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

\[(c(a|b)c)^*\]

**Answer:** Here is one possible correct answer:

![Automaton Diagram](image)

Several \(\varepsilon\)-transitions can be contracted and some states merged.

2. Using the sets-of-states approach described in class, simulate the (nondeterministic) finite automaton shown below on the input string \(w = abcca\). That is, for every prefix \(x\) of \(w\) (so \(x = \varepsilon, a, ab, abc, abcc, abcca\) in that order), list all states reachable from the start state by reading \(x\). Does the automaton accept \(w\)? Explain.

![Automaton Diagram](image)
Order of states does not matter in any of the listings. The automaton rejects \( w \), because there are no accepting states in the final set.

3. Give a single flex-suitable regular expression that matches all Pascal comments. A Pascal comment is a string \( s \) surrounded by the delimiters (* and *), which are themselves part of the comment. The string \( s \) can be any string of ASCII characters (including newline) that does not contain * as a substring. For example,

\[
(* \text{ This is a Pascal comment. } *)
\]

\[
****** \text{ So ** is ** this. ******}
\]

\[
(* \text{ (and this)* })
\]

\[
(*\text{this too}*)
\]

\[
(* \text{ Comments can’t be nested, so (* this whole line is a single, complete comment *)}
\]

\[
(*(*\text{(* This is inside a comment *) but this is not! } *)
\]

Here are two legal comments: (**) and (***)

This is not a legal comment: (*)

You are NOT allowed to use the flex “/” operator. Be as concise as possible.

[Hint: [^*] matches any non-star character, and [^\)] matches any character except a right paren.]

Answer: Here are two possible correct answers that differ only slightly:

"(*"([^-]*[^[^-]*][^[^-]*]*)*[^[^-]*]""

"(*"([^-]*[^[^-]*][^[^-]*]*)*[^[^-]*]"

The core of the comment is divided into chunks, where each chunk is either a single non-star character or a run of stars followed by something that is neither a star nor a right parenthesis. There may be some trailing stars left over, so we have to match those, too. Instead of [^*], one could alternatively use "*" or \\*, which both treat the star character literally.

4. Here is a grammar with start symbol \( S \) describing some kinds of statements in a Pascal-like programming language:

\[
S \rightarrow a \mid iS \mid iSeS \mid wS \mid \{L\}
\]

\[
L \rightarrow L;S \mid S
\]

(Just to explain—here, \( S \) stands for “statement,” \( L \) for “list” (of statements), \( a \) for “assignment,” \( i \) for “if,” \( e \) for “else,” and \( w \) for “while.”)

(a) Using this grammar, show the complete parse tree yielding the string “\{a;wiaeia\}”.

(b) Show that this grammar is ambiguous by giving two different parse trees yielding the string “\(iiaea\)”. (This is the so-called, “dangling else” ambiguity.)

Answer:

(a) This is the unique parse tree:
(b) The parse tree on the right corresponds to the usual grouping, where the “else” matches the closest unmatched “if”:

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

```plaintext
expr :
  term
  | expr '+' term
  | expr '-' term
 ;
term :
  factor
  | term '*' factor
  | term '/' factor
 ;
factor :
  CONST
  | VAR
  | '(' expr ')' 
 ;
```
Add semantic actions to this grammar so that the root of the parse tree contains as its attribute the number of parentheses in the input expression.

For example, if the input is “3*(4+5)+6”, then the root should have attribute value 2, because the expression has two parentheses (one opening, one closing).

Make your actions as simple as possible. Any omitted actions are assumed to be the default action {$$ = $1;}. 

Answer:

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

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

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