Lab 10

COMP9021, Session 2, 2016

1 ☞ Building a general tree

Consider a file named `tree.txt` containing numbers organised as a tree, a number at a depth of N in the tree being preceded with N tabs in the file. The file can also contain any number of lines with nothing but blank lines. Using the module `general_tree.py`, write a program `read_and_build_tree.py` that reads the contents of the file. If the file does not contain a proper representation of a tree then the program outputs an error message; otherwise, it builds the tree (an instance of `GeneralTree()`) and prints it out using the same representation as in the file (except for the possible blank lines of course).

```
$ python3
...
$ cat tree.txt

2
        3
                1
        4





                5
                        7
                                8
                9

                        10
                        11
                        12





        6
1
$ python3 read_and_build_tree.py
tree.txt does not contain the correct representation of a tree.
```

```
$ cat tree.txt

2
        3
                1
        4





                5
                                7
                                8
                9

        6
$ python3 read_and_build_tree.py
tree.txt does not contain the correct representation of a tree.
$ cat tree.txt

2
        3
                1
        4





                5
                        7
                                8
                9

                        10
                        11
                        12



        6
$ python3 read_and_build_tree.py
2
        3
                1
        4
                5
                        7
                                8
                9
                        10
                        11
                        12
        6
$
```

2 ☞ Back to fully parenthesised expressions

Modify the program `fully_parenthesised.py` from Lab 9, that deals with arithmetic expressions written in infix, fully parenthesised, and built from natural numbers using the binary +, -, * and / operators, still using a stack but to build an expression tree rather than to evaluate the expression (that is, representing an expression of the form `(first_argument operator second_argument)` as a tree whose value is `operator`, and whose left and right nodes are the subtrees that represent `first_argument` and `second_argument`, respectively. The function `evaluate()` is then reimplemented so as to recursively evaluate the expression from the tree.

Next is a possible interaction.

```
$ python3
...
>>> from fully_parenthesised import *
>>> parse_tree('100').print_binary_tree()
100
>>> parse_tree('[(1 - 20) + 300]').print_binary_tree()
            1
      -
            20
+

      300

>>> parse_tree('( 1 - [ 20 + 300] )').print_binary_tree()

      1

-
            20
      +
            300
>>> parse_tree('20*4/5').print_binary_tree()
            20
      *
            4
/

      5

>>> parse_tree('[ 20 * (4 / 5) ]').print_binary_tree()

      20

*
            4
      /
            5
```

```
>>> parse_tree('(1 + 20 * (30 - 400))').print_binary_tree()



      1




+

            20

      *
                  30
            -
                  400
>>> parse_tree('(1 + 20)*(30 - 400)').print_binary_tree()
            1
      +
            20
*
            30
      -
            400
```

3 Back to context free grammars (optional)

Modify the program `context_free_grammar.py` from Lab 9, that deals with a context free grammar for a set of arithmetic expressions, so that rather than implementing the function `evaluate()`, a function `parse_tree()` is implemented to build an object of type `BinaryTree` to represent the parse tree of the expression.

Next is a possible interaction.

```
$ python3
...
>>> from context_free_grammar import *
>>> parse_tree('100').print_binary_tree()
100
>>> parse_tree('1 - 20 + 300').print_binary_tree()
            1
      -
            20
+

      300

>>> parse_tree('1 - (20 + 300)').print_binary_tree()

      1


-
            20
      +
            300
>>> parse_tree('20 * 4 / 5').print_binary_tree()
            20
      *
            4
/

      5

>>> parse_tree('20 * (4 / 5)').print_binary_tree()

      20


*
            4
      /
            5
```

```
>>> parse_tree('1 + 20 * (30 - 400)').print_binary_tree()
    1
+
        20
    *
            30
        -
            400
>>> parse_tree('(1 + 20) * (30 - 400)').print_binary_tree()
        1
    +
        20
*
        30
    -
        400
```

4 Possible subtractions yielding a given sum (optional)

Write a program `subtractions.py` that takes as input an iterable `L` of nonnegative integers and an integer `N`, and displays all ways of inserting negations and parentheses in `L`, resulting in an expression that evaluates to `N`. For this question we make use of `eval()`.

Next is a possible interaction.

```
$ python3
...
>>> from subtractions import *
>>> subtractions((1, 2, 3, 4, 5), 1)
1 - ((2 - 3) - (4 - 5))
(1 - ((2 - 3) - 4)) - 5
>>> subtractions((1, 2, 3, 4, 5), 2)
>>> subtractions((1, 2, 3, 4, 5), 3)
1 - (2 - (3 - (4 - 5)))
1 - ((2 - (3 - 4)) - 5)
(1 - (2 - 3)) - (4 - 5)
>>> subtractions((1, 2, 3, 4, 5), 4)
>>> subtractions((1, 2, 3, 4, 5), 5)
(1 - 2) - ((3 - 4) - 5)
>>> subtractions((1, 3, 2, 5, 11, 9, 10, 8, 4, 7, 6), 40)
1 - ((((3 - 2) - 5) - 11) - (9 - ((((10 - 8) - 4) - 7) - 6)))
1 - ((((((3 - 2) - 5) - 11) - 9) - 10) - (8 - (4 - (7 - 6))))
1 - ((((((3 - 2) - 5) - 11) - 9) - 10) - ((8 - (4 - 7)) - 6))
1 - (((((((3 - 2) - 5) - 11) - 9) - 10) - (8 - 4)) - (7 - 6))
1 - (((((3 - 2) - 5) - 11) - (9 - (((10 - 8) - 4) - 7))) - 6)
1 - ((((((3 - 2) - 5) - 11) - (9 - ((10 - 8) - 4))) - 7) - 6)
1 - (((((((3 - 2) - 5) - 11) - (9 - (10 - 8))) - 4) - 7) - 6)
1 - ((((((((3 - 2) - 5) - 11) - (9 - 10)) - 8) - 4) - 7) - 6)
(1 - 3) - ((((2 - 5) - 11) - 9) - (10 - (((8 - 4) - 7) - 6)))
(1 - 3) - (((((2 - 5) - 11) - 9) - (10 - ((8 - 4) - 7))) - 6)
(1 - 3) - ((((((2 - 5) - 11) - 9) - (10 - (8 - 4))) - 7) - 6)
(1 - 3) - (((((((2 - 5) - 11) - 9) - (10 - 8)) - 4) - 7) - 6)
(1 - ((((3 - 2) - 5) - 11) - 9)) - ((((10 - 8) - 4) - 7) - 6)
((1 - 3) - ((((2 - 5) - 11) - 9) - 10)) - (((8 - 4) - 7) - 6)
(1 - ((((((3 - 2) - 5) - 11) - 9) - 10) - 8)) - (4 - (7 - 6))
(1 - ((((((3 - 2) - 5) - 11) - 9) - 10) - (8 - (4 - 7)))) - 6
(1 - (((((((3 - 2) - 5) - 11) - 9) - 10) - (8 - 4)) - 7)) - 6
((1 - ((((((3 - 2) - 5) - 11) - 9) - 10) - 8)) - (4 - 7)) - 6
```