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

December 5, 2013

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 Where other published and unpublished source materials have been used, these have been acknowledged.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

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.

1.1 Motivation

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).

Chapter 2: **The Language**

The Scribble language is the definition of how the language is written and what effect that will have on the program, this information is then used as the basis for a computer program which takes and executes scribble source code.

The syntax is the set of rules which define whether a given piece of source code is valid within a language and the semantics define what effect different tokens will have on the program that the source code represents.

Scribble does not use an object oriented model and instead is purely a procedural language. The was partly an experiment to see what implications this would have on the usefulness of the language however it also reflects the purpose of Scribble. The language has been designed to allow extension of existing systems in small managed ways so using objects would likely make the syntax more confusing without providing any benefits.

The language attempts to be strictly typed meaning that the compiler will usually not allow interactions between values of different types without explicit conversion. It also requires all variables to have a defined type, however it differs from most high level languages in that the type of variables and functions are defined to the right of the variable name. This approach makes function definitions more understandable and the definition of variables which have their type inferred simpler removing the need for a confusing keyword for automatic types such as 'auto' in C++11. The downside to this approach is that the syntax for expressions that require a conversion between values of different types can be quite verbose.

The language offers limited type inference on local variables during compilation, this type inference makes the definition of variables much simpler (For example var i : int := 0; can now be written as var i := 0; and the compiler will work out that the variable i is of type int) however it has the negative side effect of potentially making the type of a given variable more ambiguous to somebody examining the source code. This is usually not an issue however as the type is often clear from the expression.

2.1 Types

Scribble has three primitive types. A primitive type in Scribble represent values with fixed lengths which are not references to other values

bool - A boolean value capable of being set to only 'true' or 'false'.

int - A 32 bit signed integer capable of representing values between 2,147,483,648 and 2,147,483,647.

float32 - A 32 bit floating point value capable of representing approximations real numbers. Floats are differentiate from integers by ending with an f, so 5f would be a float32 representation of the number 5 whereas just writing 5 would be a int.

The language is also capable of storing and handling references to arrays or structures, a value of the type string for instance would be a reference to a null terminated array of characters. Array and structure reference types are also possible and are explained in the Arrays and Structures sections respectively.

2.2 Arithmetic

The + - * and / operators are used for most arithmetic in Scribble. These operators will accept a left and right hand expression, although the program will only compile successfully if they are the same type. For example 1 + 5 would compile successfully and produce the value 6 when executed but 1 + 5f would not execute as 1 is of type int and 5f is of type float32. This design choice was made to try and remove issues facing other languages in which an error would occur because a programmer would use variables of different types and cause rounding errors.

These are defined in the grammar as

```
Expression '+' Expression
Expression '-' Expression
Expression '*' Expression
Expression '/' Expression
```

Using the '-' operator with only a right hand expression is also valid and will result in the negation of the expression, so -(5 + 5) would produce the value -10 when executed.

You can also use i++, ++i, i– and –i to increment and decrement the value of integer variables by one. These operators also carry the value of the variable either before or after the increment i++ and i– will increment and decrement respectively and return the original value. ++i and –i will increment and decrement and then return the new value.

These are defined in the grammar as

```
WORD '++'
WORD '--'
'--' WORD
'++' WORD
```

2.3 Tests & Boolean operations

In Scribble there are tests for equality, greater than and less than. The tests for equality are = and != which test whether two expressions are equal or whether they are not equal respectively and the tests for less than and greater than are <, <=, >and >= which are less than, less than or equals too, greater than and greater than or equals to.

The tests for equality work with boolean and numeric expressions. The <, <=, > and >= tests only work on expressions that can be evaluated to numbers. In both cases the expression on the left hand side has to evaluate to the same type as the expression on the right hand side (So comparisons or arithmetic on different types).

The grammar for tests is defined as

```
Expression '<' Expression
Expression '<=' Expression

Expression '>' Expression
```

```
Expression '>=' Expression

Expression '=' Expression
Expression '!=' Expression
```

In addition to these tests there the boolean tests 'and' & 'or'. These tests can only be performed on boolean values.

The 'and' test will check whether the left and right hand arguments are both true and if that is the case then it will evaluate to true, in the case that the left hand side is false the 'and' test will not check the right hand side.

The 'or' test will check whether either the left and right hand arguments are true. It checks the left argument first and in the case that it is true then the right hand side will not be checked as the test is guaranteed to be true.

These are defined in the grammar as

```
Expression '&' Expression
Expression '|' Expression
```

2.4 Expressions

To allow more complex calculations to be performed the language allows sequences of the operators and operands to be combined making inputs such as '5 + 5 + 5' valid. These expressions are evaluated from left to right when they are executed which will not effect the output of many operations however I also added the rule

```
'(' Expression ')'
```

into the grammar to allow a developer to manipulate how an expression is executed. For example '5 * 10 / 5' would output the value '10' when executed however '5 * (10 / 5)' would output the value '2' when executed. This allows finer control of how expressions are evaluated whilst keeping the basic syntax simple.

2.5 Variables

Scribble was designed to be a strict language in which every piece of data had a defined type. It was also designed with type decelerations on the right hand side of data to attempt to increase the readability of code. This however made variable definitions convoluted as something like 'int A = 0;' in a language like C would become 'var A : int := 0;' in Scribble. This lead to an alternative method of defining variables was defined which allowed a developer to infer a created variables type from the type of an expression at compile time, maintaining the strict type system within the language but making the definition of variables much simpler.

Defined in the grammar as

```
'var' WORD ':' Type
```

for variables which explicitly specify their type and

```
'var' WORD ':=' Expression
```

for variables which infer their type from an expression, variables become an integral part of any non trivial function in Scribble allowing a developer to store the results of computations.

2.6 Flow Control

The language has three primary control structures. One difference between the flow control structures in Scribble and most other languages is the lack of parentheses around each expression in their definition. For instance while (true) in Java would be while true do in Scribble, I made this choice because I felt that the parentheses where unnecessary and often made flow control structures less readable. When I made this choice it had the side effect of potentially making statements like if a -a; ambiguous, both to the writer and the parser so I added an additional keyword (either 'do' for while and for statements or 'then' for if statements) before the statements in each definition to avoid this issue.

2.6.1 If

If statements execute pieces of code depending on whether a boolean condition is true.

```
if BooleanExpression then {
 Code
}
```

You can also specify statements to be executed when the statement is not true.

```
If BooleanExpression then {
 Code
} else {
 Code
}
```

and chain if statements together so

```
if FirstBooleanExpr then {
 Code
} else if SecondBooleanExpr then {
 Code
}
```

would also be valid.

This is defined in the grammar as

```
'if' Expression 'then' ( Statement | '{' Statements '}' )
```

With an optional

```
'else' ( Statement | '{' Statements '}' )
```

2.6.2 While

A while statement will repeat until the given boolean expression is false. It is written in the form

```
while BooleanExpression do {
 Code
}
```

as an example for a 10 element loop you would write

```
var i := 0;

while i < 10 do {
 i++;
}
```

This is defined in the grammar as

```
'while' Expression 'do' ( Statement | '{' Statements '}' )
```

2.6.3 For

A for statement like a while loop will continue until a condition is false. Unlike the while loop however is also contains syntax to initialize the loop and to step through it which allows for easily loop through arrays using an iterator as well as many other uses.

The syntax for the for loop is as follows

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

for example

```
for var i := 0; i < 100; i++ do {
 sys.Write("Hello World\n");
}
```

will print hello world 100 times to the screen.

This is defined in the grammar as

```
'for' Expression ';' Expression ';' Expression 'do' ( Statement | '{' Statements '}' )
```

2.7 Arrays

Arrays in Scribble have the type array(Subtype) where subtype is another type. You are allowed to have arrays and structures as a subtype (So array(array(int)) is a valid array type). This syntax was chosen over the traditional Type[] syntax to make array definition clearer in a system where types are defined on the right hand side, but has the drawback of being slightly more verbose.

Array data is located on a heap and is accessed via references. This method of access means that when two variables are assigned to the same reference they do not contain copies of the data but actually reference the same piece of data and a change in one would be the same in the other.

Arrays are initialized using the [Number]Type syntax instead of having a dedicated keyword. This syntax was chosen because it is concise and simple, an array of integers could be initialized by writing 'var j := [100]int;'. It does however have the downside of making memory allocation harder to identify, though this is probably fine in a scripting language this ambiguity could cause issues in some larger applications with lots of memory allocation.

To access an index in an array you use the [Index] syntax. So given the array var Test := [100]int; to access the 50th index you would write Test[50] or to assign to the 50th index you would write Test[50] := 50.

The length of an array can be accessed via the len(Array) function. This will return an integer with the length of the array. This was turned into a function as opposed to other possible syntax such as .length in Java because it fits in with the procedural nature of the language and most alternatives were designed for an object oriented languages.

There is no explicit array destruction in Scribble, once there are no more references to a given piece of data (Locally or in the heap) it will automatically be freed.

The initialization of an array is defined in the grammar as

```
'[' Expression ']' Type
```

The compilation will only succeed if the expression is an integer expression.

Access to array elements is defined in the grammar as

```
Sets array data at a given index
WORD '[' Expression ']' ':=' Expression

Gets array data at a given index
WORD '[' Expression ']'
```

2.8 Structures

Structures can be declared using the syntax type Name := struct Name : Type, Name : Type . Structures can only be declared outside of functions. For example

```
function main() {

 type D := struct {}

}
```

Is not valid however

```
type D := struct {
}

function main() {
}
```

is valid.

To create a new instance of a structure you use the syntax Name Data1, Data2, Data... . This will return a reference to the new object on the heap. An example would be

```
type User := struct {
 FirstName : string,
 LastName : string,
 Age : int
}

func main() {

 var blake := User {
  "Blake",
  "Loring",
  20 };

}
```

Structures are accessed through references to them. So given the example above if you then wrote

```
var john := blake;
```

Then john would be a reference to the same instance of the structure that blake does and not a copy.

To access members of a structure you use the -> operator. So to access the firstname of the User structure created in the last example you would write blake ->FirstName

A reference will default to 'nil' unless it is assigned to something. You can also assign it to nil to remove the data connected to it. The language handles all memory allocation under the hood so it will be automatically freed without any explicit deletion.

You can tell whether a reference is nil or points to data by testing equality against nil.

```
var j := nil;

if j = nil then
 sys.Write("Hello World\n");
```

References can be used to construct structures like linked lists.

```
type List := struct {
 User payload,
 List next
}
```

Structure definition is defined in the grammar as

```
DataDefinition: 'WORD' ':' Type
StructureData: DataDefinition | StructureData ',' DataDefinition

Structure:
'type' WORD ':=' 'struct' '{'
  StructureData
'}' ';'
```

The initialization of a structure is defined as

```
WORD '{' Arguments '}'
```

Where the WORD is the name of the structure and the arguments are statements with which the values are to be initialized.

The getting and setting of structure data is defined in the grammar as

```
For set
Expression '->' WORD ':=' Expression

For get
Expression '->' WORD
```

where the expression evaluates to a reference to the structure and the WORD is the name of the variable to get or set.

2.9 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) {
}
```

To exit out of a function early or return a value you can use the return keyword. In Void functions return; will exit immediately and in functions with type return X; where X is an expression or variable of the functions type will exit and return the given value. If the argument given to return differs from the functions type a syntax error will occur.

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.

```
'func' WORD '(' FunctionArguments ')' ':' Type '{' Statements '}'
```

and function calls are written into the grammar as

```
WORD '(' Arguments ')' ';'
```

2.9.1 Constraints

Functions which take the same arguments but differ by return type cannot share the same name. For instance

```
func A() : int {
 return 1;
}

func A() : float32 {
 return 1f;
}
```

is not allowed. This is because the compiler would have no way of supporting type inference on variables and matching functions.

The definition of the function in the grammar is

2.10 Packages

Every file in Scribble is seen as a separate package. These packages contain sets of functions and structures which make up the functionality of that part of the program. Scribble also defines syntax so that packages can import and use structures and functions from other packages.

I originally attempted to have every package addressed by its file name (So to use the math library you would first import math and then write math.Something();) however I ran into issues when two packages shared a name (Such as math.sc in /examples/tests/ and math.sc in /examples/math/). Because of this I devised a syntax which lets you choose how an imported package will be addressed.

You do this by writing

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

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

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

I have found so far that this approach is very effective, allowing Scribble files to be split up so that code can be split up logically and common functions can be reused without adding a difficult syntax when interacting with different packages.

This has also given me the opportunity to start building a 'library' of packages which provide common functionality. See the packages in /examples/sorts/ and /examples/math/ for examples.

Importing is defined in the grammar as

```
'package' WORD ':=' 'import' '(' STRING ')' ';'
```

and the use of an imported function is defined as

```
WORD '.' WORD '(' Arguments ')'
```

2.11 Operators Summary

```
+ Addition
```

```
- Subtraction
* Multiply
/ Divide
[ ] Either array index or array initialize
( ) Function call
++ Increment
-- Decrement
. Package entry select
-> Structure element select
= Equality test
!= Non equality test
> Greater than
>= Greater than or equal to
< Less than
<= Less than or equal to
& Logical and
:= Assignment
```

Chapter 3: **The Compiler**

The Scribble compiler takes the input files and returns either a tree or statements or intermediate code which can be used to execute the program. The compiler also checks that every input file meets all of the rules of the language (Both syntax rules and type rules), handles type inference and links function calls with their equivalent function.

3.1 Lexical Analysis

A lexical analyser takes an input string and returns a list of tokens which can be used by the parser. Tokens are strings which can represent a larger type. For instance the symbol ';' would create a 'END_STATEMENT' token or "Hello World" would create a STRING token.

3.1.1 Flex

Flex is a tool written in C to generate a lexical analyser from a set of regular expressions and rules on what to do with them. Flex generates Scribbles lexical analyser from the file src/Scribble/Parser/Lexer.l. Flex is a more modern version of the tool lex and has much of the same functionality and comes with bindings to yacc and bison.

3.2 Parsing

A parser takes the set of tokens generated by the lexical analyser and uses them to construct the statement tree. The parser is also responsible for detection of syntax errors in the source code.

3.2.1 Bison

Bison is a newer version of the tool Yacc which will construct a parser for a language from a grammar file, Scribbles grammar file is located at src/Scribble/Parser/Parser.yy. The Grammar file contains all of the rules on how to construct the statement

3.3 The Statement Tree

The statement tree is a representation of the parsed program as a tree of possible statements. This tree is constructed by the rules defined in the Bison grammar and can be used to execute the program (Using the tree execution mode) or construct intermediate code for the virtual machine.

This tree is built from children of the Statement class (See src/Scribble/Statement/Statement.hpp) and when Bison constructs each statement it supplies the information it needs to execute,

for example AssignStatement takes information about the variable it should be modifying and a pointer to the expression which the variables value should be set to.

3.4 Type Inference

Type inference is handled by examining the types of expressions after the statement tree is constructed and setting the type of the variable which needs to be inferred to the type of the expression. This type inference has to be done at the same time as other operations which could modify an expressions such as function matching and in the same order that they would execute in the code to ensure that any potential references used in the expressions being examined have been resolved.

3.5 Function Matching

As Scribble can have multiple defined functions which share a name (For an example look at Diff or Abs in examples/math/math.sc). This allows a programmer to write a function for different types without having to change its name, so the C functions abs, labs, fabs could all be defined as Abs in Scribble.

This adds an additional complexity to the compiler as it has to be able to select a function from a list of potential functions, this is achieved by examining the type of each argument and looking in the list of potential functions for a function which takes the same argument.

3.6 Error Checking

There are several kind of mistakes that can be made when writing Scribble files.

Lexical errors occur when the input string cannot be turned into tokens. These will rarely occur in practice in Scribble as most items that aren't keywords would be interpreted as Words for variable IDs or function calls. They will occur however when symbols which are not valid in function names or variable id's are used (Something like ?!*

Syntax errors occur when the set of tokens produced by the lexical analyser does not make a valid program according to the grammar.

Type errors occur in Scribble whenever an operation is performed between two separate types. This can be attempting to assign the value of variable to an expression of the wrong type, comparing expressions of different types or attempting arithmetic on expressions of different types. As Scribble is a strict language no casting is implicit these would all be issues that cause the compiler to exit without compiling the program. These errors are caught using the checkTree function in the statement tree, the reason they are not checked when compiling the grammar is because as functions can be used before they are defined the type of expressions can remain unknown until the statement trees for every function in a package are constructed.

Resolution errors occur when the programmer has tried to use a structure, function or package that do not exist or have not been imported. This is caught when the compiler tries to link all function calls or structure references to equivalent functions or structures. Whilst most

of these errors are obvious resolution errors can also be caused when a programmer uses a function which does not have arguments that match it. For example

```
func Abs(i : float32) {
  if i < 0 then return -i;
  return i;
}

func Other() {
  Abs(3);
}
```

would cause a resolution error as although the function Abs is defined the compiler would not be able to match the function call with the function as the argument is a different type to the type of the parameter i.

Chapter 4: **Intermediate Code**

To make debugging generated code easier and to separate the compiler and virtual machine logically an intermediate language was constructed. This language was modelled around assembly with each line representing a single virtual machine instruction. It follows a three address format, with most instructions taking two inputs and an output.

To differentiate between integers and registers the '$' symbol is placed before any register. So a command to load the integer 5 into register 0 would be 'load 5 $0' instead of 'load 5 0'.

Scribbles intermediate code has no type checking whatsoever, this is expected to be done by the high level compiler. Whilst the virtual machine does sanity check types, arrays lengths etc. any errors in intermediate code will only be picked up at runtime.

4.1 Parsing

The conversion of intermediate code to virtual machine instructions is done by a separate Lex based compiler (see src/SASM/). It is a much simpler compiler than the Scribble compiler capable of producing objects of the class 'InstructionSet' from a piece of intermediate code.

Chapter 5: **The Virtual Machine**

The virtual machine in Scribble is a piece of software which emulates a virtual piece of hardware. It is the part of Scribble which actually executes the set of instructions generated for each function by the compiler.

Virtual machines can vary dramatically in design, two common approaches are register based virtual machines such as the LUA VM or stack based like the Java virtual machine. The stack based architecture has every instruction manipulating a stack, popping the values at the top of the stack for operands and pushing the results to the stack. The register based architectures operate on elements of memory much like hardware registers. These 'registers' are not hardware based and thus do not net any speed benefits however this design does often lead to less instructions being generated and so less code will need to be generated however these instructions are often larger as more data needs to be encoded per instruction. There is much debate over which approach is better however in most languages a just in time (JIT) compiler which compiles the bytecode into native code will be able to optimize the bytecode for the native architecture, negating or minimizing the potential benefits of either.

5.1 Registers

ScribbleVM is a register based virtual machine. Registers are fixed size (64 bit) areas in memory used to store values. In the case of ScribbleVM a large number of registers (Up to 255) are used. Most of these registers are used to store the values of variables however there are also several registers reserved for use when computing expressions.

This approach to variable storage has some major disadvantages, if more than 255 variables are used in a single function then the function will not be able to compile and a lot of work has to be done whenever a function is called compared to the storage of variables at offsets from the stack. It does however have the advantage of being extremely easy to generate code though it will be looked at and is likely to be changed during work done over the second term.

5.2 Primitives

Primitives within the virtual machine are the name given to values which can be stored within a 64 bit register. The instruction set has support for integer values of size 1, 2, 4 and 8 bytes as well as support for floating point numbers of 4 bytes.

Support for all primitives is not yet fully complete within the virtual machine, there needs to be support for 64 bit floating point values and a need for testing how effectively primitives function on the heap.

5.3 Instructions

Instructions within ScribbleVM will all be fixed size (8 bytes). The first byte of the instruction is used to identify what operation the instruction is to perform and the remaining bytes carry the operands for the instruction.

When I designed the virtual machine I made the decision to make every instruction the same size. This choice made any flow control or jumps considerably easier as now the compiler doesn't have to take the length of instructions it is jumping over into account however it has the downside of forcing smaller operations such as the 'TestEqualsZero' to waste space. This issue is exacerbated by the large size of each instruction in Scribble which leads to a lot of instructions wasting space. I will be attempting to reduce the size of each instruction over the next term and potentially redesigning how several instructions function in order to minimize this, my aim being to ensure that each instruction can be encoded into 4 bytes instead of the current 8.

5.4 Instruction Set

The instruction set in ScribbleVM is split up into two regions, the instructions list (an array of fixed size instructions which ScribbleVM can execute) and the constant region a variable sized region that contains all of the constants used in the functions.

The instructions list is a list of fixed size instructions

The constants area was necessary to allow for instructions which needed more data than could fit within a single instruction or to store data of varying size such as a string. Currently this area is access via an index to the first byte of the data to be looked at from the instruction requiring, this approach has the drawback of requiring 4 bytes per constant address to be encoded into an instruction. I will revise this during the second term to make the constant zone more like a table with an index referencing a single element in the constant zone, this should make 2 bytes sufficient for an index and reduce the size of instructions which use the constant area.

5.5 Namespaces, Functions & Types

Instead of having every function in Scribble defined in a single instruction set with labels to jumps to access other functions like machine code I instead have a seperate instruction set for every function and have instructions and special instructions to call between them. I chose to place these instructions with the definition of types which also have to be defined so that the virtual machine can find their size and garbage collection information during runtime.

Instead of having a single large list of entries I chose to instead create 'Namespaces' within the virtual machine. These namespaces form a tree structure, having children which are either functions, types or other namespaces. This reduced the size of the list and also removed the issue of conflicts occurring when two packages declared a function of the same and tried to register them both to the virtual machine.

Calls to entries in different namespaces are resolved at runtime using ':' as a delimiter (So a call to hello:World would look for the entry World in the hello namespace). I use ':' as

the delimiter as it is not valid as a package name under Scribble and so there would not be conflicts with namespaces like 'Test:Namespace'.

Within the virtual machine these lists are implemented as hash maps in order to speed up the execution of the program. Originally I used the std::map class to form the list but I found that it was slower than a hash map when searching for elements of a namespace.

Functions and types have to be registered to namespaces within the VM before it is executed. This registration allows them to be identified and used by the VM at runtime.

5.6 Heap

The heap is the part of the virtual machine used to store arrays of data or complex structures. Each piece of data on the heap has a unique ID. When an instruction wants to access some data on the heap it does so by using it's ID, this ID is known as a reference to the heap.

5.6.1 Garbage Collection

Garbage collection is a method of automatically freeing dynamically allocated memory that is no longer being used by checking to see whether any references to it still exist.

The garbage collector in the virtual machine is very crude. It loops over every reference in the registers and stack and marks them, it then looks at all elements it has marked and marks anything that they reference (As a piece of data on the heap can contain references to other elements on the heap) and after this has completed it deletes every element on the heap which has not been flagged. This method is suitable when speed is not a concern but there are many optimizations that could be used and these should be investigated during the remainder of the project.

5.7 Instructions List

LoadConstant(ConstantIndex : Constant Location (4 bytes), Destination : Register (1 byte)) Load a constant from the constants list and place it in the specified register.

Move (from : Register(1 byte) , to Register (1 byte)) Copy whatever is at the from register to the to register.

JumpDirect(Where : 4 bytes) Jump to the specified instruction.

JumpIndrect(Register : 1 byte) Jump to the location specified by the register.

Add(left : Register (1 byte), right : Register (1 byte) , dest : Register (1 byte)) Place the addition of the left and right registers in the destination register

Subtract(left : Register (1 byte), right : Register (1 byte), dest : Register (1 byte)) Place the subtraction of the right register from the left register in the destination register

Multiply(left : Register (1 byte), right : Register (1 byte), dest : Register (1 byte)) Multiply the left register value by the right register value and place it in the dest register

Divide(left : Register (1 byte), right : Register (1 byte), dest : Register (1 byte)) Divide the left register by the right register and place it in the destination register

Equals(left : Register (1 byte), right : Register (1 byte)) Test whether the left register is equal to the right register. If it is then execute the next instruction otherwise skip an instruction.

EqualsZero(register : Register (1 byte)) Test whether the specified register equals zero. If it is then execute the next instruction else skip an instruction.

LessThan(left : Register (1 byte), right : Register (1 byte)) Test whether the left register is less than the right register. If it is then execute the next instruction otherwise skip an instruction.

LessThanOrEqual(left : Register (1 byte), right : Register (1 byte)) Test whether the left register is less than or equal to the right register. If it is then execute the next instruction otherwise skip an instruction.

NewArray(Length : Register (1 byte), Destination : Register (1 byte), TypeConstant : Constant location (4 bytes)) Create a new array of the specified length and type and place a reference to it in the Destination register.

ArraySet(ArrayReg : Register (1 byte) , IndexReg : Register (1 byte), ValueReg : Register (1 byte)) Set the value of the array at the specified index to be the value of the specified value register.

ArrayGet(ArrayReg : Register (1 byte), IndexReg : Register (1 byte), DestReg : Register (1 byte)) - Set value of the DestRegister to be the value of the array at the specified index

ArrayLength(ArrayReg : Register (1 byte), Dest : Register (1 byte)) Place the length of the array at ArrayReg into Dest

PushRegisters(Start : Register (1 byte) , N : 1 Byte) Push N registers to the stack starting from the start register.

PopRegisters(Start : Register (1 byte), N : 1 Byte) Pop N registers starting from the Start + Nth register and ending with the Start register (The reverse order is so that push 010*pop*0 10 are complimentary)

PopNil() Pop a register from the stack and discard it.

CallFunctionConstant(Constant : Int (4 bytes)) Call a function given a function name in the constant area

CallFunction(Fn : Register (1 byte)) - Call a function with the name of the string that Fn is a reference to

Return - Returns to the previous function and sets the program counter to the instruction after the function call. If there is no function to return to then the VM->execute function will return and by default the program will exit.

Chapter 6: **Progress & Second term goals**

6.1 Current Progress

So far I have spent most of my time working on designing a stable parser and a clear language syntax. Most of my development so far has gone into designing and implementing the statement tree and parser so that things like type inference and the use of functions before they are defined which I wanted the language to be able to support. Currently the grammar, parser and statement tree are fairly complete however the virtual machine, intermediate language and API still need a lot of work. In its current state Scribble is functional but difficult to embed within an application and very slow.

6.2 Future Tasks

The focus of my improvements to the virtual machine will be on redesigning the instruction set to make instructions fit within 4 bytes instead of the current 8 to reduce the size of generated code. I also intend to provide specialized instructions for more common operations in order to increase execution speed.

Currently variables are located in registers, whilst this approach was beneficial when developing the virtual machine as it meant I could test registers without implementing the virtual machine stack it has significant performance implications and I will be looking into alternatives such as storing variables on the stack and seeing what effects that this has on performance.

I will be looking into potential faster alternatives to my garbage collection routine. As the virtual machine has to be halted while the garbage collector is running any potential increases in speed of garbage collection should be investigated as they will significantly impact on the time it takes to execute a program.

A application programming interface (API) needs to be written to allow easy interaction between native and Scribble functions. Currently all interaction between the native code and the VM is done by manipulating the state of the virtual machine through a very crude interface that relies on a intricate knowledge of how the VM works. For the language to be viable as a scripting language this needs to be greatly simplified so that developers can easily call native code or Scribble functions, however this presents several technical issues regarding arguments located on the heap which will need to be overcome.

Bibliography

[1] Bison - The GNU Parser generator website http://www.gnu.org/software/bison/

[2] Flex - The fast lexical analyser website http://flex.sourceforge.net/

[3] The MSDN resource on data type ranges http://msdn.microsoft.com/en-us/library/s3f49ktz.aspx

[4] The GoLang specification http://golang.org/ref/spec

[5] A No-Frills Introduction to Lua 5.1 VM Instructions http://luaforge.net/docman/83/98/ANoFrillsIntroToLua51VMInstructions.pdf

[6] Java Virtual Machine - Wikipedia Article - http://en.wikipedia.org/wiki/Java_virtual_machine

[7] V8 Javascript Engine - https://code.google.com/p/v8/