Chapter 1: **Overview**

Chapter 2: **Language Examples**

2.1 Hello World

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

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

2.2 Variables

2.3 Loops

2.4 Functions

2.5 Arrays

2.6 Structures

Chapter 3: **Lexical**

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**

5.1 Type System

5.2 Primitive Types

5.3 Reference Types

5.4 Arrays

5.5 Structures

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.

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.

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;

var arr := [10]int; arr[0] := 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 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**

Chapter 9: **Packages & Importing**

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/