**Stacks and Compilers**

Purpose

**This programming assignment continues to extend the feature of the textbook's calculator example. You will implement unary operators and the operator= in your calculator. Furthermore, your calculator must be able to handle three variables such as a, b, and c.**

Unary Operators

**Those includes '-', '+', '!', and '~'. For those operators' precedence, refer to**[**http://www.cppreference.com/operator\_precedence.html (Links to an external site.)**](http://www.cppreference.com/operator_precedence.html)**. Among them, a special attention must be paid to '-' and '+', because you have to decide if they are binary or unary operators. You can make a decision in tokenizer.cpp.h's getToken( )as follows:**

1. Prepare **TokenType prevToken** as a private data member of **tokenizer.h**.
2. Whenever **tokenizer.cpp.h** returns a toke type, memorize it in **prevToken**.
3. When encountering '+' or '-', your **tokenizer.cpp.h** must check **prevToken** if the previous token was '('. If so, you can return the unary '+' or unary '-'. Otherwise, you should return the binary '+' or binary '-'.
4. Note that **prevToken** must be initialized with OPAREN (i.e, '(') so that your **tokenizer.cpp.h** can accept the very first '+' or '-' as a unary operator even in case if they do not follow an open parenthesis.

Variables

**Your calculator is supposed to handle only three variables such as a, b, and c. While the actual implementation is up to you, the easiest (but an unextended) implementation is as follows:**

1. **evaluator.h**: add the following private data members.
2. private:
3. // new members
4. vector postFixVarStack; // Postfix machine stack for var
5. NumericType var\_a; // variable a
6. NumericType var\_b; // variable b
7. NumericType var\_c; // variable c
8. **token.h**: add the following special tokens, each corresponding to variables **a**, **b**, and **c**.
9. enum TokenType {
10. ...
11. VAR\_A, // variable a
12. VAR\_B, // variable b
13. VAR\_C // variable c
14. };
15. **tokenizer.cpp.h**: return the corresponding token when getToken( ) encounters 'a', 'b', and 'c.
16. if ( getChar( ch ) == true ) {
17. switch( ch ) {
18. ...
19. case 'a':
20. prevToken = VAR\_A;
21. return Token<NumericType>( VAR\_A, 0 );
22. case 'b':
23. ... // the same as 'a'
24. case 'c':
25. ... // the same as 'b'
26. ...
27. default:
28. ...
29. }
30. }
31. **evaluator.cpp.h**: add three **case** statements, each corresponding to **VAR\_A**, **VAR\_B**, and **VAR\_C** and pushing the variable content in **postFixStack** and the variable name in **postFixVarStack**. For a constant value, push the constant value to**postFixStack** as you did previously and push a space ' ' to **postFixVarStack**.
32. switch( lastType ) {
33. case VALUE:
34. postFixStack.push\_back( lastToken.getValue( ) );
35. postFixVarStack.push\_back( ' ' );
36. return;
38. case VAR\_A:
39. postFixStack.push\_back( var\_a );
40. postFixVarStack.push\_back( 'a' );
41. return;
42. case VAR\_B:
43. ... // the same as VAR\_B
44. case VAR\_C:
45. ... // the same as VAR\_C

Whenever **evaluator.cpp.h** needs to pop out a value from **postFixStack**, you must also pop out a character from**postFixVarStack**. Similarly, whenever **evaluator.cpp.h** needs to push a new value to **postFixStack**, you must also push the corresponding character ('a', 'b', 'c', or ' ') to **postFixVarStack**. This way allows both **postFixStack** and **postFixVarStack**synchronously grow and shrink.

Assignment Operators

**Although C++ assignment operators include =, +=, -=, \*=, /=, %=, &=, ^=, |=, <<=, and >>=, we focus on only '='.**

Statement of Work

1. Download [cpp\_evaluator1.zip](https://canvas.uw.edu/courses/1028107/files/32574516/download?wrap=1)[View in a new window](https://canvas.uw.edu/courses/1028107/files/32574516/download?wrap=1) to your project. Replace the evaluator.cpp.h and tokenizer.cpp.h with the ones from Lab 6.
2. Implement unary operators '+', '-', '!', '~', an assignment operator '=', and variables 'a', 'b', and 'c'. You may follow the above instructions or develop your own algorithm.
3. Compile all your code.
4. Execute this **evaluator** program to verify your modification.
5. $ ./a.out
6. 8 + (-5 + 2)
7. 5
8. 8 + (+5 + 2)
9. 15
10. -5 + 8 + 2
11. 5
12. +5 + 8 + 2
13. 15
14. !(3 == 3)
15. 0
16. !3 == 0
17. 1
18. ~2
19. -3
20. a = 10 \* 3
21. 30
22. b = 5 \* 2
23. 10
24. c = 8 % 3
25. 2
26. (a = 10 \* 3) + (b = 30 / 3) + (c = 50 - 25) + a + (-b) + ~c \* 3
27. 7

^c (to exit the program)