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

December 3, 2013

Declaration

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 Where other published and unpublished source materials have been used, these have been acknowledged.

Word Count:

Student Name: Blake Loring

Date of Submission:

Signature:

Table of Contents

Abstract

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

Project Specification

Your project specification goes here.

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 whilst allowing enough access to native functions to perform native of time dependant operation. The design of my project was split into three key sections, the compiler handles the process of turning the input source into executable code using the syntax I have defined. The intermediate language acts as a stepping stone, a language the compiler can relitively easily generate code for which can then be turned into instructions for the virtual machine which will do the actual execution.

Chapter 2: **The Language**

This chapter defines the syntax and rules of the high level language. The language has been designed to attempt make functions and structures more immediately readable when compared to other scripting languages like Javascript.

2.1 Types

Scribble has three primitive types.

The type bool is a primitive type capable of representing true or false.

The type int is a 32 bit signed integer capable of representing values between 2,147,483,648 and 2,147,483,647.

The type float32 is a 32 bit floating point value.

The type string stores references to null terminated strings on the heap.

2.2 Arithmetic

There are four arithmetic operators. + - * and /. These operators only work on numeric expressions of the same type.

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

2.3 Tests

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

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

All of these tests will result in a boolean value with the result being produced.

Some examples would be

5 <10; would be true

10 = 10; would be true

true > false; would cause a syntax error as > only works on numeric expressions

There is also a logical and test. Using the syntax left & right will result in a boolean expression that is only true if both left and right are true. So true & true will equal true however any other variation will return false.

2.4 Expressions

Sequences of tests, function calls or arithmetic operations can be combined. Parentheses can also be used to control the order in which things are execute. For example

```
(5 * 10) + (5 * 10)
5 * 10 + 5 * 10
```

2.5 Variables

Primitive data can be represented either a boolean, integer or string. Booleans can only store the values true or false, Integers are capable of storing 32 bit signed integers and strings are capable of storing strings of text of varying length.

To declare a variable of a specific type the syntax var Name : type; is used. For example var i : int := 0; is valid.

Assignment is done using the := operator. J := 15; would set the value of the variable J to be 15.

Scribble also has the ability to infer the type of a variable when it is declared. So if you have an expression such as 15 + 96 + 4 you can define a variable which automatically infers the type. So var Test := 15 + 96 + 4; would create a new variable of the type integer.

This inference also works with the values obtained from function calls. So in the example

```
func Hello() : int {
 return 30;
}

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

The type of variable j will be set to int.

2.6 Flow Control

The language has three primary control structures.

2.6.1 If

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

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

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

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

and chain if statements together so

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

would also be valid.

2.6.2 While

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

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

as an example for a 10 element loop you would write

```
var i := 0;

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

2.6.3 For

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

The syntax for the for loop is as follows

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

for example

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

will print hello world 100 times to the screen.

2.7 Arrays

Arrays in Scribble have the type array(Subtype) where subtype is another type. You are allowed to have arrays and structures as a subtype (So array(array(int)) is a valid array type).

Like structures arrays are accessed through references to the heap. To create a reference to an array you can write var I : array(int); In this case I would be nil by default as it by default is not a reference to nil.

To create a new array you use the [NumberOfElements]Type syntax. So an array of 100 integers would be constructed using [100]int;

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

The length of an array can be accessed via the len(Array) function. This will return an integer with the length of the array.

2.8 Structures

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

```
function main() {
```

```
 type D := struct {}

}
```

Is not valid however

```
type D := struct {
}

function main() {
}
```

is valid.

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

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

func main() {

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

}
```

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

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

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

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

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

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

```
var j := nil;

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

References can be used to construct structures like linked lists.

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

2.9 Functions

A function is declared using the syntax

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

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

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

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

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

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

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

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

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

```
func PrintTrue(i : bool) {

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

}

func PrintTrue(i : int) {

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

}
```

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

2.9.1 Constraints

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

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

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

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

2.10 Packages

A package is represented as a separate file and interaction between methods and functions in a different package to the one being compiled must be explicit (For example if function A in Hello wants to call function B in world it must use the syntax world.B(); and not just B();).

To use any structures or functions declared in another namespace you must first import it. This will load and compile the file if it has not already been compiled or link it to the current if it has.

An example of this is the sys package which provides functions that allow input and output.

```
import(sys);

func main() {
 sys.Write(Hello World);
}
```

in this example the Write function in name space sys is called which should cause 'Hello World' to be written to the console.

2.11 Operators Summary

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

Chapter 3: **The Compiler**

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

3.1 Lexical Analysis

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

3.1.1 Flex

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

3.2 Parsing

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

3.2.1 Bison

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

3.3 The Statement Tree

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

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

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

3.4 Type Inference

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

3.5 Function Matching

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

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

3.6 Error Checking

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

Syntax errors occur when the user has written something which cannot be handled by the lexical analyser or grammar. An example of this would be mistyping 'for' when writing a for loop.

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

Resolution errors occur when the programmer has tried to use a structure, function or package that do not exist or have not been imported. This is caught when the compiler tries to link all function calls or structure references to equivilent functions or structures. Whilst most of these errors are obvious resolution errors can also be caused when a programmer uses a function which does not have arguments that match it. For example

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

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

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

Chapter 4: **Intermediate Code**

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

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

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

4.1 Instructions

Valid Scribble ASM instructions are

"load" Value Register

"add" Left Right Destination - Add the left and the right register and place the result in the destination register.

"sub" Left Right Destination - Subtract the left and the right register and place the result in the destination register.

"mul" Left Right Destination - Multiply the left and the right register and place the result in the destination register.

"div" Left Right Destination - Divide the left and the right register and place the result in the destination register.

"addf32" Left Right Destination - Add the left and the right register as 32bit floating point values and place the result in the destination register.

"subf32" Left Right Destination - Subtract the left and the right register as 32bit floating point values and place the result in the destination register.

"mulf32" Left Right Destination - Multiply the left and the right register as 32bit floating point values and place the result in the destination register.

"call" Register - Call the function referenced by the register specified (The register must point to a string on the heap with the name of the function)

"move" From To - Copy the value of the From register and place it in the To register

"popn" - Pop and discard a single value from the stack

"neq" Left Right - Test whether the left and right registers are not equal

"eq" Left Right - Test whether the left and right registers are equal

"eqz" Register - Test whether the specified register is equal to zero

"inc" Register - Increment the value of the register by one

"dec" Register - Decrement the value of the register by one

"lt" Left Right - Test whether the left register is less than the right one

"le" Left Right - Test whether the left register is less than or equal to the right one

"gt" Left Right - Test whether the left register is greater than the right register

"ge" Left Right - Test whether the left register is greater than or equal to the right register

"ret" - Return from the current function

"jmp" Instruction - Jump to the specified instruction

"jmpr" Instructions - Jump relative to the current PC by the specified number of instructions

"newarray" Type Length Dest - Create a new array of the specified type and length and place a reference to it in dest

"aset" Array Index Value - Set the register Value to the value of the array at the specified index

"aget" Array Index Value - Set the array at the given index to the value

"alen" Array Dest - Place the length of the array specified into Dest "pushr" From N - Push N registers to the stack starting from the register From "popr" From N - Pop N values into registers from the stack in reverse order ending with From

4.2 Parsing

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

Chapter 5: **The Virtual Machine**

5.1 Registers

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

5.2 Primitives

The primitives in ScribbleVM are byte, short, int, long which are 1, 2, 4 and 8 bytes wide accordingly. When in registers they are all seen as 8 bytes wide (So the opcodes do not have to differentiate between primitive type before executing) and these sizes only apply to values within heap memory or in the constant region.

5.3 Instructions

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

5.4 Instruction Set

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

5.5 Namespaces, Functions & Types

Functions and types are registered to namespaces within the VM before it is executed. This registration allows them to be identified by the VM at runtime. Namespaces can also have namespaces as children, so the root namespace may contain the namespaces 'sys' and 'main'.

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

5.6 Heap

The VM heap is garbage collected and very heavily policed. Each entry inside it will have a type from a set of types that have been registered with the virtual machine and these types will contain information about what is contained within each element on the heap.

5.7 Instructions List

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Bibliography