Final Year Project Report

Full Unit - Final 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 25, 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 Where other published and unpublished source materials have been used, these have been acknowledged.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: 20,597

Student Name: Blake Loring

Date of Submission: Wednesday, 26 March

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 program's 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 speeding up 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 such as 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: 1) the compiler, which handles the process of turning the input source into executable code using the syntax that I have defined, 2) the intermediate language, which represents byte code instruction sets in a human readable form, and 3) the virtual machine, which executes compiled programs.

I decided to create a scripting language because they are an increasingly important part of software engineering. Scripting languages have a large number of uses in a wide variety of applications, such as enabling user add-ons within Microsoft's Office suite, or allowing modifications to virtual reality software. I have used scripting languages like LUA and Javascript (using Google's V8) in previous projects and have wanted to experiment with my own ideas about how a language should be written based on my experiences.

My choices when defining the syntax of Scribble are the result of a desire to take what I consider to be the better parts of imperative programming languages such as C and Go. The Go developers' compelling arguments against object orientation also made me very interested in seeing whether a syntax could be defined so that a non object oriented language was functional and readable when writing equivalent programs, as its object oriented alternatives (in this case Javascript would likely be the closest comparison) and many of my design choices reflect this desire to avoid a traditional object oriented paradigm.

For this report I have produced a user manual and details about the language implementation. Due to the scope of the project it would be difficult to appropriately explain the entire implementation, so I have instead chosen to briefly cover the overall design and development of Scribble, as well as providing documentation on the theory and implementation of several important parts of the Scribble language. I have also provided a user manual which outlines the syntax and semantics of the action, as well as explaining how things like function matching and type inference should be used.

Chapter 2: **Design and Development**

The design of Scribble was split into three sections: the compiler, the virtual machine and the intermediate language. The aim of this design was to make each of these three sections independent of one another so that they could be deployed as separate programs which produce output for each other.

2.1 The Compiler

The compiler is the part of the project which takes the high level Scribble source code and turns it into a set of simple instructions which a computer is capable of executing.

The first step of this process is to parse the source text using a parser generated from a grammar and lexical rules. This parser takes a single piece of source text and generates statement trees for each function and structure type information from structure definitions, as well as marking all of the packages which are imported by the source text. The parser runs recursively on any imported packages, however it checks whether a package has been parsed before to avoid infinite recursion and unnecessary computation. During this step the parser will detect any syntax errors in the source code and if one occurs it will exit with a message informing the user of which line it occurs on.

Once all of the source files for a program have been parsed the compiler attempts to resolve and function and type references found. The reason why the compiler has to resolve references after the entire program has been parsed is because in Scribble a function does not have to be defined before it is used. This requires extra computation when compiling but makes the language much easier to use.

The compiler will then verify that the program is correctly typed and that all references have been resolved properly. This is achieved by the checkTree function which recursively descends down the generated tree using post-order tree traversal, and for each statement in the tree checking that all of its children are the correct type for the program.

Finally, the compiler will generate a set of intermediate code which represents the original source code as a set of low level instructions. This intermediate code is written to a std::stringstream using the generateCode function. This function recursively descends down the statement tree similarly to the checkTree function. However, instead of verifying the statements it generates a low level equivalent in the Scribble ASM language.

2.2 Scribble ASM

Scribble ASM is an intermediate low level language which is able to represent the program in a low level form which is much closer to the virtual machine code. The Scribble ASM compiler has a simple design and requires only a basic single file parser. The job of the compiler is to take intermediate code and generate bytecode instructions which are specific to the virtual machine. This is explained in more detail in Chapter 9.

2.3 The Virtual Machine

The job of the Scribble virtual machines is to take the compiled bytecode produced by the Scribble ASM compiler and execute it. To be able to execute the program, the virtual machine must be capable of doing several things.

The core functionality of the virtual machine is the execution of sets of instructions. Each Scribble instruction is encoded into 8 bytes of memory which stores the opcode, which is used to identify which operation is meant to be performed, and the operands, which give the virtual machine the information required to carry out the instruction. These sets of instructions are represented by an InstructionList class which stores the instructions and constant values required for each part of a program. This is covered in more detail in Chapter 4 on instruction encoding.

In addition to the basic execution of instructions, the virtual machine has to be capable of dynamically allocating memory onto a heap at run time and freeing it when it is no longer used. In order to do this the virtual machine needs to know the amount of memory required, as well as having some way to reference a piece of allocated memory. Instead of allowing the allocator to specify the amount of memory in bytes, It is required that every piece of allocated memory has a type which dictates its size, for example an array(int) of size 4 would have a size of 16 bytes in memory. Memory allocation requires type information in order to enable the virtual machine to inspect dynamically allocated memory at run time. Without this additional type information it would be impossible to work out whether a piece of memory referenced another, and garbage collection would be impossible.

The virtual machine also has to be able to store all of the instruction sets it will execute, and all of the registered types. To do this the VMFunc and VMType classes were devices. The VMFunc class represents a function within a virtual machine and stores its instruction set as well as an identifier (currently the name is stored as a string, however this will likely be replaced with an integer identifier later). The VMType class represents a type within the virtual machine. This type can be one of the primitives types, a structure, or an array, and it stores information on whether it is a reference, what its subtype is and information about a structure's fields. This information is used by the garbage collector to perform reference counting and could also be expanded to allow for something like Java's object reflection whereby type information can be gathered from a structure at run time. Finally, the VMNamespace class provides a method of storing VMFunc and VMType instances in a manner which allows the virtual machine to identify them.

Chapter 3: **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.

3.1 Language Examples

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

3.1.1 Testing

In order to make the process of testing code easier I have developed an online tool which is capable of compiling, executing and displaying the output of single package Scribble programs. All of the examples listed below are easily accessible using a bar on the right of the web page.

The tool is available at `http://www.parsed.co.uk/test.html`

3.1.2 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");
}
```

3.1.3 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");
}
```

3.1.4 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--;
 }

}
```

3.1.5 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");
 }

}
```

3.1.6 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");
 }

}
```

3.1.7 Structures

The next example defines a structure User to hold a user's 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 user's 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);

}
```

3.1.8 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);
}
```

3.2 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.

3.2.1 Identifiers

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

the regular expression:

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

3.2.2 Value Constants

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

Integer

In Scribble an integer constant is a string of digits with no decimal place, prefix or suffix.

Integers are defined in Scribble by the regular expression:

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

which will accept strings of digits between 0 and 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.

Floating point numbers are 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 an array of characters within two double quotation marks.

Strings are defined by the regular expression:

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

.

3.2.3 Operators List

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

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

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

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

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

3.2.4 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.

3.2.5 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;
```

3.3 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.

3.3.1 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 affect 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.

3.4 Types

Scribble is a strictly typed language. This means that every variable, function and value has a specific type. Unlike other languages, 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.

Scribble's 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);
}
```

3.4.1 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 that 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.

3.4.2 Reference Types

A reference type represents a piece of data which is 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 syntactic sugar for an array of characters. When a string constant is written in Scribble, the expression generated will create a new array on the heap and then load the string 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 containing 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 meant 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 were made in order to make type information more clear, 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 can have a with its own type, so a Tree could be represented by writing something like:

```
type Tree := struct {
 payload : int,
 left : Tree
 right : Tree
};
```

A structure is defined by writing:

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

3.4.3 Type Inference

Scribble allows for a variable's type to be inferred when it is declared, instead of explicitly requiring that it be typed. This reduces some of the typed overhead caused by strict languages, without introducing some of the issues which lazy typing systems can experience (such as being unable to easily identify the type of a value as well as rounding errors due to implicit 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 tree's 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. Alternatively 5f + 5f would be inferred to be a float32.

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

3.4.4 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 {
}
```

3.5 Operators & Expressions

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

3.5.1 Constant Expressions

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

Booleans

A boolean constant expression occurs whenever the keywords 'true' or 'false' are used.

Boolean expressions are defined in the grammar as:

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

Integers

An integer constant expression occur whenever the lexical analyser recognizes an integer using the regular expression defined in the lexical analysis chapter.

integer expressions are defined in the grammar as:

```
INT {
}
```

Floats

A floating point constant expression occurs whenever the lexical analyser recognizes a float using the regular expression defined in the lexical analysis chapter.

float expressions are defined in the grammar as:

```
FLOAT32 {
}
```

Strings

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

As a string is not a primitive type. W1hen 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 {
}
```

3.5.2 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 type of the expression that is assigned to. This is done 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 {
}
```

3.5.3 Variable Expressions

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

This is defined in the grammar as:

```
ID {
}
```

3.5.4 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.

Array construction expressions are defined in the grammar as:

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

3.5.5 Structure Construction

Structures are constructed using the StructureTypeName Initial Values separated 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.

Structure construction expressions are defined in the grammar as:

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

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

Type LBRACKET Arguments RBRACKET {
}
```

3.5.6 Array Index Expression

An array index expression returns the value of an array at a given index, written using the notation ArrayExpression[IndexExpression].

This is defined in the grammar as:

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

3.5.7 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))

Structure field expressions are defined in the grammar as:

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

3.5.8 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.

Arithmetic expressions are in the grammar 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.

A minus expression is defined in the grammar as:

```
MINUS Expression {
}
```

Increment & Decrement

In addition to the basic arithmetic operators developers 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 between Scribble and C 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.

Logical operator expressions are defined in the grammar as:

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

3.5.9 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 if the expression to be assigned is 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 {
}
```

3.5.10 Array Length Expression

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

Examples:

```
len([100]int) = 100
len([50]int) = 50
```

would both be true.

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

3.5.11 Parenthesis Expressions

Parenthesis are used 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.

This is defined in the grammar as:

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

3.5.12 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.

Function call expressions are 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 {
};
```

3.6 Statements

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

3.6.1 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
}
```

If statements are defined in the grammar as:

```
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
}
```

For statements are defined in the grammar as:

```
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 {
}
```

While statements are defined in the grammar as:

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

3.6.2 Expression Statements

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

The syntax:

```
Expression;
```

Expression statements are defined in the grammar as:

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

3.6.3 Return Statements

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

Return statements are defined in the grammar as:

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

3.7 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 {
}
;
```

3.8 Packages and 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 are defined in the grammar as:

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

3.8.1 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 4: **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 encodings 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.

4.1 ScribbleVM Instructions

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.

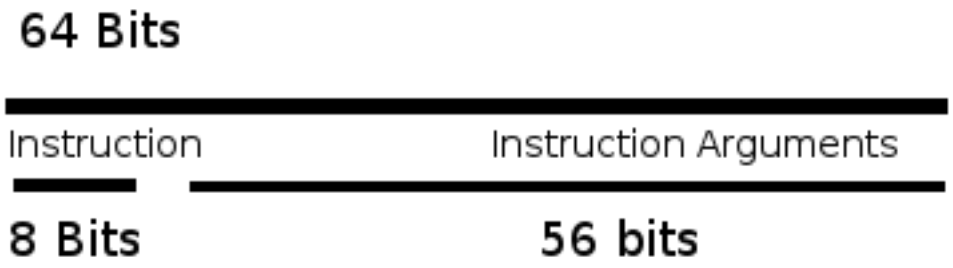

Fig 1. Scribble Instruction

In order to make the virtual machine useful I had to ensure that there were 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.

4.2 The Constants Table

Some instructions in Scribble such as the load string instruction need more than 56 bits of space or a variable amount of space in order to perform there desired function. To allow for this I created a table which accompanies every instruction set and provides a mapping between a small (32 bit) integer and a location in the table.

With this table in place the extra information an instruction needs to execute can be stored in the constant table and the instruction can encode the index required to access the correct constant entry with its instruction. One example of this is the load constant instruction, as the load constant instruction can load values 8, 16, 32 or 64 bits wide instead of encoding the value into the instruction instead it encodes a byte which represents the type of the value as well as the actual value into the constant region. When the VM encounters a load instruction it looks at the constant table entry referenced and uses the data stored in that entry in order to load the desired value.

In order to use the constants table the instructions which required extra memory each had to be modified so that they used constant indices instead of storing the data. In addition to

this the Scribble ASM (SASM) compiler had to be modified so that it is capable of adding entries to the constant table whenever it encounters an instruction that requires it.

One potential improvement here would be to reduce the size of each constant index down from 32 bits to 16 bits. Having 32 bit indices allows for $2^{32} - 1$ unique entries into the constants table for each instruction set. It was designed this way as the additional constant region was setup as flat memory rather than a table and since that change the large amount of space per index is unnecessary. By choosing a 16 bit integer for constant indices each function would be left with a maximum 65535 per instruction set which is more than enough as each function has its own instruction set. This change would reduce the size of the constants and potentially allow a reduction in the size of each instruction for the program.

4.3 Performance

The way in which instructions are encoded has a direct impact upon the performance of the virtual machine. If an instruction encoding is complex then it will require more computation to extract the op code and arguments. As the instruction decoding is performed each time an instruction is executed this additional computation could have an impact on performance.

The instruction encoding in Scribble requires that all arguments are aligned on byte boundaries. The minimum size for an argument is 8 bits and every argument will start at an offset which is multiple of 8 bits from the instruction start. The benefit of this is that it is extremely easy to efficiently extract the argument from the instruction by either treating the instruction as an array of bytes.

The reason why byte aligned arguments are easy to decode is because processor architectures have been developed to offer extremely fast move instructions for byte aligned of a variety of sizes. The process of extracting a byte from an instruction therefore is as simple as using a processor move instruction offset by a certain number of bytes from the base address of the instruction.

The downside to byte aligned arguments is that it can increase the instruction size as not every instruction will require a multiple of 8 bits to represent its data, for some 4 or 12 bits would be more appropriate and the larger 8 or 16 bit widths would lead to wasted space.

Currently the main hit on the virtual machine performance is not the instruction encoding as it is easy to extract the required parts of the instruction, but is instead the large switch statement used to select the action that should be taken for the decoded op code. Currently the virtual machine will loop through all the possible instructions one by one until it finds the correct action to take and then execute it, this is all within a large switch statement. The performance cost of this massively outweighs the complexity of decoding the instruction.

The reason why I had more of a focus on making sure that the instructions could be decoded easily rather than the performance of the op code lookup was because it is much harder to change the instruction format later. While the op code lookup has a much greater impact on performance it could be modified to use a function pointer lookup array or another method with changes to only one source file. A change to the instruction encoding on the other hand would mandate changes not only to the virtual machine but also to the SASM intermediate language and anything that generated code for it would to have to be updated.

4.4 Potential Improvements

One way in which the instruction encoding in Scribbles virtual machine could be improved would be to allow the storage of function variables on the stack rather than in registers.

The benefit of this is that it would free some registers which could then be used to store temporary or intermediate values when calculating expressions, reducing the number of push, pop and move instructions which would need to be executed. It would also allow an unlimited number of variables per function (currently each function is limited to 32 local variables).

This could be achieved by allocating stack space for each variable at the start of a function and introducing OpCodes allow the the getting and setting of specific locations on the stack (Currently the only stack op codes are Push and Pop).

The reason I did not implement this was because it introduced an issue with my method of detection of references within the stack required for garbage collection. This issue could have been sold by dedicated set variable and get variable op codes as variables do not change type however I did not realize this when I was implementing the statements regarding to the storage, getting and setting of variables.

Chapter 5: **Type Inference**

Type inference is the automatic inference of the type of an expression within a programming language. This feature can be used to allow for many useful features within a programming language in many different areas such as variable assignment, error checking and matching functions.

Type inference is a feature most commonly found within functional programming languages, in these languages both functions and data can have their arguments and return type inferred based upon the body of their code. It's purpose in these languages is to free the programmer from the responsibility of assigning types to data.

One example of a language which uses a large amount of type inference is Haskell. Haskell is a functional programming language in which functions are all high order and treated the same as any other piece of data. In Haskell you are never required to explicitly specify the type of arguments or the return type of any function, though you can add type annotation to limit the types of data which a function will accept.

Haskell has a type system which allows for type polymorphism. This means that within Haskell a value can have more than one type. For example the value 1 is both a value of type Integer and a value of type Num. This system is often used in combination with type inference. When a function is declared the most generic type which can be applied to it is inferred from its body, for example f(x) = x+1 would have the type Num -¿ Num inferred from its body and not int, meaning that any numeric type could be used as an argument for this function. This makes functions much more reusable as the same function can be used on data from many types, reducing the amount of duplicated code in a program without using a confusing syntax involved in using C++ templates or Java's generic classes.

This approach also allows Haskell to apply additional error checking onto functions. Haskell can check the inferred type of input arguments to a function and make sure that they match with one of the accepted types that the function can take. Haskell can also check the type annotations which the programmer has applied to the function and make sure that these constraints make sense, for example if a programmer tried to tell Haskell that f(x) = 1 would return a boolean the type mismatch would be detected and Haskell would give an error.

The downside to integrating type inference with so many features is that it can lead to cryptic errors within a program which can be extremely difficult to debug. If there is a syntax or type error with a program Haskell will often spit out a confusing error message rather than something that helps resolve the issue. This happens because the compiler will try to infer a type for a piece of data which matches with what it is expecting and if it cannot it produces a confusing message. For example if the function f x = x + 1 is called with "Hello" as its x parameter the error message

```
No instance for (Num [Char]) arising from a use of ‘f’
Possible fix: add an instance declaration for (Num [Char])
In the expression: f "Hello"
In an equation for ‘it’: it = f "Hello"
```

is produced.

This issue will not occur within Scribble as type polymorphism is not supported and type inference is only has a few uses, many of which are not visible. These issues could become an issue later in the languages development however as features such as high order functions

or polymorphic types would bring a new set of challenges which the type inference system would have to be able to deal with.

5.1 Type inference for variables

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 it allows the language to avoid rounding errors which are caused by implicit casting and also reduces the ambiguity over what the type of an expression will be whilst programming. 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 value which the function will return is made explicit in its definition.

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 as it makes programs more readable and allows the compiler to perform better optimizations, 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 variable 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.

5.2 Type inference in function matching

Scribbles ability to infer the type of expressions is required when trying to select the correct version of a function for a function call.

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.

This feature seeks to remedy an issue found in other imperative languages like C.

```
void Start(char* ProgramName, int argc, char** argv) {
 //Run the specified program
}

void Start() {
 //Start the program
}

int main() {
 Start("BackgroundProcess.exe", 0, 0);
 Start();
}
```

In this example both have used the function Start in different ways which would cause a conflict and confuse any developers. When multiple functions can be defined this would not be an issue as the compiler would be capable of selecting the correct version of Start by inspecting the types of the arguments.

One limit to the type inference within this pattern matching technique is that two functions cannot be defined with the same input types and different output types. This is because it could lead to ambiguous types when combined with the variable type inference used above.

One example of such an ambiguity is

```
func Test(x:int):string {
 return "Hello World";
}

func Test(x:int):int {
 return 5;
}

func main() {
 var j := Test(5);
}
```

The pseudo code shows an ambiguous variable expression var j := Test(5) as the variable j looks to the expression it is being assigned to to define its type however without explicit type information for the variable j it would be impossible to know which version of the function Test should be selected.

While I could have supported both features and added a method which would allow the compiler to check and throw errors whenever such an ambiguity occurred it could have potentially led to some cryptic error messages and programs which were hard to debug - something I wished to avoid with my type inference. Supporting this type of matching would have also led to other confusing inference rules which could lead to confusing code such as which functions would be selected with the expression 'Test(5) = Test(5)'.

5.3 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 which generate data of the same type. This makes issues such as rounding errors that commonly occur 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. In Scribble this problem is solved by combining the same expression type inference functionality which is used to infer the type of variables and combining it with a recursive algorithm which verifies that there are no issues with each node in the program's 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. One example of this is the AddStatement's CheckForIssues function which uses type inference to ensure

that it is adding two expressions of the same type, and also to check that the addition can be performed on values of that type. Alternatively, the CheckForIssues function in an array assignment node checks that the its child expressions types are an array to be assigned to, an integer which identifies the index and a type which is the same as the array subtype.

Chapter 6: **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 by allowing multiple namespaces however they add additional complexity to the code and naming conflicts can still occur.

As an example imagine a C source file called printer.c with the following

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

and main.c with the following

```
#include <printer.h>

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

int main(int argc, char** argv) {
 print("I'm printing a file\n");
 print(getPrinterId(), "HelloWorld.txt");
}
```

In this example the two definitions of the function print within the namespace would cause the compiler to exit with an error. In languages like C++ which support function prototyping the program would compile but the code would still be more confusing to maintain because of the redefinition of print as it is not clear where each is defined.

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 made up from a collection of classes. 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 conflicts still lead to issues (though they are more easily resolved by using the full path to a class).

This approach is designed for object oriented languages and would not work in languages like C, as there could be several different types defined in a single file.

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 work within 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. I found that this approach quickly led to issues when I had packages of the same name, such as importing a set of 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, so it had to be revised.

To avoid this issue 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 7: **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, which has been known to be so severe that they cause developers to avoid allocating memory as often as possible in order to avoid the garbage collector. The impact on performance is caused by the difficulty of finding every reference to memory which has been allocated as it requires a search for all active references. 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, through requiring confusing programming practices, than the convenience gained from automatic memory management.

The Scribble virtual machine performs garbage collection by marking every piece of memory that is referenced by the stack or the 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 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.

```
garbageCollection() {
 toInvestigate = An empty list of heap references;

 for i from 0 to the number of registers {

  if registers[i] is a reference then {
   mark the reference in registers[i] as used
   add the reference at registers[i] to the toInvestigate list
  }

 }

 for each heap reference i on the stack {
  mark the reference i as used
  add the reference i to the toInvestigate list
 }

 for each reference i in toInvestigate {

  if i is a reference to an array of references {
   mark every reference in the array as used
   add every reference in the array to the toInvestigate list
  } else if i is a reference to a structure {
   for each field j of i which is a reference {
    mark the reference j as used
    add the reference j to the toInvestigate list
   }
  }

 }

 for every heap item i which is not marked as used {
  delete i and remove it from the heap
 }

}
```

Above is a example of the garbage collection routine is pseudo code.

One way that Scribbles garbage collection could be improved is by separating allocated mem-

ory into categories based on how long they have existed. Analysis of programs shows that most of the memory allocations within a program are only used for a short period of time, from this we can infer that the younger objects should be checked more often while the older memory allocations need to be inspected less often. If a garbage collector were to separate its memory into two or more sets each representing objects of a certain age 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, which inspects recently created objects, often whilst only executing 'major' garbage collection, which looks at older objects, rarely[4].

There is one other common technique when trying to automatically free memory. 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 and potential issues regarding objects which have references to themselves, as if three pieces of memory reference each other but are no longer within the scope of the program then they will never be deallocated.

Chapter 8: **Application programming interface**

The application programming interface (API) is the way that a scripting language such as Scribble, LUA or V8 interact with the program which is executing them.

The API is an extremely important part of the programming language, as the methods which it provides define how a developer can interact with Scribble programs. An API should expose any parts of the programming language that the developer could need, for example if the API 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 the abstraction helps maintain a backwards compatible API for as long as possible which is useful because developers will not adopt a language which breaks every time there is an update.

This interface is provided in the form of a C++ application programming interface (API). The Scribble API specifies how to make native functions visible to the language, call functions which are visible to the language 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.

8.1 Structure

The Scribble API provides classes which can represent values, types and function, it 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 the API were far to complex and required a detailed knowledge of how the virtual machine allocated memory to use, for example to create a structure it was required that you request from the virtual machine the size of that structure in bytes. There was also no wrapper around the compilation and execution of programs, making compilation, execution and error checking confusing, and all arguments were returned as register values with no indication of what type of data they represented. This system was incredibly difficult to use so I designed a set of classes which I could use to provide a simpler interface to the language.

The first of these classes is the Scribble class. Located in the folder /src/Scribble/ it is the first part of the wrapper that a developer will use when adding Scribble to their application. The Scribble class compiles and prepares a package for execution, ensuring that if any errors occur an appropriate exception is thrown so that the issue can be handled by the developer. The Scribble class also makes it easy to execute Scribble functions with the execute function. This function allows a developer to execute a Scribble function with a set of arguments by using the same expression type inference function matching technique which is used by the

compiler to resolve function calls.

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

```
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

```
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 programs. 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.

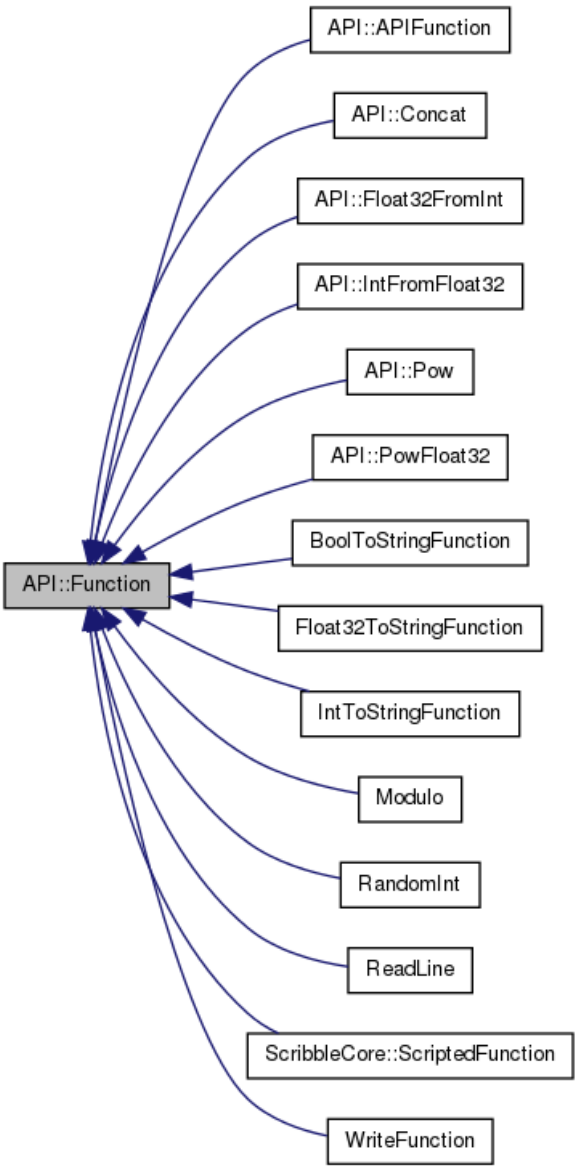

Fig 1. The Class hierarchy of from API::Function showing all of the built in native functions as well as the ScriptedFunction which represents Scribble methods in C++. Generated using graphviz

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

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

to

```
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 complex APIs 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 is difficult within C++ and it would be hard to verify whether a function was being called with the number of arguments which it was supposed to take.

Chapter 9: **Intermediate Code**

The Scribble compiler generates Scribble assembly (SASM). SASM is a language capable of representing every valid function in Scribble at a much lower level than the high level Scribble syntax. The generated SASM code is then converted into bytecode which is compatible with ScribbleVM before it can be executed.

Using an intermediate language over direct translation to bytecode is not necessary however it does have several advantages.

One of these advantages is that it makes debugging the generated low level code easier when trying to fix issues or looking for potential improvements. As a high level compiler runs it generates a statement tree which is then translated into either low level instructions such as bytecode, or to a intermediate language. By using an intermediate language when developing Scribble I am able to inspect the code that is being generated without having to use a complex tool to inspect bytecode. This proved helpful throughout the development of the virtual machine, statement tree and the compiler as there were often non obvious issues with the code generator that I was only able to identify when looking at the low level output.

Another benefit of this approach is that the SASM compiler could be modified to generate bytecode for other virtual machines, platforms, or even generate native code for a processor without requiring any changes or branching of the high level compiler. This would allow Scribble to leverage faster or more mature platforms such as the JVM which could potentially improve performance or increase compatibility with other languages. The independence from the high level language also means that several versions for many different platforms could be maintained without having to worry about the potential maintenance issues caused by maintaining a different version of the language for each platform. Whilst it could be argued that many reasonably low level languages or instruction set could be translated into each other this is made considerably easier by the simple, platform independent syntax of intermediate languages like SASM.

While SASM can be used as a intermediate for conversion to other languages it may not produce good results as the language is tailored toward the virtual machine that it was designed with and conversions could lead to the generation of inefficient code for other platforms. While SASM instruction set is based on a register based virtual machines and could be converted easily to machines which used register based architectures there are many potential issues converting it to other architectures such as stack based machines like the Java virtual machine. If Scribble did begin to use other virtual machines or compile into a different language SASM would first need to be modified to be more abstract so that it could be converted to virtual machines which had fundamentally different designs.

One example of this type of conversion in practice is ASM.js. The project aims to enable the advantages in both speed and syntax of many compiled languages on a browser by taking the intermediate code generated by a compiler which uses the LLVM architecture and translating it into equivalent Javascript[10]. This project shows the wealth of possibilities which a well designed intermediate code architecture could provide, in a small amount of time they have allowed any of the many LLVM compatible languages to execute within a web browser without requiring a custom compiler or even detailed knowledge of the languages being converted.

The downside to having an intermediate language is that it adds an additional overhead to the process of compiling and executing code. Compiling SASM into a bytecode language that can be executed requires the execution of another, albeit simpler, parser and this increases both the amount of memory and processing power required to take a piece of source code and prepare it for execution. This additional overhead may cause issues with the viability of the

language on low powered systems such as robots of televisions. The performance issues could be fixed with custom compilers for those systems or with pre compiled code, these features however would require an investment in both time and resources which make the language less viable.

9.1 Design

SASM had to be specified in such a way that it could be used to represent any program which could be written in Scribble in a low level form which could easily be translated into bytecode.

The language is designed to allow multiple instruction sets to be defined within the same file. This is done by naming each function using the Name { Instructions } syntax. By doing this an entire package can be expressed in a single file, allowing for a reduction in the number of files required to represent a compiled program when compared to a format which only allowed for a single instruction set per source file. The following SASM code is an extract from the compiled output of math/math.sc

```
Diff#0 {
omitted...
ret
}

Exp#0 {
omitted...
ret
}

Exp#1 {
omitted...
ret
}
```

In the example above three different named instruction sets are defined. Two of these sets share the same name and therefore the #Version syntax is added onto the name to differentiate between them. This is how functions with the same names and different signatures are separated once the code has been compiled into SASM, this is required because instruction sets carry no type information and operate using much lower level operations therefore the same type inference techniques used in Scribble cannot be applied.

Every line of SASM instruction code represents a single instruction in bytecode. The line starts with the type of instruction, is followed by the input arguments for that instruction and ends with the destination register for the instruction. Registers are differentiated from other numbers by being prefixed with a $ symbol. For example the following code sample will load the integer 15 into register 0, the integer 30 into register 1 and then add them together and place the result in register 2.

```
load 15 $0
load 30 $1
add $0, $1, $2
```

This convention of instruction input arguments and output register is applied to every instruction. The consistent syntax is an attempt to make instructions easier to understand, if add began with inputs and ended with an output and subtract began with the output and ended with inputs then user errors would be much more likely.

Instructions follow the convention that arguments are used in the order which they are supplied. For example

```
load 5 $0
load 10 $1
sub $0, $1, $2
```

is equivalent to the expression 5 - 10 and not 10 - 5. By specifying this convention across all applicable instructions it makes the semantics of instructions much more predictable without having to double check it's specification, this makes writing code generators easier.

To write a comment in SASM code the – prefix can be used. This will tell the lexical analyser to ignore the rest of the line. Multi-line comments are possible only through repeated use of the – prefix and there is no equivalent to the Scribble multi-line /* */ comment. This approach was chosen because the low level language has been designed to be generated by a machine and comments are only used as a method of identifying specific blocks of code while debugging or looking for potential optimizations, therefore allowing for multi-line comments would add extra complexity to the parser while having no real benefits.

One area in which my design should be improved in the future is regarding type definition. Currently there is no formal specification in SASM for type definition and type information is passed directly from the high level language to the virtual machine. The major drawback of this is that compiler will need to use some non standard method to pass type information instead of SASM to the virtual machine, confusing the code and unnecessarily tying components together. This drawback is currently the reason why Scribbles compiler and virtual machine have to be within the same executable and why code cannot be pre compiled before a programs execution to speed up load times.

9.2 Implementation

SASM is parsed using a lexical analyser and parser generated using the tools Flex and Bison. These tools take a lex file which defines regular expressions that identify blocks of text as specific data types (such as integers, identifiers or registers) and a grammar file which defines the syntax and semantic actions of the language.

In order to take some code written in SASM and compile it into Scribble bytecode the source code must first be loaded into a temporary buffer, that buffer is passed to the generated parser using the sasm_scan_string function. The parser is then executed using the sasm_parse function. The parser will then take the source code and return an instance of VM::VMNamespace which is populated with all of the functions that were defined within the intermediate file. This is all done by the Parser::parse function which is defined as the following within SASM/Parser.hpp.

```
VM::VMNamespace Parser::parse(std::string text);
```

This is repeated for the source code which is output for each package compiled until every

package is loaded into the compiler.

Chapter 10: **Literature Review**

Throughout the project I used various different resources in order to gain the knowledge required to develop Scribble.

10.1 Parsing

While developing the parser I used several resources such as the Lan Gao's Bison calculator[1] example and the Aquamentus Flex and Bison tutorial[2] in order to learn how to construct a grammar for the language that I wanted to create.

When initially approaching the project I had little knowledge regarding the process of making a parser, nor did I have any experience with Bison or Flex. The Bison and Flex files which were provided by Lan Gao served as a reference during my initial attempts to construct simple parsers. The files include a simple grammar, a lexical analysis file and a c file which uses the parser, for a simple calculator.

I used the grammar provided to gain an understanding of what Bison grammar rules looked like, how to apply semantic actions onto the grammar and how to structure a Bison grammar appropriately. This knowledge was useful throughout the project, as whenever I wished to add to or modify the syntax a modification of the Scribble grammar was required.

The lexical analysis file provided gave examples of the format in which regular expressions are interpreted by Flex and how to take data which has been returned by the lexical analyser and pass it to the grammar. This Flex file was used as a template for my initial parsing attempts and my final lex file follows a very similar structure, albeit with more operators, keywords and regular expressions, as well as support for comments.

The example main.cc file provided showed how to use the parser generated by Flex and Bison from C. This proved useful as a starting point when developing my parser, although the way in which Scribble interacts with the parser uses a slightly different set of functions now so that text strings can be used for input as well as files.

The Aquamentus Flex and Bison guide[2] provided a good set of examples covering a range of uses for the tools. I used this document as a reference often throughout development whenever I was trying to implement additions to the syntax which required more advanced features of Bison or Flex. A good example of when I used this resource was when I had to implement single and multi-line comments in the language. When the lexical analyser encounters a language comment it should ignore it and continue analysing from the end of the comment, however the Flex tool does not provide any simple syntax to ignore source code, which meant that with my understanding of Flex I could not allow multi-line comments in Scribble. This is when I turned to the guide and found the Flex "start states" feature which allowed me to match regular expressions only when the lexical analyser was in a specific state. This feature enabled multi-line comments as I could then specify two separate comment states, the single line state which is entered with a // and exited with a new line and the multi-line state which is entered with a /* and exited with a */.

Later during the parser's development I used information from the CS3470 lecture notes[3] as well as knowledge I had learned on the course to modify the grammar in an attempt to remove any ambiguities. While initially developing the language I had little understanding of what an ambiguous grammar was and this led to many mistakes, such as not requiring a

';' at the end of the line, which could cause an incorrect derivation tree to be constructed. The information provided regarding ambiguity on Elizabeth Scott's Compilers And Code Generation course allowed me to spot these issues and adjust the syntax of the language so that they would no longer occur.

10.2 Virtual Machine

When designing my virtual machine I had a hard time deciding on how to represent instructions. From past experiments and a knowledge of Intel x86 assembly language I had a rough ideas about how an instruction set was devised however I had very little knowledge regarding how they were commonly encoded, whether they are fixed size or how things like jumps are handled.

I used the reference guide *A No-Frills Guide to LUA 5.1 VM Instructions*[9] while considering how to design the instruction set for my own virtual machine. The guide covers the instruction encoding and virtual machine implementation used in the popular LUA scripting language. This guide contains information regarding not only how the instruction encoding in the LUA virtual machine works but also how these instructions were displayed as text and how the virtual machine worked overall.

This guide gave me a better understanding of how the instruction encoding would effect the overall performance and functionality of the virtual machine. The explanation of how each instruction modifies the state of the virtual machine and how the programs variables are represented on a stack frame were the most useful parts of the document as they gave me a reference on what mechanism would work in practice when I began prototyping my initial virtual machine. Though the Scribble virtual machine differs greatly from the LUA VM structurally the explanations given about how the LUA VM stores and differentiates between registered functions were also helpful as having knowledge of an existing system to separate out code within the virtual machine helped me to make informed decisions when designing my own.

I used *The Java Virtual Machine Instruction Set*[5] and *Java Garbage Collection Basics*[4] documents much later in the development of my virtual machine when I was beginning to think about how to handle the dynamic allocation and automatic garbage collection of memory.

The methods and constraints regarding how a language allocates and manages memory have a significant impact on every programming language, affecting both performance and syntax. There are two common approaches to memory allocation, unmanaged and managed. In an unmanaged system all dynamic memory has to be allocated and freed by the memory. A good example of this style of memory allocation is the language C in which the malloc and free functions are used to perform all memory allocation. Alternatively, in a managed system there will be some form of automatic garbage collection which can detect when memory is no longer used and automatically free it.

I planned for Scribble to have a managed memory model from the outset, however when faced with the challenge of implementing my own garbage collection routines I found that I did not have the required technical knowledge to design an appropriate system. By reading through the *The Java Virtual Machine Instruction Set*[5] document I gained an understanding of how memory was allocated by the instruction set within a strictly typed and managed language similar to Scribble. This helped me design a set of instructions which allow for the allocation and modification of dynamically allocated memory.

Scribble is executed in a managed environment with predefined types, similar to the Java virtual machine, and so the description of Java's method of garbage collection in the *Java Garbage Collection Basics*[4] document gave me a good understanding of the technical issues I was going to face and one method of overcoming them. The document also introduced me to the idea that a garbage collector could be made faster by separating the allocations into generations based on their age, a feature which could dramatically improve the performance of the Scribble virtual machine.

Chapter 11: **Professional Issues**

There were several issues that arose regarding licensing during development which had to be reviewed.

11.1 Parser Licenses

A portion of the source code which goes into a Scribble build is automatically generated using two tools called Bison and Flex. These tools were released under the GPL license and this causes potential issues as a standard GPL license requires that any modifications to source code must be released under the GPL license and this would by default extend to the generated parser.

The developers of the tools wanted developers to be able to produce proprietary software using generated using Flex and Bison and so they added the exception "The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work." into the version of the GPL license for these tools. This exception permits the distribution of a the source code which these tools generated without the GPL license or the release of the source code and solves the issue covered above[6].

Some older versions of Bison & Flex were not completely covered by this exception, however, as they used the 'yyparse' from their standard GPL predecessor called yacc. If the developers were to compile their program using one of these older versions then portions of their source code would be subject to the GPL license which could be an issue if Scribble was ever used within a proprietary application.

To mitigate the potential license issue covered above, and also make the grammar easier to change, Scribble automatically generates the Bison and Flex files every time it is compiled and none of the generated output is stored within the repository. By removing the generated output I can ensure that there is no risk of running into licensing issues regarding what is in the repository, even if I was using older versions of the Bison or Flex tools on my development machine. This also solves a potential licensing issue regarding the publishing of my source code under a open source license later on as the GPL license stipulates that any source code must be distributed in "the preferred form of the work for making modifications to it". As the grammar is the preferred form for making modifications to a Bison based parser I would be required to distribute it with the source code if Scribble was released under the GPL.

11.2 Licenses of third party libraries

Scribble relies on one third party library, the C++11 standard library. Scribble has been developed using the GNU G++ compiler and the GNU packaged libstd++ and the following licensing details may not be true for versions of libstd++ from other sources.

Because the C++ standard library makes heavy use of templating and inline functions it cannot be linked with an executable without including any source code in the way that programs would be in C. This raises several licensing issues as, while a program written in C can rely on a standard library without actually having to distribute parts of it, a program written with the C++ standard library will often have parts of the standard library embedded

into it during compilation. This makes the licensing of the C++ library the developer is using extremely important as there could be clauses which stop the distribution of the application.

The GNU C++ standard library has been released with a version of the GPL license which adds an exception to allow for proprietary executables to be distributed without making the source code public as long as that program does not make changes to the GNU library it is being compiled with[7]. Scribble falls into this category and therefore there will be no issues with using the GNU g++ compiler to generate a proprietary binary for Scribble. If Scribble was compiled using a different version of libstd++ then there could potentially be licensing issues. This should be looked into by anybody wishing to distribute Scribble in situations where using g++ is not possible.

11.3 Licensing Scribble

It is my plan to release Scribble under a open source license such as the GNU GPLv3 license. This license would allow any interested people to copy, modify and redistribute my code as they liked and even to use it within their projects.

This license is one of the most commonly used within open source software like Linux, Git or the GNU compiler suite and it has been developed for software which is meant to be truly open source, free for anybody to download, modify or redistribute.

One drawback of a GPL license is that by requiring any changes to the original source code to be made public there are potential licensing issues for people which wish to use Scribble programs in a non open source project. This is because when Scribble compiles a program some parts of the Scribble libraries covered by the GPL are embedded into the compiled code. I would need to add exceptions to allow for any output from the Scribble compiler to be exempt from the license in order to avoid this issue.

There is a divide amongst the developers of open source compiler software over whether proprietary third party plug-ins should be allowed or whether they should also be covered by the open source license. By allowing proprietary plug-ins businesses can come in and build closed source tools which use that platform. This could improve the quality of the project as the developers would likely also contribute to the open source portions of the project however it also encourages third party "walled gardens", where a business could come along and attempt to make their variant of the software incompatible with the open source projects through the use of proprietary additions or tools because they believe it will increase market share or profit.

In practice a language will usually have compilers which allow proprietary plugins, and compilers which do not. For example in the case of C and C++ the GNU GCC and G++ compilers do not accept proprietary plug ins due to the copy left requirement in the GPL license however Clang/LLVM does.

It would probably be best to initially use a license which does not allow these proprietary additions initially for Scribble, although this could always be revised later if there was a compelling reason to.

Bibliography

[1] Lan Gao *Bison calculator tutorial* - http://alumni.cs.ucr.edu/~lgao/teaching/bison.html

[2] Aquamentus *Flex and Bison tutorial* - http://aquamentus.com/flex_bison.html

[3] Elizabeth Scott *CS3470 Lecture Notes*

[4] Oracle *Java Garbage Collection Basics* - http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html

[5] Oracle *The Java Virtual Machine Instruction Set* - http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html#jvms-6.2

[6] Free Software Foundation, Inc *Bison GPL license* - http://www.gnu.org/software/bison/manual/bison.html#Copying

[7] Free Software Foundation, Inc *GCC GPL license* - http://gcc.gnu.org/onlinedocs/gcc-3.1/gcc/Copying.html#Copying

[8] *LLVM Language Reference Manual* - http://llvm.org/docs/LangRef.html

[9] Kein-Hong Man *A No-Frills Guide to LUA 5.1 VM Instructions* - http://luaforge.net/docman/83/98/ANoFrillsIntroToLua51VMInstructions

[10] *ASM.js Specification* - http://asmjs.org/spec/latest