31251 Data Structures and Algorithms
Assignment 1

1 Overview

Your task for this assignment is to complete a class representing an expression tree. The functions you will complete, collectively, take in a string in infix notation, parse it to build an expression tree, evaluate the tree and print out the tree in infix, prefix and postfix notation.

The expressions you will deal with are simple arithmetic expressions on non-negative integers. The allowed operators are $+$, $-$, $*$ and $/$, and sub-expressions may be enclosed in parentheses ( ).

2 The Code

You are provided with 5 C++ files:

- `TreeNode.h`
- `TreeNode.cpp`
- `ExprTree.h`
- `ExprTree.cpp`
- `Assignment1Tests.h`

The only file you need to modify is `ExprTree.cpp`. As this will also be the only file you submit, if you modify the other files, it won't work during marking!

`TreeNode.h` and `TreeNode.cpp` define a node class for the expression tree

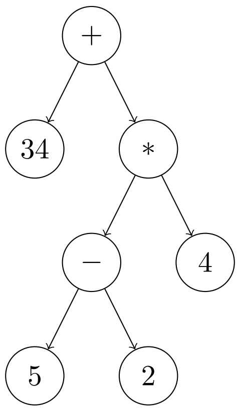
3 An Example

An arithmetic expression in infix notation is the normal way we write arithmetic expressions, e.g.:

$$34 + (5 - 2) * 4$$

This can be tokenised into its individual components: "34", "+", "(", "5", "$-$", "2", ")", "$*$" and "4".

With these we can build an expression tree that represents the same expression:



By traversing the tree we can do a number of things:

- We can evaluate the expression to get 46.
- We can print out the prefix version of the expression: $+ \ 34 \ * \ - \ 5 \ 2 \ 4$.
- We can print out the infix version of the expression: $34 \ + \ 5 \ - \ 2 \ * \ 4$.
- We can print out the postfix version of the expression: $34 \ 5 \ 2 \ - \ 4 \ * \ +$.

The functions in `ExprTree.cpp` will implement this functionality.

4 A Precise Definition of an Arithmetic Expression

An arithmetic expression can be recursively defined as follows:

- A single number is an arithmetic expression.
- If a and b are arithmetic expressions, the following are also arithmetic expressions:
  - $a + b$
  - $a - b$
  - $a * b$
  - a/b
  - (a)

5 The `ExprTree` Class

To complete the assignment, you must complete the code in `ExprTree.cpp`. You may add additional global functions if you find them useful. As this is the only file you will submit, you should not change the other files, and you should complete the existing functions, as these are the ones that will be marked (note that the functionality marking will be automatic, so it must work with the functions as given).

`ExprTree` contains the following functions, which are to have the described behaviour:

- `ExprTree()` – The basic constructor that initialises an empty `ExprTree`.

- `ExprTree(TreeNode *)` – A constructor that takes a `TreeNode*` and creates an expression tree that has that node as the root, along with any children it has.
- `~ExprTree()` – Destructor to clean up the tree. This will not be tested, but you should probably implement it to stop memory leaks.
- `static vector<string> tokenise(string)` – Reads in a string, then breaks it up into the individual components like in the example above. Returns the components grouped together as a vector of strings. The input string may or may not separate the components with white space.
- `static ExprTree buildTree(vector<string>)` – Takes a vector of strings where each element is an individual component of an expression (as produced by `tokenise`), and returns a tree representing the expression as in the example above.
- `static int evaluate(TreeNode *)` – Computes the actual value the expression starting at the node given as a parameter produces.
- `int evaluateWholeTree()` – Computes the value of the whole tree (i.e. starts at the root).
- `static string prefixOrder(const ExprTree &)` – Returns the expression as a string in prefix order. Each component should be separated by a single space character.
- `static string infixOrder(const ExprTree &)` – Returns the expression as a string in infix order. Each component should be separated by a single space character.
- `static string postfixOrder(const ExprTree &)` – Returns the expression as a string in postfix order. Each component should be separated by a single space character.
- `int size()` – Returns the number of nodes in the tree. This is already implemented.
- `bool isEmpty()` – Returns true if the tree contains no nodes, false otherwise. This is already implemented.
- `TreeNode * getRoot()` – Returns the root of the tree. This is already implemented.

`ExprTree.cpp` also contains four predefined helper functions to do some basic conversions between types (which is not always straightforward in C++98), check if a string is a number and produce operator TreeNodes from strings. Note that the parameters to these functions (and three of the others) are `const <type> &`. This should make no difference to your code – references in C++ work like Java references, so you can use them as if they were the actual variable (i.e. no dereferencing), it was done for technical reasons.

6 Some Notes About the Input and Output

You will not be expected to handle incorrect or broken input. Expressions given as input strings (for the `tokenise(string)` function) may or may not include additional space separating the components. For the three functions that produce string versions of the expression, it is expected that a single space will separate each component (as with the example).

7 The Test File

For the functionality testing, we will use CxxTest. You are provided with a copy of the test file so you can check your progress as you complete the assignment. To make the scoring work correctly, the test file needs to be built with some additional options we have not used before. It will work without these options, but the scoring will be inaccurate.

To get the scoring working:

```
>cxxtestgen --have-eh --abort-on-fail --error-printer -o Assignment1Tests.cpp Assignment1Tests.h
```

Of course you will have to run `cxxtestgen` as appropriate given your local setup. After this however, the compilation works as before (and you can pick whatever output name you want).

8 Marking

The overall mark for the assignment consists of three parts, functionality, design and style.

8.1 Functionality

Functionality will be assessed by the automated test file and is worth 50% of the total mark. The marks will be assigned according to the following rubric:

Pass	`ExprTree()`, `ExprTree(TreeNode*)`, `prefixOrder()`, `infixOrder()` and `postfixOrder` must be properly implemented. `buildTree(vector<string>)`, `evaluate(TreeNode*)` and `evaluateWholeTree()` must support expressions involving only addition (i.e. you don't have to worry about other operators or parentheses.
Credit	All Pass level implementation complete, plus `evaluate(TreeNode*)` and `evaluateWholeTree()` must support all arithmetic operators.
Distinction	All Credit level requirements, plus `buildTree(vector<string>)` must support all operators (including operator precedence) and expressions with parentheses.
High Distinction	All Distinction level requirements, plus `tokenise(string)` must be implemented.

8.2 Design

The design of your assignment will be assessed by your tutor during the in-class demonstration and is worth 35%. The marks will be assigned according to the following rubric:

Pass	The code shows basic understanding of how to employ data structures to achieve a goal. The design should avoid unnecessary data structures and should make reasonable use of iteration and recursion when appropriate.
Credit	The design shows a solid understanding of data structures and demonstrate effective use of control structures to achieve the program's goals.
Distinction	The design shows a high degree of understanding of how to use data structures to achieve a goal efficiently, and demonstrate some evidence that the design does not use unnecessary resources. The design should be clean and efficient.
High Distinction	The design demonstrates a high degree of understanding of data structures and how to efficiently employ them to build algorithms that not only meet technical goals, but support maintenance and future development.

8.3 Coding Style

Coding style will also be marked by your tutor during the in-class demonstration against the following rubric:

Pass	The code mostly uses some formatting standard and is somewhat readable.
Credit	The code adheres well to a formatting standard and variables are well named.
Distinction	At least as well formatted as for Credit standards, along with sufficient inline commenting to explain the code.
High Distinction	Excellent formatting and variable naming. Excellent, judiciously employed comments that explain the code without just repeating the code.

8.4 Marking Schedule

If you submit your assignment on time, and no other issues arise, we expect to return your marks within a week after your in-class demonstration. We do however reserve the right to delay this schedule if necessary.

9 Plagiarism Detection Software

You assignment may be submitted to plagiarism detection software at the course coordinator's discretion.

10 Submission

You will submit your `ExprTree.cpp` file using UTSOnline. Submit only this file.

11 Due Date

The assignment is due by 5pm, Friday, May 12th. See the subject outline for late submission penalties.