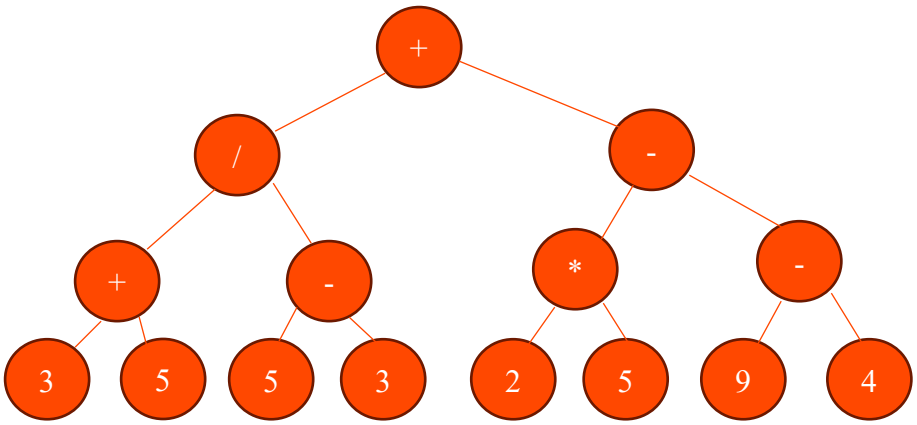
Mälardalens universitet

DVA245, Laboration 4 - Aritmetiskt uttryckträd

Mål
1- kunna använda abstrakta datatyper i programmeringsuppgifter
2 - kunna definiera och implementera abstrakta datatyper utifrån informella problembeskrivningar
3- kunna implementera algoritmer utifrån beskrivningar i pseudokod

Ni ska implementera ett aritmetiskt uttryckträd, dvs ett träd som kan representera ett uttryck. Exempelvis:
(((3+5)/(5-3))+((2*5)-(9-4)))
representas i trädform:



I er implementation ska ni från roten kunna:

- Returnera en textsträng som representerar uttrycket som ovan: ”(((3+5)/(5-3))+((2*5)-(9-4)))”.
- Returnera värdet när beräkningen utförts

Ni ska också i steg 2 skapa ett träd från ett uttryck i form av en textsträng. En hjälpfunktion för detta steg finns i filen `expression_tree.py`. Ni får lägga all implementation i den filen om ni vill, eller dela upp i flera.

Steg 1 – Noder

Ni behöver skapa en eller flera klasser för noderna (delträden).

Noderna/ delträden i uttryckträdet är antingen löv som representerar tal/ numeriska värden, eller inre noder som representerar operatorer (+, -, / eller *) och kopplar ihop två delträd (som kan vara löv/ tal eller operatorer).

Ni kan välja att skapa olika klasser för lövnoder och operatornoder, eller i medlemsfunktionerna hantera fallen olika. Ni kan också välja att skapa olika klasser för operatorerna eller hantera de olika operatorerna med villkor i funktionen som gör beräkningen.

- Vilken/ vilka klasser ska ni ha? Vad är bra namn?
- Hur ser konstruktorn/ konstruktorerna ut?
- Vilka medlemsvariabler behövs?

Klassen/ klasserna ska ha två medlemsfunktioner, en för att returnera textsträngen som representerar uttrycket för delträdet där noden är rot, och en för att beräkna värdet av uttrycket för detta delträd.

- Vad är bra namn för medlemsfunktionerna?

Om du har flera klasser, använd samma namn för funktionerna. Då kan en operatornod bygga sin sträng eller beräkna värdet av uttrycket genom att anropa barnens funktioner oavsett om de är andra operatornoder eller numeriska noder (tal/ löv).

Tips för att förenkla för er: Testa i en main-funktion eller i en separat fil att skapa numeriska noder (tal/ löv) och testa deras funktioner. Koppla sedan ihop två numeriska noder med en operatornod och testa, och bygg successivt ett större träd – till exempel det i illustrationen ovanför.

Steg 2 – Bygg uttryckskräd från en textsträng

I detta steg ska ni bygga ett uttryckskräd från en textsträng. Till er hjälp har ni en funktion `tokenize` som tar en textsträng, och delar upp innehållet i kortare textsträngar i en lista, Till exempel blir strängen '(43-(3*10))' listan ['(', '43', '-', '(', '3', '*', '10', ')', ')']. Funktionen sätter operatorer och parenteser i separata listelement, men samlar ihop tecken som hör till ett tal till samma listelement.

Pseudokod för en funktion som bygger ett uttryckskräd från en lista med strängar hittar ni på nästa sida. Funktionen använder en stack för att bygga delträd och sätta ihop dem i rätt ordning.

Funktionen går igenom listans element i tur och ordning. Om ett element i listan innehåller en operatorsymbol så läggs denna på stacken. Om elementet innehåller ett tal så skapas en lövnod/ numerisk nod och läggs på stacken. Om elementet innehåller en högerparentes så tas tre element ut från stacken – de kommer att vara två delträd och en operatorsymbol. En operatornod skapas som kopplar ihop delträden och den nya noden läggs på stacken. När man gått igenom hela listan innehåller stacken uttryckskrädet.

I pseudokoden finns några rader som ni kanske undrar över:

- IS_OPERATOR får ni implementera som en funktion eller direkt som ett villkor (ni kan titta på `tokenize`-funktionen och SYMBOLS och göra liknande).
- IS_NUMERIC: pythons strängklass str har en isnumeric-funktion som ni kan använda er av: https://docs.python.org/3/library/stdtypes.html?highlight=isnumeric#str.isnumeric
- CREATE_NODE-anropen innebär att ni skapar objekt av er klass/ era klasser och det ser olika ut beroende på hur ni gjort i steg 1.
- Ni får använda en python-lista som stack, eller en annan stack-implementation.

Implementera `build_expression_tree` och lägg till tester för den med olika strängar och/ eller möjligheten att mata in en uttrycksträng interaktivt. Testa att trädet ger rätt uttrycksträng och ger rätt resultat.

```
function BUILD_EXPRESSION_TREE(token_list)
         returns resulting expression tree
input:   token_list – list of strings with numeric
values, parentheses, and operator symbols
local variables:  token_tree_stack – stack for
tokens and subtrees during creation
                  token – current token to check
                  node – created tree node
                  left – subtree node from stack
                  right – subtree node from stack
                  op – operator token from stack


token_tree_stack←the empty stack
for each token in token_list do
         if IS_OPERATOR(token) then
                  # push operator symbol
                  token_tree_stack.PUSH(token)
         else if IS_NUMERIC(token) then
                  # push a leaf node

                  node←CREATE_NODE(token)
                  token_tree_stack.PUSH(node)
         else if token = ')' then

                  right← token_tree_stack.POP()

                  op← token_tree_stack.POP()

                  left← token_tree_stack.POP()

                  node←CREATE_NODE(op,left,right)
                  token_tree_stack.PUSH(node)

         # left parentheses are ignored
# resulting expression tree is the last on stack
return token_tree_stack.POP()
```

För redovisning:
Ni måste kunna förklara hur er klass/ era klasser fungerar, och hur ni testar det.
Ni måste också kunna förklara hur ni testar `build_expression_tree`.

English:

Learning objectives
1- be able to use abstract data types in programming assignments
2 - be able to define and implement abstract data types when given informal problem statements
3- be able to implement algorithms given descriptions as pseudocode

You shall implement an arithmetic expression tree, (see example and figure)

In your implementation you shall be able to from the root:

- Return a text string that represents the expression like: ”(((3+5)/(5-3))+((2*5)-(9-4)))”.
- Return the value of the evaluated expression

You shall also in step 2 create a tree from an expression in the form of a text string. A helper function for this step is in the file `expression_tree.py`. You can put all implementation in this file or split it in several files.

Step 1 – Nodes

You need to create one or more classes for the nodes (subtrees).

The nodes/ subtrees in the expression tree are leaves that represent numerical values or inner nodes that represent operators (+, -, / eller *) and connect two subtrees (that are leaves/ numbers or operators).

You can create different classes for leaf nodes and operator nodes or handle the cases differently in member functions. You can also create different classes for the operators or handle the different operators with conditions in the evaluating function.

- What class/ classes will you have? What are good names?
- How does the constructor/ constructors look like?
- What member variables are needed?

The class/ classes shall have two member functions, one to return the text string representing the expression of the subtree where the node is root, and one to calculate the value of the expression for this subtree.

- What are good names for the member functions?

If you have several classes, use the same names for the functions. Then an operator node can build the string or calculate the value of the expression by calling the children’s functions no matter if they are other operator nodes or numerical nodes (numbers/ leaves).

Tip to make it easier for you: Test in a main-function or in a separate file to create numerical nodes (numbers/ leaves) and test their functions. Connect two

numerical nodes with an operator node and test, and build a larger tree successively – for example the tree illustrated above.

Step 2 – Build an expression tree from a text string

In this step you shall build an expression tree from a text string. For your help you have a function `tokenize` that takes a text string and separates it into shorter text strings in a list, For example the string '(43-(3*10))' becomes the list ['(', '43', '-', '(', '3', '*', '10', ')', ')']. The function puts operators and parentheses in separate list elements but collects characters that belong to a number into the same list element.

Above you find pseudocode for a function that builds an expression tree from a list of strings. The function uses a stack to build subtrees and connect them in the right order.

The function goes through the list elements one by one. If an element in the list contains an operator symbol, it is pushed on the stack. If the element contains a number a leaf node/ numerical node is created and pushed on the stack. If the element contains a right parenthesis three elements are popped from the stack – they will be two subtrees and an operator symbol. An operator node is created that connects the subtrees and the new node is pushed on the stack. When the entire list is processed the stack contains the expression tree.

In the pseudo code there are some lines that you perhaps wonder about:

- IS_OPERATOR you can implement as a function or directly as a condition (you can look at the `tokenize` function and SYMBOLS and do something similar).
- IS_NUMERIC: python's strring class str has an isnumeric-function that you can use: https://docs.python.org/3/library/stdtypes.html?highlight=isnumeric#str.isnumeric
- CREATE_NODE- the calls mean that you create objects of your class/ classes and this will look different depending on what you did in step 1.
- You can use a python-list as a stack, or another stack implementation.

Implemente `build_expression_tree` and add tests for it with different strings and/ or the possibility to enter an expression string interactively. Test that the tree gives the right expression string and the right result.

For presentation:
You need to be able to explain how your class/ classes work, and how you test. You also need to be able to explain how you test `build_expression_tree`.