniScriptTypes.cs" section of Chapter 5. The most important types are `ValNumber`, `ValString`, `ValList`, and `ValMap`. If you are pretty sure you'll get a specific type, use the C# `as` operator to convert it:

```
ValMap data = interpreter.GetGlobalValue("data") as ValMap;
```

Alternatively, of course, you can pull out a `Value`, check the type with `is`, and then do different things depending on what sort of data you find. In either case, always be sure to check for null, which GetGlobalValue will return if the variable you're asking for is not found.

If your data is a `ValList` or `ValMap`, those types provide public fields to let you access their contents. For a list, you can access a `List<Value>` called `values`, and for a map, there is a `Dictionary<Value, Value>` called `map`. ValMap also overrides the square-brackets operator, allowing you to easily get and set values by a string index. For example, if `data` is a `ValMap`, then `data["x"]` is a useful (and more efficient) shorthand for `data.map[new StringValue("x")]`.

When setting or returning MiniScript values, you typically create a new instance of some Value subclass, as shown in the first example above (`new ValNumber(42)`). However

there are some special static values which are often convenient to use, as well as more efficient. These include:

- `ValNumber.zero`: equivalent to `new ValNumber(0)`
- `ValNumber.one`: equivalent to `new ValNumber(1)`
- `ValNumber.Truth(bool t)`: equivalent to `new ValNumber(t ? 1 : 0)`
- `ValString.magicIsA`: equivalent to `new ValString("__isa")`
- `ValString.empty`: equivalent to `new ValString("")`

Adding Custom Intrinsics

Instead of or in addition to exchanging data via global variables, you can define new intrinsic functions for your interpreter context. User scripts can then invoke those functions, which run your own C# code. These functions can accept parameters and return a value, allowing you to exchange data both directions.

Intrinsic functions are global (or more technically, static) — they apply to all MiniScripts running within your application. So, you should only define them once. Here's an example (from the RoboReindeer demo) that defines a new intrinsic called “throw” that takes one parameter.

```
1.  // throw(energy=20)
2.  // fires a snowball in our current heading;
3.  // returns 1 if success, 0 if failed.
4.  f = Intrinsic.Create("throw");
5.  f.AddParam("energy", 20);
6.  f.code = (context, partialResult) => {
7.    var rs = context.interpreter.hostData as ReindeerScript;
8.    Reindeer reindeer = rs.reindeer;
9.    float eCost = (float)context.GetVar("energy").DoubleValue();
10.   if (reindeer.Throw(eCost)) return Intrinsic.Result.True;
11.   return new Intrinsic.Result.False;
12. };
```

Let’s go over this in some detail. Line 4 actually creates a new intrinsic function called `throw`. Line 5 adds a parameter called “energy” with a default value of 20. To define more parameters, simply make more calls to `AddParam` here.

Line 6 defines the C# code that should be invoked when the intrinsic is called from a user script. This is a function delegate of this form:

```
Intrinsic.Result IntrinsicCode(TAC.Context context,
    Intrinsic.Result partialResult)
```

So it takes a context object that gives you access to the local variables (including arguments to the function), and it returns a result. (We'll explain more about that `partialResult` parameter in a moment.)

Lines 7-11 in the example above get executed when the intrinsic is called. Lines 7-8 get the interpreter from the context, and from there grab a reference to a game-specific class stored in its `hostData`. Line 9 is looking up the value of the "energy" parameter passed in by the user. Finally, line 10 invokes a custom function on our C# script, and returns True if it succeeded, or False otherwise.

Note that we don't have to do anything with our intrinsic reference (`f`) once we're done defining the intrinsic; as soon as we call `Intrinsic.Create`, it exists in the global intrinsic function space. The local variable `f` here is only needed to give us a chance to fill out the function definition. A possible exception is if you create an unnamed intrinsic, with `Intrinsic.Create("")`. In this case, you would use your reference (`f`) to provide this code to the user in some other way, for example, as an entry in a map that you provide through some other intrinsic.

After you've done whatever processing you need to do in your intrinsic, you must return an `Intrinsic.Result`. This is a little class with two fields: `done` (a bool), and `result` (a Value). In most cases you would set `done` to true (or use one of the several constructors that do that for you), indicating that this intrinsic's work is done, and the result is final. MiniScript will then continue on with the next line of the user script.

However, if you have more processing to do, you can set `done` to false, and then the MiniScript engine will invoke your intrinsic code again, passing back the whole result object in the `partialResult` parameter. The built-in `wait` intrinsic uses this, only reporting that its work is done when the requested amount of time has passed. You might use it if you add an intrinsic that should block (wait) until some game event occurs, or until some computation done in a separate thread has finished.

The Result class includes a number of handy constructors and static methods for common needs, including returning a number, returning a string, or returning true (1), false (0), or null. See the source code in MiniscriptIntrinsics.cs for details.

As mentioned before, you only want to add the intrinsics to MiniScript once. So typically you would put these in a static method, and use a static flag to indicate when you've done it. Here's an outline showing a good way to do it.

```
using UnityEngine;
using MiniScript;

public class MyClass : MonoBehaviour {
  static bool intrinsicsAdded = false;

  void Awake() {
    AddIntrinsics();
  }

  static void AddIntrinsics() {
    if (intrinsicsAdded) return;   // already done!
    intrinsicsAdded = true;

    Intrinsic f;

    f = Intrinsic.Create("foo");
    …
    f = Intrinsic.Create("bar");
    …
  }
}
```

Map Assignment Override

If you're making a sophisticated MiniScript interface, you probably have intrinsics that return maps to be used as classes, modules, or objects encapsulating your game API. In that case, you may need to do something special when the user assigns to one of those maps. For example, in Mini Micro, there is a TextDisplay class that has members that represent the text foreground color, background color, etc., and when the user assigns to one of those members, we need to update the Unity data behind the scenes.

You can accomplish this by assigning to the `assignOverride` field of your ValMap before returning it to the user. This is a function that takes a key and value, and returns true to cancel (override) the assignment, or false to let it proceed as normal. The aforementioned code for TextDisplay looks like this:

```
map.assignOverride = (key, value) => {
	if (value == null) value = ValNumber.zero;
	switch (key.ToString()) {
	case "color":
		textColor = value.ToString().ToColor();
		break;
	...
	}
	return true;  // cancel normal assignment
}
```

We check the key (as a string), and if it's "color" we convert the value to a color and store it in textColor, a C# property that's part of our game. Other cases check for other special keys (not shown). At the end, we return true, which cancels the standard assignment behavior. This works out great because "color" in this map is in fact a function that constructs its return value on demand from the behind-the-scenes C# property. We wouldn't want users to lose that functionality by assigning a new value.

There are other, older means of doing this sort of interface, such giving users an ordinary map of values and then checking/updating it on every frame. But this method — using functions to look up values when needed, and `assignOverride` to intercept assignment of new values — is flexible and far more performant, since it only runs code when the user actually reads or writes the value.

3. Making a REPL Loop
Read, Eval, Print Loop (...Loop)

This chapter describes an advanced form of MiniScript usage which most games will not need. You should probably skip right on to the next chapter until you're already comfortable using MiniScript in other ways.

The usage here is making a Read-Eval-Print Loop or REPL. This is an interactive MiniScript context where the end user can type a bit of MiniScript code, and immediately get the results, line by line. You see this in the MiniScript demo, in the "REPL" mode.

The good thing about a REPL is that the user can very quickly play around with MiniScript syntax and try things out. The big disadvantage is that typing any function or loop more than a couple of lines long is cumbersome and error-prone, and if you make a mistake you have to do the whole thing over. Thus, for any nontrivial scripting, you'll certainly want to instead provide a multi-line edit field, or read text files edited outside your game. And in that case, you don't need this chapter.

But if you really do want to provide an interactive MiniScript console, then read on!

Setup

The setup for a REPL is actually the same as for any other usage; you create an Interpreter, and hook up its standardOutput, implicitOutput, and errorOutput callbacks. In case of errorOutput, you probably also want to stop the interpreter, as it could be in a wonky state.

```
interpreter.standardOutput = (string s) => output.PrintLine(s);
interpreter.implicitOutput = (string s) => output.PrintLine(
        "<color=#66bb66>" + s + "</color>");
interpreter.errorOutput = (string s) => {
  output.PrintLine("<color=red>" + s + "</color>");
  // ...and in case of error, we'll also stop the interpreter.
  demoScript.interpreter.Stop();
};
```

Here we assume we have some output script with a PrintLine method; this could, for example, append the given text to the content of a UI.Text object.

Providing a Prompt

There are three different states the interpreter might be in at any moment. It could be idle, running, or waiting for additional input (for example, waiting for the end of a loop or

function definition). To help your users understand what's going on, you should change the prompt to indicate the state.

The following UpdatePrompt method shows how you might do this.

```
1.void UpdatePrompt() {
2.   var promptR = prompt.GetComponent<RectTransform>();
3.   string promptStr;
4.   if (interpreter.NeedMoreInput()) {
5.        promptStr = ". . . > ";
6.        promptR.sizeDelta = new Vector2(60, promptR.sizeDelta.y);
7.   } else if (interpreter.Running()) {
8.        promptStr = null;
9.   } else {
10.       promptStr = "> ";
11.       promptR.sizeDelta = new Vector2(25, promptR.sizeDelta.y);
12.  }
13.  prompt.text = promptStr;
14.
15.  RectTransform inputR = input.GetComponent<RectTransform>();
16.  inputR.anchoredPosition = new Vector2(promptR.sizeDelta.x,
       inputR.anchoredPosition.y);
17.  if (promptStr == null) {
18.       input.DeactivateInputField();
19.       input.gameObject.SetActive(false);
20.  } else {
21.       input.gameObject.SetActive(true);
22.       input.ActivateInputField();
23.  }
24.}
```

Lines 3-13 set the prompt text (a UI.Text) according to the interpreter state, by checking the `NeedMoreInput` and `Running` methods. They also resize the prompt text accordingly, since the prompt we use in the "need more input" state is significantly longer than the usual prompt.

Then lines 15-23 update the UI.InputField where the user actually types their input. We need to move it over so that it sits next to our (possibly resized) prompt, and even deactivate it while the script is running.

This UpdatePrompt method should be called from your Update method, so that the prompt will continuously update as the interpreter state changes.

Handling Input

Now that we're set up and we have given the user a prompt, we need to do something with their input! Make a public function in your script, and hook it up to the onEndEdit event of your input field. The function should look something like this.

```
1.public void HandleInput(string line) {
2.        if (string.IsNullOrEmpty(line)) return;
3.        output.PrintLine("<color=grey>" + line + "</color>");
4.        interpreter.REPL(line, 0.01);
5.    }
```

Line 2 ignores empty input. Line 3 echoes the input to the output — this is optional, of course, but comforting to the user.

The key bit is line 4. We pass the input to the `REPL` method on the interpreter, along with a timeout (again, 0.01 seconds is typical). This tells the interpreter to attempt to interpret the given input in REPL mode.

There are a couple of differences between running script in REPL mode, and running it via the usual `RunUntilDone` method. First, the interpreter will not expect a complete program; if there are open code blocks (e.g. `if`, `for`, `function`, etc.), then it will go into the "need more input" state. Second, it will allow an input which is only an expression, rather than a full statement, and in this case convert the value of that expression to a string and send it to standardOutput. This allows the user to enter something like `2+2` and see the result (`4`) in the output. Code running via `RunUntilDone` does not do this; only values explicitly printed with `print` go to standard output.

Keep the Engine Running

The code above would be sufficient if the user never enters any loops, nor calls any intrinsics that don't return immediately (such as the `wait` function). If they do, though, then that 0.01-second timeout in your REPL call will time out before the script is done, leaving the interpreter in the Running state. We need to be sure we give it a chance to finish. So, your Unity script will need an Update method that looks something like this.

```
1.void Update() {
2.    if (interpreter.Running()) interpreter.RunUntilDone(0.01);
3.    UpdateFromScript();
4.    UpdatePrompt();
5.}
```

Line 2 is the key bit; if the interpreter is still in a running state, then we tell it to run a little more, until finished or until another 0.01 seconds has passed.

Line 3 here calls an UpdateFromScript method you might want to have to update whatever Unity state you want to reflect the values of global variables in your script, as described in chapter 2.

Finally, line 4 is the call to that UpdatePrompt method discussed in the previous section.

4. Making an Event Handler
scripting reactions to stuff that happens

This chapter describes another somewhat advanced topic, but one that is more commonly needed: invoking specific functions in a script in order to handle events that happen at runtime.

For example, you might have user scripts controlling some GameObject, and want those scripts to define specific MiniScript methods to get invoked when the level starts, when the object collides with something, when it takes damage, and so on. Or you might have some scriptable UI object, and want to invoke certain methods on mouse enter, mouse exit, click, etc.

There is certainly more than one way to do this, but here's a technique known to work well. We will instruct users to write functions with specific names for each event. They can also write some global code to run when the script is first loaded, but it should not go into an infinite loop; it should instead fall through to the end of the script.

Preparing the Event Loop

To the bottom of the user's script, we will sneakily append an event loop, which constantly checks a global queue of function pointers, and invokes any functions it finds. Then on the C# side, when we want to invoke a particular function in response to some event, we simply look up that function by name in the global space, and append it to the event queue.

Here's the C# setup code, that appends the event loop before compiling the user code:

```
    // Grab the source code from the user
    string sourceCode = sourceField.text;  // (or whatever)

    // Append our secret sauce: the main event loop.
    sourceCode += @"
_events = []
while true
  if _events.len > 0 then
    _nextEvent = _events.pull
    _nextEvent
  end if
  yield
end while";

    // Reset the interpreter with this combined source code.
    interpreter.Reset(sourceCode);
    interpreter.Compile();
```

Nothing tricky here; we're just adding a half-dozen lines of MiniScript code, which declare a global list called `_events`, and implement a simple event loop. (Because `_events` is a global variable in the user's script space, we begin it with an underscore; the MiniScript Manual advises users to avoid such names.)

Invoking Events

Then, here's the C# code that adds to this event queue in order to invoke a function.

```
void Invoke(string funcName) {
  Value handler = interpreter.GetGlobalValue(funcName);
  if (handler == null) return;   // no handler defined
  var eventQ = interpreter.GetGlobalValue("_events") as ValList;
  if (eventQ == null) {
    eventQ = new ValList();
    interpreter.SetGlobalValue("_events", eventQ);
  }
  // Add a reference to the handler function onto the event queue.
  // The main event loop will pick this up and invoke it ASAP.
  eventQ.values.Add(handler);
}
```

This code looks up the function in the global space by name, and also looks up the event queue (making sure that exists as a list). Once we have these, it's a simple matter of adding the function to the list.

And that's it! The next time through its event loop, the MiniScript will find the entry in its `_events` list and invoke it. That function can then do whatever the user wants to do to handle the event, communicating with the Unity side via intrinsics or other globals.

Note that the technique as shown does not allow for passing any data to the event handlers. To do that, you'd need a slightly more complex setup; instead of just a list of function pointers, the event queue would be a list of maps. Each map would contain the function reference, along with whatever other data is needed. The event loop would then pull out the function pointer, and invoke it with the remaining map as an argument.

The MiniScript package comes with a demo scene called "EventPumpDemo" that illustrates this technique. All the relevant code is in ScriptableButton.cs.

5. MiniScript Class Overview
what's what and where it lives

This chapter provides an overview of the classes in the MiniScript namespace. For details, you can always go to the source code, where you should find detailed comments on any class or function intended for public use. But this chapter will give you the big picture, so you know where to look and how it all fits together.

MiniScriptErrors.cs

SourceLoc: represents a source location, i.e. a line number and optional file name. This is used in error reporting.

MiniscriptException: base class for all exceptions raised by MiniScript. Includes a SourceLoc, whenever possible, to provide more information about where the error was found. Call the Description method to get a nice error message, including location. A number of subclasses provide more specific exception types.

LexerException: represents an error in the MiniScript lexer (the first stage of parsing). For example, this might be thrown for an unterminated string.

CompilerException: represents an error found while compiling, other than lexer errors. Typically these represent some incorrect syntax in the MiniScript code.

RuntimeException: base class of any exception thrown while MiniScript code is executing. Several subclasses are used for more specific cases.

IndexException: thrown when MiniScript code attempts to look up an element by index, but the index is invalid (out of range).

KeyException: thrown when code attempts to access a map element by key, but the key is not found. (Use `m.hasIndex(k)` to see if map `m` contains a key `k`.)

TypeException: thrown when attempting to do something with the wrong type of data. For example, trying to use a function as if it were a map or list will throw this error.

TooManyArgumentsException: thrown when a function call includes more arguments than the function has defined parameters.

UndefinedIdentifierException: thrown whenever trying to get the value of a variable that does not exist.

Check: this is a module (static class) that provides some convenience functions for error-checking at runtime (for example, in your custom intrinsic methods). Currently the only method in here is Range; call Check.Range with an actual value, and the allowed minimum and maximum values, and it will throw an IndexException if the value is out of bounds.

MiniscriptInterpreter.cs

The only class in this file is **Interpreter**, which is your main interface to the MiniScript system. You give Interpreter some MiniScript source code, and tell it where to send its output (via delegate functions called **TextOutputMethod**). Then you typically call `RunUntilDone`, which returns when either the script has stopped, the given timeout has passed, or script code has called `yield`. For details, see Chapters 1-3 of this manual.

MiniscriptIntrinsics.cs

This file defines the **Intrinsic** class, which represents a built-in function available to MiniScript code. All intrinsics are held in static storage, so this class includes static functions such as `GetByName` to look up already-defined intrinsics. See Chapter 2 for details on adding your own intrinsics.

This file also contains the **Intrinsics** static class, where all of the standard intrinsics are defined. This is initialized automatically, so normally you don't need to worry about it, though it is a good place to look for examples of how to write intrinsic functions.

Note that you should put any intrinsics you add in a separate file; leave the MiniScript source files untouched, so you can easily replace them when updates become available.

MiniscriptKeywords.cs

This file defines a little **Keywords** class, which contains all the MiniScript reserved words (`break`, `for`, etc.). It might be useful if you are doing something like syntax coloring, or want to make sure some user-entered identifier isn't going to conflict with a reserved word.

MiniscriptLexer.cs

This file is used internally during parsing of the code, breaking source code text into a series of tokens. Unless you're writing a fancy MiniScript code editor, you probably don't need to worry about this stuff. It includes the following classes.

Token: represents one elemental part of MiniScript code. Tokens come in many different types, including keywords, numbers, various operators, and more. A token instance represents the token type, and in cases where the content is variable (such as a number rather than, say, a left parenthesis), it includes the actual text of the token as well.

Lexer: this class is responsible for taking MiniScript source code and breaking it up into a series of tokens.

MiniscriptParser.cs

This file is responsible for parsing MiniScript source code, and converting it into an internal format (a three-address byte code) that is considerably faster to execute. Again, because this is normally wrapped by the **Interpreter** class, you rarely need to deal with **Parser** directly.

MiniscriptTAC.cs

This file defines the three-address code (TAC) which represents compiled MiniScript code. TAC is sort of a pseudo-assembly language, composed of simple instructions containing an opcode and up to three variable/value references.

TAC is actually a static class that contains several public classes all related to three-address code. **TAC.Line** represents one TAC instruction, and includes a reference to the source code line it came from, for better error reporting. It also includes the code to actually execute all the instructions; this is where the "engine" is that makes MiniScript actually run.

TAC.Context keeps track of the runtime environment, including local variables. Context objects form a linked list via a "parent" reference, with a new context formed on each function call (this is known as the call stack).

Finally, **TAC.Machine** implements a complete MiniScript virtual machine. It keeps the context stack, keeps track of run time, and provides methods to step, stop, or reset the program.

MiniscriptTypes.cs

A number of small classes in this file represent the MiniScript type system. **Value** is the abstract base class, from which more specific classes are derived:

- **ValNumber**: a double-precision numeric value.
- **ValString**: a string value.
- **ValList**: a MiniScript list (sequence of other values).
- **ValMap**: a MiniScript map (dictionary of value/value pairs).
- **ValFunction**: a reference to a function defined in MiniScript.
- **ValTemp**: used internally to represent a temporary value while evaluating an expression.

- **ValVar**: rather than an actual value, this is a variable reference.
- **ValSeqElem**: represents a sequence element reference, that is, an indexed list or map.

MiniscriptUnitTest.cs

This file provides a bit of unit-testing support for other MiniScript classes (in particular, the lexer and parser). This is mainly used in contexts other than Unity by the MiniScript development team, and isn't something you should need to worry about.