Computational Problem Solving CSCI-603
PreTee Interpreter Lab 8

1 Introduction

In this lab, you will get experience developing a small interpreter for a very simple mathematical oriented language named *PreTee*.

The mathematical expressions will be formulated in `prefix` form and it supports only the following binary operations: addition (`+`), subtraction (`-`), floor integer division (`//`), and multiplication (`*`). The operands are integer values (negative or positive), and variables. A data structure called a *symbol table* is used by the interpreter to associate a variable with its integer value.

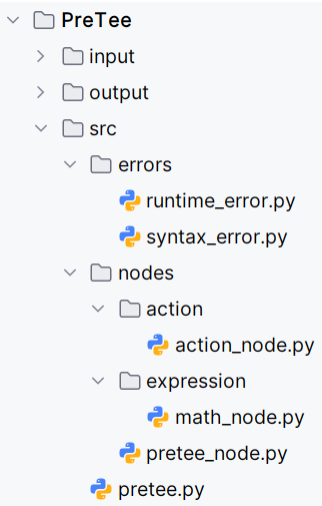
2 Implementation

2.1 Starter code

Download the starter code from `here`.

Extract the zip file, and then copy the `input`, `output` and `src` directories into the root of your project folder in PyCharm. To make PyCharm aware the `src` directory has source code, you should right click on the `src` directory and select **Mark directory as... Sources root**. The `src` directory should be blue color.

Your project structure should look like the image below. The sections below will cover details of the classes provided in the starter code.



2.2 PreTee language

A source program in the PreTee language is a sequence of statements. There is no punctuation between statements. Spaces are required between each element, or token, of every statement.

The source program statements must begin with one of four tokens:

- `#`: The line is a comment. This line is ignored when parsed but it counts as a line number in the source program.
- `=`: The line is an assignment statement.
- `@`: The line is a print statement.
- `‘‘’’`: A blank line (counted as a line number but ignored).

You have been provided with some PreTee program files. Take a look at the files from the `input` folder in the starter code.

2.2.1 Statements Types: Assignment and Print

Assignment	
Token	=
Syntax	= {`identifier`} {`expression`}
Description	Assigns the evaluation of the expression to a variable named by the identifier.
Print	
Token	@
Syntax	@ {`expression`}
Description	Prints to the standard output the evaluation of the expression.

2.2.2 Expression Types: Literal, Variable and BinaryOperator

These are the various expressions that can be encountered when parsing statements.

Literal	
Syntax	A sequence of digits representing an integer number, possibly preceded by a minus sign.
Variable	
Syntax	A sequence of character when the first one is guaranteed to be alphabetic.
BinaryOperator	
Syntax	{`binary-operator`} {`expression1`} {`expression2`}
Binary operators	`+, -, //, *`
Description	The binary operation will be evaluated in infix form: {`expression1`} {`binary-operator`} {`expression2`}

2.3 Program Operation

The interpreter, `pretee.py`, is run by providing the source code file as a command line argument. Example:

```
$ python3 pretree.py prog1.pre
```

When run, the three stages of interpretation occur in order:

1. **Parsing**: The source code is compiled into a sequential collection of parse trees.
2. **Emitting**: The parse trees are traversed inorder to generate infix strings of the statements.
3. **Evaluating**: The parse trees are executed and any printed output is generated.

The main method of the `pretee.py` module is given to you completely and should not be changed. You will be implementing all the methods for the `PreTee` class.

2.4 Documentation

Here is the documentation for all the classes you must implement for this lab. **It is strongly suggested you look at the docstrings because they explain, in detail, how each method should operate.**

2.5 Design

To implement the interpreter we will heavily rely in core OOP principles such as inheritance and polymorphism. We will also design our solution using a structural design pattern called *Composite*.

The class diagram below details all the classes and their relationships. We strongly suggest that you follow this design.

UML class diagram

The core abstract classes in the hierarchy of node classes are already provided to you. **Do not modify these classes**. Here is a description of those classes:

- `PreTeeNode`: This is the most-top class in the hierarchy. PreTeeNode is an abstract class describing the method (`emit()`) that all nodes in the parse trees must implement regardless its type.
- `ActionNode`: This is an abstract subclass of `PreTeeNode` that describe the methods that the statement type nodes must implement. `Print` and `Assignment` are subclasses of this class. Each parse tree will have as root an object of an `ActionNode` subclass.
- `MathNode`: This is an abstract subclass of `PreTeeNode` that describe the methods that the expression type nodes must implement. `Literal`, `Variable`, and `BinaryOperatpr` are subclasses of this class.

2.6 Design Constraints

You are not allowed to use Python's build in `eval()` method for evaluating nodes. All nodes must be evaluating by traversing the parse trees from the root.

Here is the suggested order of implementation for this lab:

1. Write the constructor for `PreTee` so it initializes all the instance fields.

2. Work on the parsing, starting with `PreTee.parse` method. This is analogous to the work you did on the last question in problem solving. You should build this up incrementally.
   First work on implementing the `Assignment` class. After that, implement the constructor for `Literal` and `Variable` and see if you can correctly build the parse tree for an assignment of a literal to a variable.
3. In order to tell if the tree is being built correct, implement the `emit` method in `Literal` and `Variable`. Now implement `PreTee.emit` to call `emit` on the root parse node/s and verify their infix form.
4. Add in the `Print` implementation of the constructor and `emit`. Try printing some literals and variables to see if they work correctly.
5. Implement the `BinaryOperator` class constructor and `emit` and see if you can handle math expressions.
6. Now move on to execution of the parse trees. Implement `PreTee.evaluate`, along with all the other node classes `evaluate` method, and make sure you can handle the full grammar of the language.
7. Next work on handling syntax errors. They are raised in the various node classes and should be handled in `PreTee.parse`.
8. Finally work on handling runtime errors. Again, these are raised in the various node classes. You do not need to handle them because the supplied `main` function already does.

2.6.1 Error Handling

There are seven distinct type of errors the interpreter must handle when compiling and evaluating the PreTee programs. You can find more information about these errors in the documentation provided and the output files. Here is a summary of all those errors:

1. Encountering an illegal action token (not commenting, printing, assigning nor empty line).
2. Attempting to create a variable with an invalid identifier.
3. Attempting to assign variable to a non-math expression.
4. Attempting to use a variable that has not been assigned to yet.
5. Encountering an illegal token while parsing an expression.
6. Ran out of required input when parsing an expression.
7. Attempting to divide by a zero denominator.

2.6.2 Input and Output files

You have been provided with some PreTee program files to verify the correctness of your solution. The files in the `input` folder contain the PreTee source code files required to run the program and the files in the `output` folder contain the output generated by the interpreter when executed with its corresponding file from the `input` folder.

Notice that these input files are not comprehensive and do not test every possible case scenario. We encourage you to create your own test cases for a more thorough testing.

3 Grading

- 20% Problem Solving
- 15% Design
- 55% Functionality:
  - Expression nodes: 15%
  - Action nodes: 10%
  - Compilation: 10 %
  - Emitting: 5%
  - Evaluation: 5%
  - Error handling: 10%
- 10% Style: Proper commenting of modules, classes and methods (e.g. using docstring's).

4 Submission

Create a ZIP file named `lab8.zip` that contains all your source code. Please note you are not submitting the supplied modules that should not be changed. Submit the ZIP file to the MyCourses assignment before the due date (if you submit another format, such as 7-Zip, WinRAR, or Tar, you will not receive credit for this lab).