Computational Problem Solving CSCI-603
PreTee Interpreter Lab 8

1 Introduction

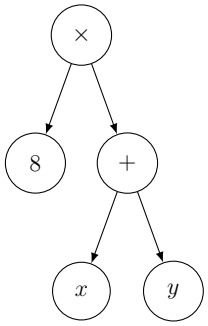
An *interpreter* is a program that executes instructions for a programming language. The `python3` interpreter executes Python programs. This assignment involves writing an interpreter for a very simple language called **PreTee**.

The interpreter accepts *prefix* mathematical expressions as input, where a prefix expression is one in which the mathematical operation is written at the beginning rather than in the middle. Inside the interpreter is a *parser*, whose job is to convert prefix expressions, represented as a string, into a collection of tree structures known as *parse trees*. The PreTee interpreter can evaluate mathematical expressions using the operators `+`, `-`, `*`, and `//`. The supported operands are positive integer literals (e.g., `8`) and variables (e.g., `'x'`). A data structure called a *symbol table* is used by the interpreter to associate a variable with its integer value. When given a prefix expression, PreTee displays the *infix* form of the expression and evaluates the result.

Consider the following example using the prefix expression: ``` ``* 8 + x y'' ```

Variable	Value
``x''	10
``y''	20
``z''	30

tr:



Infix expression: ``(8 * (x + y))''

Evaluation: 240

Let's assume that the parse tree has already been constructed from the prefix expression. The infix form of the expression can be obtained by doing an `inorder` (left, parent, right) traversal of the tree from the root and constructing a string. Recall the private `__inorder` helper function from lecture that `__str__` called when getting a string representation of a general purpose tree.

```
def __inorder(self, node):
    if not node:
        return ' '
    else:
        return self.__inorder(node.left) + \
               str(node.val) + \
               self.__inorder(node.right)
```

The result of the expression can be found by evaluating the root node in `preorder` fashion (parent, left, right).

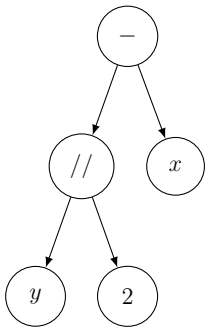
The implementation stage will cover details of the symbol table and evaluation of the parses tree to generate the result of the expression.

2 Problem Solving

Work in a team of three to four students as determined by the instructor. Each team will work together to complete a set of activities. Please complete the questions in order. Do not read ahead until you have finished the first two questions.

1. Consider the following tree.

   tr:



   Write the prefix expression for the tree, *tr*, above. Recall in a prefix traversal, the order is parent, left then right.

2. Write the infix expression for the tree, *tr*, from problem 1. Use parentheses to specify the order of operations.

3. The parse tree can be built by reading the individual tokens in the prefix expression and constructing the appropriate node types. Given this prefix expression as the token list:

   ```
   [‘‘*’’, ‘‘//’’, ‘‘-’’, ‘‘x’’, ‘‘10’’, ‘‘y’’, ‘‘+’’, ‘‘4’’, ‘‘x’’]
   ```

   (a) Draw the parse tree for the expression using the diagramming method above.
   (b) Write the infix expression for the parse tree. Make sure you include parentheses to preserve the operator precedence.
   (c) What would be the result of evaluating the parse tree? Assume the symbol table on the first page is in effect.

We will be representing the various types of nodes in the tree as Python classes.

Node class	Slot name(s)	Type(s)	Constructor
LiteralNode	val	int	__init__(self, val)
VariableNode	id	str	__init__(self, id)
MathNode	left right token	object (a Node class) object (a Node class) str ("+", "-", "*", or "//")	__init__(self, left, right, token)

For example, the code below constructs a parse tree for the expression tree in the first question:

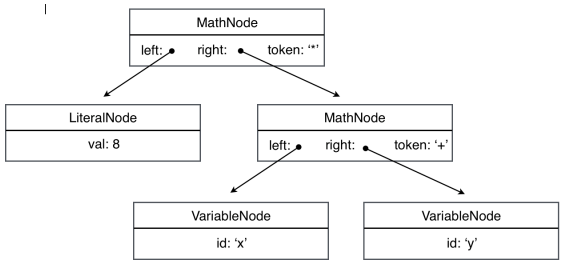
```
>>> tr = MathNode( \
              MathNode( \
                    VariableNode('y'), \
                    LiteralNode(2), \
                    '\\'), \
              VariableNode('x'), \
              '-')
```

Assume the prefix expression has been converted into a list of string tokens:

```
>>> prefixExp = '* 8 + x y'
>>> tokens = prefixExp.split()
>>> tokens
['*', '8', '+', 'x', 'y']
```

Now let's build and return a parse tree using the token list and node classes. The diagram below shows the node structure of the parse tree. For this problem, we will call that tree *tr*. The root of the tree *tr* is a MathNode. Each node is indicated by its type (one of the 3 possible node classes), and its internal slot values are also shown. Arrows indicate that a node has a child node. Here is a picture of the parse tree for the prefix expression, ``'* 8 + x y''``, which is stored in the token list as ``['*', '8', '+', 'x', 'y']``:

tr:



4. Write a function `parse`. It takes a list of tokens for a prefix expression and returns a parse tree. The parse tree for the token list `[‘‘*’’, ‘‘8’’, ‘‘+’’, ‘‘x’’, ‘‘y’’]` should produce the tree corresponding to the image above.

   Note that a prefix expression has one of the following forms:
   - A number that is non-negative in value (0 or greater).
   - A variable.
   - $+\ e_1\ e_2$, where e_1 and e_2 are prefix expressions.
   - $-\ e_1\ e_2$, where e_1 and e_2 are prefix expressions.
   - $*\ e_1\ e_2$, where e_1 and e_2 are prefix expressions.
   - $//\ e_1\ e_2$, where e_1 and e_2 are prefix expressions.

   The string function `isdigit` is useful for testing whether or not a string looks like a number. The string function `isidentifier` is useful for testing whether or not a string looks like a variable.

```
>>> str1 = '123'
>>> str1.isdigit()
True
>>> str2 = '3x'
>>> str2.isdigit()
False
>>> str3 = 'x'
>>> str3.isidentifier()
True
>>> str4 = '2x'
>>> str4.isidentifier()
False
```

3 Design

3.1 PreTee Language

The source code lines begin with one of four tokens:

- `#`: The line is a comment.
- `=`: The line is an assignment statement.
- `@`: The line is a print statement.
- ‘‘’’: A blank line (counted but ignored).

3.1.1 Comment

Any line that is a comment is ignored when parsed. It still counts as a line number. For example:

```
# this line is a comment
```

3.1.2 Assignment

An assignment statement is of the prefix form:

```
= {variable} {expression}
```

For example:

```
= x 10
= y 20
= z + x y
= x 40
```

When emitted, the variable is emitted, followed by the equals sign, followed by the emission of the expression that followed.

```
x = 10
y = 20
z = (x + y)
x = 40
```

When evaluated, the expression is evaluated and its result is stored in the symbol table for the variable:

```
# symbol table: {..., 'x': 10, ...}
# symbol table: {..., 'y': 20, ...}
# symbol table: {..., 'z': 30, ...}
# symbol table: {..., 'x': 40, ...}
```

A syntax error exception will be raised for the following:

1. Assignment to a non-variable node, e.g.:
   ```
   = 10 10     # error message: Bad assignment to non-variable
   ```
2. Bad expression for assignment, e.g.:
   ```
   = x @       # error message: Bad assignment expression
   ```

3.1.3 Print

A print statement is of the prefix form, where expression is optional:

```
@ {expression}
```

For example:

```
@                # prints a new line
@ 10
@ + 10 20'
@ x              # assuming x = 10
```

When emitted, the string ‘‘`print`’’ is emitted, followed by a space, and the emission of the expression that follows.

```
print
print 10
print (10 + 20)
print x
```

When evaluated, the expression is evaluated and its result, if present, is displayed:

```
                 # just a newline is printed
10
30
10               # the value of x is printed
```

3.2 Expressions

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

3.2.1 Literal

A literal expression is of the prefix form, where value is an integer:

```
{value}
```

For example:

```
10
4
```

When emitted, the expressions are displayed infix as strings:

```
10
4
```

When evaluated, their integer form is returns:

```
10
4
```

3.2.2 Variable

A variable expression is a legal identifier in Python:

```
{id}
```

For example:

```
x
y
variable
```

When emitted, the variable name is returned as a string, e.g.:

```
x
y
variable
```

When evaluated, the value associated with the variable name in the symbol table is returned. For example if the symbol table contains {..., ‘‘x’’: 10, ‘‘y’’ : 20, ‘‘z’’ : 30, ...}, the evaluations would be:

```
10
20
30
```

A runtime exception is raised if the variable is not in the symbol table, e.g.:

```
a             # error message: Unrecognized variable a
```

3.2.3 Math

A math expression is of the prefix form:

```
{operator} {left-expression} {right-expression}
```

For example:

```
+ 10 20
* 3 5
- 2 4
// 13 2
+ 2 * 8 7
```

When emitted, the statement is converted into an infix string:

```
(10 + 20)
(3 * 5)
(2 - 4)
(13 //2 )
(2 + (8 * 7))
```

When evaluated, integer result is returned:

```
30
15
-2
6
58
```

A runtime exception is raised division by 0 is attempted:

```
// 4 0            # error message: Division by zero error
```

3.3 Full Example

The full grammar can be seen in this example source file, `prog1.pre`.

```
# Create three variables
= x 10
= y 20
= z + x y

# Print some expressions
@
@ + 10 20
@ - 10 20
@ * 10 20
@ // 20 10
@ z
@ * x + y z
@ // * x + y 10 - y * 10 z
```

3.4 Sample Run

The interpreter is run by providing the source code file as a command line argument:

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

When run, the three stages of interpretation occur:

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.

Here is the full interpretation output for `prog1.pre`:

```
PRETEE: Compiling prog1.pre...

PRETEE: Infix source...
x = 10
y = 20
z = (x + y)
print
print (10 + 20)
print (10 - 20)
print (10 * 20)
print (20 // 10)
print z
print (x * (y + z))
print ((x * (y + 10)) // (y - (10 * z)))

PRETEE: Executing...

30
```

```
-10
200
2
30
500
-2
```

3.5 Errors

There are two types of errors that can occur when interpreting:

3.5.1 Syntax Errors

These occur when parsing the statements and a violation of the grammar is encountered. Under these circumstances, the interpreter will not generate a parse tree for this statement (and thus will not emit in infix form). *The interpreter must display the line number the syntax error occurs on.*

Syntax errors do not halt parsing. The parsing continues as normal with the next statement, and any further syntax errors encountered will also be displayed. However, if there are any syntax errors in the program, it will not execute.

Using the example above, these are the syntax errors you need to deal with:

```
=              # Incomplete statement
= 10 10        # Bad assignment to non-variable
= x @          # Bad assignment expression
^              # Invalid token ^
```

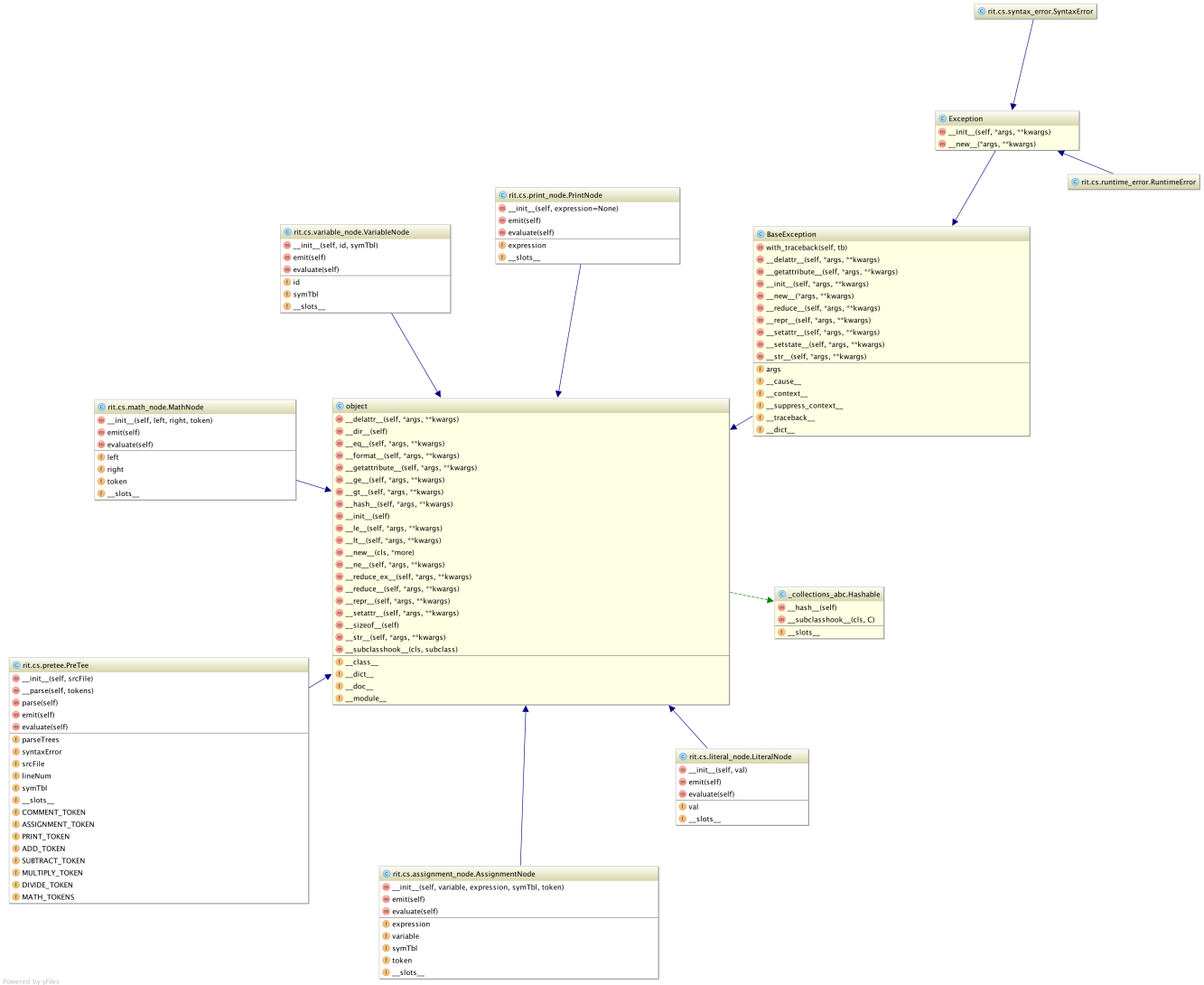
3.5.2 Runtime Errors

Runtime errors occur during execution of the parse trees. These can only happen if there are no syntax errors. When the first runtime error is encountered, the program should display the error and then halt execution of the program.

Using the example above, these are the runtime errors you need to deal with:

```
@ a            # Unrecognized variable a
@ // 5 0       # Division by zero error
```

3.6 PreTee UML



The full object oriented design is diagrammed above. It involves the following classes

- `PreTee`: The main interpreter class. It is responsible for parsing the PreTee source code, displaying it in infix, and then executing it.
- `AssignmentNode`: A class to represent the assignment node. It assigns the result of an expression to a variable.
- `LiteralNode`: A class to represent a literal node containing a positive integer.
- `MathNode`: A class to represent a mathematical node. It contains the operation, plus the left and right expressions.
- `PrintNode`: A class to to represent a print node. It displays the result of an expression to standard output.
- `VariableNode`: A class to represent a variable node. It stores the id of the variable and can retrieve its stored value from the symbol table.
- `SyntaxError`: An exception class used to indicate syntax errors when compiling the

code.

- `RuntimeError`: An exception class used to indicate runtime errors that occur during execution.

3.7 Node Design

The Node classes have different slots depending on their type. Please refer to the source code for more information.

All Node classes implement three methods:

- A constructor for initializing the slots.
- A `emit` function that takes nothing and returns an infix string for the node and its children
- An `evaluate` function that takes nothing and returns the integer result of evaluating this node and all of its children.

4 Implementation

4.1 Provided Files

- The `doc` folder contains the pydocs for all the classes in HTML.
- The `src` folder contains starter code.
  - `pretee.py`: The main program which contains the `PreTree` class. The main program is given to you completely and should not be changed. You will be implementing all the methods for the `PreTree` class.

    For `PreTee` (and all other classes), the class, slots, methods and docstring's are all given to you. You are not allowed to change any of the slots, or method names. You should write your code in the provided methods and not create any additional ones. You are free to use whatever local variables you want in the methods, but there should be no use of globals. For any files you contribute to, make sure your name/s are added as authors in the top docstring for the module.
  - `assignment_node.py`: This is a complete implementation of the `AssignmentNode` class. You should not alter this file at all. It is being given to you as a demonstration of how one node is implemented.
  - `literal_node.py`. Contains the `LiteralNode` class that you will provide implementation for the methods contained within.
  - `math_node.py`. Contains the `MathNode` class that you will provide implementation for the methods contained within.
  - `print_node.py`. Contains the `PrintNode` class that you will provide implementation for the methods contained within.
  - `variable_node.py`. Contains the `VariableNode` class that you will provide implementation for the methods contained within.
  - `runtime_error.py`. The definition of the `RuntimeError` exception class. You should not modify this file.

– `syntax_error.py`. The definition of the `SyntaxError` exception class. You should not modify this file.

4.1.1 Restrictions

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.

4.2 Suggested Approach

Do things in this order.

1. Write the constructor for `PreTee` so it initializes all the slots.
2. Work on the parsing, starting with `PreTee.parse` and the private recursive helper function `PreTee.__parse`. This is analogous to the work you did on the last question in problem solving. You should build this up incrementally and use the already implemented `AssignmentNode` to start with. First implement the constructor for `LiteralNode` and `VariableNode` 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 `LiteralNode` and `VariableNode`. Now implement `PreTee.emit` to call `emit` on the root parse node/s and verify their infix form.
4. Add in the `PrintNode` implementation of the constructor and `emit`. Try printing some literals and variables to see if they work correctly.
5. Implement the `MathNode` 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.

5 Grading

This assignment will be graded using the following rubric:

- (20%) Problem Solving
- (20%) Design: Following the prescribed design when implemented.
- (55%) Functionality: Correctly handles compiling, emitting and evaluating, along with error handling.
- (5%) Style and Documentation

6 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).