Final Year Project Report

Full Unit - Interim Report

Scribble Programming Language

Blake Loring

A report submitted in part fulfilment of the degree of

BSc (Hons) in Computer Science

Supervisor: Adrian Johnstone

Department of Computer Science
Royal Holloway, University of London

March 4, 2014

Declaration

This report has been prepared on the basis of my own work. Where other published and unpublished source materials have been used, these have been acknowledged.

Word Count: 8,674

Student Name: Blake Loring

Date of Submission:

Signature:

Table of Contents

Abstract . . . 3

1 Introduction . . . 4

2 User Manual . . . 5

2.1 Language Examples . . . 5

2.2 Lexical . . . 12

2.3 Identifiers . . . 15

2.4 Types . . . 16

2.5 Operators & Expressions . . . 20

2.6 Statements . . . 27

2.7 Functions . . . 29

2.8 Packages & Importing . . . 31

3 Instruction Encoding . . . 33

3.1 Instructions List . . . 34

4 Type Inference . . . 36

4.1 Expression type inference in function matching . . . 37

4.2 Expression type inference in error checking . . . 38

5 Package Handling . . . 40

6 Garbage Collection . . . 42

7 Native Interface . . . 44

7.1 Structure . . . 44

Bibliography . . . 48

Abstract

The aim of the project is to create a platform independent programming language, parser and virtual machine capable of being embedded as a scripting language inside larger C++ applications.

Chapter 1: **Introduction**

The aim of my project was to create a programming language which could be used within C++ applications to make the development process easier and allow the extension of existing functionality. The main purpose of Scribble is to provide developers with a way to allow a programs users to extend or modify their program in an easy way without having to release the entire source code of the program. It was also designed with the intention of allowing more rapid development as a script can be recompiled and executed without having to recompile the entire program, often without even requiring the program to restart.

This approach is most commonly taken within the games industry with languages like LUA, UnrealScript, TorqueScript and JavaScript being used to let developers create a scene or manipulate a virtual world without needing to directly interact with the substantially more complex C or C++ source code in which these engines are usually written.

The design of my project was split into three key sections, the compiler handles the process of turning the input source into executable code using the syntax I have defined. The intermediate language acts as a stepping stone, a language the compiler can relatively easily generate code for which can then be turned into instructions for the virtual machine which will do the actual execution.

I decided to create a scripting language because they are an increasingly important part of software engineering, uses range from everyday software such as Microsoft office to highly specific software like virtual reality. I have used languages like LUA and Javascript (Through V8) in previous projects and have wanted to experiment with my own ideas about how a language should be written based on my experiences with the others.

My choices when defining syntax are the result of a desire to take what I consider the better parts of the Go programming language and make them usable within the context of a scripting language. GoLang also made me very interested in seeing whether a syntax could be defined so that non object oriented language was as capable and readable when writing equivalent programs as its OO alternatives (In this case Javascript would likely be the closest comparison).

Due to the scope of the project it would be extremely difficult to try and appropriately explain the entire language therefore this document there I have instead opted to include a user manual which details the syntax and semantics of the language as well as chapters on important parts of the implementation.

Chapter 2: **User Manual**

The following user manual describes the core features of the language syntax and semantics as well as information on how to use features like package importing and type inference.

2.1 Language Examples

The following examples will outline some of the features of the language by showing them in use in small programs.

2.1.1 Hello World

The hello world example will print "Hello World" to the screen.

```
//Import the console package allowing us to write to stdout
package console := import("console");

func main() {
 console.Log("Hello World\n");
}
```

2.1.2 Variables

The example below creates a variable and sets its value to a random integer between 0 and 1500 then prints it to the screen.

```
//Import the console package allowing us to write to stdout
package console := import("console");

//Import the sys package for the RandomInt function and string casts
package sys := import("sys");

func main() {

 //Write a message about generating the value
 console.Log("Generating a random number between 0 and 1500\n");

 //Create a new variable with a random number between 0 and 1500 in it
 var random := sys.RandomInt(1500);

 //Write the random number to stdout
 console.Log(sys.String(random));
 console.Log("\n");
}
```

2.1.3 Flow Control

The next example uses if statements to control the flow of execution, generating a random value between 0 and 2000 and then executing different code depending on whether the value is greater than 1000

```
//Import the console package allowing us to write to stdout
package console := import("console");

//Import the sys package for the RandomInt function and string casts
package sys := import("sys");

func main() {

 //Generate the random integer value j
 var j := sys.RandomInt(2000);

 if j > 1000 then {
  console.Log("J is > 1000\n");
 } else if j < 1000 then {
  console.Log("J is < 1000\n");
 } else {
  console.Log("J is 1000\n");
 }

}
```

The example below uses a for loop to sum all the values between 0 and a randomly generated value and then print it out to the screen.

```
//Import the console package allowing us to write to stdout
package console := import("console");

//Import the sys package for the RandomInt function and string casts
package sys := import("sys");

func main() {

 //Generate a number between 0 and 150 to find the sum of
 var random := sys.RandomInt(150);

 //Write a message about finding the sum
 console.Log("Finding the sum of all values between 0 and ");
 console.Log(sys.String(random));
 console.Log("\n");

 //Create a sum value to store the result
 var sum := 0;

 //Loop i between 0 and random and add i to sum at each iteration
 for var i := 0; i < random; i++ do {
  sum := sum + i;
 }
```

```
 //Write the sum to stdout
 console.Log(sys.String(sum));
 console.Log("\n");
}
```

The last flow control example uses a while loop to loop the variable i between 100 and 0, printing out the value of i at each step in the execution.

```
//Import the console package allowing us to write to stdout
package console := import("console");

//Import the sys package for the RandomInt function and string casts
package sys := import("sys");

func main() {

 //Generate the random integer value j
 var j := sys.RandomInt(2000);

 //Initialize the variable i to 100
 var i := 100;

 //Loop while i > 0 print i and then decrement it by 1
 while i > 0 do {
  console.Log(sys.String(i));
  console.Log("\n");
  i--;
 }

}
```

2.1.4 Functions

In the following example two mutually recursive functions will be defined to check whether a value is odd or even. It also uses if statements to detect base cases for the recursive functions.

```
//Import the console package allowing us to write to stdout
package console := import("console");

//Import the sys package for the RandomInt function and string casts
package sys := import("sys");

func Even(n : int) : bool {

 //If n = 0 then it is a base case return true
 if n = 0 then {
  return true;
 }

 return Odd(n-1);
```

```
}

func Odd(n : int) : bool {

 //If base case then return false
 if n = 0 then {
  return false;
 }

 return Even(n-1);
}

func main() {

 //Generate a number between 0 and 150 to find the sum of
 var random := sys.RandomInt(150);

 //Write a message about odd/even
 console.Log("Checking whether ");
 console.Log(sys.String(random));
 console.Log(" is even\n");

 var even := Even(random);

 if even then {
  console.Log("The value is even\n");
 } else {
    console.Log("The function is odd\n");
 }

}
```

2.1.5 Arrays

This example will populate an array with values from 0 to 100 and then print out all of its elements to the screen in reverse order (starting at index 99 and going until index 0)

```
//Import the console package allowing us to write to stdout
package console := import("console");

//Import the sys package for the RandomInt function and string casts
package sys := import("sys");

func main() {

 //Create an array of 100 integers
 var intArray := [100]int;

 //Initialize the array so that its values range between 0 and len(array)
 for var i := 0; i < len(intArray); i++ do {
  intArray[i] := i;
 }
```

```
 //Loop between len(intArray) - 1 and 0 printing out the value at each index.
 for i := len(intArray) - 1; i > -1; i-- do {
  console.Log(sys.String(intArray[i]));
  console.Log("\n");
 }

}
```

2.1.6 Structures

The next example defines a structure User to hold a users information, it then defines a function which takes a user as an argument and prints it to the screen and a main function which creates and outputs a users data.

```
//Import the packages for console and system libraries
package console := import("console");
package sys := import("sys");

//Define the user structure with name, email and age fields
type User := struct {
 name : string,
 email : string,
 age : int
}

//Define the function PrintUser which writes
//the users name email address and age to stdout
func PrintUser(user : User) {
 console.Log("Name: ");
 console.Log(user->name);

 console.Log("\nEmail: ");
 console.Log(user->email);

 console.Log("\nAge: ");
 console.Log(sys.String(user->age));
 console.Log("\n");
}

func main() {

 //Create a user for John Smith aged 32
 var user := User { "John Smith", "js@email.com", 32 };

 //Print the users details to the screen.
 PrintUser(user);

}
```

2.1.7 Quick Sort

This example takes the functions and user structure defined in the previous example and implements a function to sort an array of these users based on their age.

```
package sys := import("sys");
package console := import("console");

type User := struct {
 name : string,
 email : string,
 age : int
}

/**
 * The function Younger will return true if a user is younger than
 * another, used by the QuickSort function when comparing users
 */

func Younger( left : User, right : User) : bool {

 if left->age < right->age then {
  return true;
 }

 return false;
}

/**
 * The function Older will return true if a user is older than
 * another, used by the QuickSort function when comparing users
 */

func Older( left : User, right : User) : bool {

 if left->age > right->age then {
  return true;
 }

 return false;
}

/**
 * The print user function outputs information about a
 * generated user to the screen
 */

func PrintUser(user : User) {
 console.Log("Name: ");
 console.Log(user->name);

 console.Log("\nEmail: ");
 console.Log(user->email);
```

```
 console.Log("\nAge: ");
 console.Log(sys.String(user->age));
 console.Log("\n");
}

/**
 * This function calls PrintUser for every element of a user array.
 */

func PrintUsers(users : array(User)) {

 for var i := 0; i < len(users); i++ do {
  PrintUser(users[i]);
  console.Log("\n");
 }

}

/**
 * The QuickSort function takes an array and the index of the lowest and
 * highest element it should sort between and sorts it by moving any value
 * lower than a selected pivot value to the left of it and any higher value
 * to the right and then repeating for the arrays to the left and right of
 * the pivot value until the array is sorted.
 */

func QuickSort( n:array(User), low : int, high : int) {

 var i := low;
 var j := high;

 //Take the pivot value to be the value in the middle
 var pivot := n[i];

 while i <= j do {

  while Younger(n[i], pivot) do {
   i++;
  }

  while Older(n[j], pivot) do {
   j--;
  }

  // As long as i <= j swap n[i] and n[j] and increment them both
  if i <= j then {
   var temp := n[i];
   n[i] := n[j];
   n[j] := temp;

   i++;
   j--;
  }
```

```
 }

 if low < j then
  QuickSort(n, low, j);

 if i < high then
  QuickSort(n, i, high);
}

func main() {

 //Create an array of 5 user references
 var users := [5]User;

 //Create the users instances and assign them to elements of the array
 users[0] := User{"Jil", "jil@email.com", 29 };
 users[1] := User{"Zox", "zox@alien.com", 1500 };
 users[2] := User{"John", "j@email.com", 22 };
 users[3] := User{"Prim", "prim@email.com", 90 };
 users[4] := User{"Jim", "jim@email.com", 30 };

 //Print out the list of users before searching
 console.Log("Users before sort: \n\n");
 PrintUsers(users);

 //User the QuickSort defined above to sort the list of users by age
 QuickSort(users, 0, 4);

 console.Log("-----------------------------------------\nUsers after sort: \n\n");

 //Print out the list of users after sorting
 PrintUsers(users);
}
```

2.2 Lexical

The lexical chapter outlines all of the important language features defined in the lexical analyser such as the regular expressions for constant values and the language keywords.

2.2.1 Identifiers

In Scribble the lexical analyser defined an identifier (or ID) as one or more characters, underscores and digits starting with a character that is not a digit. It is defined by the regular expression

```
id [_|a-z|A-Z][a-z|A-Z|0-9|_]*
```

2.2.2 Value Constants

There are several constant values which are identified by the lexical analyser.

Integer

In Scribble a integer constant is a string of digits with no decimal place, prefix or suffix. it is defined in Scribble by the regular expression

```
digit [0-9]
integer {digit}+
```

which will accept strings of the digits 0-9.

For example, 5 would be identified an integer but 5f, 5.0 or i5 would not be.

Floats

A float constant is a string of digits, optionally followed up with a second string of digits with the character 'f' as a suffix to identify it from other types which have decimal places. It is defined by the regular expression

```
real {integer}("."{integer})*
float {real}f
```

For example 5f and 5.43f would be examples of floating point constants however 5.0, 5 or f5 would not be.

Boolean

A boolean is defined in Scribble by the two keywords true or false.

String

A string is a string of characters within two sets of '"'. It is defined by the regular expression

```
string \"[^\"\n]*\"
```

.

2.2.3 Operators List

```
"+" - PLUS;
"-" - MINUS;
"*" - TIMES;
"/" - DIVIDE;
```

```
":=" - ASSIGN;
"=" - EQUALS;

">" - GREATER;
"<" - LESSER;

"." - LINK;
"->" - POINT;

"++" - INCREMENT;
"--" - DECREMENT;
```

2.2.4 Comments

Scribble uses the //Comment Line and /* Block of text */ notation for comments.

This means that when you write // the rest of the current line will be seen as a comment and ignored. This is how single line comments are achieved.

It also means that anything in between /* and */ will be ignored, allowing multi line comments.

2.2.5 Keywords List

```
"and" - AND;
"or" - OR;
"package" - PACKAGE;
"then" - THEN;
"if" - IF;
"else" - ELSE;
"struct" - STRUCT;
"func" - FUNCTION;
"for" - FOR;
"var" - VARIABLE;
"int" - TYPE_INT;
"bool" - TYPE_BOOL;
"float32" - TYPE_FLOAT32;
"nil" - NIL;
"string" - TYPE_STRING;
"void" - TYPE_VOID;
"return" - RETURN;
"while" - WHILE;
"import" - IMPORT;
"true" - TRUE;
"false" - FALSE;
"type" - TYPE;
"do" - DO;
"array" - TYPE_ARRAY;
"len" - LENGTH;
```

2.3 Identifiers

An identifier in Scribble is either the local package name, the name of a function, a type or a variable.

No two identifiers should have the same name (Regardless of what they are identifying or their scope) and this will raise a parsing error.

Identifiers are always local to the package and are not accessible from other packages except for functions and types using an explicit syntax.

2.3.1 Scope

Scribble currently has a simple very scoping system.

Package, function and type identifiers are all global and their names are shared across all functions within a package.

Variable names are always local to the function that they are defined in from that line onwards. Variable redefinition is not allowed and statement blocks (Such as declaring it inside a flow control statement) will not effect this.

For example:

```
func main() {
 var j := 15;

 if true then {
  j := 30;
 }

}
```

and

```
func main() {

 if true then {
  var j := 30;
 }

 j := 10;
}
```

are valid, however

```
func main() {
 var j := 15;

 if true then {
  var j := false;
```

```
 }

}
```

would cause a parsing error.

2.4 Types

Scribble is a strictly typed language. This means that every variable, function and value has a specific type. Unlike other language Scribble will never implicitly cast another object. A programmer will have to use one of the casting functions provided or write their own to cast between types, even the basic primitive types such as integers and floats.

For example

```
var j : int := 5;
var k : float32 := j;
```

would cause a parsing error but

```
var j : int := 5;
var k : float32 := sys.Float32(j);
```

would be correct.

Scribbles type system is right handed. This means that you declare the type of a variable, argument or function on the right hand side of its deceleration instead of the left.

If a function is not given a type then it is assumed to be of type void. You cannot explicitly create a function or variable of void type.

Examples:

```
//Void function which takes an argument of type int
func vFunction(j : int) {
 //Do something with j
}

//Function that returns an int
func iFunction() : int {
 var j : int := 15;
 return j;
}

//Main function which creates some variables
func main() {
 var j := iFunction();
 vFunction(j);
}
```

2.4.1 Primitive Types

There are several primitive types in Scribble. Primitive types are the building blocks of a program, representing fixed size pieces of data within the language giving them a fixed range.

Primitive types are passed around as values and not references, this means that if you create a primitive variable like var j : int and pass it to a function which modifies the copy of j in the calling function will not be affected.

For example

```
func Hello() {
 var j := 10;
 World(j);
 //Here j = 10 would still be true.
}

func World(x : int) {
 x := 12;
}
```

This difference is the primary difference between referenced values, where a change will affect the value anywhere where the reference exists.

bool

A boolean value is capable of representing one of two values, true or false. They are identified with the bool keyword.

Examples:

```
var j : bool := false;
var r : bool := true;
```

int

The int type represents a 32 bit signed integer value which is capable of representing integers between -2147483648 and 2147483647.

They are identified by the keyword int.

float32

The float32 type represents 32 bit floating point numbers allowing approximations of numbers with decimal points to be used in a program. They are identified by the float32 keyword.

2.4.2 Reference Types

A reference type are types of a variable size which are stored on the heap, an area in memory which allows for dynamic memory allocation.

Reference types are garbage collected, this means that like Java the data is automatically destructed when a reference to it no longer exists.

The values (arrays, structures or strings) which are referenced only exist once in memory and any change to these values will be reflected across any functions which have a reference to it. The references themselves are like C pointers and are local to the function.

For example

```
func ModifyArray(arr : array(int)) {

 if len(arr) > 0 then {
  arr[0] := 5;
 }

}

func DoStuff() {
 var j := [10]int;
 ModifyArray(j);
}
```

In this case the value of j[0] after ModifyArray has been called would be 5 and not the initial value 0.

However in the case

```
func ModifyArray(arr : array(int)) {

 //Create a new array and assign it to arr
 arr := [10]int;

 if len(arr) > 0 then {
  arr[0] := 5;
 }

}

func DoStuff() {
 var j := [10]int;
 ModifyArray(j);
}
```

the value of j[0] in DoStuff would remain unchanged, as the reference arr in the ModifyArray function was modified to point to a new array rather than the one referenced by j in DoStuff.

string

The string type represents a string of characters. It is really syntax sugar for an array of characters. When a string constant is written in Scribble the expression generated will be to create a new item on he heap and then load that data into it, so "Hello World" would make a new array of 12 characters (Including the null terminator) and then load the string into it as initial values.

Strings are immutable. Any operation on them using the functions provided will not change the memory of the string and will instead create a new string with which to place the result of the operations, this can have some performance implications as repeated string manipulations cause the heap and garbage collector to perform a lot of expensive operations.

The string keyword identifies this type.

Arrays

Arrays are capable of contain a set number of values of the same type, including reference types. Two arrays which represent different types are not equivalent (So you couldn't assign an array of integers to a variable which is mean to store an array of booleans).

Arrays are written in Scribble as array(subType), this differs from the subtype[] syntax often used in other languages. The changes where made in order to make type information clearer especially when dealing with a right handed type system.

The length of an array can also be obtained after it is declared by writing len(arrayValue). This will return the number of elements in the array, not the size in bytes.

Structures

Structures are a predefined collection of fields with specific types. They are used to build complex types from collections of primitive data.

A structure is defined by writing

```
type Name := struct {
 FieldName : Type,
 FieldName : Type
}
```

2.4.3 Type Inference

Scribble allows for a variables type to be inferred when it is declared instead of explicitly typed. This reduces some of the typed overhead caused by strict languages without allowing the issues which lazy typing systems can cause (Such as being unable to easily identify the type of a value as well as errors caused by casting).

The type is inferred from the type of the expression that it is being assigned to. The type of the expression is identified by the recursive inspection of the generated statement tree. Each element on the trees type is inferred from the type of its children down to the basic constant values.

For example the expression 5 + 5 will be inferred to be of type integer as the two constant values are of type int and the addition expression will produce a value of the same type as the values it is adding, additionally 5f + 5f would be inferred to be float32.

If the type of an expression is inferred to be 'Void' (Scribbles keyword for expressions or functions which have no type) then it cannot be used in variable type inference and a parser error will be raised.

2.4.4 Types in the grammar

The types defined above are expressed in the grammar as

```
Type: TYPE_INT {
} | TYPE_STRING {
} | TYPE_FLOAT32 {
} | TYPE_BOOL {
} | TYPE_ARRAY LPAREN Type RPAREN {
/* Structure types name from this and other packages */
} | WORD {
} | WORD LINK WORD {
}
;

/**
 * BaseStructureInfo is the definition of each field within a structure in the form Name
 * Accepts 1 or more definition.
 */

BaseStructureInfo: WORD COLON Type {
} | BaseStructureInfo COMMA WORD COLON Type {
}
;

/**
 * Defines a structure within a package
 */

Program TYPE WORD ASSIGN STRUCT LBRACKET BaseStructureInfo RBRACKET {
}
```

2.5 Operators & Expressions

The next section covers the operators and expressions that drive Scribble and enable information to be processed.

2.5.1 Constant Expressions

Constant expressions in Scribble are the basic building blocks of the language. Allowing you to enter constant values into the language so that they can be used in calculations or assigned to variables.

Booleans

A boolean constant is expressed using the keywords 'true' or 'false'.

It is defined in the grammar as

```
TRUE {
} | FALSE {
}
```

Integers

A integer constant is expressed whenever the lexical analyser recognizes an integer using the regular expression defined in the lexical analysis chapter.

It is defined in the grammar as

```
INT {
}
```

Floats

A floating point constant is expressed whenever the lexical analyser recognizes an float using the regular expression defined in the lexical analysis chapter.

It is defined in the grammar as

```
FLOAT32 {
}
```

Strings

A string constant is expressed whenever the lexical analyser recognizes a string using the regular expression defined in the lexical analysis chapter.

As string a not a primitive type when a string constant expression is executed it will create a new string on the heap and return a reference to it, this differs from all of the other constant types which are all primitives and have no effect on the heap.

```
STRING {
}
```

2.5.2 Variable Definition

A variable can be defined in Scribble either with an explicit type or with its type inferred from its initial assignment. This will create a new identifier by which that variable is referenced for any line after the definition in that function.

To explicitly specify the type of a variable you write var Name : Type;

Using type inference allows you to automatically set the type of a created variable based on the expression that it being signed to initially. You do this by writing var Name := Expression; If no expression is supplied or the expression is Void then a parser error will be thrown otherwise the type of the new variable will be set to the type of the expression.

examples:

```
var j := 5; //Create a new variable of type int
var j : int := 5; //Explicitly create a new variable of type int.
var j := (5 + 5) * 2; //Create a variable of type int from a more complex expression
```

variable definition is defined in the grammar as

```
Variable {
} | Variable ASSIGN Expression {
} | VARIABLE WORD ASSIGN Expression {
}
```

2.5.3 Variable Expressions

A variable expression will evaluate to the value of the specified variable.

This is defined in the grammar as

```
ID {
}
```

2.5.4 Array Construction

Arrays in Scribble are constructed using the [ArraySize]type notation. The initial value will be set to zero for numeric types, false for booleans and null for reference types.

Examples: [100]int will construct an array of 100 integers. [100]array(int) will construct an array of 100 integer arrays.

It is defined in the grammar as

```
LSQBRACKET Expression RSQBRACKET Type {
}
```

2.5.5 Structure Construction

Structures are constructed using the StructureTypeName Initial Values seperated by a comma notation. Currently every field in a structure has to be given an initial value and they are assigned in the order that they are defined in the structure definition.

Examples:

type Hello name : string, age : int ;

var J := Hello "Bobby", 18 ; Will create a variable J which is an instance of the structure Hello with the name "Bobby" and the age 18.

It is defined in the grammar as

```
Arguments: {
```

```
} | Arguments_2 {
};

Arguments_2: Expression {
} | Arguments COMMA Expression {
};

Type LBRACKET Arguments RBRACKET {
}
```

2.5.6 Array Index Expression

An array index expression evaluates to the value of an array at a given index written in the notation ArrayExpression[IndexExpression].

This is defined in the grammar as

```
Expression LSQBRACKET Expression RSQBRACKET {
}
```

2.5.7 Structure Field Expression

A structure field expression evaluates to a specified field within a given Structure expression (An expression which evaluates to the type struct(Something))

It is defined in the grammar as

```
Expression '->' ID {
}
```

2.5.8 Operators

The following sections define the core operators in Scribble (+, -, =, /, *, etc..)

Arithmetic

There are four basic arithmetic operators in Scribble. These are addition '+', subtraction '-', multiplication '*' and division '/'. These operators take a left and right numeric expression of the same type and perform the appropriate operation on it.

Examples: 5 + 7 * 2 4 / 2 * 8

If the expressions are not numeric or not of the same type then a parsing exception will occur. For example 5 * 5 would be valid but "Hello" + "World" or 5f + 5 would not be.

It is defined in the syntax as

```
Expression PLUS Expression {
} | Expression MINUS Expression {
```

```
} | Expression TIMES Expression {
} | Expression DIVIDE Expression {
}
```

Minus Expression

In addition to these arithmetic operators -Expression is also allowed to allow negative expressions like -5 or -x to be constructed. It is the same as writing 0 - 5 or 0 - x and is valid for any numeric type. It is defined in the grammar as

```
MINUS Expression {
}
```

Increment & Decrement

In addition to the basic arithmetic operators you can also use the '++' and '–' operators to increment or decrement a integer variable by 1. An increment or decrement expression also returns the value of the variable. If the operator is placed before the variable name then it will return the value of the variable after it is modified and if it is placed after the id of the variable then it will return the value of the variable before it is modified.

Examples: var i := 0; i++ = 0 is true

var j := 0; ++i = 1 is true

These operators are defined in the grammar as

```
ID INCREMENT {
} | INCREMENT ID {
} | ID DECREMENT {
} | DECREMENT ID {
}
```

Comparison

There are six basic comparison operators in Scribble. Equal '=', Not equal '!=', greater than '¿', less than '¡', less than or equal to '¡=' and greater than or equal to '¿='. These operators allow a comparison between two expressions. All primitive types can be compared using the = and != operators however only numerical types can be compared using the ¡, ¿, ¡=, ¿= operators.

One difference from C and many of the languages based of it is that assignment is done using the ':=' operator and a single '=' is now used for comparison instead of '=='.

These operators are only capable of comparing expressions of the same type. Mixing two different types, even numerical types, will cause a parsing error.

They are defined in the grammar as

```
Expression EQUALS Expression {
} | Expression NOT EQUALS Expression {
} | Expression GREATER Expression {
```

```
} | Expression LESSER Expression {
} | Expression LESSER EQUALS Expression {
} | Expression GREATER EQUALS Expression {
}
```

Logical Operators

And and Or are the two currently supported logical operators in Scribble. These operators take two boolean expressions and return a boolean value of the logical and or or. They are written using the 'and' and 'or' keywords instead of the traditional '&' and '—'. This was done in an effort to make blocks of code more readable, especially since Scribble does not require parenthesis around flow control statements.

The and keyword will only execute the expression on its right hand side if the value on the left hand side is true.

Examples: true or false, true or true, false or true will all return true. false or false will return false. true and true will return true. true and false, false and false, false and true will all return false. 5 = 1 and 6 = 2 - The right hand side expression would never be checked because 5 does not equal 1.

It is defined in the grammar as

```
Expression AND Expression {
} | Expression OR Expression {
}
```

2.5.9 String Concatenation

In Scribble the '$' operator is used to concatenate two expressions of the type string.

Writing "Hello" $ " World" is equivalent to writing sys.Concat("Hello", " World"), this is the only syntax feature in Scribble that currently depends on the built in 'sys' package. If sys is removed then the $ operator will not work and will have to be switched off or reimplemented.

Note: While this feature relies on the sys library it does not rely on the user importing explicitly into a package and will be automatically imported by the compiler if required.

```
Expression '$' Expression {
}
```

Assignment

Assignment in Scribble is done using the ':=' operator. It takes a variable ID and an expression and sets the value of that variable to the value given when the expression is evaluated.

Assignment will only take place of the expression to be assigned is exactly the same type as the variable being assigned to. Any other values will have to be cast using a custom or built in function.

Assignments can be done on the index's of an array by using the Expression[IndexExpression] := Expression notation.

Assignment can also be done on the fields of a structure using the Structure Expression -¿ FieldId := Expression notation.

Examples:

```
var j := 65;
j := 192;
```

Would create a variable j, set j's value to 65 and then set j's value to 192.

```
var arr := [10]int;
arr[0] := 5;
```

Would create the array arr with 10 integers and then assign arr's 0th index to the value 5.

```
type J := struct {
name : string
}

var j := J{"Hello"};
j->name := "Hello World";
```

Will result in a structure of type J being created with the name field set to "Hello" and then the name field of j being set to "Hello World"

If the variable and expression types differ then a parsing error will occur.

Assignment is defined in the grammar as

```
LPAREN Expression RPAREN {
} | ID  Expression {
} | Expression '->' ID ':=' Expression {
} | Expression LSQBRACKET Expression RSQBRACKET ASSIGN Expression {
}
```

2.5.10 Array Length Expression

The length of an array can be obtained by using the len(ArrayExpression) notation.

examples: len([100]int) = 100 and len([50]int) = 50

```
LENGTH LPAREN Expression RPAREN {
}
```

2.5.11 Parenthesis Expressions

Parenthesis to control the order in which expressions are evaluated. This is helpful in large or complex expressions.

Examples: (5 + 5) * (5 + 5) would evaluate to 10 * 10 which would be 100. 5 + 5 * 5 + 5 would evaluate to 10 * 5 + 5 which would be 55.

defined in the grammar as

```
'(' Expression ')' {
}
```

2.5.12 Function Call Expressions

A function call is written as FunctionName(Arguments) or PackageName.FunctionName(Arguments). The type of a function call is the type of the function that is being called, for example sys.String(15) is of type string and sys.Log("Hello World"); is of type void.

Examples:

sys.Log(sys.String(15)); will print the number 15 to the screen.

sys.Pow(2, 4); will return 16.

defined in the grammar as

```
Arguments: {
} | Arguments_2 {
};

Arguments_2: Expression {
} | Arguments COMMA Expression {
};

FunctionCall: WORD LPAREN Arguments RPAREN {
} | WORD LINK WORD LPAREN Arguments RPAREN {
};
```

2.6 Statements

The following section describes the statements which include expression statements, flow control and return statements that allow a program to be effectively structured.

2.6.1 Flow Control

There are three flow control structures in Scribble.

If

The If statement will execute the code in its body if the given boolean expression evaluates to true. You can also supply an optional set of code to be execute if the statement evaluates to false.

Syntax:

```
if Expression then {
Code
}

or

if Expression then {
Code
} else {
Code
}
```

Is is defined in the grammar by

```
IfStatements:
Statement {
} | LBRACKET Statements RBRACKET {
}

IF Expression THEN IfStatements  {
} | IF Expression THEN IfStatements ELSE IfStatements {
}
```

For

A for loop is constructed with an initializer, a condition and a step expression. It allows for controlled loops.

The syntax:

```
for Initializer ; Condition ; Step do {
Code
}
```

It is defined in the grammar by

```
FOR Expression ';' Expression ';' Expression DO IfStatements {
}
```

While

A while loop is given a boolean expression and will continue to execute a given piece of code until that expression evaluates to false.

The Syntax:

```
while Expression do {
}
```

It is defined in the grammar by

```
WHILE Expression DO LBRACKET Statements RBRACKET {
}
```

2.6.2 Expression Statements

An expression statement is a statement such as console.Log("Hello World"); or j := 5 * 4;

The syntax:

```
Expression;
```

It is defined in the grammar as

```
Expression ';' {}
```

2.6.3 Return Statements

The return statement exits the current function and returns a given expression value.

It is defined in the grammar as

```
'return' ';' | 'return' Expression ';'
```

2.7 Functions

A function is declared using the syntax

```
function Name( One or more arguments ) : Type {
 Statements
}
```

If type is not set then the function is assumed to be of a Void type. A function of Void type does not return any values.

Arguments are written in the form Name : Type, Name : Type, Name : Type so to declare a function which takes two integers as arguments you would write

```
func TestArgs(A : int, B : int) {
}
```

The order in which a function is defined does not matter. Unlike languages such as C you are allowed to call a function before it is defined. This allows mutual recursion as shown in the example

```
func even(x : int) : bool {
 if x = 0 then
   return true;
 return odd(x-1);
}

func odd(x : int) : bool {
 if x = 0 then
  return false;
 return even(x-1);
}
```

Multiple functions can share the same name as long as they take different arguments and have the same return type. An example of this is the system String function which has two version one which takes a boolean and one which takes a number. A practical example of this would be

```
func PrintTrue(i : bool) {

 if i then
  sys.Write("True");
 else
  sys.Write("False");

}

func PrintTrue(i : int) {

 if i = 1 then
  sys.Write("True");
 else
  sys.Write("False");

}
```

this would compile successfully and upon any call to PrintTrue the compiler would resolve the correct function to be executed.

Functions are defined in the grammar as

```
/**
 * Accept zero or more argument definitions
 */

OptionalArgumentDefinitions: {
} | ArgumentDefinitions {
}
;
```

```
/**
 * Accept one or more argument definitions in the form Name : Type, Name : Type..
 */

ArgumentDefinitions: ArgumentDefinition {
} | ArgumentDefinitions COMMA ArgumentDefinition {
}
;

/**
 * The definition of a function. func Name ( Arguments ) { Code }
 * defines a function of void type and func Name ( Arguments ) : Type { Code }
 * defines a function of a specific type.
 */

Function: FUNCTION WORD LPAREN OptionalArgumentDefinitions
RPAREN COLON Type LBRACKET Statements RBRACKET {  }
| FUNCTION WORD LPAREN OptionalArgumentDefinitions
RPAREN LBRACKET Statements RBRACKET {
}
;
```

2.8 Packages & Importing

Every file in Scribble is seen as a separate package. These packages contain sets of functions and structures which can be used to perform some function, for example one package may contain a linked list structure and a set of functions for inserting and removing elements from said list. Scribble also defines syntax so that packages can import and use structures and functions from other packages.

Packages in the grammar are defined as

```
Package: {
} | Package PACKAGE WORD ASSIGN IMPORT LPAREN STRING RPAREN END {
} | Package Function {
} | Package TYPE WORD ASSIGN STRUCT LBRACKET BaseStructureInfo RBRACKET {
}
;
```

2.8.1 Importing Packages

One package can be used from another by using the

```
package LocalName := import("Package/Path");
```

syntax.

This syntax allows you to import a package from a given path and give it a local identifier which can be used to reference it. The local identifier allows the avoidance of package

naming conflicts, as a importing package can give two packages of the same name (Such as /tests/math.sc and /math/math.sc) different local names to be identified by.

To use a function or structure from a different package you first write the packages local name then a '.' and then the name of the function or structure you wish to use. For example to print some output to the screen you write.

```
console.Write("Hello World");
```

Chapter 3: **Instruction Encoding**

The instruction encoding is one of the most vital parts of a virtual machine.

Every operation that the virtual machine is able to perform from addition to memory allocation is able to be represented as an instruction or a set of instructions within the virtual machine. These instructions are executed one after another by the processor each one performing an action which will change the state of the virtual machine, for example one instruction may jump to an earlier set of instruction while another may load an integer constant like the number 5. This means that the instruction set and instruction encoding for a virtual machine has a very significant impact on the speed and compiled code size of your programs.

The most common approach to instruction encoding is to represent every instruction and the arguments required for the instruction to operate within a fixed size region of memory. Having fixed size instructions can lead to wasted space however it makes instruction stepping and jumping significantly simpler because the virtual machine can calculate the location of the next instruction without needing to inspect the instructions which are being skipped.

One thing that can have a dramatic effect on the instruction encoding is whether the virtual machine is stack or register based. A stack based virtual machine will store all of the data used in operations on a stack for example to perform the expression 5 + 4 you would load 5 to the stack, load 4 to the stack and the perform the add operation which would pop the top two values from the stack and add them together then place the result of the addition back into the stack. This approach leads to extremely small instruction sizes, usually less than 32 bits, as instructions do not require much additional information to execute, this can however lead to a larger number of instructions being used because of the stack operations required to make the program run. A register based virtual machine instead chooses to emulate a real hardware CPU more closely. In this model the virtual machine has a fixed number of registers and instructions modify these instead of the stack for most operations. The example given about the stack would instead be load 5 into register 0, load 4 into register one, add registers 5 and 4 and place the result into register 2. This approach requires larger instructions because they have to carry data about the registers they are operating on however it potentially decreases the number of instructions that have to be executed to perform the same operation. It has also been argued that register based instruction sets translate better to machine code when used with a ByteCode to native code compiler used in technologies such as JIT which is common within modern virtual machines. Which model is better is a wide area of debate and there is no clear answer as each has clear advantages and disadvantages and strong support for both sides from developers with languages like Java using stack based virtual machines and technologies like Microsofts .NET and Googles Dalvik using register based virtual machines.

Scribble uses a register based virtual machine, I decided to use a register based virtual machine over a stack based virtual machine as I was more familiar with register based environments.

The instructions in Scribble are 64 bits wide, the first 8 bits always represent the operation to be perform (For example OpAdd, OpNewArray, etc) and the purpose rest of the data will vary depending on the instruction. This much space per instruction is probably not necessary and the wasted space will result in an increased size of compiled programs over equivalent programs in other languages such as LUA however when I was developing the instruction set I did not know what constraints on instructions I would find acceptable and decided to give myself as much overhead as seemed reasonable.

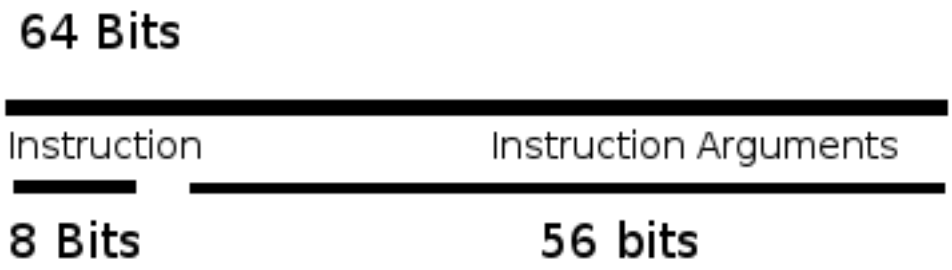

Fig 1. Scribble Instruction

In order to make the virtual machine useful I had to ensure that there where enough instructions available to achieve anything that was valid within the higher level language. This involved looking at each of the language features and working out a way of expressing them as a set of instructions, for example the for loop is achieved by using assign, compare and jump operators in order to iterate until a condition is met. The number and complexity of the instructions that a virtual machine can perform can have a significant impact on its performance. If the instructions are to specialized then a larger number of possible instructions will be needed which could potentially slow down the virtual machine as it has to spend more time with each instruction or increase the size required to store the part of the instruction that tells the virtual machine what to do. Alternatively having less complex instructions will increase the number of instructions that need to be executed to perform each task which and could potentially lead to the virtual machine not being able to complete a task or support a feature later on.

3.1 Instructions List

The code below lists all of the possible instructions that are compatible with ScribbleVM. These OpCodes form the first 8 bits of every instruction and are used to tell the VM how to handle the instruction.

```
enum OpCodes {

OpLoadConstant = 0,
OpMove,
OpJump,

/**
 * Arithmetic operations
 */

OpAdd,
OpSub,
OpMul,
OpDiv,
OpInc,
OpDec,

/**
 * Float operations
```

```
 */

OpAddFloat32,
OpSubFloat32,
OpMulFloat32,
OpDivFloat32,
OpCmpFloat32,

/**
 *  Tests
 */

OpEqual,
OpNotEqual,
OpEqualZero,
OpLessThan,
OpLessThanOrEqual,
OpGreaterThan,
OpGreaterThanOrEqual,

/**
 * Array operators
 */

OpNewArray,
OpArraySet,
OpArrayGet,
OpArrayLength,

/**
 * Structure operators
 */

OpNewStruct,
OpStructSetField,
OpStructGetField,

/**
 * Function & Stack operators
 */

OpPushRegisters,
OpPopNil,
OpPopRegisters,
OpCallFn,
OpReturn
};
```

Chapter 4: **Type Inference**

In a strictly typed language such as Scribble every expression, argument, variable, function or piece of data has a defined type.

The advantage of a strictly typed language is that is allows the language to avoid common errors which casting cause and also reduces the ambiguity over the type of a function or expression as every cast has to be explicit and everything has a defined type. For example when calling the function var val = doSomething(); in Javascript it is unclear what type of data val will contain after the function call. Within a strictly typed language this ambiguity is removed as the type of val will have to be declared before it is used as does the type which doSomething() would return.

The downside of a strictly typed syntax is that it can lead to additional overhead for the programmer. Whilst it is reasonable to force that a function or structure field be given an explicit type because it makes programs much clearer, often when implementing a function the programmer does not want to have to explicitly specify the type of every variable that they create. In an attempt to reduce the time a programmer has to spend writing unnecessary definitions the type of a variable in Scribble can be inferred from the expression that it is being assigned to by omitting the type declaration. Therefore what was

```
var j : int := 15;
```

can become

```
var j := 15;
```

This language feature is made much simpler type information is declared to the right of a deceleration in Scribble as it allows the use of the var Name := Expression; syntax which would not be possible if the definition was Type name = Expression. This issue is visible in the C++11 standard in which the confusing auto keyword was introduced to allow basic type inference.

The following is an example of a piece of code written in C and an equivalent piece of code in Scribble.

```
struct ALongName {
int i;
};

int main() {

 ALongName* names = new ALongName[100];

 for (int i = 0; i < 100; i++) {
  names[i] = ALongName();
  names[i].i = 5;
 }

}
```

And in Scribble

```
type ALongName := struct {
 i : int
}

func main() {
 var names := [100]ALongName;
 for var i := 0; i < len(names); i++ do {
  names[i] = ALongName{5};
 }
}
```

The example above shows that the variable type inference makes the definition of the variable much simpler while it is still clear that 'names' is an array of structures and 'i' is an integer.

One potential negative to this approach is that it will be difficult to immediately work out the type of a variable when the expression it is assigned to involves a function call which they do not know the type of. This issue is likely to be mitigated by good naming practices when naming functions, platform documentation and IDEs however it could cause issues in large poorly documented programs.

4.1 Expression type inference in function matching

Scribble ability to infer the type of expressions is also extremely useful when used with matching a function call with the appropriate function. In Scribble a function name may be used to call several functions which accept different parameters removing the need for confusing naming conventions to differentiate between types such as functions like sqrt and sqrtf in the C standard library. One example of this is the sys.String function, for which there is a definition which takes each of the primitive types in Scribble and returns it as a string.

```
String ( int ) : string
String ( float32 ) : string
String ( bool ) : string
```

Each definition of a function is an independent function which can be identified by using its signature, a combination of its name, the number and type of parameters and the type of the value that it returns.

This feature requires the compiler to be able to be able to identify the correct function signature from a function call. In Scribble this can be achieved because the type of each of the arguments can be inferred by inspecting the tree and a function with the correct signature will be selected.

4.2 Expression type inference in error checking

Scribble enforces a strictly typed syntax in an attempt to avoid many of the common issues that arise and can go undetected when a language implicitly converts values between two types.

One very simple example of these issues would be the issues that arise when performing arithmetic on values of different types in C.

```
#include <stdio.h>

int main(int argc, char** argv) {
 int i = 5.0 / 2.0;
 printf("%i", i);
}
```

In this example the program would print the value 2 to the screen. Though issues arising from this example seem unlikely (And a good compiler is also likely to warn a developer about it) when there are complex expressions of several different types which involve large amounts of arithmetic this can lead to extremely cryptic issues which are incredibly hard to debug.

In Scribble operations which compare or modify values can only be performed on expressions of the same type, this makes issues with type conversion less likely as no cast is implicit.

This adds complexity to the compiler as it has to compute and compare the type of the expressions involved in many operations, additionally as some type or function references are resolved after the file is parsed. In Scribble this problem is solved by combining the same expression type inference functionality used to infer the type of variables and combining it with a recursive descent algorithm which verifies there are no issues with each node in the programs statement tree.

```
function CheckTree(Tree* root) {

 For each child i of root do {

   if CheckTree(i) throws an exception then
   exit and return to the try catch handler
   around the initial call to CheckTree

 }

 if CheckForIssues() finds an error then throw an exception
   detailing the error and where it occurs
   in the program.

}
```

Above is an example of the CheckTree function in pseudo code.

By using the function detailed above each node in the statement tree can be checked. The CheckForIssues function will be overloaded for each type of node on the tree, for instance

a node of type AddStatement's CheckForIssues function would use type inference to ensure that it was adding two expressions of the same type, and also to check that the type of the expressions being added together was valid (So nobody tried to add together two references or booleans) whereas the CheckForIssues function in an array assignment node would check that the type of the expression to be assigned to was the same as the type of data stored in the array, whilst also checking that the index expression was an integer.

Chapter 5: **Package Handling**

There are two common approaches to handling multiple source code files within a program. Languages such as C tend to include anything loaded from an external file into the existing namespace, for example if file A includes file B which defines the function hello_world then the function hello_world would be usable from A. This approach often leads to naming conflicts, especially when middleware or libraries are being used as several developers will often choose the same name for different functions or classes. C++ attempts to avoid this problem with the concept of namespaces, however they add additional complexity to the code and can lead to names becoming confusing within complex systems.

For example in C the code printer.c

```
void print(int printerID, char const* filePath) {
 //Print the contents of a given file to a printer
}
```

and main.c

```
#include <printer.h>

void print(char const* line) {
 //Print something to the screen
}
```

both have used the function print in different ways which would cause a conflict and confuse any developers.

The other common approach taken is the idea of having software packages that are accessed by adding package identifier to any calls outside of the current package. One example of this is a Java class, in Java programs are expressed as objects of a certain class. These classes can only interact with each other by explicitly naming the class that they want to use. For example in the following code one class uses a method from another.

```
public class A {
    public static void someFunction() {}
};


public class B {

    public static void doComplexThing() {
        //Do Stuff
        A.someFunction();
    }

};
```

This approach is vastly more user friendly and much safer than the approach taken in C however naming issues still lead to issues (Though they are more easily resolved by using the

full name of a class) and common applications of this approach have tended to be in object oriented languages however Scribble is a procedural language and doesn't have classes.

The approach that Scribble takes is similar to the way classes are separated in Java however it has a couple of differences in order for it to make sense for a procedural language and to making naming conflicts easier to resolve. Every file in Scribble is seen as a separate package and these packages contain sets of functions and structures which make up the functionality of that part of the program. For one package to use another package it has to "import" it which gives that package an identifier with which its functions can be accessed and telling the compiler that that package must be loaded for the importing package to execute properly.

In the first implementation of this the identifier of the package would be assumed from that packages file name, for example import("examples/helloworld"); would import the package examples/helloworld.sc and give it the identifier helloworld, however I found that this approach quickly lead to issues when I had packages of the same name such as importing a set of unit tests for the math library being placed in /test/math and a package providing common mathematical functions being placed in /math/math from the same package.

To avoid these issues I introduced local package identifiers to Scribble. Local package identifiers allow a developer to choose the local identifier for a package when importing that package. For example the issue listed in the last paragraph would now be solved by writing

```
package MathTests := import("tests/math");
package math := import("math/math");
```

This method of package handling allows me the to almost completely avoid naming issues and have a useful way for developers to structure their programs as packages can be used by several different parts of a program to avoid having to rewrite code.

Local package names are implemented by maintaining a mapping of local package names to global package identifiers for each package. Whenever a reference to a function or structure in another package occurs the resolve function first replaces the local package name with that packages global identifier by looking for the mapping and then resolves then continues to resolve the reference the way it would any other.

If no mapping between a local package identifier and a real package exists then an error will be detected during the parsing step where references are resolved, however line and symbol information on where the error occurs is not available at this point. Instead the reference is marked so that the statement tree can identify that there has been an error with resolution because there is no mapping and an error will be raised when the checkTree function is called. By doing this detailed information can be given on where the error occurs and also caused the issue.

Chapter 6: **Garbage Collection**

Scribble automatically frees up any dynamically allocated data which is no longer referenced by using a technique called garbage collection. Garbage collection is useful as it removes the burden of having to manually free memory from the programmer and it also reduces the risk of memory leaks.

For example the piece of C++

```
int main(int argc, char** argv) {

int** arr = new int[100];

for (int i = 0; i < 100; i++) {
arr[i] = new int[100];
}

//Do stuff with the 2D array

for (int i = 0; i < 100; i++) {
delete[] arr[i];
}

delete[] arr;
}
```

has several lines of overhead as the programmer has to free each piece of data that has been manually allocated, however in Scribble an equivalent program would be

```
func main() {

var arr := [100]int;

for var i := 0; i < 100; i++ do {
arr[i] := [100]int;
}

//Use 2D array

arr := nil;
}
```

As you can see from the examples garbage collection is significantly more convenient for the programmer, reducing the amount of the time that they have to spend thinking about managing the system rather than solving the problem that they are trying to solve.

The main issue with common garbage collection techniques is the impact that they have on the performance of a program. The complexity that these techniques add to a program have a significant impact on its performance, leading to developers on platforms such as Android to develop codes of practice that avoided allocating memory as much as possible in order to try and stop the garbage collector executing. The impact on performance is caused by the difficulty of finding every reference to memory on the heap during the execution as it requires

the current execution to be halted until every value on the register and the stack can be inspected. The performance of a virtual machines garbage collector is extremely important as having a slow garbage collector or one that executes to often can force developers to try and work around it which could have more of a negative impact than the convenience gained from automatic memory management.

Scribbles approach to garbage collection frees memory by marking every piece of memory that is referenced by the stack or the in a register. The garbage collector will then look at every marked item to see if that references any allocated memory and it will mark that, this is repeated until all marked items have been checked. At this point it is safe to delete any allocated memory which has not been marked by the garbage collector as the program no longer has any references to it. This is a naive approach to garbage collection which has to check every single reference in the application every time it executes, however it is also fairly accurate ensuring that there are no memory items flagged up for deletion which are still in use while making sure that any unreferenced memory is freed.

One way that Scribbles garbage collection could be improved is by separating allocated memory up based on how long they have existed. Empirical analysis of programs shows that most of the memory allocations within a program are only used for a short period of time. If a garbage collector where to separate its memory into two or more 'generations' then a program could garbage collect the younger objects more often with less of an impact on performance whilst only checking the older objects occasionally. This is the approach taken by the programming language Java which executed 'minor' garbage collection often whilst only executing 'major' garbage collection which looks at older objects rarely. This is described in the 'JVM Garbage Collection' article listed in the bibliography.

There is one other common technique when trying to automatically free memory that is no longer used. This technique is called reference counting, in a reference counted virtual machine every piece of allocated memory has a counter which records the number of current references to that memory exist. This approach is much simpler than the approach that Scribble uses and results in memory being immediately freed when it is no longer used however it has the downside of every operation on dynamic data taking an increased amount of time.

Chapter 7: **Native Interface**

The native interface is the way that a scripting language such as Scribble, LUA or V8 interact with the program which is executing them. The native interface is an extremely important part of the programming language as the functionality that is provides dictates what a user can do with the language. The native interface should expose any parts of the programming language that the developer could need, for example if the native interface did not allow a developer to get elements from an array then the usefulness of arrays within the language would be severely diminished. The interface should also aim to be as abstract as possible and avoid as much coupling to the underlying implementation. There are two reasons for this, a developer does not want to have to understand how ScribbleVM actually executes the program in order to extract useful information from it and you want to be able to maintain a backwards compatible native interface indefinitely, if a developer has to rewrite their native code for every version of Scribble then they will not use the language.

This interface is provided in the form of a C++ application programming interface (API). The Scribble API specifies how to add native code functions to Scribble, call Scribble functions and also provides a class which allows you to more easily work with function arguments and return values. In addition to this the API also provides a set of helper functions to make some of the more complex issues like allocating memory on the heap or checking for compilation errors simpler.

One downside to writing the API in C++ is that many languages do not have easy ways to wrap around C++ code which would reduce the number of languages which Scribble would be compatible with. One way in which this could be improved would be by implementing a version of the API in standard C as most languages provide a way of using or writing a wrapper which can call code which has been written in C.

7.1 Structure

The Scribble native interface exposes information about types and functions whilst providing and also provides a wrapper to compile and execute Scribble programs without having to manually invoke each step of compilation and VM initialisation.

The first iterations of my native interface where far to complex and required a detailed knowledge of how the virtual machine allocated memory. There was also no wrapper around the compilation and execution of programs and all arguments where returned as register values, not automatically type cast to the correct type. This system was incredibly difficult to use so I designed a set of abstract classes which I could overload to provide an intuitive and simple interface with the language.

The first of these classes is the Scribble class located in the folder /src/Scribble/ is the first part of the wrapper that a developer will use when adding Scribble to their application. It is the class that compiles and prepares a package for execution. The Scribble class also makes it easy to execute specific functions within a package by using the execute function you can execute a function with a specific number of arguments without having to look into the generated namespace for its declaration. The Scribble class will also detect whether any errors occur during compilation or whether the function that is being executed exists and raise an exception if there are any issues.

The APIValue class represents a value within Scribble. The class has functions to expose the

type and data of the value that it represents. This class allows every value which is valid in Scribble to be accessed easily within the native code of the application.

The alternative to this approach would be to use the type information that exists about each value to automatically cast it to its native system type before passing it to the execute function.

The downsides to using a class to represent values are that it can lead to more complex code, for example the Scribble console.Log function in native code is

```
virtual API::APIValue execute(API::APIValue* values, VM::VirtualMachine* virt) {
 printf("%s", (char*) values[0].getReferencePointer());
 return API::APIValue();
}
```

whereas if I used the type information to cast to the native system type it would look like

```
virtual void execute(char* argument, VM::VirtualMachine* virt) {
printf("%s", argument);
}
```

The advantage however is that the API value class can contain more than just the value. In the case of an array or structure the APIValue provides helper functions to easily set or get the value of an index or a field. The APIValue can also store information about the array length or provide a convenient way to retrieve it.

Finally there is the Function class, this class provides a way for a developer to register a Function so that it is executable from Scribble. This class is capable of returning the type, number of arguments and the type of arguments a function takes and an execute in which the function is actually implemented.

The Scribble sys and console built in libraries are both implemented by overloading the Function class.

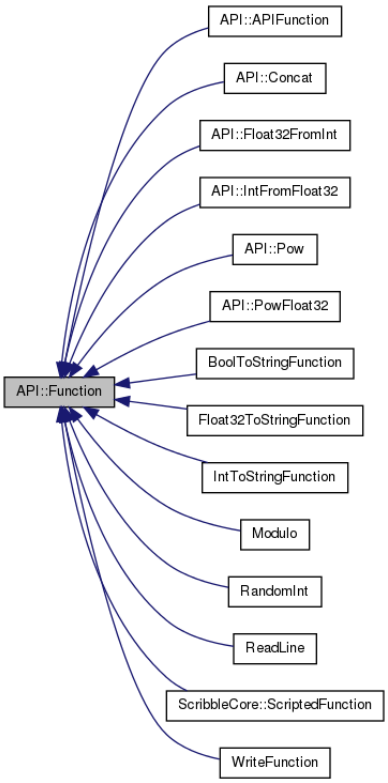

Fig 1. The Class hierarchy of from API::Function showing all of the built in native functions as well as the ScriptedFunction. Generated using graphviz

Functions which are written in Scribble are also instances of the Function parent class (Created from instances of ScribbleCore::ScriptedFunction) and can be executed in the same way that a native function could. This provides a standard interface with which to access any function that is visible to the Scribble virtual machine.

This approach gives every function which is visible to Scribble a common interface which can be extremely useful in complex systems as it means a developer can execute a function and get the result without knowing or caring about whether it is executing within Scribble or is executing in native code.

One way to improve my current APIValue / Function implementation would be to allow for arguments in the execute function to be passed as they would in native code instead of passing them as an array.

So the Mod function would go from

```
virtual API::APIValue execute(API::APIValue* values, VM::VirtualMachine* virt) {
 return APIValue(values[0].getIntValue() % values[1].getIntValue());
}
```

to

```
virtual API::APIValue execute(API::APIValue target,
    API::APIValue divisor, VM::VirtualMachine* virt) {
 return APIValue(target.getIntValue() % divisor.getIntValue());
}
```

This technique would provide a more intuitive interface and is used in more developed native

interfaces such as the Java native interface, however it adds additional complexity to the Function class as defining a function with a variable number of arguments without using the complex and confusing virtual arguments syntax is not easily achieved in C++.

Bibliography

JVM Garbage Collection - http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/ind