Recursion

# Problem 1. Srinian Exercise

# Problem 2. Matching Parenthesis.

Parenthesis operate according to a very simple rule: if you open a parenthesis, you have to close it. For this problem, we are going to write a function that will check if a given expression is properly parenthesized.

To make the problem more interesting, we are going to have two kinds of parenthesis: round () and square [].

The function should return true if a string is correctly parenthesized and false if it is not. The function should ignore any character that is not one of the three kinds of parenthesis.

For example, the following expression is correctly parenthesized:  
 '[ x (y + z ) w(a b c)][t + x[z]]'  
, but this one is not:   
 '(x + y [z]'

# Structuring a recursive solution

Our solution will be in terms of three recursive functions outlined below:

### checkParenthesis(word):

The function checks if a string is properly parenthesized. The function returns a boolean True or false depending on whether the word is properly parenthesized or not. The function is defined recursively as follows:

* Base case: Words of length zero are always properly parenthesized.
* Inductive case 1: If the first letter in a word is not a parenthesis or a bracket, the word is properly parenthesized if the rest of the word is properly parenthesized.
* Inductive cases 2 and 3: If the first letter is a '[' or '(', call completeRound or completeSquare to check until the matching ']' or ')' respectively. If they failed to find a match, return False. Otherwise, check the rest of the word.

### def completeRound(word):

The function assumes that word contains the text immediately after an opening parenthesis. It will check parenthesization between the beginning of word and the matching closing parenthesis. The function returns a pair **(b, rest)**, where **b** is a boolean describing whether it encountered an error, and **rest** is the rest of the word after the matching closing parenthesis.

* Base case 1: If the word is of length zero, it means there was no text after the opening parenthesis. This is an error, so the function should return (False, something). It doesn't matter what something is, because it should be ignored.
* Base case 2: If the first character is a closing round parenthesis ')', we have found a match, so we can return (True, rest), where rest is the rest of the word after the closing parenthesis.
* Base case 3: If the first character is a closing square bracket this is an error, because we were looking for a closing \*round\* parenthesis. We should then return (False, something)
* Inductive case 1: If the first letter is not a parenthesis or a bracket, it can be ignored. The return values should be the same as if the function had been called with word[1:] instead of with word.
* Inductive case 2: Can you figure out the details of what happens when the first character is an opening parenthesis '('? Hint: we need to find a match for this new open parenthesis before we can continue looking for a match for the current open parenthesis in the rest of the word.
* Inductive case 3: Can you figure out the details of what happens when the first character is an opening square bracket '['? Hint: we need to find a match for this new open square bracket before we can continue looking for a match for the current open parenthesis in the rest of the word.

### def completeSquare(word):

The function is very similar to completeRound, but now we assume the word contains the text immediately after an opening square bracket, so now we are looking for a closing square bracket.

# Testing your solution

The code we have provided you includes a tester function appropriately called testerFunction. The function currently contains two tests, one positive and one negative. You need to provide additional tests to ensure that your code is working correctly. We will be grading your tester by running it against functions with subtle bugs and making sure that it is able to catch all of them. In particular, you need to make sure your tests cover all the corner cases inherent in the recursive definitions of the functions.

# Problem 3. Lisp calculator

For this last exercise, you are going to write a simple calculator. The input language for the calculator has two kinds of expressions:

Integer constants: (n) for simplicity you can assume that n is a one digit number.

Binary expressions: (op e1 e2) , where op can be + or \*.

For example, the following are legal expressions:

(+ (5) (3)) which should evaluate to 5 + 3 = 8

(\* (+ (2) (1)) (\* (5) (4)) ) which should evaluate to (2+1) \* (5\*4) = 60

Your goal is to write a program that given a string with an expression, returns the value of that expression. If the input is an invalid expression, the program should return zero.

TODO: Need to write descriptions of all the functions involved.

# Testing your solution

The code we have provided you includes a tester function appropriately called testerFunction. The function currently contains a small number of tests, one positive and one negative. You need to provide additional tests to ensure that your code is working correctly. We will be grading your tester by running it against functions with subtle bugs and making sure that it is able to catch all of them. In particular, you need to make sure your tests cover all the corner cases inherent in the recursive definitions of the functions.