GLaDOS

Language Specification

Version 42

Geeky Language and Depraved Original Syntax

"pa' DIchDaq taH ro'qegh'Iwchab"
"There will be blood pie."

"ro'qegh'Iwchab ghaH Qot"
"The blood pie is a lie."

A Neal Tanner Production

WARNING:

The following document was prepared by a sarcastic and snarky individual. You have been warned.

1. Introduction

GLaDOS is a bizarre, geeky, twisted, object-oriented, interpreted and strongly-typed programming language. Based on Javascript and LOGO, but differing in the following ways:

1. All keywords in the language have been translated into Klingon.
2. The number of basic types has been expanded from Var, adding Number, Word, and Boolean.
3. GLaDOS uses the interpreted aspects of LOGO and Javascript.
4. GLaDOS is designed to be used with an integrated IDE, similar to the one LOGO uses.
5. And whatever other goofy stuff I decide to throw in.

1.1 Hello world

```
Qap helloWorld ()
tagh
        jatlh("Hello World");
pItlh


helloWorld();    // Outputs Hello World
```

1.2 Program structure

The key organizational concepts in GLaDOS are as follows:

1. There is no required `Main` necessary to start a GLaDOS program.
2. Functions can be defined individually or as part of a class.
3. Individual statements can also be entered, on a line by line basis.
4. GLaDOS operates in an interpreted environment.
5. Individual statements, including class declarations are entered into the interpreter.
6. GLaDOS functions and class definition are entered in the built-in editor.

This example

```
Segh arrayTest
tagh
      'ang mI'DaH dataArray;
      Qap arrayTest()
      tagh
            dataArray = new DaH();
```

```
        pItlh

        'ang Qap add(mI' x)
        tagh
              dataArray.chel(x);
        pItlh

        'ang Qap retrieve(mI' index)
        tagh
              mI' returnValue := 0;
              if((index >= 0) && (index < dataArray.toghta'))
              tagh
                    returnValue := dataArray[index];
              pItlh
              tatlh returnValue;
        pItlh
  pItlh

  arrayTest myTest = chu' arrayTest();
```

declares a class named `Test`. The fully qualified name of this class is `Test`. The class contains several members: an array field named `dataArray`, a constructor, and a method named `add` and `retrieve`. An instance of the class is then declared.

1.3 Types and variables

There are two kinds of types in GLaDOS: ***value types*** and ***reference types***. Variables of value types directly contain their data whereas variables of reference types store references to their data, the latter being known as objects. With reference types, it is possible for two variables to reference the same object and thus possible for operations on one variable to affect the object referenced by the other variable.

1.4 Statements Differing from Javascript and LOGO

1.4.1 Translation Guide

Why is this here? The obvious reason. It had to go SOMEWHERE. It also helps explain the next section.

English meaning	Klingon keyword		English meaning	Klingon keyword
class	Segh		new	chu'
object	bep		if	chugh
public	'ang		more (else)	latlh
protected	QaD		truth	vIt
private	So'		lie	Qot
this	vam		open	poSmoH
function	Qap		speak (print)	jatlh
beginning	tagh		cut	pe'
end	pItlh		to / at	Daq
return	tatlh		blood (for drawing)	'Iw
while	vIS		on	Daq
for	vaD		off	lItha'
words	mu'mey		go	jaH
number	mI'		back	DoH
number be accurate (float)	mI'qar		right	nIH
array	DaH		left	poS
read only (constant)	laDneH		filled	tebta'
boolean (which is true)	teHbogh		nothing (null)	pagh
add	chel		numbered (size)	toghta'
remove	teq		battle (event)	may'
deal (handle)	Da			

1.4.2 Statement List

Statement	Example

Expression statement	`mI' x;`
	`x := 5;`
	`mI' y = x + 1;`

If (chugh) statement	`mI' x := 5;`
	`chugh(x > 4)`
	`tagh`
	`  jatlh("X is greater than 4.\n");`
	`pItlh`

Else/More (latlh) statement	`mI' x := 5;`
	`chugh(x == 4)`
	`tagh`
	`  jatlh("X is 4\n");`
	`pItlh`
	`latlh`
	`tagh`
	`  jatlh("X is not 4\n");`
	`pItlh`

While (vIS) statement	`mI' x := 5;`
	`vIS(x == 5)`
	`tagh`
	`  jatlh("X is: " + mI'.Daqmu'mey(x) + "\n");`
	`  x--;`
	`pItlh`

For (vaD) statement	`vaD(mI' i := 0; i < 5; i++)`
	`tagh`
	`  jatlh(mI'.Daqmu'mey(i));`
	`pItlh`

Output (jatlh) statement	`jatlh("This is some output.");`

Function (Qap) statement	`Qap printWords(mu'mey inputWords)`
	`tagh`
	`  jatlh(inputWords);`

	`pItlh`

Return (tatlh) statement	`Qap calculate(mI' x, mI' y)`
	`tagh`
	`  tatlh x + y;`
	`pItlh`

1.5 Classes and objects

New classes are created using class declarations.

The following is a declaration of a simple class named `Point`:

```
'ang Segh Point
tagh
      'ang mI' x, y;
      'ang Point(mI' x, mI' y)
            tagh
           vam.x := x;
           vam.y := y;
      pItlh // end constructor
pItlh // end class
```

Instances of classes are created using the `new` operator, which allocates memory for a new instance, invokes a constructor to initialize the instance, and returns a reference to the instance.

```
Point p1 := chu' Point(0, 0);
Point p2 := chu' Point(10, 20);
```

The memory occupied by an object is automatically reclaimed when the object is no longer in use. It is neither necessary nor possible to explicitly deallocate objects in GLaDOS, though you may be able to coax the garbage collector into action early.

1.5.1 Accessibility

Each member of a class has an associated accessibility, which controls the regions of program text that are able to access the member. There are three possible forms of accessibility. These are summarized in the following table.

Accessibility	English word	Meaning
`'ang`	public	Access not limited.
`QaD`	protected	Access limited to this class or classes derived from this class
`So'`	private	Access limited to this class. This is the default accessibility, if not otherwise stated.

1.5.2 Fields

A field is a variable that is associated with a class or with an instance of a class.

```
'ang Segh Color
tagh
      'ang laDneH Color Black := new Color(0, 0, 0);
      'ang laDneH Color White := new Color(255, 255, 255);

      So' mI' red, green, blue;

      'ang Color(mI' red, mI' green, mI' blue)
      tagh
            this.red   := red;
            this.green := green;
            this.blue  := blue;
      pItlh // end constructor
pItlh // end class
```

1.5.3 Methods

A ***method*** is a member that implements a computation or action that can be performed by an object or class.

The ***signature*** of a method must be unique in the class in which the method is declared.

1.5.3.1 Constructors

GLaDOS supports both instance and static constructors. An ***instance constructor*** is a member that implements the actions required to initialize an instance of a class. A ***static constructor*** is a member that implements the actions required to initialize a class itself when it is first loaded.

1.5.3.2 Properties

Properties are a natural extension of fields. Both are named members with associated types, and the syntax for accessing fields and properties is the same. However, unlike fields, properties do not denote storage locations. Instead, properties have ***accessors*** that specify the statements to be executed when their values are read or written.

1.5.3.3 Events

An ***event*** (may') is a member that enables a class or object to provide notifications. Clients react to events through ***event handlers***. Event handlers are attached using the `chelmay'Da` function and removed using the `teqmay'Da` function. The following example attaches an event handler for a `mouseClick` event of `myList`.

```
List myList := chu' List();
myList.chelmay'Da(mouseClick);  // Add the event handler.
myList.teqmay'Da(mouseClick);  // Remove the event handler.
```

1.6 Arrays

An ***array*** is a data structure that contains a number of variables that are accessed through computed indices. The variables contained in an array, also called the ***elements*** of the array, are all of the same type, and this type is called the ***element type*** of the array. Arrays can be dynamically increased, after being defined. The following example allocates a one-dimensional, a two-dimensional, and a three-dimensional array.

```
mI'DaH a1 := new DaH(10);
mI'DaH a2 := new DaH(10, 5);
mI'DaH a3 := new DaH(10, 5, 2);
```

The `a1` array contains 10 elements, the `a2` array contains 50 (10 × 5) elements, and the `a3` array contains 100 (10 × 5 × 2) elements.

2. Lexical structure

1. Programs

A GLaDOS ***program*** consists of one or more ***source files***. A source file is an ordered sequence of (probably Unicode) characters.

Conceptually speaking, a program is compiled using three steps:

1. Transformation, which converts a file from a particular character repertoire and encoding scheme into a sequence of Unicode characters.
2. Lexical analysis, which translates a stream of Unicode input characters into a stream of tokens.
3. Syntactic analysis, which translates the stream of tokens into executable code.

2. Grammars

This specification presents the syntax of the GLaDOS programming language where it differs from Javascript and LOGO.

2.2.1 Lexical grammar where different from Javascript and LOGO

string	::==	" *character list*	*space*	*character list*"
identifier	::==	*character*	*character list*	*digit list*
	::==	*underscore*	*character list*	*digit list*
character list	::==	*character*	*character list*	
	::==	*character*		
digit list	::==	*digit*	*digit list*	
	::==	*digit*		
character	::==	a | b | ... | y | z		
	::==	A | B | ... | Y | Z		
digit	::==	0 | 1 | ... | 8 | 9		
space	::==	SPACE character		
underscore	::==	_		

2.2.2 Syntactic ("parse") grammar where different from Javascript and LOGO

function-statement:

Qap *function-identifier* (*argument-list*)

tagh

embedded-statement

pItlh

if-statement:

```
chugh ( boolean-expression )
tagh
        embedded-statement
pItlh
```

if-else-statement:

```
chugh ( boolean-expression )
tagh
        embedded-statement
pItlh
latlh
tagh
        embedded-statement
pItlh
```

if-else-if-statement:

```
chugh ( boolean-expression )
tagh
        embedded-statement
pItlh
latlh chugh ( boolean-expression )
tagh
        embedded-statement
pItlh
```

for-statement:

```
vaD ( initialization-statement ; boolean-expression ; increment-statement )
tagh
        embedded-statement
pItlh
```

while-statement:

```
vIS ( boolean-expression )
```

```
tagh
        embedded-statement
pItlh
```

2.2.3 Grammar notation

The lexical and syntactic grammars are presented using **BNF *grammar productions***. Each grammar production defines a non-terminal symbol and the possible expansions of that non-terminal symbol into sequences of non-terminal or terminal symbols. In grammar productions, *non-terminal* symbols are shown in italic type, and `terminal` symbols are shown in a fixed-width font.

The first line of a grammar production is the name of the non-terminal symbol being defined, followed by a colon. Each successive indented line contains a possible expansion of the non-terminal given as a sequence of non-terminal or terminal symbols. For example, the production:

while-statement:
```
vIS  (  boolean-expression  )
tagh
        embedded-statement
pItlh
```

defines a *while-statement* to consist of the token `vIS`, followed by the token "`(`", followed by a *boolean-expression*, followed by the token "`)`", followed by `tagh`, followed by an *embedded-statement* and finishing with `pItlh`.

When there is more than one possible expansion of a non-terminal symbol, the alternatives are listed on separate lines. For example, the production:

statement-list:
statement
statement-list statement

defines a *statement-list* to either consist of a *statement* or consist of a *statement-list* followed by a *statement*. In other words, the definition is recursive and specifies that a statement list consists of one or more statements.

Alternatives are normally listed on separate lines, though in cases where there are many alternatives, the phrase "one of" may precede a list of expansions given on a single line. This is simply shorthand for listing each of the alternatives on a separate line. For example, the production:

real-type-suffix: one of
F f D d M m

is shorthand for:

real-type-suffix:
F
f
D
d
M
m

2.3 Lexical analysis

2.3.1 Line terminators

Line terminators divide the characters of a GLaDOS source file into lines.

new-line:
Carriage return character (U+000D)
Line feed character (U+000A)
Carriage return character (U+000D) followed by line feed character (U+000A)
Next line character (U+0085)
Line separator character (U+2028)
Paragraph separator character (U+2029)

2.3.2 Comments

GLaDOS inherits its comment structure from Javascript. Two forms of comments are supported: single-line comments and delimited comments. ***Single-line comments*** start with the characters `//` and extend to the end of the source line. ***Delimited comments*** start with the characters `/*` and end with the characters `*/`. Delimited comments may span multiple lines. Comments do not nest.

2.3.3 White space

White space is defined as any character with Unicode class Zs (which includes the space character) as well as the horizontal tab character, the vertical tab character, and the form feed character.

whitespace:
Any character with Unicode class Zs
Horizontal tab character (U+0009)
Vertical tab character (U+000B)
Form feed character (U+000C)

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
digit-list
real-literal
digit-list . *digit-list*
character-literal:
character
string-literal:
" *character-list* "
" *character-list* *space* *character-list* "
mathematical-operator:

```
+       -       *       /       %
^       +=      -=
```

logical-operator:

```
==      !=      >=      <=
```

punctuator:

```
(       )       [       ]       ;
"       '               .
```

2.4.1 Keywords different from Javascript or LOGO

A ***keyword*** is an identifier-like sequence of characters that is reserved, and cannot be used as an identifier except when prefaced by the `@` character.

Removed keywords:

All of them. GLaDOS uses no English words as keywords. Therefore, it uses NO keywords from Javascript or LOGO.

Come on, really, the whole language is in Klingon. What exactly did you expect?

New keywords:

All of them. Alright, alright, here's a list of them, too. And their translations from English. This list is also **not** exhaustive by any means. It is simply the words that were found to be necessary at the publication time.

English meaning	Klingon keyword		English meaning	Klingon keyword
public	`'ang`		words	`mu'mey`
protected	`QaD`		number	`mI'`
private	`So'`		number be precise (for floats)	`mI'qar`
function	`Qap`		if	`chugh`
beginning	`tagh`		more (else)	`latlh`
end	`pItlh`		truth	`vIt`
return	`tatlh`		lie	`Qot`
while	`vIS`		blood (for drawing)	`'Iw`
for	`vaD`		new	`chu'`
array	`DaH`		speak (print)	`jatlh`
boolean	`teHbogh`		nothing (null)	`pagh`

3. Basic concepts

3.1 Application Startup

Application startup occurs when a user enters their first command into the interpreter's command line. This can either be a user-define function, created using the built-in editor, or code typed directly into the command line, such as drawing commands.

GLaDOS programs do not require a defined Main method, or other such predefined startup point. They begin when and where the user wants them to.

3.2 Application termination

Application termination occurs when the currently active function has completed all its steps, or the user entered command has been fully interpreted and executed, or rejected. If a command is found to have the incorrect form, an error message is returned, and no execution occurs.

The same is true when GLaDOS executes a function. When a error is reached, function execution stops immediately, and returns an error message. Further execution is terminated, and control returns to the command line.

3.3 Scope

GLaDOS utilizes static scope, just as Javascript does.

```
Qap printStuff()
tagh
      laDneH mI' TIMES_TO_PRINT := 5;
      mI' x := 0;
      vaD(mI' i := 0; i < TIMES_TO_PRINT; i++)
      tagh
            jatlh("The value i of is: " + mI'.Daqmu'mey(i) + "\n");
      pItlh
      vIS(x < TIMES_TO_PRINT)
      tagh
            jatlh("The value of x is: " + mI'.Daqmu'mey(x) + "\n");
            x++;
      pItlh
pItlh

mI' x := 10;
mI' i := 9;
printStuff();
```

Scope	Name	Type	Value
0 ("Main")	x	mI'	10
	i	mI'	9
	printStuff	Qap	
1 (printStuff)	TIMES_TO_PRINT	laDneH mI'	5
	x	mI'	0 (initial value) 5 (after vIS)
2 (vaD)	i	mI'	5 (final value)
3 (vIS)	None	None	None

```
Qap printArray(mI'DaH someArray)
tagh
      mI' size := someArray.length;
      vaD(mI' i := 0; i < size; i++)
      tagh
            jatlh(mI'.Daqmu'mey(i) + ": " +
                  mI'.Daqmu'mey(someArray[i]) + "\n");
      pItlh
pItlh

Qap printStuff(mu'mey inputWords)
tagh
      jatlh("The inputted value was: " + inputWords + "\n");
pItlh

Qap addStuff(mI' a, mI' b)
tagh
      tatlh a + b;
pItlh

mI' x := 5;
mI' y := 6;
mI' z := addStuff(x, y);
mI'DaH myArray := new DaH[1, 2, 3, 4];
printArray(myArray);
```

```
printStuff(mI'.Daqmu'mey(z));
```

Scope	Name	Type	Value
0 ("Main")	x	mI'	5
	y	mI'	6
	z	mI'	11
	myArray	mI'DaH	1, 2, 3, 4
	addStuff	Qap	
	printArray	Qap	
	printStuff	Qap	
1 (addStuff)	a	mI'	5
	b	mI'	6
2 (printArray)	someArray	mI'DaH	1, 2, 3, 4
	size	mI'	4
3 (vaD in printArray)	i	mI'	4 (final value)
4 (printStuff)	inputWords	mu'mey	"11"

3.4 Automatic memory management

GLaDOS employs automatic memory management, which frees developers from manually allocating and freeing the memory occupied by objects. Automatic memory management policies are implemented by a ***garbage collector***. It differs from Javascript and LOGO in the following ways:

1. As GLaDOS is built on Javascript, it relies extensively on Javascript's own garbage collector.
2. Being an educational language, in the style of LOGO, GLaDOS performs its most intense garbage collection after a workspace has been completely closed out. Only when a workspace has been closed out can GLaDOS be sure that it is completely safe to garbage collect.
3. Active programming workspaces are garbage collected, but in a minimal fashion. It is impossible to be sure when an element loaded into the workspace will be needed.
4. Note: At this time, GLaDOS' author believes this to be the correct approach to garbage collection. However, he reserves the right to more closely examine the garbage collection practices of both LOGO and Javascript, and later revise his approach to garbage collection.

4. Types

GLaDOS types are divided into two main categories: ***Value types*** and ***reference types***.

4.1 Value types (different from Javascript and LOGO)

There are four value types in GLaDOS, which are the four basic data types defined in the language. They are `mI'` (integer), `mI'qar` (float), `mu'mey` (string), and `teHbogh` (boolean).

```
mI' x := 5;
mI'qar y := 1.05;
mu'mey someWords := "some words";
teHbogh test1 := vIt;   // True
teHbogh test2 := Qot;   // False
```

4.2 Reference types (differing from Javascript and LOGO)

All `DaH`s (arrays) and `bep`s (objects) are handled as reference types.

Arrays:

```
mI'DaH intArray := new DaH(3);
intArray[0] := 1;
intArray[1] := 2;
intArray[2] := 3;
mI'qarDaH floatArray := new DaH[1.3, 1.2, 1.1];
mu'meyDaH wordArray := new DaH["test", "test", "test"];
teHboghDaH booleanArray := new DaH();
booleanArray[0] := vIt;
booleanArray[1] := Qot;
```

Objects:

```
myClass test := pagh;
test := myClass();
myClass test2 := myClass();
```

5. Variables

Variables represent storage locations. Every variable has a type that determines what values can be stored in the variable. GLaDOS is a type-safe and strongly-typed language.

All basic variables (`mI'`, `mI'qar`, `mu'mey`, and `teHbogh`) are stack-dynamic. This is designed to allow for easy recursion with these variable types.

Objects and arrays (`bep` and `DaH`) are explicit heap-dynamic. This allows for easy management of dynamic storage.

For a detailed parameter passing explanation and example, please see section 6.

5.1 Variable categories

```
mI' – Number / Integer
      mI' x := 5;
      laDneH x1 := 6; // Constant integer
mI'qar – Accurate Number / Float
      mI'qar y := 1.05;
      laDneH mI'qar y1 := 1.20; // Constant float
mu'mey – Words / String
      mu'mey myWords := "my string"; // String
      laDneH mu'mey myWords2 := "some string"; // Constant string
teHbogh – Boolean
      teHbogh test1 := vIt;  // True
      teHbogh test2 := Qot; // False
      laDneH teHbogh test3 := vIt; // Constant true
DaH -  Array
      mI'DaH myArray := new DaH(5);
      myArray[0] := 1;
      myArray[1] := 2;
      myArray[2] := 3;
      myArray[3] := 4;
      myArray[4] := 5;
bep - Object
      Point p1 := new Point(5, 6);
```

6. Parameter Passing

6.1 Method

GLaDOS uses pass-by-value (In Mode) for the basic variable types (`mI'`, `mI'qar`, `mu'mey` and `teHbogh`). Pass-by-value is implemented by copying. Parameters passed have their values copied into new storage, set aside for the function parameters.

GLaDOS uses pass-by-reference (InOut mode) for objects and arrays. While these references are pointers back to the object or array, it is not possible to do pointer math on them. The only way to change the memory address they point at is to assign the reference to a new object or array.

6.2 Examples

Example:

```
Qap printArray(mI'DaH someArray)
tagh
      mI' size := someArray.toghta';
      vaD(mI' i := 0; i < size; i++)
      tagh
            jatlh(mI'.Daqmu'mey(i) + ": " + mI'.Daqmu'mey(someArray[i]) + "\n");
      pItlh
pItlh

Qap printStuff(mu'mey inputWords)
tagh
      jatlh("The inputted value was: " + inputWords + "\n");
pItlh

Qap addStuff(mI' a, mI' b)
tagh
      tatlh a + b;
pItlh

mI' x := 5;
mI' y := 6;
mI' z := addStuff(x, y);
mI'DaH myArray := new DaH[1, 2, 3, 4];
printArray(myArray);
```

```
printStuff(mI'.Daqmu'mey(z));
```

Activation Record:

Data	"Main"	*Local Variables:* *Name Type Value* x mI' 5 y mI' 6 z mI' 11 (after code execution)
	addStuff	*Local variables:* *None* *Parameters:* *Name Type Value* a mI' 5 b mI' 6 *Return Address:* ("Main", line 3)
	printArray	*Local variables:* *Name Type Value* size mI' 4 i mI' 4 *Parameters:* *Name Type Value* someArray mI'Dah myArray, "Main" *Return Address:* ("Main", line 5)
	printStuff	*Local variables:* *None* *Parameters:* *Name Type Value* inputWords mu'mey 11 *Return Address:*

		("Main", line 6)

Code	"Main"	mI' x := 5;
		mI' y := 6;
		mI' z := addStuff(x, y);
		mI'DaH myArray := new DaH[1, 2, 3, 4];
		printArray(myArray);
		printStuff(mI'.Daqmu'mey(z));

	addStuff	tatlh a + b;

	printArray	mI' size := someArray.length;
		vaD(mI' i := 0; i < size; i++)
		tagh
		jatlh(mI'.Daqmu'mey(i) + ": " +
		mI'.Daqmu'mey(someArray[i]) + "\n");
		pItlh

	printStuff	jatlh("The inputted value was: " + inputWords + "\n");

7. Conversions

A ***conversion*** enables an expression to be treated as being of a particluar type.

7.1 Implicit conversions (are bad in GLaDOS)

There are no implicit conversions conversions in GLaDOS. Passing a parameter that doesn't match the required type will trigger an interpreter error.

Even the output functions take a string as a parameter. All data to be outputted needs to be explicitly converted to mu'mey (translation: words) before it can be outputted.

Implicit conversions are not for true Klingon warriors. Only a worthless p'tak would expect a language to employ them.

7.2 Explicit conversions (much better in GLaDOS)

All type conversions in GLaDOS are explicit. Each meta class library possesses a `Daqmu'mey` function. (translation: To Words) This can be called, and then the matching data type passed in, for conversion to *words*. The function then returns the appropriate *words*.

For example:

```
mI' x := 5;
jatlh("The value of X is: " + mI'.Daqmu'mey(x) + "\n"); // Outputs 5
```

In this example, `x` is declared as `mI'`, assigned a value of 5, and passed to the output function. Notice that the meta-class function `mI'.Daqmu'mey` is called, and `x` is passed to it, for the explicit conversion.

Let's consider a more complicated example:

```
mu'mey y := "42";
mI' z := mu'mey.DaqmI'(y) + 5;
jatlh("The value of z is: " + mI'.Daqmu'mey(z) + "\n"); // Outputs 47
```

In this example, `y` is declared as `mu'mey`, and assigned a value of `"42"`. `z` is declared as `mI'`, then assigned the value of `y`, which is converted by a meta-class conversion function, which has also 5 added to it. Notice that the meta-class function `mI'.Daqmu'mey` is called, and `z` is passed to it, for the explicit conversion.

8. Statements

Regarding statements, GLaDOS differs from Javascript and LOGO in the following areas:

Statements:

1. GLaDOS follows the Javascript style of requiring a semi-colon at the end of each statement.
2. Statements can also be assignment or declaration statements.
3. Statements can be contained inside of blocks.

Declaration Statements:

1. Declaration statements consist of a type and an identifier name.
2. All variables must be declared, before they can be used.
3. Variables can be assigned a value when they are declared.

Assignment Statements:

1. Assignment statements use the `:=` operator.
2. `=` is not a valid operator for assignments.

Blocks:

1. Blocks do not use curly braces.
2. Blocks begin with `tagh`, and end with `pItlh`.
3. Blocks can be associated with loops (vaD or vIS), or chugh, latlh, and chugh latlh.

9. Example Programs

Simple Loop Example:

```
mI' x := 4;

vaD(mI' i := 0; i < x; i++)
tagh
      jatlh("The number is:");
      jatlh(mI'.Daqmu'mey(x + i) + "\n");
pItlh

vIS(x > 0)
tagh
      jatlh("The number is:");
      jatlh(mI'.Daqmu'mey(x) + "\n");
      x--;
pItlh
```

Change example:

```
Qap makeChange(mI' startAmount)
tagh
      mI' amount  := 0;
      mI' quarters      := 0;
      mI' dimes   := 0;
      mI' nickels := 0;
      mI' pennies := 0;
      amount := startAmount;
      quarters := mI'.bIng(amount / 25);
      amount := amount % 25;
      dimes := mI'.bIng(amount / 10);
      amount := amount % 10;
      nickels := mI'.bIng(amount / 5);
      amount := amount % 5;
      pennies := amount;
      jatlh("Original input was: " + mI'.Daqmu'mey(startAmount) + "\n");
      jatlh("Quarters: " +  mI'.Daqmu'mey(quarters) + "\n");
      jatlh("Dimes: " +  mI'.Daqmu'mey(dimes) + "\n");
      jatlh("Nickels: " +  mI'.Daqmu'mey(nickels) + "\n");
```

```
      jatlh("Pennies: " +  mI'.Daqmu'mey(pennies) + "\n");
pItlh // end makeChange

// Begin use example
makeChange(42);
// End use example
```

Drawing example:

```
// Draw a triangle
'Iw.nIH(45);
'Iw.cher(255, 0, 0);
'Iw.jaH(50);
'Iw.nIH(90);
'Iw.jaH (50);
'Iw.nIH(135);
'Iw.jaH(70);
'Iw.nIH(135);
// Turn off the blood
'Iw.lItha'();
// Move into the center
'Iw.jaH(20);
'Iw.nIH(45);
'Iw.jaH(20);
// Set the blood color
'Iw.cher(255, 0, 0);
// Fill the triangle
'Iw.tebta'();
'Iw.nIH(180);
'Iw.jaH(100);
'Iw.cher(0, 0, 0);
// Turn on the blood
'Iw.Daq();
// Perform a for loop, to draw a square.
vaD(mI' i := 0; i < 4; i++)
tagh
      'Iw.jaH(100);
      'Iw.nIH(90);
pItlh
// Turn off the blood
```

```
'Iw.lItha'();
// Move inside the square
'Iw.jaH(5O);
'Iw.nIH(9O);
'Iw.jaH(2O);
// Fill the square.
'Iw.tebta'();
```

Encrypt / Decrypt example:

```
mI' CHAR_CODE_A := 65;
mI' CHAR_CODE_Z := 90;
Qap decryptUnknown (mu'mey inputWords, mI' shiftValue)
tagh
      chugh(maxShiftValue >= 0)
      tagh
            mu'mey outputWords := "";
            outputWords += "Caesar " + shiftValue + ": ";
            outputWords += decrypt(inputWords, shiftValue);
            outputWords += "\n";
             jatlh(outputWords);
            decryptUnknown(inputWords, (shiftValue - 1));
      pItlh // end chugh
pItlh // end decryptUnknown

Qap decrypt (mu'mey inputWords, mI' shiftValue)
tagh
      tatlh shiftWords(mu'mey.DaqDung(inputWords), -shiftValue);
pItlh // end decrypt

Qap encrypt (mu'mey inputWords, mI' shiftValue)
tagh
      tatlh shiftWords(mu'mey.DaqDung(inputWords), shiftValue);
pItlh // end encrypt

Qap shiftWords (mu'mey inputWords, mI' shiftValue)
tagh
      mu'mey shiftedWords     := "";
      mI' shiftedCharCode     := 0;
      mI' shift         := shiftValue % 26;
```

```
      mI' difference          := 0;
      chugh(inputWords != "")
      tagh
            mI' currentCharCode := mu'mey.mI'Daq(inputWords, 0);

            chugh((currentCharCode >= CHAR_CODE_A) &&
                  (currentCharCode <= CHAR_CODE_Z))
            tagh
                  shiftedCharCode := currentCharCode + shift;
                  chugh(shiftedCharCode < CHAR_CODE_A)
                  tagh
                        difference := CHAR_CODE_A - shiftedCharCode;
                        shiftedCharCode := CHAR_CODE_Z - difference + 1;
                  pItlh  // end chugh
                  latlh chugh(shiftedCharCode > CHAR_CODE_Z)
                  tagh
                        difference := shiftedCharCode - CHAR_CODE_Z;
                        shiftedCharCode := CHAR_CODE_A + difference - 1;
                  pItlh  // end latlh chugh
                  shiftedWords += mu'mey.vo'mI'(shiftedCharCode);
            pItlh  // end chugh
            latlh
            tagh
                  // Character is whitespace.  Add to the string without shifting.
                  shiftedWords += mu'mey.vo'mI'(currentCharCode);
            pItlh  // end latlh
            shiftedWords += shiftWords(mu'mey.pe'(inputWords, 1),  shiftValue);
      pItlh  // end chugh
      tatlh shiftedWords;
pItlh // end shiftWords

// Begin use example
mu'mey testWords := "THIS IS A TEST STRING";
mu'mey resultWords := "";
mu'mey secondResultWords := "";
resultWords := encrypt(testWords, 3);
jatlh(resultWords);
secondResultWords := decrypt(resultWords, 3);
jatlh(secondResultWords);
```

```
decryptUnknown("HAL", 26);
// End use example
```

10. Conclusion

The language presented here is obviously an attempt at making as depraved a language as possible. If it wasn't, what other good reason is there for developing a entire programming language in Klingon? However, the language is not without its useful points. It combines elements from two very different languages, Javascript and LOGO, and melds them in a very interesting manner. If another language, with an English syntax, but a similar structure to GLaDOS were to be developed, it would have a number of advantages, especially for educational use. Let's look at Javascript and LOGO, the strengths of each, the weaknesses that have been left behind, and finally the advantages of the resulting language.

Javascript has a number of strengths as a development language, and for educational use. Javascript was intended as a client side web scripting language. Its popularity has allowed it to become prevalent in all modern web browsers. In a sense, Javascript is even more platform independent than Java. Javascript code that works in Mozilla Firefox on a PC will work equally well in Mozilla Firefox on a Mac, with no modifications. Stylistically, Javascript is similar to C, C++, C# and Java. This allows students of Javascript to find themselves at home in C, C++, C# and Java. Javascript is also object-oriented, allowing students to be introduced to these concepts, while at the same time teaching them the language syntax.

LOGO's principle strengths lie in its abilities as an educational language, something it was designed exclusively for. Its development environment is command-driven. A programmer enters a command, and sees an immediate result. This simplicity is valuable for students, allowing them to easily experiment with new commands, without the heavy framework of a full program to worry about. LOGO is strongly-typed, a concept the author finds valuable in enforcing good programming habits and styles. This is especially important for teaching students, as a weakly-typed language allows too many opportunities to develop sloppy programming habits.

Javascript does have several weaknesses that make it difficult to use as an educational language. First, it is weakly-typed. While weak typing is not necessarily a bad thing, the author believes that it can lead to beginning programmers forming sloppy habits and styles. Strong typing, as previously stated, is thought to help curb some of these tendencies. Second, it is difficult to program in Javascript without knowing some HTML and CSS as well. Producing a stylish looking program in Javascript requires knowledge of HTML and CSS as well. Even trying to format output from Javascript requires knowledge of HTML. This leads to teaching not only Javascript, but HTML and CSS as well. This takes away from time spent learning Javascript, and general programming concepts, which isn't desirable when teaching beginning programmers.

LOGO has several weaknesses that limit its usefulness as an educational language. First, it uses a syntax that is utterly dissimilar to any other modern language. While the plain English approach it takes to identifiers and function names makes it desirable for teaching completely new programmers, it is difficult to transition to another language's syntax from LOGO. When you get right down to it, many modern languages look like C. If you want to teach someone to program, and wanted them to be comfortable and familiar with the syntax of another language, you teach them a language that derives its style and syntax from C. Since LOGO doesn't do this, it would be a bit confusing for a new programmer to move from LOGO to Java. Second, LOGO is not object-oriented. It is a functional language, and doesn't support object-oriented design. This makes it impossible to teach any object-oriented design in LOGO, or even introduce the concept.

GLaDOS, or its English equivalent, LOGOScript, moves beyond these disadvantages, while retaining the strengths of each language. It embraces strong typing and uses simple names for its various variable types. Rather than integer, it uses *number* as its base numeric notation. Rather than string, it uses *words* as its multi-character storage. It utilizes the C-like syntax of Javascript, as seen in the example programs. Each statement ends with a semi-colon, mimicking all languages utilizing the syntax of C. It allows both a functional and object-oriented approach to programming. Since functional programming is simpler to work in for a beginning programmer, it allows this to start with. As the programmer grows more confident and skilled, they can add

functions to their programs, followed by objects when they are ready. The language's development environment is even intended to be platform independent. Designed to be built on Javascript, it is meant to run equally well in any modern browser, on any modern operating system.

GLaDOS / LOGOScript is also equally appropriate for any age programmer. The drawing commands are simple and straight forward, a clear inheritance from LOGO. This allows young learners to marvel at how easily they can make the language's on-screen avatar draw pictures. As this is a programming language, they can even save the steps, and reproduce the drawing at any time. Older students will find more value in the language's more advances features. Loops and if / else statements allow functional programs to be easily implemented. These can vary from something as simple as the change calculator, with its relatively uncomplicated math, to more complex character / string manipulation program like the encrypt / decrypt example.

In conclusion, GLaDOS / LOGOScript has the makings of a premiere educational language. Traditional educators will prefer the idea of GLaDOS' English-language cousin, LOGOScript. However, the GLaDOS will certainly appeal to one educator in particular. Given his love of Portal, Star Trek, Javascript and learning through pain, Alan Labouseur is sure to enjoy GLaDOS. Its use of Javascript-like syntax, combined with the discomfort and insanity of attempting to program in Klingon is sure to entertain and delight his twisted and perverse sensibilities.

11. Sources

English to Klingon:
http://klv.mrklingon.org/klvtab.html

Javascript:
http://www.w3schools.com/js/default.asp

Javascript Grammar / Statements:
http://home.cogeco.ca/~ve3ll/jstutor2.htm#co
http://www.wdvl.com/Authoring/JavaScript/Tutorial/grammar.html

Javascript reserved words:
http://en.wikibooks.org/wiki/JavaScript/Reserved_Words