Interpreter Code

Introduction

This project is to apply syntax and semantics that the professor given in conjunction with the Lexer code that transform a sequence of the character into token, and the parser that take the sequence of the parser and produces an abstract syntax tree created in the previous part to modify the feature of the language to create a Interpreter code. I attach 2 types of the code for consideration. In the project2_Compiler zip file, there are the Lexer code, Parser code and Interpreter code so that it can run as the compiler. This project is written in python to implement the codes.

File in the Archive

1. project3.py

This file contains the Lexer code, the Parser code and the Interpreter Code and other classes and functions that is necessary to run as the compiler for the language.

2. shell.py

This file is used to compile in order to run the compiler

3. Strings_with_arroaw.py

This file contains the helper function used with the project1.py file

Procedure

1) to run project3_Compiler

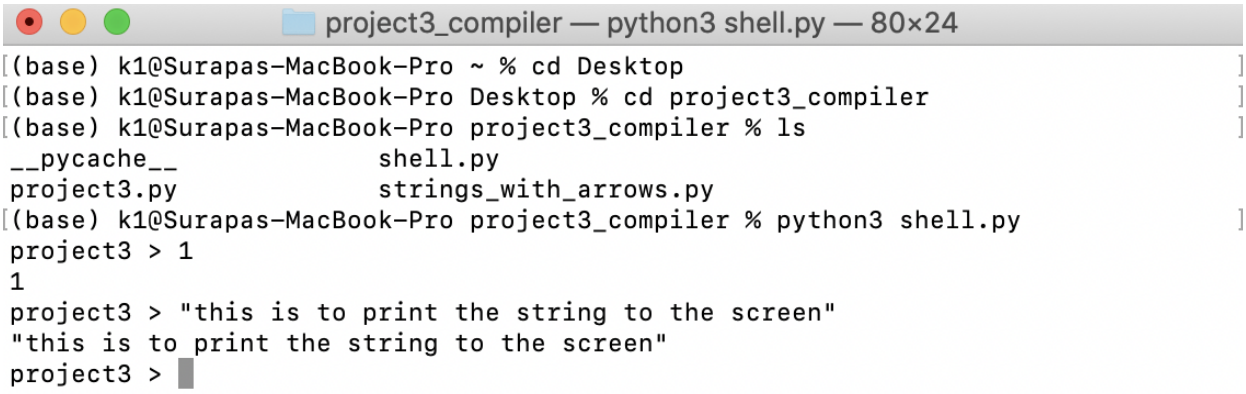
To use the interpreter for the language, the user must compile the shell.py file on the terminal or on the CS1 or CS2. Note that the user must contain project3.py and Strings_with_arroaw.py on the same directory while compiling

Compile as:

Python3 shell.py

For the purposes of consideration of the Parse code, please read the Node, Parser Result and Parser section.

A picture below shows how to compile project3_Compiler to run the interpreter



Result

In project3.py, it contains the Lexer code, the Parser code and the Interpreter Code (for the purpose to run as the compiler) and other classes and functions that is necessary to run as the compiler for the language.

The Lexer code transforms the source code and breaks into the token stream. There are many type of token stream that Lexer code can break into string, operator, and number. The string and the number together might be identifier or the keyword. So, the token stream that break into has the name itself and the value attach to it.

The parser Code contains the implementation of the grammar rule according to the Professor assign. It converts the text into the new structure according to the grammar rule provide. And I adjust some syntax and sematic. Please see the example below for your consideration. For the Parser code part, I create the parser node according to the syntax and semantic the professor provided and adjust some rule. There are may type of node such as : NumberNode, StringNode, ListNode, VarAccessNode, VarAssignNode, BinaryOperationNode, UnaryOperationNode and so on. Plese see the node section for your consideration.

After that I built the tree using the Parser Node. The parser is syntax tree of the program. The Parser will keep track of the token index, similar to the Lexer. This is used to check whether the input token follow the grammar rule generated by the tree or not otherwise it will return error. There is the Invalid syntax error. If the invalid grammar is input to as in Example 6 below.

After that the interpreter code will interpret the parse tree created by using the traversing the tree that I built from many tyoe of the node according to my rule and interpreted it. And the program will print the result the screen. My interpreter will make use of our Lexer and parser to get the AST of our input expression and then evaluate that AST whichever way I want.

Please consider line 1666 to 1900 to understand the my interpreter code from the interpreter part and run part.

In addition, while using the interpreter, the error might occur. So I create the error class to track the error while using the interpreter. There are many type of the error that can occur from the Lexer code part, the Parser code part, and the interpreter code part. For the interpreter part, please consider the RTError which is the run time error in line 54. The Example 6 also shows the run time error that prints into the screen.

Compiler Example

This is the implementation of the compiler

1. **Type the int value on the terminal**

```
project3 > 1
```

2. **Type the string with the quote on the terminal**

```
project3 > "this is to print the string to the screen"
"this is to print the string to the screen"
```

3. **Assignment of the string to the identifier and print the varible**

```
project3 > get x = "hello world"
"hello world"
project3 > print(x)
hello world
0
```

3. **Assignment of the value on the terminal**

project3 > get a = 1

1

project3 > get b = 2

2

4. **Apply the if else statement togther with 3.**

 # case1

project3 > if a > b then a+3 else a+b

3

 # case 2

project1 > get k = 5

5

project3> if k == 5 then print("k is euqal 5") else (" k is not equal 5")

k is euqal 5

5. **Error case, and the compiler trace back to the error**

project3 > get y = 3

3

project3 > get z = "hello"

"hello"

project3 > print(y+z)

Traceback (most recent call last):

File <stdin>, line 1, in <program>

Runtime Error: Illegal operation

6. Implementation of a while loop

```
project3 > get x = 3
3
project3 > while x < 5 then get x = x+1
[4, 5]
project3 > while x < 5 then get y = y+1
[ ]
```

 # Error Case

```
project3 > get z = 3
3
project3 > while z < 5 then get d = d+2
Traceback (most recent call last):
  File <stdin>, line 1, in <program>
Runtime Error: 'd' is not defined
```

7. Implement of the input method (the assign method)

```
project3 > get x = INPUT()
string
"string"
project3 > print(x)
string
0
project3 > get x = INPUT()
```

1

"1"

project3 > print(x)

1

0