Programs

Scribble programs are comprised of functions and data structures split up into different name spaces.

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.

Namespaces

A name space is represented as a separate file and interaction between methods and functions in a different name space 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” name space which provides basic IO functionality as demonstrated by this hello world example.

```
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.

Data

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.

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 respectivelly and return the original value. ++i and --i will increment and decrement and then return the new value.

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 <, <=, >, >= 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 < <= > >= 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 no float > int you would have to cast)

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

Some examples would be

var j := 5 < 10; would leave the variable j as true

var t := 10 = 10; would leave the variable t as true

var j := 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.

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

would produce different results.

These are executed from left to right unless split up with parenthesis.

Function Call

To execute a different function and get the result you use the syntax Name(arguments);

so to write hello world to the screen using the sys.Write function you would write sys.Write("Hello World\n");

These can be used in expressions if the function has a return type other than void. So sys.RandomInt(100) * 10; will multiply the result of getting a random integer by 10

If a there are multiple definitions of a function which take different arguments then the compiler will use the function that matches the arguments provided or throw an error.

Control

The language has three primary control structures.

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 {
} else {
  Code
}
```

and chain if statements together so

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

would also be valid.

While

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

```
while BooleanExpression do {
}
```

as an example for a 10 element loop you would write

```
var I := 0;

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

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.

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 50^{th} index you would write Test[50] or to assign to the 50^{th} 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.

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 → keyword. 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
}
```

Library

The 'sys' library is provided by the system and provides several helpful functions.

sys.String(boolean or number) - This function takes a given integer or boolean argument and returns a string representation of it

sys.Write(string) - This function outputs a given string to the console

sys.RandomInt(max : int) – This function returns a random integer between 0 and max

sys.Mod(left : int, right : int) – This function returns left argument modulo right argument.

sys.ReadLine() - This function blocks until a line is entered in the console and then returns it

sys.Concat(a : string, b : string) – This function returns the concatenation of the strings a and b together.

Operator 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