So What Can It Do?

Kit Lao

June 12, 2020

Primitives

Numbers

Pretty straight forward. Integers and floats.

```
>>> 4
4
>>> 76642
76642
>>> 009732
9732
>>> 12.00045
1200045.0
```

Booleans

```
>>> True
True
>>> False
False
```

Strings

Haven't implemented yet.

Lists

Haven't implemented yet.

Arithmetic

Signed Numbers

```
<<< ----5
5
<<< +3
3
<<< +++3
3
<<< +--+6
6
```

Addition

```
<<< 5 + 2
7
<<< 0 + 12 + 3 + 10
25
```

Subtraction

```
<<< 10 - 9
1
<<< 19 - 1 - 2 - 15
1
```

Multiplication

```
<<< 4 * 2
8
<<< 1 * 3 * 5 * 2
30
```

Division

```
<<< 10 / 2
5.0
<<< 15 / 3 / 2
2.5
<<< 10 / 0
Traceback (most recent call last):
File <stdin>, line 1, in <program>
RunTime error: Division by zero

10 / 0
```

Modulo

```
<<< 27 % 3
0
<<< 5 % 2
1
<<< 0 % 3
0
<<< 5 % 0
Traceback (most recent call last):
File <stdin>, line 1, in <program>
RunTime error: Modulo by zero

5 % 0
```

Exponent

```
<<< 2 ^ 3 ^ 2
512
<<< 2 ^ 0
1
<<< 5 ^ 2
25
```

Parenthesis

```
<<< (4 + 6) * 2
20
<<< 3 + 4 * 2 * (1 + 1) / (0.6 + 0.4 - 0.5)
6.2
```

This shit literally works like PEMDAS, but with modulo. The modulo operator has same priority has multiplication and division.

Comparison Operators

Equality

```
<<< 5 == 6
False
<<< 5 == 5
True
<<< 5 == 5 == True
True
<<< 5 == (5 == True)
False
```

Inequality

```
<<< 5 /= 5
False
<<< 5 /= 6
True
<<< 5 /= 5 == False
True
```

Greater Than

```
<<< 5 > 4
True
<<< 4 > 5
False
```

Greater Than or Equal to

```
<<< 5 >= 4
True
<<< 4 >= 5
False
<<< 5 >= 5
True
```

Less Than

```
<<< 4 < 5
True
<<< 6 < 5
False
```

Less Than or Equal to

```
<<< 5 <= 6
True
<<< 6 <=6
True
<<< 7 <=6
False
```

So like, this shit is pretty straight forward... All operators have equal precedence.

Logical Operators

Negation

```
<<< not True
False
<<< not False
```

```
True
<<< not 5 + 6
False
```

AND

```
<<< True and False
False
<<< True and True
True
```

OR

```
<<< True or False
True
<<< False or True and False
False
```

This shit works just like discrete math. AND operators takes most priority, followed or OR, followed by NOT.

Variables

Declare and initialize variables using the "var" keyword. Everything to the right of the equal sign could be an expression or another variable assignment. Access variables by just stating it.

```
<<< var x = 5
5
<<< var x = 4 * (2 + 3) / 2
10.0
<<< var x = var y = var z = 20
20
<<< x
20
<<< y
20
<<< z
20
<<< var test_variable = True
True
<<< var test_variable_2 = True or (not x and True)
True
```

Error will be thrown when trying to access a variable that is not defined or outside of the scope.

```
<<< variable_that_doesnt_exist
Traceback (most recent call last):
File <stdin>, line 1, in <program>
RunTime error: 'variable_that_doesnt_exist' is not defined

variable_that_doesnt_exist
```

Variables must be initialized upon declaration.

```
<<< var x
Invalid syntax: Expected '=' File <stdin>, line 1

var x
```

Variable names need to start with a letter or underscore. The rest of the characters can only be numbers, letters, or underscore.

```
<<< var 86 = 9
Invalid syntax: Expected identifier File <stdin>, line 1

var 86 = 9

<<< var 8var_name = 2
Invalid syntax: Expected identifier File <stdin>, line 1

var 8var_name = 2

<<< var _xyx23-?. = 10
Illegal symbol: '_' File <stdin>, line 1

var _xyx23-?. = 10
    ^
```

Conditional Statements

Conditional statements starts with the "if" keyword. After the condition, there needs to be a "then" keyword before it leads to the expression. Multiple conditions are also implemented using the "elif" keyword. The "else" keyword may be used or not. All conditional statements must end in the "elif" keyword. The if statement is written in one line. To interpret expressions with multiple lines, put the expression in a file.

```
<<< if True then 10 endif
10
<<< var x = 10
```

```
10
<<< var y = 15
15
<<< var z = 20
20
<<< if x == y then 25
    elif x == 20 then 30
    elif z - 2 * 5 == x then 40
    else 50 endif
40
```

Variables inside the scope of a conditional statement has access to all variables from it's parent scope. However, the parent score does not have access to what's declared in the conditional scope. In the follow code, the variable 'x' gets updated by being reinitialized and declared using the "var" keyword.

```
<<< var x = 10
10
<<< if True then var x = x + 5 else var x = x - 5 endif
15
<<< x
15
<<< if False then x else var y = 10 endif
10
<<< y
Traceback (most recent call last):
File <stdin>, line 1, in <program>
RunTime error: 'y' is not defined

y
```

Loops

For loops are initiated using the "for" keyword, a variable declaration that indicates the initial value, the "to" keyword, a numeric expression that indicates the final value, the "do" keyword, the expression inside the loop, and the "endfor" keyword.
The variable used as the iterator is only within the scope of the loop. The scope of the loop has access to it's parent's environment, but not vice versa.

```
<<< var x = 0
0
<<< for var i = 0 to 10 do var x = x + i endfor
45
<<< x
```

```
45
<<< i
Traceback (most recent call last):
File <stdin>, line 1, in <program>
RunTime error: 'i' is not defined

i
```

The nested for loop is all written in one line from the interpreter. Multiple line expressions needs to be inside a file.

```
<<< var res = 0
0
<<< var x = 12
12
<<< for var i = 0 - 2 * 2  - 1  to x - 2 do
        for var j = 0 to 10 do
            if (i + j) % 2 == 0 then
                var res = res + 1
            endif
        endfor
    endfor
>>> 75
```

While loops initiated using the "while" keyword, a logical, arithmetic, or comparison expression, the "do" keyword, the loop expression, and the "endwhile" keyword. Pretty much the same as for loop except it runs until the condition is false.

```
<<< var x = -2
-2
<<< while x /= 10 + 2 do var x = x + 1 endwhile
12
```

Functions

Functions are basically sub-programs, with the possibility of pre-defined variables from parameters. The body of the functions could literally contain any type of expressions that's described above. Whatever the expressions interprets to is the returned value.

```
>>> var n = 0
>>> def f(x, y, m) {
        for var i = x to y do
            var n = n + m * i
        endfor
```

```
    }
>>> f(10, 40, 2)
1470
```

Each function has their own scope, who is a child of wherever it is declared. The example above shows the variable 'n' declared in the parent scope but is accessible within the function. This example below shows that whatever is declared inside a function, stays inside the function.

```
>>> def f() {
        def g(a) {
            a
        }
    }
>>> g(2)
Traceback (most recent call last):
File <stdin>, line 1, in <program>
RunTime error: 'g' function is not defined

g(2)
 ^
```

However, in this language, functions are values. So functions that are not within a scope of another function can still be accessed if they are passed in. The example below shows that.

```
>>> def f() {
        def g(n) {
            if n <= 0 then
                True
            else
                h(g, n-1)
            endif
            }
        }
>>> def h(func, n) {func(n)}
>>> f()(1)
True
```

And of course, there is recursion.

```
>>> def fib(n) {
        if n <= 2 then
            1
        else
            fib(n-1) + fib(n-2)
        endif
    }
```

```
>>> fib(8)
21
```

These are just general examples of what my compiler can do. The test cases will have more examples.