Computer Science I — CSCI-141
Derp the Interpreter — Lab 9

1 Introduction

11/04/2020

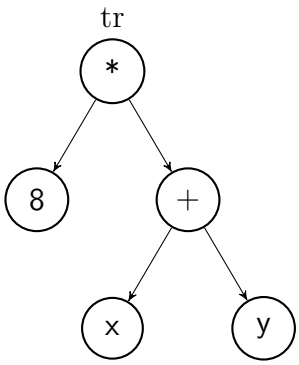
An *interpreter* is a program that executes instructions for a programming language. The `Python` interpreter executes Python programs. This assignment involves writing an interpreter for a very simple language called **Derp**.

The Derp interpreter accepts *prefix mathematical expressions* as input. In a prefix expression, the mathematical operator is written at the beginning rather than in the middle. Derp converts a prefix expression represented as a string into a tree structure called a *parse tree*. Only the operators `+`, `-`, `*`, `//` are allowed in the Derp language. The supported operands are 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, Derp 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



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

Evaluation: 240

Let's assume that the parse tree has already been constructed from the prefix expression. The prefix form of the expression can be produced by performing a `preorder` traversal (parent, left, right) of the tree from the root. Here is pseudocode for a pre-order traversal:

```
function preorder(node):
    if node is empty then return
    else:
        print node's value
        preorder(node's left child)
        preorder(node's right child)
```

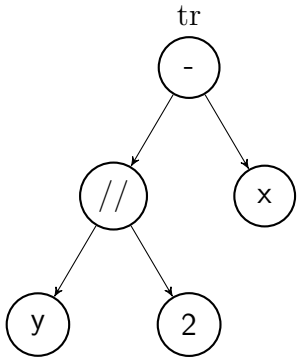
Likewise, the infix expression can be constructed by performing an `inorder` traversal (left, parent, right) of the tree from the root. This is similar to the `bst_to_string` function of a binary search tree problem.

2 Problem-solving

Work in a team determined by the instructor. Each team will complete these activities.

Please complete the questions in order.

1. Consider the following tree.



   Write the prefix expression for the tree, *tr*, above.

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. Assume for the moment that the following node classes exist.

Node class	Slot name(s)	Type(s)
LiteralNode	val	int
VariableNode	name	str
MathNode	left	object (reference to any node class object)
	operator	str (string representation of the operator)
	right	object (reference to any node class object)

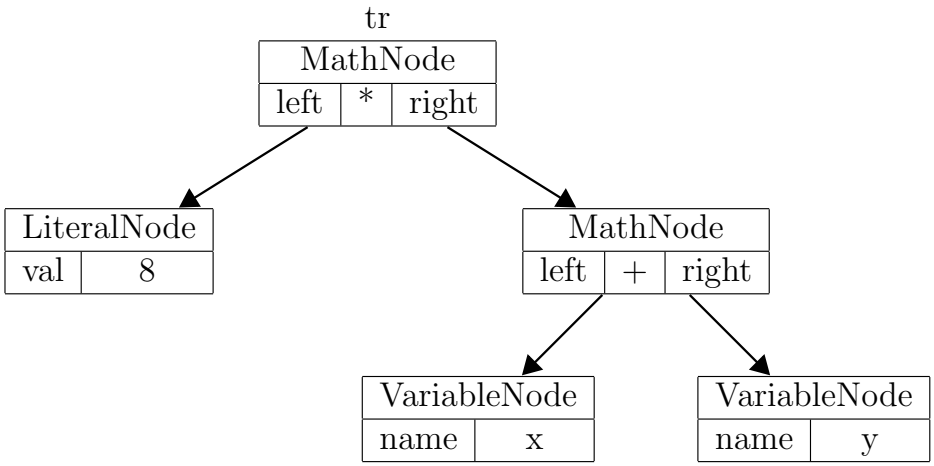
   For example, the code below constructs a parse tree for the above expression tree:

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

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

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

Given that list, you have to build and return a parse tree using the token list and node classes. The diagram below shows the node structure of the parse tree. Let this tree be called *tr*. The root of tree *tr* is a MathNode containing a multiplication operation. 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']`:



Given this prefix expression as the token list:

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

(a) Draw the parse tree for the expression. It is ok to use circles for the nodes.
(b) Write the infix expression for the tree you drew.

4. Write the function `parse`. It takes a list of tokens for the prefix expression and returns the root of the 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.

Note that this is a recursive procedure that consumes the list. Since a list is passed by reference, changes to the list will be seen when the function returns. For example, if `result = parse(lst)` changes `lst`, then the changes to that list will propagate to subsequent statements. The next subsection has some tools to help with this function.

2.1 Tools

When processing the input list, `pop(0)` is a handy way to remove and fetch the first element.

2.1.1 The `isdigit` function

The `isdigit` function tests whether or not a string contains all numeric digits.

```
>>> str1 = '123'
>>> str1.isdigit()
True
>>> str2 = '3x'
>>> str2.isdigit()
False
```

2.1.2 The `isidentifier` function

The function `isidentifier` tests whether or not a string looks like a variable name.

```
>>> str3 = 'x'
>>> str3.isidentifier()
True
>>> str4 = '2x'
>>> str4.isidentifier()
False
```

3 Post-PSS

1. Download supplied materials here:
   `http://www.cs.rit.edu/~csci141/Labs/09/Code.zip`
   Inside is the types file, `derp_types.py`, and the main program file, `derp.py`, which contains stubbed out and commented functions that you will need to implement. Study the supplied code. Do not remove the comments; place the code that relates to each comment below that comment.
2. Write a function that takes the name of a symbol table file, reads the file, and returns the symbol table. Study the example symbol table file called `vars.txt`. You will store the symbol table using a Python dictionary.
3. Using the `LiteralNode` and `MathNode` in the supplied code, **hard-code** a structure that represents this expression: `* 3 * 2 1`
4. Implement the `infix` and `evaluate` functions so that they process your hard-coded structure. Your program must implement these functions *exactly* because an automated test program will call them directly. The Required Functions section will guide you.

Zip `derp.py` and `derp_types.py` to `pss09.zip`, and upload that to the dropbox for this Post-PSS by the due date.

4 Implementation

4.1 Symbol Tables

For this assignment, the symbol table will be specified in a separate text file. Each line will contain one variable name, followed by a space, followed by its integer value. For example, here is a file named `vars.txt`:

```
x 10
y 20
z 30
```

4.2 Required Functions

You will write the following functions for this assignment.

Note that the functions' names, parameters, and behaviors must correspond exactly to what is specified below.

- The function `infix` takes a parse tree and returns a string that represents the fully parenthesized, infix form of the expression. Only `MathNode` expressions get parentheses.

```
>>> infix(tr)
'(8 * (x + y))'
```

- The function `evaluate` takes a tree and a symbol table as arguments. The table is a dictionary with entries: key type=string, value type=integer. The function traverses the tree to evaluate the expression, and returns the integer result for the expression. The traversal starts at the root node of the tree and asks for its value. This should return the value of the tree.

```
>>> symbol_table
{'y': 20, 'x': 10, 'z': 30}
>>> evaluate(tr, symbol_table)
240
```

- The `parse` function in `derp.py` takes a list of string *tokens* and constructs the tree.

```
>>> tr = parse(['*', '8', '+', 'x', 'y'])
>>> isinstance(tr, MathNode)
True
```

4.2.1 Additional Functions

You may choose to include additional functions as you see fit.

4.3 The `isinstance` function

Python has a built-in function, `isinstance(obj, type)`, that can assess whether or not the `obj` is of the type `type`.

```
>>> isinstance(3, int)
True
>>> isinstance(4.5, str)
False
>>> isinstance(tr, MathNode)
True
```

If `tr` is a reference to a node of a parse tree, this function can ascertain what node type of a reference.

4.4 Example Run Details

Derp processes an infix expression to construct a tree using the node classes, and then print the infix expression. Lastly, it evaluates the expression using the tree.

Initially, the interpreter (Derp) greets the user (Herp) and prompts for the symbol table file.

```
$ python3 derp.py
Hello Herp, welcome to Derp v1.0 :)
Herp, enter symbol table file: vars.txt
```

Next, the program reads the symbol table into a Python dictionary, and displays the variable names and values in the order of iteration through the dictionary.

```
Derping the symbol table (variable name => integer value)...
x => 10
y => 20
z => 30
```

Finally, the program enters a loop that runs until the user hits the ENTER key by itself. Each cycle of the loop prompts for a prefix expression, and then prints both the infix expression and the evaluated result.

```
Herp, enter prefix expressions, e.g.: + 10 20 (ENTER to quit)...
derp> + 10 * x y
Derping the infix expression: (10 + (x * y))
Derping the evaluation: 210
derp> // * x 5 + - 10 y z
Derping the infix expression: ((x * 5) // ((10 - y) + z))
Derping the evaluation: 2
derp>
Goodbye Herp :(
```

4.5 Constraints and Prohibitions

You cannot use Python's `eval` function to directly evaluate expressions stored as infix strings.

4.6 Guarantees

The following guarantees exist for this assignment:

- The symbol table will exist and follow the described format.
- The user will enter valid prefix expressions in the correct format.
- The user will not enter expressions that cause a divide by zero.
- Any variables used in the prefix expressions will exist in the symbol table file.

4.7 Testing

A test program will execute your solution, and **that program expects the exact function names, arguments and argument order.**

If you do not follow the directions exactly, your submission will fail to pass the tests of the test program, and you will lose significant points.

4.8 Grading

- 15%: Problem Solving
- 10%: Post-PSS
- 75%: Implementation
  - 5%: symbol table printout
  - 25%: `parse` function
  - 20%: `infix` function
  - 25%: `evaluate` function
- up to 10% off: Naming, Documentation and Style: The files and functions must have the specified names, and the code must follow course guidelines with author, docstrings, etc. This includes the name and type of zip file and the Python file name.
- up to 20% off: the program fails to pass tests because the program **does not follow the exact function name, arguments and argument ordering requirements.**

4.9 Submission

Zip your source files into a file called **lab09.zip**. Submit **lab09.zip** to the myCourses dropbox. Failure to zip or the use of any other file format, including rar, 7zip, etc, will result in a 10% penalty.

Be sure to put the file in the correct dropbox. Graders/SLIs will not go to other dropboxes searching for submissions.