Polish notation project

by Lugojan Emanuel

The steps I followed in order to achieve this project:

* I created the class Node(), that has 5 attributes, but each of them is kep in a dictionary (the attribute self.\_\_content), and so the keys are:
  + value – which holds the number or the symbol (15, or “-“)
  + role – keeps track if the value is a symbol or a number
  + right – the right chilld
  + left – the left child
  + father – the father Node
* The I created the getter and setter, and I used a dictionary for my class, so I could create only a single setter or getter, which accept as parameters, a key and a value, meaning I want to change the current content I have at key, with value
* Once the class was ready, I created the .txt file
* I separated each symbol/number by a space
* Then, I created a function read\_expression() that splits the input by space, and retruns a list of each symbol/number
* The next step was to create the tree
  + Firstly, an observation: Only the symbols/operators can have childs, the numbers are leafs in the tree
  + Using this information, I made a recursive function inside create\_tree(), called generating\_tree(father: Node, l: list)
  + Why I want to have the father? Because the call of the function Is a step ahead, at first, the first created node in the function would be the child of the root, so I call the function with father, at first father being the root
  + Why I used a list ? To know what symbol/number need to be inserted in the tree, I always insert the l[0] element, then I pop the element at index 0
  + The logic behind it:
    - If list gets empty, stop
    - I’m starting the function from root, and I pop root before entering the function
    - If l[0] is:
      * A symbol -> then I should go recursively in the free child, first I try if the left child is free, then if the right child is free. Whener the if is true, I call the function recursively in the new child, after popping it from the list
      * A number -> because the number is a leaf, we can’t go in the left or right child, so when this happens, instead of going recursively in the new node, I go recursively in father after I pop the first index
* The last step would be to evaluate the expression:
  + For this I used divide and conquer, at each given time the problem gets cut in half
  + I start in root, and the root will be the symbol, so I give the function calculator() the role to calculate the value at each step, so it will return the result.
  + Because of this, calculator(root), if root was “-“, then calculator(root) = calculator(left) – calculator(right)
  + To make the clode cleaner, I made an additional function operation(left, right, symbol, fn), where fn Is the function used, but it will always be calculator()
  + So now the calculator() only has like 10 lines of code, the calculating part takes place in operation()
* For visual representation of the code I used the module “graphviz” which allows you to create nodes, make edges between them, and then creates a pdf after the code runs. Watch the pdf examples for reference. After running the code, your default browser should be open with the pdf.
* The most difficult part for me was writing the function generating\_tree() I had like 3 different implementations, but they had to much ifs, so then I tried making it more dinamic, with more smaller functions and I cam up with the function I have right now, and the code looks much better and I think is easier to understand. The thing with recursivenes, some cases were not generated correctly because a younger recursive call would get in some if’s that it shouldn’t have and using while fixed the problem, and inside asking whether nou is a symbol or a number. At first I had 2 if’s instead of the while to check whether the value is a symbol or a number, but that would become problematic, cause in some calls, l[0] would change and I would get in an incorrect if, but that is not a problem anymore, using while and then asking each time if I have a symbol or number.