Using BotL from C#

BotL code is exposed the C# programmers through the BotL namespace. The primary entry points for programmers are:

- public static void **KB.Load**(string *path*)
  Compiles and loads a .bot file.
- public static void **KB.LoadTable**(string *path*)
  Loads a CSV file into a table whose name is the filename component of *path*.
- public static void **KB.Compile**(string *assertion*)
  Compiles a single rule or fact.
- public static void **KB.IsTrue**(string *predicateName*)
  Runs the specified predicate (with no arguments) and returns true if it succeeds, else false. The globals $this and $gameobject are set to null. $time is set to the value of Time.time.
- public void *component*.**IsTrue**(string *predicateName*)
  Extension method. Runs the specified predicate (with no arguments) and returns true if it succeeds, else false. The global $this is set to *component* and $gameobject is set to *component*.gameObject. $time is set to the value of Time.time.
- public void *gameObject*.**IsTrue**(string *predicateName*)
  Extension method. Runs the specified predicate (with no arguments) and returns true if it succeeds, else false. The global $this is set to null and $gameobject is set to *gameObject*. $time is set to the value of Time.time.

There isn't presently a good mechanism for passing arguments to/from the predicate, since it's hard to design an API for that that doesn't involve boxing and unboxing data (i.e. allocating memory). You main options for communication between C# and BotL are:

- Have BotL code **call your C# methods** directly
  This is less performant because it goes through reflection, which allocates memory. But if you're starting out, it's probably the right thing to use.
- Have BotL code and C# code communicate through BotL **global variables**
  This will let the two sides communicate without memory allocation.
- Use **Eremic logic**
  This is probably preferred, but won't be implemented for a couple more weeks.
- Write your own **primops** (primitive predicates) for BotL that call directly into your C# code
  This is the most efficient way of doing this, but is primarily for power-users.

Interoperation using Reflection

The easiest way to have the two systems talk is to write methods in C# that your BotL code can call to read and write data from the C# side. This is really easy to do and if you're doing something simple, it's probably sufficiently performant. However, calls from BotL to C# go by way of reflection, which is somewhat slow, and which allocates memory (e.g. for argument lists, for boxing numbers, etc.). So

while you probably want to try this mechanism first, if you find your game is doing lots of garbage collections, then you might want to optimize it using one of the alternative mechanisms.

Using BotL global variables from C#

BotL global variables can be read and written from C#. You can get a pointer to the internal GlobalVariable object using the method:

> public static **GlobalVariable.DefineGlobal**(string *name*, object *initialValue*)

This will create a global with the specified *name* and *initial value*, if one does not already exist, and return it, or return the existing one if one has already been made.

The current value of the global is stored in its **Value** property. For efficiency reasons discussed below, Value isn't of type object, it's of type **TaggedValue**, and you need to use special methods to read and write it. The short version is that to set it, do one of:

- *global*.**Value.Set**(*numberOrBoolean*)
  To set it to an int, float, or bool
- *global*.**Value.SetReference**(*object*)
  To set it to a value you know will never be an int, float, or bool, or
- *global*.**Value.SetGeneral**(*object*)
  To set it to a value that truly could be anything. This will do run-time checks of the argument and dispatch to the appropriate method so as to handle it without having to allocate memory (see below).

To read the value of a global variable, use:

- *global*.**Value.AsFloat**
  To get the value of the variable as a float.
- *global*.**Value.Value**
  To get the value as a reference of type object. Note that if the value was a number or Boolean, this will allocate memory because C# allocates memory when you convert a struct to an object (see below)

A TaggedValue also has a Type field to indicate what the actual type stored in it is. I will happily document this if and when a user actually needs to know it, but I'm guessing that's deeper in the weeds than people need to know about and I don't want to scare potential users off with needless detail.

Why we have to use TaggedValue rather than just object

Unlike older dynamics languages such as Lisp and Prolog, storing a number (or other struct type) into a variable of type object causes the runtime to allocate memory through a process called *boxing*. If one executes the code fragment:

```
int x=10;
object y = x;
```

then x is stored on the stack as a normal integer. However, y, being of type object, must be stored as a pointer to an object on the heap. So the act of setting y to a number causes the system to allocate a tiny

object on the heap (a box) to hold the integer, store the integer in the object, and then store the pointer to the object in y. If you run the fragment:

```
for (int i=0; i<1000; i++) y = i;
```

the system will allocate 1000 boxes. This means that if one wants to write an efficient interpreter for a scripting language, one has to essentially make a new data type to substitute for the old "object" type, one that has separate fields for numbers (floats, integers) and for references (everything else). In BotL, that substitute type is called **TaggedValue**. It's a struct, and has methods for reading and writing its value without boxing. All intermediate results in BotL are stored internals as TaggedValues in a central stack.

Writing your own primops

This is for power users. I'm happy to help you with it, but I'm not going to document it until someone needs to do it, especially since the internals of the primop API might evolve, whereas the rest will likely remain stable.