Chapter 1: **Overview**

Chapter 2: **Language Examples**

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

2.1 Hello World

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

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

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

2.2 Variables

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

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

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

func main() {

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

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

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

2.3 Flow Control

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

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

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

func main() {

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

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

}
```

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

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

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

func main() {

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

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

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

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

```
 }

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

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

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

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

func main() {

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

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

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

}
```

2.4 Functions

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

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

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

func Even(n : int) : bool {

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

```
 return Odd(n-1);
}

func Odd(n : int) : bool {

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

 return Even(n-1);
}

func main() {

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

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

 var even := Even(random);

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

}
```

2.5 Arrays

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

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

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

func main() {

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

 //Initialize the array so that its values range between 0 and len(array)
```

```
 for var i := 0; i < len(intArray); i++ do {
  intArray[i] := i;
 }

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

}
```

2.6 Structures

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

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

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

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

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

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

func main() {

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

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

```
}
```

2.7 Quick Sort

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

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

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

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

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

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

 return false;
}

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

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

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

 return false;
}

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

func PrintUser(user : User) {
 console.Log("Name: ");
```

```
 console.Log(user->name);

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

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

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

func PrintUsers(users : array(User)) {

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

}

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

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

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

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

 while i <= j do {

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

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

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

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

 }

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

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

func main() {

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

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

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

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

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

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

2.8 Native Interface

Chapter 3: **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.

3.1 Identifiers

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

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

3.2 Value Constants

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

3.2.1 Integer

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

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

which will accept strings of the digits 0-9.

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

3.2.2 Floats

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

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

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

3.2.3 Boolean

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

3.2.4 String

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

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

.

3.3 Operators List

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

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

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

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

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

3.4 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.

3.5 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;
```

Chapter 4: **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.

4.0.1 Scope

Scribble currently has a simple very scoping system.

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

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

For example:

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

 if true then {
  j := 30;
 }

}
```

and

```
func main() {

 if true then {
  var j := 30;
 }

 j := 10;
}
```

are valid, however

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

 if true then {
```

```
  var j := false;
 }

}
```

would cause a parsing error.

Chapter 5: **Types**

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

For example

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

would cause a parsing error but

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

would be correct.

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

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

Examples:

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

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

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

5.1 Primitive Types

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

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

For example

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

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

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

5.1.1 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;
```

5.1.2 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.

5.1.3 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.

5.2 Reference Types

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

Reference types are garbage collected, this means that like Java the data is automatically

destructed when a reference to it no longer exists.

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

For example

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

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

}

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

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

However in the case

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

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

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

}

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

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

5.2.1 string

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

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

The string keyword identifies this type.

5.2.2 Arrays

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

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

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

5.2.3 Structures

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

A structure is defined by writing

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

5.3 Type Inference

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

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

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

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

error will be raised.

5.4 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 {
}
```

Chapter 6: **Operators & Expressions**

6.1 Constant Expressions

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

6.1.1 Booleans

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

It is defined in the grammar as

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

6.1.2 Integers

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

It is defined in the grammar as

```
INT {
}
```

6.1.3 Floats

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

It is defined in the grammar as

```
FLOAT32 {
}
```

6.1.4 Strings

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

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

```
STRING {
}
```

6.2 Variable Definition

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

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

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

examples:

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

variable definition is defined in the grammar as

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

6.3 Variable Expressions

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

This is defined in the grammar as

```
ID {
}
```

6.4 Array Construction

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

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

It is defined in the grammar as

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

6.5 Structure Construction

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

Examples:

type Hello name : string, age : int ;

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

It is defined in the grammar as

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

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

Type LBRACKET Arguments RBRACKET {
}
```

6.6 Array Index Expression

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

This is defined in the grammar as

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

6.7 Structure Field Expression

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

It is defined in the grammar as

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

6.8 Operators

6.8.1 Arithmetic

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

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

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

It is defined in the syntax as

```
Expression PLUS Expression {
} | Expression MINUS Expression {
} | Expression TIMES Expression {
} | Expression DIVIDE Expression {
}
```

Minus Expression

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

```
MINUS Expression {
}
```

6.8.2 Increment & Decrement

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

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

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

These operators are defined in the grammar as

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

6.8.3 Comparison

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

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

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

They are defined in the grammar as

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

6.8.4 Logical Operators

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

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

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

It is defined in the grammar as

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

6.8.5 Assignment

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

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

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

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

Examples:

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

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

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

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

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

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

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

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

Assignment is defined in the grammar as

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

6.9 Array Length Expression

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

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

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

6.10 Parenthesis Expressions

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

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

defined in the grammar as

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

6.11 Function Call Expressions

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

Examples:

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

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

defined in the grammar as

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

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

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

Chapter 7: **Statements**

7.1 Flow Control

There are three flow control structures in Scribble.

7.1.1 If

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

Syntax:

```
if Expression then {
Code
}

or

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

Is is defined in the grammar by

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

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

7.1.2 For

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

The syntax:

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

It is defined in the grammar by

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

7.1.3 While

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

The Syntax:

```
while Expression do {
}
```

It is defined in the grammar by

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

7.2 Expression Statements

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

The syntax:

```
Expression;
```

It is defined in the grammar as

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

7.3 Return Statements

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

It is defined in the grammar as

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

Chapter 8: **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 {
}
;
```

Chapter 9: **Packages & Importing**

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

9.0.1 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 10: **Standard Functions**

Bibliography

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

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