Project 3

- Choose either project 3A or 3B. **Deadline: April 30, 2015.** All the submissions should be made electronically on Blackboard by the end of the day.
- ***Those who worked on project 2A are encouraged to work on project 3B.***
- Please stick to the teams that I assigned you with.

Project 3A

- *Using binary expression trees*, write an infix expression parser. Here are a few examples of expressions your program should parse and evaluate:

Expression	Result
`1+2*3`	`7`
`2+2^2*3`	`14`
`1==2`	`0 //or false if the type is bool`
`1+3 > 2`	`1 // or true if the type is bool`
`(4>=4) && 0`	`0 //or false if the type is bool`
`(1+2)*3`	`9`
`++++2-5*(3^2)`	`-41`

Technical Requirements

- (Weight: 20%) Your parser should parse an infix expression that supports the following arithmetic and logical operators with the specified precedencies:

Operator	Precedence	Example
`! //logical not`	8	`! 1 // = 0 (false)`
`++ //prefix increment`	8	`++2 //3`
`-- //prefix decrement`	8	`--2 //1`
`- //negative`	8	`-1 //-1`
`^ //power`	7	`2^3 // 8`
`*, /, % //arithmetic`	6	`6 * 2 // 12`
`+, - //arithmetic`	5	`6 - 2 //4`
`>, >=, <, <= //comparison`	4	`6 > 5 // 1(true)`
`==, != //equality comparison`	3	`6!=5 // 1(true)`
`&& //logical and`	2	`6>5 && 4>5 //0(false)`
`|| //logical OR`	1	`1 || 0 //1 (true)`

- (Weight: 30%) Parse an expression given in a string format. Your program should be flexible with the given expressions. For instance, 1+2 is the same as 1 + 2. The user should not worry about writing the spaces between operands and operators.
- (Weight: 20%) Your program should check for invalid expressions, and produce meaningful error messages when necessary. Further, the error message should indicate whether the error happened (Only report the first error the program encounters and exits the program).
  Here are a few examples:

What the user enters	Error message
)3+2	Expression can’t start with a closing parenthesis @ char: 0
<3+2	Expression can’t start with a binary operator @ char: 0
3&&&& 5	Two binary operators in a row @ char 3
15+3 2	Two operands in a row @ char 5
10+ ++<3	A unary operand can’t be followed by a binary operator @ char 6
1/0	Division by zero @ char 2

- (Weight: 30%) Evaluate the given expressions efficiently.

Facts and Assumptions

- You may assume that all operands are integers. The result of a comparison is a boolean (e.g. 6==6 is true). However, C++ compiler allows a boolean can be converted to an integer according to this logic: true =1 and false = 0. An integer can be converted to a boolean according to this logic: a number that is equal to 0 is false. Otherwise, it is true.
- You may just call the evaluate function in the main function. There is no need for getting input from the user or a menu-based system. For instance, this is an example of how the evaluator is used in the main function:

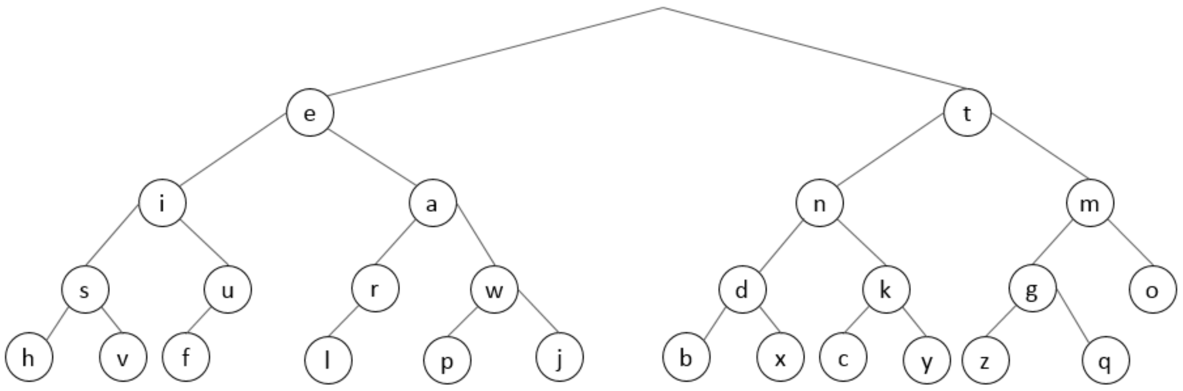
```
int main(){
Evaluator eval;
int result=eval.eval("!!!3+2");

return 0;
}
```

- Take a look at the algorithm in the following link for inspiration (This algorithm constructs a binary expression tree from a postfix expression. **The required one should build the tree it from an infix expression instead**): http://en.wikipedia.org/wiki/Binary_expression_tree#Construction_of_an_Expression_Tree
- You are welcome use the source code on Blackboard (e.g. `Binary_Tree`).
- Negative numbers and unary operators (e.g. !, ++, --) should be supported.

Project 3B

- Morse code (see the table below) is a common code that is used to encode messages consisting of letters and digits. Each letter consists of a series of dots and dashes, for example, the code for the letter a is •- and the code for the letter b is -•••. Store each letter of the alphabet in a node of a binary tree of depth 4. The root node is at depth 0 and stores no letter. The left node at depth 1 stores the letter e (code •) and the right node stores the letter t (code is -). The four nodes at depth 2 stores the letters with codes (••,•-, -•, --). To build the tree (See the figure below), read a file in which each line consists of a letter followed by its code. The letters should be ordered by tree depth. To find the position for a letter in the tree, scan the code and branch left for a dot and branch right for a dash. Encode a message by replacing each letter by its code symbol. Then decode the message using Morse code tree. Make sure you use a delimiter symbol between coded letters.

a ●-	b -●●●	c -●-●
d -●●	e ●	f ●●-●
g --●	h ●●●●	i ●●
j ●---	k -●-	l ●-●●
m --	n -●	o ---
p ●--●	q --●-	r ●-●
s ●●●	t -	u ●●-
v ●●●-	w ●--	x -●●-
y -●--	z --●●	



Technical Requirements

- (Weight: 40%) Write a function that builds the morse tree shown in the figure above. The information of the tree (the letters and the codes) are to be stored in a file. You can find the file here: https://www.dropbox.com/s/3cj8yb8gcdsrefg/morse.txt?dl=0 (Notice that the file hasn't been laid out in convenient order for building the tree).
- (Weight: 30%) Your system should be able to decode a message using the morse tree that you built. For example, decoding -●● --● results in "dg". The problem text briefly explains how you can do that. Notice that between the symbols (dots and dashes) is a space. The space is a delimiter that separates the codes for letters.
- (Weight: 30%) Your system should also encode a message. For example, encoding "ac" results in ●- -●-●.

  You may use a binary search tree or a map to store the codes for letters.

Facts and Assumptions

- You may assume that the delimiters are simply spaces.
- You are welcome to use the source code on Blackboard (e.g. `Binary_Tree`, `Binary_Search_Tree`).
- You may just call the decode and encode functions in the main function. There is no need for getting input from the user or a menu-based system.