G-Script

Language Specification

Version 3.11

Notice: If you are not cleared to read this document, then you are obligated to burn the copy you hold in your hands and assume the party submission position. If you wish to gain clearance to view the following pages, contact your supervisor. Failure to comply will result in death.

This Page Intentionally Left Blank

1. Introduction

G-Script is an experimental language for our experimental Genetic Life-form and Disk Operating System (or as we refer to it GLaDOS). This language is being developed in the hope that it will become the standard language to write programs in for GLaDOS. We based the language largely off of JavaScript, but it is influenced by C#. A few things were influenced by Go and will be noted in those sections. The following points describe, in general, what is borrowed from each language and what differs from both.

1. G-Script can assign a reference to a library in a variable when it is imported through the "as" keyword. Ex: import GLaDOS.IO.out as Out.
2. G-Script lines are terminated by whitespace rather than a semi-colon (';').
3. G-Script has an "Incinerator" which acts, for the most part, as a standard Garbage Collector.
4. G-Script utilizes Cores rather than functions
5. G-Script does not have classes, instead Cores are used to wrap data and functions into a class.
6. G-Script Cores have multiple return types.

Note: The origin of the name G-Script is not known to anyone but the creators of the language, any attempt to find out what G-Script stands for will result in death.

1.1 Hello World

```
import GLaDOS.IO.out as Out
core main()
{
     Out.print("Hello World!")
}
```

1.2 Program Structure

G-Script is organized in a slightly different way than other languages. The key concepts are:

1. Programs are referred to as Cores. Every program in GLaDOS is run as a modular component that can be added, removed, or modified on the fly.
2. Cores can be treated as objects.
3. Specified functions can be treated as data members in all ways except values cannot be assigned to them.

1.3 Types and Variables

G-Script has two variable types, primitives and objects. Primitive variables are those that simply store raw data. Primitive example types are, int, float and string. Object example types are arrays and programmer defined object types. Primitives are declared by their type followed by a variable name.

1.4 Statements Differing from JavaScript and C#

Statement	Example

Experiment…fail	`Experiment`
	`{`
	`  ……`
	`  /*Code to test goes here*/`
	`  ……`
	`}`
	`Fail`
	`{`
	`  ……`
	`  /*Code for failure goes here*/`
	`  ……`
	`}`

Main Core Statement	`/*Import statements*/`
	
	`Core TestCore()`
	`{`
	`  Core Main()`
	`  {`
	`  /*Main code to execute goes here*/`
	`  }`
	
	`  /* rest of program Cores go here */`
	`}`

Data member declaration Statement	`Type |name| = value`
	`lie |killEveryone| = false`
	`Core |name|(){...}`

Import statement	import *library* as *variable*

1.5 Classes and Objects

New classes are declared by using a Core statement. Data members in classes are declared by wrapping the variable name in '|' symbols. This automatically creates getters and setters for the class data members. Cores can be wrapped in '|' symbol's as well, this allows the core to be treated as a data member. Wrapping a variable in '|' symbols also acts as the `this` operator. Here is an example of a class named `WeightedCompanionCube`.

```
Core WeightedCompanionCube(sideLength=3)
{
     int |sideLength| = sideLength
     int |weight| = 42

     Core |volume|()
     {
          return |sideLength| ^ 3
     }


     Core incinerate()
     {
          euthanize self
     }
}
```

Instances of classes as objects are created using the `create` operator, which allocates memory for the instance, and calls the constructor to create the instance and then returns the reference to that instance.

```
obj WCC1 = create WeightedCompanionCube()
obj WCC2 = create WeightedCompanionCube(5)
```

The Incinerator automatically detects "orphan" objects and promptly kills them off, freeing up the memory locations. The `euthanize` operator can also be used to "throw" objects into the Incinerator immediately freeing up the memory.

1.5.1 Accessibility

Accessibility types determine what regions of code can access specific data members. Each data members and functions, by default, are set to public access. However, there are a few other access types: (note: variables not declared as data members are always private).

Accessibility Level	Meaning
`public`	No access restriction
`private`	Access is restricted to within the current Core scope
`protected`	Access is restricted to within the class or classes related to the class

1.5.2 Cores

A **Core** serves as a wrapper for object declarations and to declare blocks of code that can be executed, much like functions. The **signature** of a Core must be unique within its parent Core.

Cores can be nested infinitely, and any child core can be accessed through its parent core by using the '.' operator.

Anonymous cores can be used to define blocks of code that cannot be accessed. These are primarily used when the functional language aspects of G-Script are implemented.

1.5.2.1 Constructors

G-Script allows both instance and static constructors. Constructors are defined as such:

```
Core testCore
{
     /* Constructor 1 */
     testCore(int i, string ghost)
     {
          ...
     }
     /* Default Constructor */
     testCore()
     {
          ...
     }
}
```

If no constructors are declared, then the parameters can be declared in the class declaration as in previous examples.

1.5.2.2 Properties

Properties in G-script are automatically set up when a variable is declared as a data member using the '|' symbols.

1.5.2.3 Flinging

Flinging is how classes and objects to send out notifications. Notifications are flung out from the event horizon and caught by FlingCatchers. FlingCatchers are implemented by using the "<-<" to add a FlingCatcher and "<x<" to delete a FlingCatcher. The following example attaches a FlingCatcher to a Turret Core.

```
Core Turret()
{
     int |ammunition| = 9001
     lie |ICanSeeYou| = false
     float |velocity| = 0.0
     Core flung()
     {
          while (self.move)
          {
               |velocity|++
          }
     }
     Core Main()
     {
          Self.move <-< FlingCatcher(flung)
          ...
     }

     ...
}
```

Flinging can also be used in conjunction with anonymous Cores. These can be implemented in the following format.

```
Object.action <-< FlingCatcher(Core{... code here ...})
```

1.6 Arrays

Arrays in G-Script work like standard arrays in JavaScript. Every array element is stored as an object, so arrays can contain any number of different types of elements The following code shows how an array is created and how to store elements in the array.

```
a1 = create ArrayCore()
a1[0] = create Turret()
a2 = create ArrayCore(10)
```

That code creates an array of unset size and stores it in a1. Then it creates a Turret and stores that in a1 at element 0. This type of array can automatically expand to any number of

elements. The last line of code, however, creates an array of size 10. No more than 10 elements can ever be stored in that array.

2. Lexical Structure

2.1 Programs

A G-Script program core is comprised of one or more script files. A script file contains all of the code, formatted usually as Unicode characters.

A program core is compiled using {{X}} steps:

1. *Recode*. This takes the Unicode / ascii / encoding of choice and converts it to our proprietary ASCE (Aperture Science Character Encoding). In this state GLaDOS can understand the code.
2. *Lexicore*. This is the Lexical analysis which translates the ASCE input stream to a series of tokens.
3. *Synthesis*. This analyzes the syntax and creates executable code for GLaDOS.

Note: in the recode step we convert the file to our ASCE format so any competitor, namely Black Mesa, will not even be able to open our files correctly. This way, we never need to worry about industrial espionage. The files are useless to anyone but us.

2.2 Lexical Analysis

2.2.1 Line Terminators

Line terminators divide the characters of G-Script source files into lines.

new-line:

Carriage return character (U+000D) or (ASCE+001D)
Line feed character (U+000A) or (ASCE+0010)
Next line character (U+0085) or (ASCE+2406)
Line separator character (U+2028) or (ASCE+1040)
Paragraph separator character (U+2029) or (ASCE+1041)
Core separator character (ASCE+0070)
Material Emancipation character (ASCE+0042)
Enrichment Center Termination character (ASCE+EC00)

2.2.2 Comments

Three forms of comments are supported: Single-line comments, multi-line comments, and break-point comments. ***Single-line comments*** are started by the // characters and end at the end of the line. ***Multi-line comments*** are started by /* characters and extend through multiple lines until the */ character sequence is encountered. ***Break-point comments*** are started by using the !!break!! character sequence. This comments out the rest of the document, it "breaks" the document into two sections: code and comments.

2.2.3 White Space

White space is defined as any character with Unicode class Zs, as specified below. White space characters are defined in ASCE by class S.

Whitespace:
- Any character with Unicode class Zs
  - Horizontal tab character (U+0009)
  - Vertical tab character (U+000B)
  - Form feed character (U+000C)
- Any character with ASCE class S
  - Horizontal tab character (ASCE+3012)
  - Vertical tab character (ASCE+3014)
  - Form feed character (ASCE+3015)
  - General space character (ASCE+3000)
  - Non-general space character (ASCE+3001)

2.4 Tokens

There are several kinds of tokens: identifiers, keywords, literals, operators, and punctuators. White space and comments are not tokens, though they act as separators for tokens.

token:
- *identifier*
- *keyword*
- *integer-literal*
- *float-literal*
- *character-literal*
- *string-literal*
- *operator-or-punctuator*

Valid Tokens: `Core, Euthanize, as, string, int, float, char, lie, import, experiment, fail, fling, flingCatcher, goto`

2.4.1 Keywords different from JavaScript and C#

A ***keyword*** is a reserved word that cannot be used as an identifier.

New Keywords:

```
Core
FlingCatcher
Euthanize
Experiment
Fail
lie
```

Removed Keywords:

```
Class
Function
Try
Catch
boolean
EventHandler
```

3. Basic Concepts

3.1 Application Startup

Application startup occurs when the runtime environment calls a designated Core, which is the application's entry point. This entry point Core can only have one or two signatures:

```
Core Main(){...}
Core Main(argumentArray){...}
```

The runtime starts the program by executing that Core within the program's Core. Every runtime is completely independent of one another. If one runtime crashes it has no effect on the other active cores. In this way the GLaDOS runtime system is resilient.

3.2 Application Termination

Application termination returns control to the execution environment. The Main Core will continuously loop automatically, unless specified otherwise. The Main Core can be terminated by the `Euthanize Main` command. When the Main Core is euthanized, the entire runtime for that program is shutdown. All the memory is returned to the system, and any trace of the program is erased from active memory.

3.3 Scope

G-Script utilizes dynamic scope. While designing the language we decided that if some value is computed at one point in the program, it should be available to be used at any point after it was computed. Dynamic scope does this wonderfully.

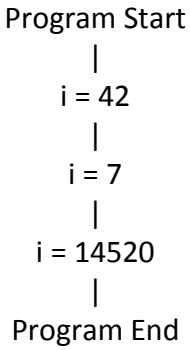
On the next page, the code on the left is an example that shows how scope works in G-Script. It will output the integer 14520. Because of dynamic scope the last value to be set to i gets printed out. If this was static scope then 42 would be printed out. On the right is a scope diagram

```
import GLaDOS.IO.out as Out

Core ScopeExample()
{
     Core Main()
     {
          int i = 42
          A()
          B()
          C()
          Out.print(i)
     }

     Core A()
     {
          int i = 7
     }

     Core B()
     {
          Int i = 14520
     }
```



3.4 Automatic Memory Management

G-Script automatically manages memory for the developer. This makes it much easier to concentrate on designing the program. Most modern programming languages use a *garbage collector*, G-Script, however, uses an *incinerator*. The *incinerator* is actually a part of the GLaDOS and is in fact an artificial intelligence that scans the active memory. This artificial intelligence can detect memory leaks, orphaned objects, and anything else that might be detrimental to the program or the system as a whole. The thing that sets the *incinerator* apart from normal *garbage collectors*, besides the fact that it is an artificial intelligence and better in every way, is the ability to call upon the *incinerator* in code to "terminate" any object that the developer knows will not be needed anymore.

4. Types

4.1 Primitive types

```
int, float, string, char, lie
```

To clarify `lie`: `lie` is the Boolean primitive type in G-Script. But it does have the distinct difference that the artificial intelligence in GLaDOS will be very disappointed if `lie` equals `false`.

4.2 Object Types

```
arrays, flingCatchers, user defined Cores
```

5. Variables

Variables point to storage locations. In G-Script primitives have types that are declared at the time of variable creation. All other variables are considered object and do not need to be declared in any special way, although objects **must** be created by using the `create` operator. G-Script is what we like to call a "type-protected" language. In operations that use more than one type of variable, one or more of the variables **must** be converted so that every variable in the operation is of the same type. This protects variables from having strange, unwanted values stored in them. Objects though have less strict restrictions on type conversions. This allows primitives values to be protected but gives the developer much more flexibility when creating and using their own objects.

Variable types:

1. Static
    a. static int count = 0
    b. static turret1 = create Turret()
2. Normal:
    a. int count = 0
    b. turret1 = create Turret()
3. Parameter
    a. int count = 0
    b. turret1 = create Turret(count)

6. Parameter Passing

6.1 Method

Parameter passing in G-Script is fairly straight forward. Parameters can **only** be passed into cores. There is no Out, or InOut methods of parameter passing in G-Script. This is done to keep things simple. In order to get data out of cores, we decided to allow multiple return types. This is directly influenced by Go.

Parameters are passed into the core by order, just like in JavaScript and C#. Primitives are passed in by value and all other objects are passed in by reference. This is done because passing all objects in by value would create too much overhead, but in general passing everything in by reference could cause too many problems. This is why we pass primitives in by value, there is little to no overhead to do so.

6.2 Examples

Examples of passing parameters into a core:

```
Cake(“flour”, “eggs”, “Alpha Resins”, “rhubarb”)
Turret(9001, arrayOfTargets)
```

Examples of returning one and more than one value:

```
Incinerated = turret1.isIncinerated()
output1, output2, outputN = experiment1()
```

7. Conversions

Conversions are fairly simple in G-Script. There are no implicit conversions in G-Script, however we packed G-Script with a vast library of cores that convert any primitive into any other. They all follow the same naming convention. The general format for these predefined functions is: `TypeConvertingFrom-to-TypeConvertingTo(Variable)`. For example:

Convert an int to a string: `int answer = int-to-string(42)`
Convert a string to an int: `string agent = string-to-int("007")`

Although we provide these cores in the base library that are included in every program, we allow these to be overridden if the developer wishes to do so.

In the event of a conversion not being possible, the core will return the default value for the type it is supposed to convert to.

8. Statements

8.1 Experiment…Fail

The experiment…fail structure is exactly like try…catch, in C#. The only exception is that in G-Script when an experiment block fails, the artificial intelligence in GLaDOS might execute a scientist. This is an unfortunate glitch in the GLaDOS subsystem right now. We are working to find a fix, just be careful when testing code until further notice.

```
Experiment
{
	……
	/*Code to test goes here*/
	……
}
Fail
{
	……
	/*Code for failure goes here*/
	……
}
```

8.2 import *library* as *variable*

The import statement, imports the entire libraries that are specified into the current runtime. All runtimes pull from the same library set, so any upgrades to the libraries are automatically implemented; the program simply has to be restarted. The as keyword puts the reference to the library into a specified variable, this allows code to be a little more readable. Instead of having half a line of code dedicated to drilling down into the library, that part of the library can be set to a single variable. This keeps code nice and clean, and more confusing for Black Mesa if they try to steal our code. The general format is the section title, convenient isn't it?

9. Example Programs

9.1 Example

```
import GLaDOS.IO as IO
import GLaDOS.dataNet as DATA
import GLaDOS.labControls.lethal as killswitch

Core Example1()
{
     Core Main()
     {
          IO.out.print("Enter your name:")
          string userInput = IO.in.keyboard()
          IO.out.blankLine()

          Experiment
          {
               target = DATA.find(userInput)
          }
          Fail
          {
               IO.out.print("There was an error, you\n" +
                    "do not exist.  Therefor you are an" +
                    "anomaly.\n I cannot stand for this.\n")
               IO.out.print("'Fixing' anomaly now...")
               neurotoxin = killswitch.readyNeuroToxin()
               neurotoxin.ready <-< FlingCatcher(Core{
                    neurotoxin.release(1, 5000)
                    euthanize self // terminates program
               })
          }

          IO.out.print("Hello " + userInput + ", you can" +
               "trust me, I have your best interest in mind.")

          target = DATA.find(userInput)
          targetHome = target.home
          targetHome.dispatchPartyAssociate()
     }
}
```

9.2 Example

```
import ApertureScience.ExperimentObjects as ASObjects

Core Example2()
{
     Core Main()
```

```
	{
		d1, d2, d3, Ch01 = setupExperiment(42)
		Ch01.subjectEnter <-< FlingCatcher(Core{
			d1.activate()
			d2.announce()
			d3.lie()
			euthanize self
		})
	}

	Core setupExperiment(number)
	{
		chamber01 = create ASObjects.createTestChamber(1, 50,
			142, 3.141592654, true, 9001, number, 62045, 10,
		72)
		Danger = create Array()
		Danger[0] = chamber01.turrets
		Danger[1] = chamber.AcidDepth
		Danger[2] = “Cake”
		Return Danger[0], Danger[1], Danger[2], chamber01
	}
}
```

9.3 Example

```
import GLaDOS.IO.Out as Out
import AperatureScience.portalEnviornment as portal

Core startTest()
{
	Core Main()
	{
		if(subject.awake)
		{
			Out.speak(“Hello and again welcome to the
				Aperture Science Computer-Aided
				Enrichment Center. We hope your brief
				detention in the relaxation vault has
				been a pleasant one. Your specimen has
				been processed and we are now ready to
				begin the test proper. Before we start,
				however, keep in mind that although fun
				and learning are the primary goals of
				the enrichment center activities,
				serious injuries may occur. For your
				own safety, and the safety of others,
				please refrain from”)

			portal.open(50, 10042, 65845, 60)
		}
	}
}
```

10. Conclusion

At this point you might be asking yourself, “Why did you guys invent G-Script.” The answer is not simple, but if there was a simple answer it would be, “why not?” The longer answer has two parts. The first reason is we needed a language to write programs for GLaDOS. When considering different languages to use on GLaDOS, we decided none were a perfect fit, so G-Script was born. The second part to that is, if we developed a proprietary language, then it would be harder for our competitors, *ahem* Black Mesa, to steal our ideas and technology if they were to procure any of our code. That is why G-Script has a few quirks about it. It is not supposed to be all too easy to read, but we want it to be easy to write.

G-Script is better than JavaScript and C#, the following sections explains why.

C#, no offence to the designers, can be annoying to work with at times if you are not experienced in the language. This is mostly because the syntax is very heavy. Too much has to be typed to do so little. Compared to other languages of the same level, C# is fairly lightweight, but we needed something even more so. Also C# run in a runtime not developed by us, this is problematic. We simply cannot deploy technologies based upon a technology that one of our competitors developed, it is simply not acceptable.

JavaScript, while is a very good language, is not good enough for us. We need a fast language, this means it has to be compiled. JavaScript is interpreted, so it is slower. But not only is it slower, it is usually run in a browser, which means it is further removed from direct access to the hardware that we need to interface with.

G-Script is not really all too different from any modern language, but we needed something only slightly different from what was out there.