CIIC4030 PL Assignment #4 Report

Implementation

The interpreter works of the previous assignments, specifically the scanner and parser. A brief overview of both. The scanner reads a text file and if it finds certain words or combinations of letters converts them into a pre-assigned list of tokens. The parser evaluates the syntax and grammar of these tokens, in essence classifies the different possible interactions between tokens, and uses these to create an abstract syntax tree (AST). Both of these are in python, using a library called ply.lex and ply.yacc. The interpreter uses this AST that the parser create to actually execute a given program. When given the stms of the AST, it will check what type it is and execute the corresponding evaluation. For example, if the type is equal to stm_op and that operation is sum, it extracts the first and second value from that statement and adds them together. It can also understand variables, and it does so by having an internal dictionary called env, short for environment. This functions as our scope. For example, functions when called don't share our global env instead opting to have their own new env, which keeps the terms defined internally restricted to that scope. As is customary in most modern languages. Functions are also a parameter of the interpreter function, this is used to validate that a given function does exist before committing to trying to execute it.

Testing

Test 1:

Here's a basic piece of example code given in a previous assignment.

```
1   func SomeFunction[n] :=
2       let
3           val r := 15 end
4       in
5           n*r
6       end
7   end
8
9   exec SomeFunction[3]
```

Evaluating it by hand, we can see that all it does is multiply 15 by any input, in this case 3. So we expect it to return 45.

```
- Interpreter/main.py"
45
```

And when we run the program, we get the expected output.

Test 2:

Another arithmetic problem, but this time using parentheses to confirm that the recursion in the interpreter works well. It also checks uminus and constants.

```
1    func SomeFunction[n] :=
2        let
3            val r := 15 end
4            val s := 2 end
5        in
6            -((((n*r) - n)/s) + 4)
7        end
8    end
9
10   exec SomeFunction[3]
```

We expect this to return -25.

```
- Interpreter/main.py"
-25.0
```

Test 3:

Now let's try out the same but for strings. Some basic string concatination.

```
1    func SomeFunction[n] :=
2        n + "World"
3    end
4
5    exec SomeFunction["Hello"]
```

We expect this to output the classic “Hello World” message.

```
- Interpreter/main.py"
"Hello""World"
```

And it would seem correct, but there’s something off. At first, I thought that it might be printing the string one ofter another, but no, it is concatenating them. The issue stems from the way that the scanner stores the string, when it does so it does not remove the quotation marks. Since this assignment is about the interpreter and therefore this is out of scope, I’ll leave this as a fix for the future if I decide to expand my project. As for the interpreter itself, it correctly interpreted the string as expected, the issue stems from a different part of the code.

Test 4:

For the final test, I wrote raise exceptions for certain common errors, such as forgetting to define variables or functions. So let's test to see if one of them is working as intended.

```
1    func SomeFunction[n] :=
2        n + y
3    end
4
5    exec SomeFunction[3]
6
```

Here, y is undefined, so we expect the program to return an “undefined var: y error”

```
Exception: Undefined var: y
```

And we got the correct error message as expected.