Software Testing project

Team members :

- Aditya Sheth (IMT2018003)
- Dev Patel (IMT2018021)
- Prateek Kamboj (IMT2018057)

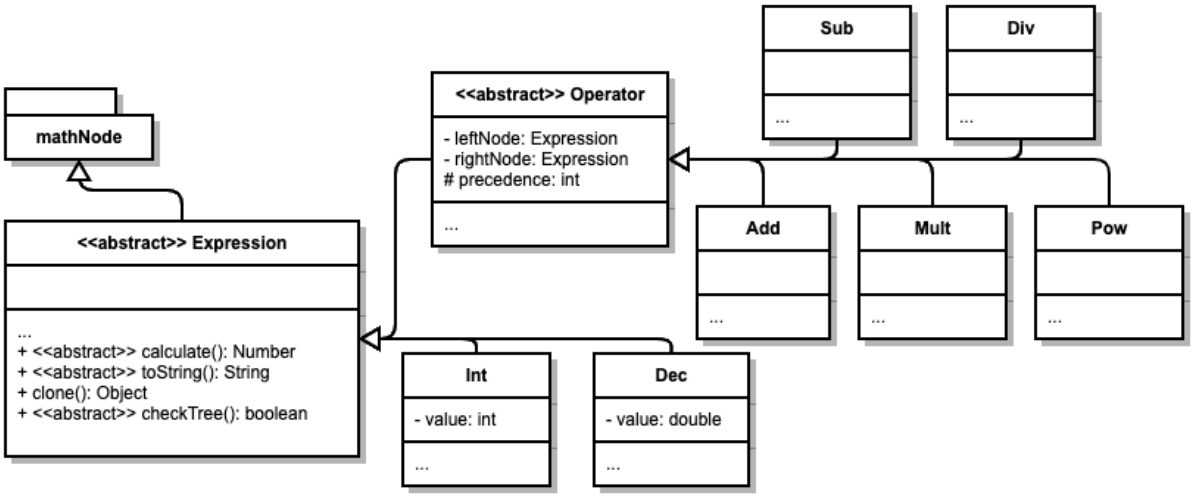
Method used : Control flow graph

Introduction :

A control flow graph is the graphical representation of control flow or computation during the execution of programs or applications. Control flow graphs are mostly used in static analysis as well as compiler applications, as they can accurately represent the flow inside a program unit.

About the code :

We tested the code on a simple math expression parser built in java. There is a mathTree class that takes an input string and builds a parsing tree. If it fails because the input string is an invalid math statement, it deletes the tree and returns false.
There is a string scanner class that breaks the input string into tokens which can be later used to build the tree.
The expression tree is built is based on math precedence rules.
Expressions are solved from left to right unless the operation on right has higher precedence.



Tools and technologies used :

- **Programming language :** Java
- **IDE used :** IntelliJ IDE
- **Testing tool used :** JUnit 5.0
- **Control flow graph construction tool :** https://code2flow.com/
- **Test path and graph coverage tool :** https://cs.gmu.edu:8443/offutt/coverage/GraphCoverage

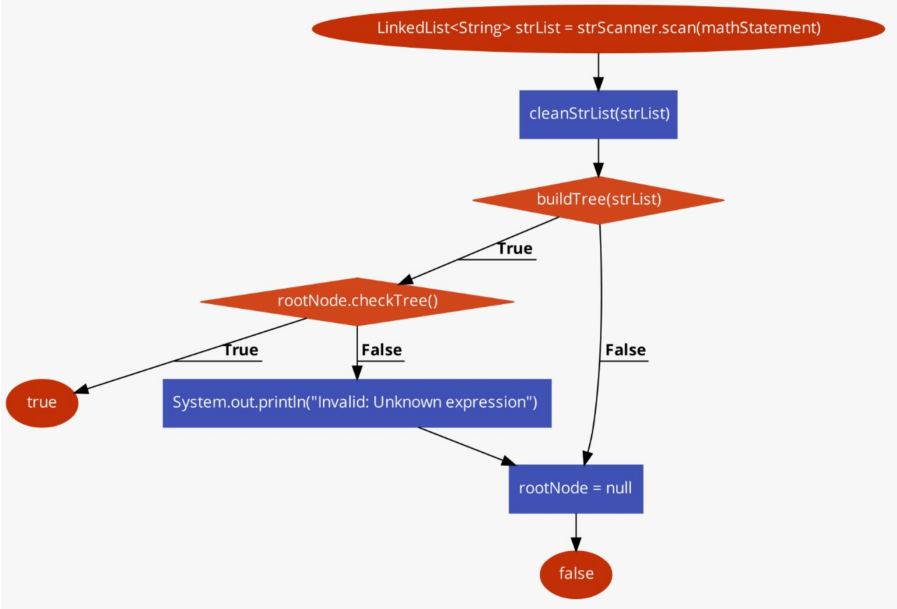
Instruction to run the code :

1. Download the github repository from https://github.com/adityasheth2000/simple-math-parser .
2. Use IntelliJ ide for opening the code repository.
3. Add Junit 5.0 to classpath using intelliJ ide suggestions.
4. Now, we can run all the tests written in the folder : /src/test/

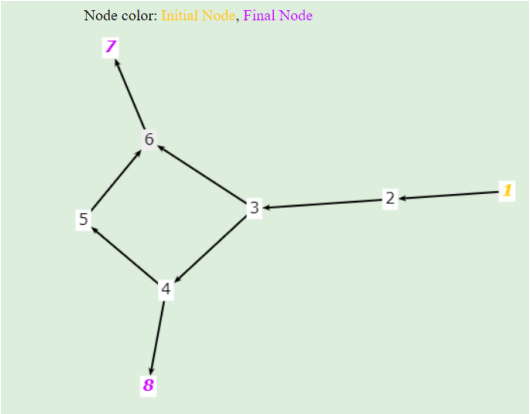
Tested functions:

Class : *MathTree*, Function : *init*

Control flow graph :



Numbered graph form :

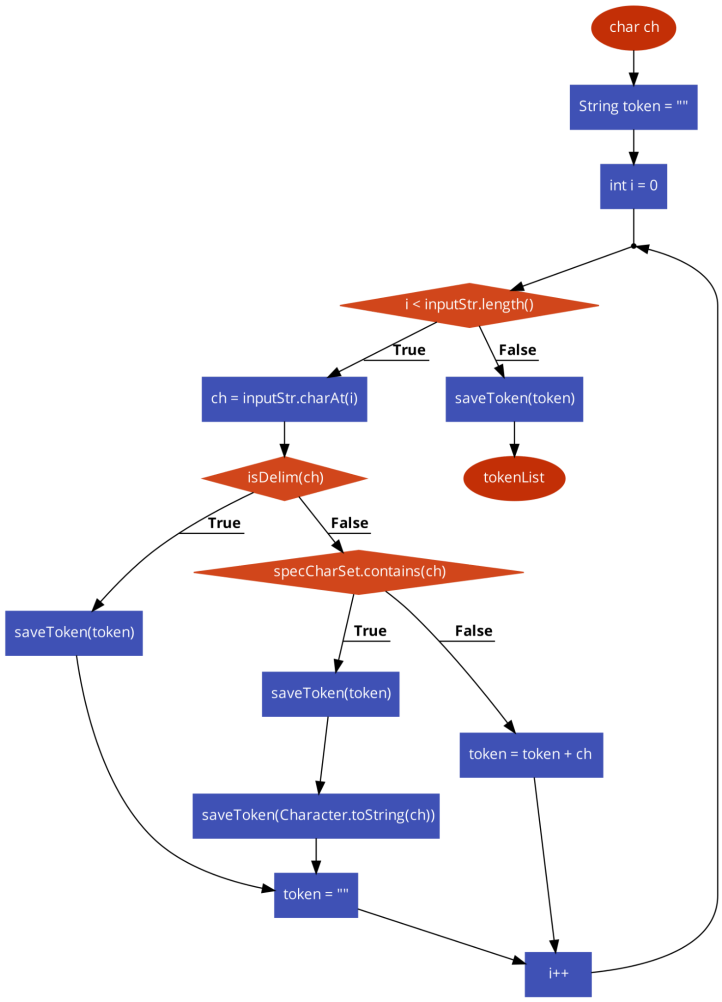


Test paths and corresponding test input:
[1,2,3,4,8] -> "4+4"
[1,2,3,6,7] -> "4 + "
[1,2,3,4,5,6,7] -> ""

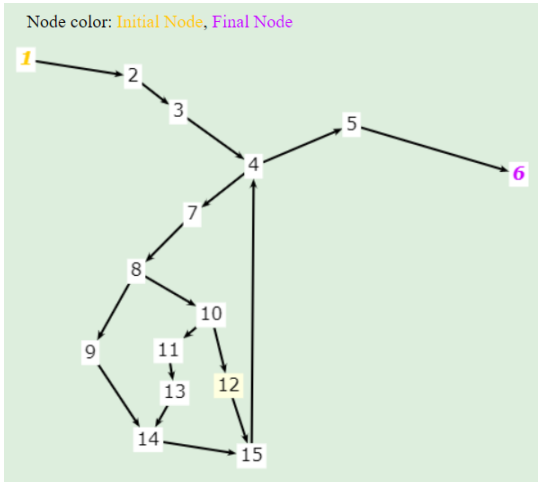
Note : Other test paths were infeasible, so we don't consider them here.

Class : *StringScanner*, Function : *scan*

Control flow graph:



Numbered graph form :



Test paths and corresponding test input:

special character set : {'.', ',', '?', ';', '(', ')'}

[1,2,3,4,7,8,9,14,15,4,7,8,10,11,13,14,15,4,7,8,10,12,15,4,5,6]
input string : " .a"

[1,2,3,4,7,8,10,12,15,4,7,8,10,11,13,14,15,4,7,8,10,11,13,14,15,4,7,8,9,14,
15,4,7,8,9,14,15,4,7,8,10,11,13,14,15,4,7,8,10,11,13,14,15,4,5,6]
input string : "a.. .."

[1,2,3,4,5,6]
input string : ""

[1,2,3,4,7,8,10,12,15,4,7,8,10,12,15,4,5,6]
input string : "aa"

[1,2,3,4,7,8,9,14,15,4,5,6]
input string : " "

[1,2,3,4,7,8,9,14,15,4,7,8,10,12,15,4,5,6]
input string : " a"

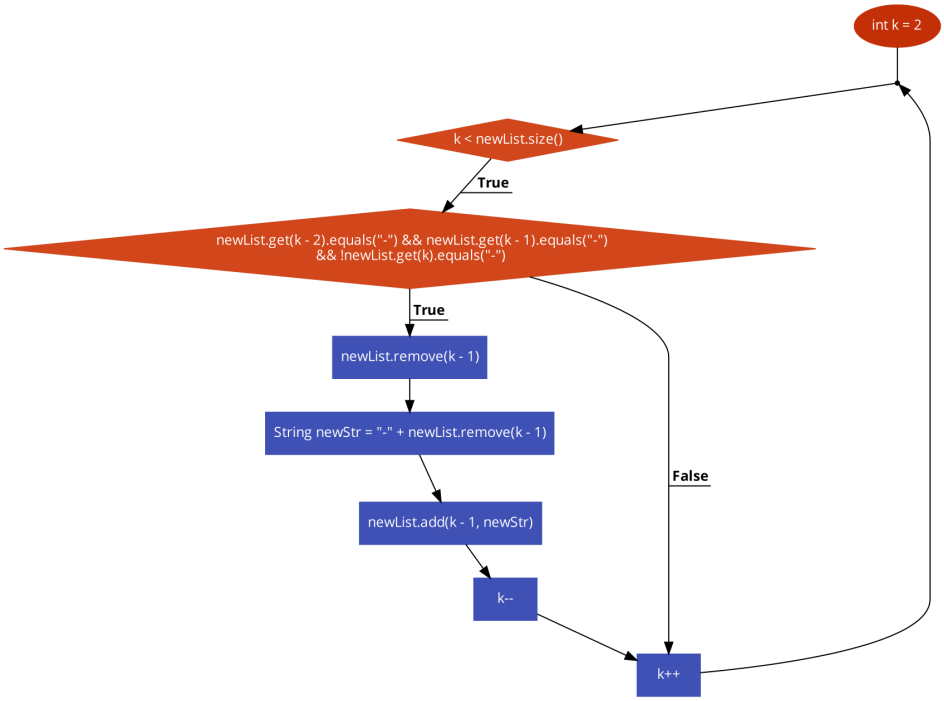
[1,2,3,4,7,8,10,12,15,4,7,8,9,14,15,4,5,6]
input string : "a "

[1,2,3,4,7,8,10,11,13,14,15,4,5,6]
input string : "."

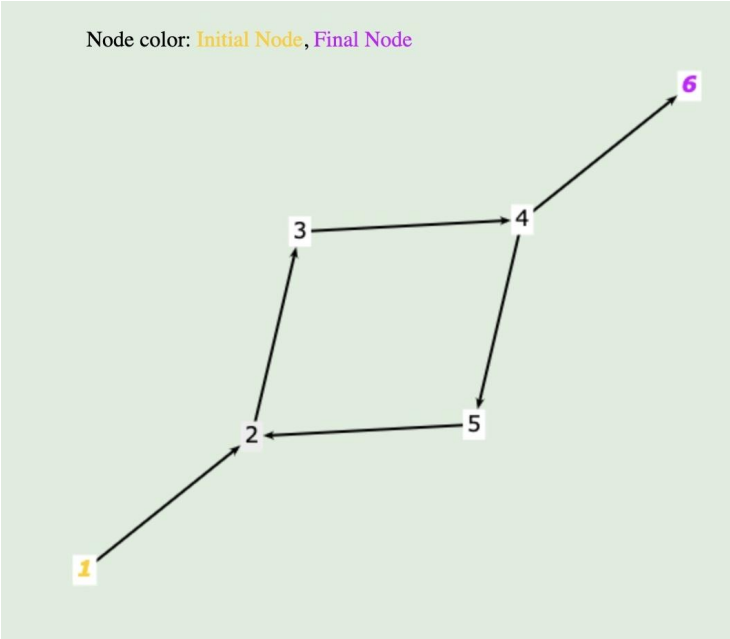
Note : Other test paths were infeasible, so we don't consider them here.

Class : *MathTree*, function: *check_neg2*

Control flow graph :



Numbered graph form :



Test paths :

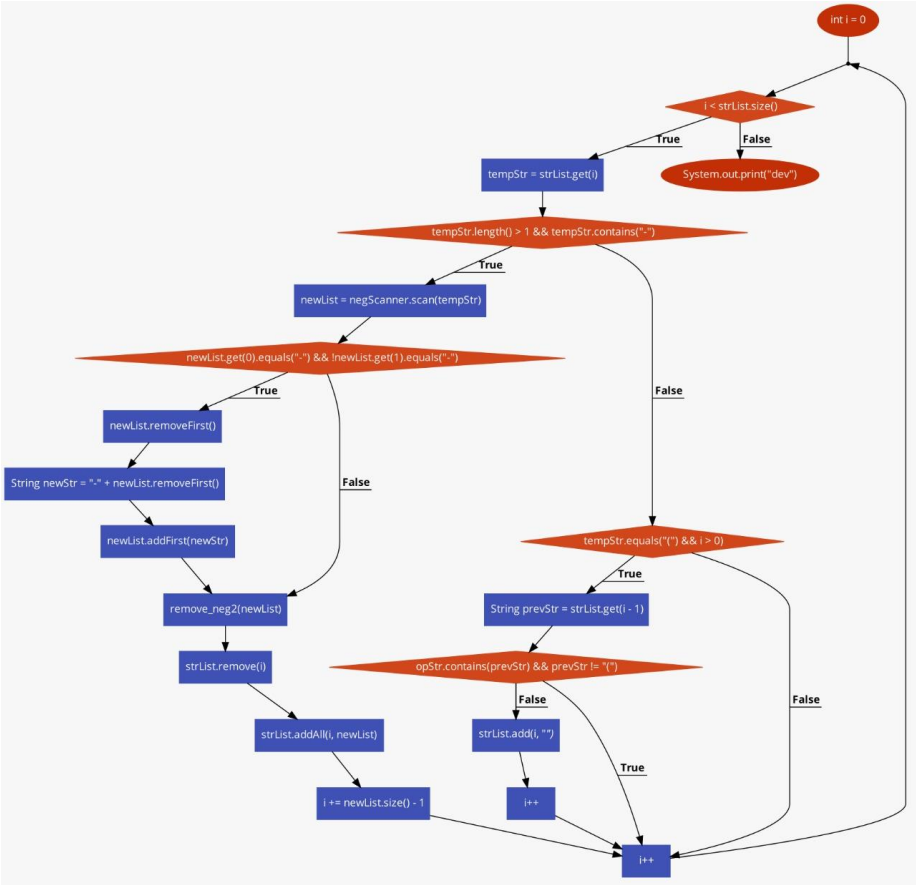
Test 1: [1,2,3,4,5,2,3,4,6]
Test string : ["3","-","-","4"]

Test 2: [1,2,3,4,6]
Test String : ["3", "-", "+", "4"]

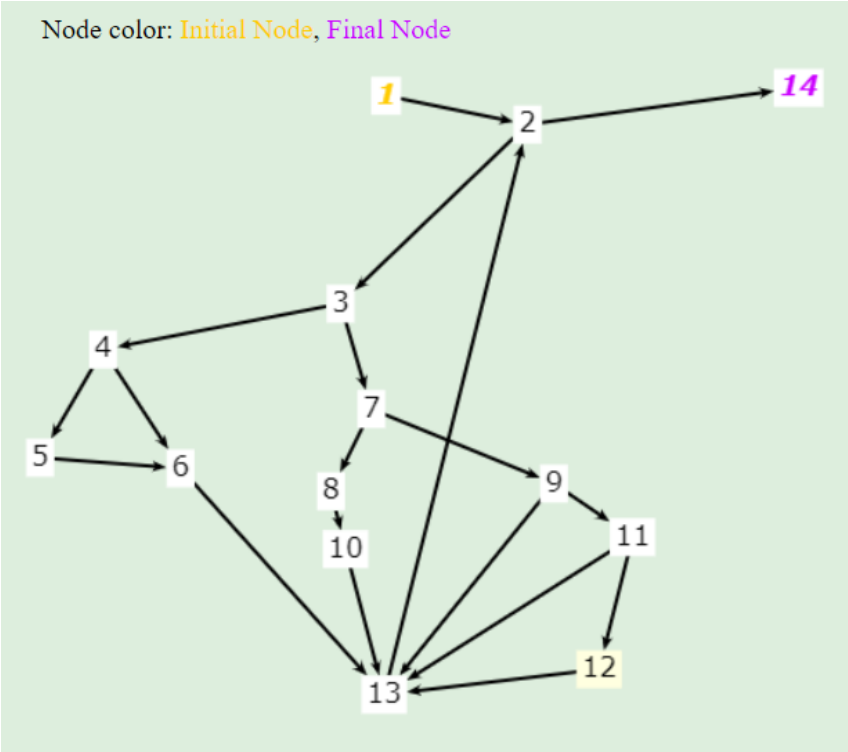
Test 3: [1,2,3,4,5,2,3,4,5,2,3,4,6]
Test String : Impossible case.

Note : Other test paths were infeasible, so we don't consider them here.

Class : *MathTree,* Function: *CleanStrList*
Control flow graph :



Numbered graph form :



Test paths and corresponding input string :
[1,2,14] => ””
[1,2,3,7,9,13,2,14] => ”2”
[1,2,3,7,9,13,2,3,7,9,13,2,14] => ”-3”
[1,2,3,7,9,13,2,3,7,8,10,13,2,7,9,13,2,3,7,9,11,13,14] => "2(3)"

Note : Other test paths were infeasible, so we don't consider them here.

Conclusion:

In this project we successfully performed testing on various branching conditions like:

- nested if statements
- nested for loops inside if
- nested if inside for loop

These testings resulted in formation of complex control flow structures and writing input test cases for various feasible paths gave us great learning experience and exposure to testing tools.