1. Give a flex-suitable regular expression that matches all words that do not have three adjacent vowels anywhere in the word.

<table>
<thead>
<tr>
<th>match</th>
<th>not a match</th>
</tr>
</thead>
<tbody>
<tr>
<td>ugliness</td>
<td>beauty</td>
</tr>
<tr>
<td>integral</td>
<td>extraneous</td>
</tr>
<tr>
<td>principal</td>
<td>coauthor</td>
</tr>
</tbody>
</table>

For our purposes, a *word* is any string of lower-case letters (alphabetic characters), i.e., something that matches [a-z]*, and a vowel is anything that matches [aeiou]. You may assume that the entire input consists of a single word; that is, if the input in its entirety does not match [a-z]*, then I don’t care whether your pattern matches or not. You are NOT allowed to use the flex "/" operator. Be as concise as possible.

**Answer:** Here are two possible solutions that are equally good:

\[
([aeiou]?[aeiou]?[^aeiou][aeiou]?[aeiou]?)*
\]

\[
([aeiou]?[aeiou]?[^aeiou])*[aeiou]?[aeiou]?
\]

2. Using the method described in the book or in class, convert the following regular expression into an equivalent (nondeterministic) finite automaton:

\[
(((bc)|")aa)*
\]

**Answer:** I’ll accept anything reasonably close to this:

![Automaton Diagram]

3. Using the sets-of-states approach described in class, simulate the (nondeterministic) finite automaton shown below on the input string \( w = bbaab \). That is, for every prefix \( x \) of \( w \) (so \( x = \varepsilon, b, bb, bba, bbaa, bbaab \) in that order), list all states reachable from the start state by reading \( x \). Does the automaton accept \( w \)? Explain.
Answer:

<table>
<thead>
<tr>
<th>prefix</th>
<th>set of states</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\varepsilon$</td>
<td>1,2,3,5,6,7</td>
</tr>
<tr>
<td>b</td>
<td>8,9</td>
</tr>
<tr>
<td>bb</td>
<td>10,1,2,3,5,6,7</td>
</tr>
<tr>
<td>bba</td>
<td>4,5,2,3,6,7</td>
</tr>
<tr>
<td>bbaab</td>
<td>4,5,2,3,6,7</td>
</tr>
</tbody>
</table>

Order of states does not matter in any of the listings. The automaton rejects $w$, because neither state 8 or 9 is an accepting state.

4. Recall our usual unambiguous grammar for arithmetic expressions with start symbol $E$:

$$
E \rightarrow T \mid E + T \mid E - T
$$

$$
T \rightarrow F \mid T \ast F \mid T/F
$$

$$
F \rightarrow c \mid v \mid (E)
$$

Using this grammar, show a complete parse tree yielding the string “$v/c \ast c - (v + c)$”.

Answer:

```
            E
           /|
          |
         E  -  T
        /   |
       T     F
      /   |
     T     F
    /   |
   T     F
   /   |
  T     F
   /   |
  T     F
  /   |
  T     F
   /   |
  T     F
   /   |
  T     F
    /   |
   v     c
     /   |
    v     c
      |
      v
```
5. The following grammar generates all strings of balanced parentheses:

\[ S \rightarrow (S) | SS | \varepsilon \]

Show that this grammar is ambiguous by finding a string of parentheses that has two distinct parse trees. Also show the two parse trees. Your string should be as short as possible. Hint: There is a string of length six (6) that has two distinct parse trees.

**Answer:** Here are two parse trees yielding \( \varepsilon \). I’ll allow any string up to 6 symbols.

![Parse Trees](image)

6. Recall one of our standard, simplified grammars for arithmetic expressions, given in yacc/bison form (\texttt{expr} is the start symbol):

```yacc
expr :
  term
  | expr '+' term { $$ = $1 + 1 + $3; }%\texttt{semantic action}
  | expr '-' term { $$ = $1 + 1 + $3; }
;

term :
  factor
  | term '*' factor { $$ = $1 + 1 + $3; }
  | term '/' factor { $$ = $1 + 1 + $3; }
;

factor :
  CONST
  | VAR
  | '(' expr ')'
;
```

Add semantic actions to this grammar that compute—as a synthesized attribute of the root—the number of occurrences of operation symbols in the input expression. Note that this value is of type \texttt{int}, the default attribute type. You may assume that the input is a well-formed expression.

For example, if the input is “3*(4+5)+6”, then the root should have attribute value 3, because the expression has three occurrences of operation symbols (two + and one *).

**Answer:**

```yacc
expr :
  term
  | expr '+' term { $$ = $1 + 1 + $3; }
  | expr '-' term { $$ = $1 + 1 + $3; }
;

term :
  factor
  | term '*' factor { $$ = $1 + 1 + $3; }
  | term '/' factor { $$ = $1 + 1 + $3; }
;
```

3
factor :
  CONST          { $$ = 0; }  
| VAR            { $$ = 0; }  
| '(' expr ')'   { $$ = $2; } 
;