COSC2320: Data Structures and Algorithms
HW3: List Expression Evaluation Using Stacks

1 Introduction

You will create a C++ program to evaluate expressions combining set union, set intersection and parentheses. The program must exploit a stack to convert the matrix expressions from infix notation to postfix notation and a second stack to evaluate the postfix expression.

In order to maintain similar notation to arithmetic expressions, + will denote union and * will be intersection. You will evaluate expressions like $A * B + C$, $(A + B) * (D + E * F)$ or $A * B + (C * D)$, where A, B, C and D are sets.

We will define the operation " $+$ " of two sets A and B as the union of the contents of both sets:

$$A + B \equiv A \cup B \tag{1}$$

Similarly, we will define the operation " $*$ " of A and B as the intersection of A and B:

$$A * B \equiv A \cap B \tag{2}$$

These definitions allow us to write long algebraic equations. We assign to the " $+$ " a lower precedence than " $*$ " (in the same way as we do in algebra) and parentheses change the precedence of an expression.

2 Input

The input is one script file, in the same format as the script file from previous homeworks, and two or more data files containing words. All operations supported in the previous homework (read(), write()) will still be supported. You will need to extend the script parser in order to accept expressions such as

```
R0=(A+B)*C
```

There will be **one** expression per line, and there might be multiple expressions in each script file. You are expected to be able to reuse the results from previous expressions.

3 Examples

INPUTS

```
   A.txt
   ---------------------------------------------
   alpha beta gamma delta Alpha

   <EOF>

   B.txt
   ---------------------------------------------
   cat                dog bird<EOF>

   C.txt
   ---------------------------------------------
   alphA

   dog
   <EOF>
```

```
input.script1
---------------------------------------------
read(A,'A.txt')
read(B,'B.txt')
R1=A+B
write(R1,'R1.txt')
read(C,'C.txt')
R2=A+B*C
write(R2,'R2.txt')
```

```
input.script2
---------------------------------------------
read(A,'A.txt')
read(B,'B.txt')
read(C,'C.txt')
R1=A+B
R2=A*B
R3=A+B*C
R4=(A+B)*C
R5=(A+B)*R4+A
write(R5,'out5.txt',reverse)
```

RESULTS

```
   out1.txt
   ---------------------------------------------
   alpha 2
   beta 1
```

```
bird 1
cat 1
delta 1
dog 1
gamma 1
<EOF>

out2.txt
---------------------------------------------
<EOF>

out3.txt
---------------------------------------------
alpha 2
beta 1
delta 1
dog 1
gamma 1
<EOF>

out4.txt
---------------------------------------------
dog 1
alpha 1
<EOF>

out5.txt
---------------------------------------------
gamma 1
dog 1
delta 1
cat 1
beta 1
alpha 3
<EOF>
```

4 Program and output specification

The main program should be called ”expressions”. Syntax:

```
expressions script=input.script
```

When asked to output a doubly linked list to a file you must write one word per line followed by a space and the number of occurrences of the word. Please note that there is no change from the previous homework. If the number of occurrences is 0 you must not output the word. Similarly, if the linked list is empty, the file created will be empty. When calculating the intersection of two linked lists you must only keep one copy of each word in the output list.

5 Requirements

- Your program must use stacks to evaluate algebraic expressions, one for infix to posfix conversion and another for posfix evaluation.
- You are allowed to build your stacks in any way you prefer, as long as the fundamental stack operations ”push”, ”pop” and ”top” exist. Your program must not crash due to stack overflows or underflows.
- You must use linked lists to manipulate lists of words. However, for a 20% penalty you can use arrays in this homework (indicate it in your README file). If the TAs dertect you use arrays without disclosing it your maximum grade will be 60.
- Notice the stack allows you to detect invalid expressions, at conversion time or during evaluation time. Your program must be able to detect invalid expressions: undefined sets, malformed expressions, unbalanced parenthesis, missing assignment operators, missing operators or missing operands. Your program must not crash if such an error is detected. You must send an error message to the log file (and optionally to the screen), and continue reading the script file. There will be two hard test cases with invalid expressions.
- You will modify the script parser to accept expressions. We will provide guidance, but we will not generate a new version of the parser.
- When writing to a file you will be asked to write either in alphabetical order or in reverse alphabetical order. Since your linked lists will be created in order, this will involve writing starting from the head or from the tail. You don’t need to use recursion to print your files.
- You must create a log file that records the length of each linked list after each operation. This log file should also be used to output any warnings and errors you might encounter.
- The program should not halt when encountering errors in the script. It should just send a message to the log file and continue with the next line. The only error that is unrecoverable is a missing script file.
- Do not use the STL library.
- Your program should write error messages to a log file (and optionally to the screen). Your program should not crash, halt unexpectedly or produce unhandled exceptions. Consider empty input, zeroes and inconsistent information. Each exception will be -10.
- Test cases: Your program will be tested with 10 test scripts, going from easy to difficult. You can assume 80% of test cases will be clean, valid input files. If your program fails an easy test script 10-20 points will be deducted. A medium difficulty test case is 10 points off. Difficult cases with specific input issues or complex algorithmic aspects are worth 5 points.
- A program not submitted by the deadline is zero points. A non-working program is worth 10 points. A program that does some computations correctly, but fails several test cases (especially the easy ones) is worth 50 points. Only programs that work correctly with most input files that produce correct results will get a score of 80 or higher. In general, correctness is more important than speed.