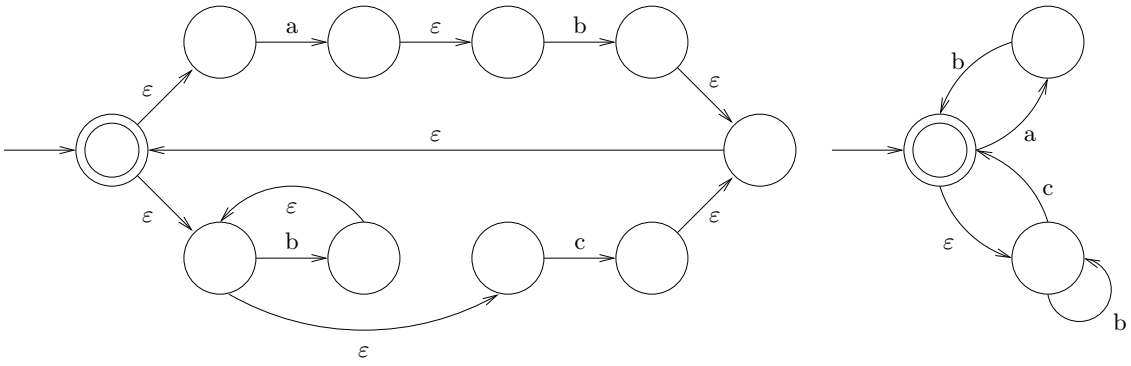
CSCE 531, Spring 2017, Midterm Exam
Answer Key

1. (15 points) Using the method described in the book or in class, convert the following regular expression into an equivalent (nondeterministic) finite automaton:

   ```
   (ab|b*c)*
   ```

   You may contract ε-transitions provided the resulting automaton is equivalent.

   Answer: Any equivalent answer between the these two extremes is correct.



   There are other possible correct answers.

2. (15 points) Using the sets-of-states approach described in class, simulate the (nondeterministic) finite automaton given below (in tabular form) on the input string $w = bcabcab$. That is, for every prefix x of w (so $x = \varepsilon$, b, bc, bca, $bcab$, $bcabc$, $bcabca$, $bcabcab$ in that order), list all states reachable from the start state by reading x. For each prefix, list the states in increasing order without duplicates. Does the automaton accept w? Why or why not?

	a	b	c	ε
$\rightarrow 0$	5			1, 2
1		3	2, 3	
2	1		2	4
3		3, 6	0	
4	7	4, 5		
5			7	2, 4
6	6	0	6	
$*7$				3, 6

Answer:

prefix	states
ε	$0, 1, 2, 4$
b	$2, 3, 4, 5$
bc	$0, 1, 2, 3, 4, 6, 7$ (all but 5)
bca	$1, 2, 3, 4, 5, 6, 7$ (all but 0)
$bcab$	$0, 1, 2, 3, 4, 5, 6$ (all but 7)
$bcabc$	$0, 1, 2, 3, 4, 6, 7$ (all but 5)
$bcabca$	$1, 2, 3, 4, 5, 6, 7$ (all but 0)
$bcabcab$	$0, 1, 2, 3, 4, 5, 6$ (all but 7)

The input is rejected, because the final set of states does not contain an accepting state (i.e., state 7).

3. (10 points) Give a single flex-suitable regular expression that matches all Pascal-style comments. That is, your regex should match all strings that start with "`(*`", end with "`*)`", and do not have any occurrence of "`*)`" in between as a substring. For example,

```
 match       |  not a match
-------------+------------------
(* hi *)     | (* hi *) (* ho *)
(*********)  | (*)
(*(*(* **)   | (* (* nest *) *)
(*)))*(*)    | (*    *
```

Note that opening and closing delimiters cannot overlap.

Your answer may include any named subexpressions in curly braces, provided you define them fully as you would in the preamble. You are NOT allowed to use the flex "/" operator. Be as concise as possible. [Keep in mind that the flex scanner will look for the longest match possible.]

Answer: There are a number of correct answers. To avoid confusion, I will define some named subexpressions, then list some equivalent patterns.

```
star      "*"
rparen    ")"
nonstar   [^*]
neither   [^)*]
open      "(*"
close     "*)"
%%
{open}{nonstar}*{star}(({neither}{nonstar}*)?{star})*{rparen}

{open}({nonstar}|{star}+{neither})*{star}+{rparen}

{open}{nonstar}*({star}({neither}{nonstar}*)?)*{close}

{open}({nonstar}|{star}+{neither})*{star}*{close}
```

There are other correct answers.

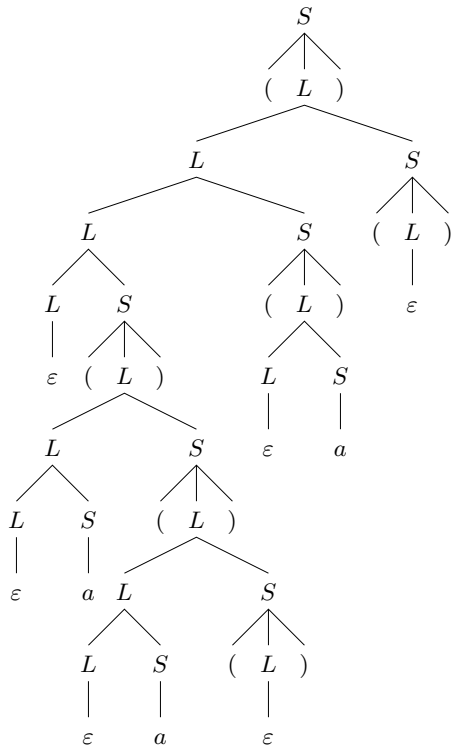
4. (15 points total) Here is a grammar with start symbol S:

$$\begin{aligned} S &\rightarrow a \mid (L) \\ L &\rightarrow \varepsilon \mid SL \mid LS \end{aligned}$$

   (a) (10 points) Using this grammar, show a complete parse tree yielding the string "$((a(a()))(a)())$".

   (b) (5 points) Is this grammar ambiguous? Explain.

   Answer: Here is a parse tree that does not use the second L-production, that is, it does not use $L \rightarrow SL$. There are other parse trees for the same string that do use this production.



   The grammar is ambiguous, because another string, "(a)", is yielded by two different parse trees (equivalently, two different leftmost derivations):

$$S \Rightarrow (L) \Rightarrow (SL) \Rightarrow (S) \Rightarrow (a)$$
$$S \Rightarrow (L) \Rightarrow (LS) \Rightarrow (S) \Rightarrow (a)$$

5. (25 points total) Recall one of our standard, simplified grammars for arithmetic expressions, given in yacc/bison form (`expr` is the start symbol):

```
expr
    : term
    | expr '+' term
    | expr '-' term
    ;
term
    : factor
    | term '*' factor
    | term '/' factor
    ;
factor
    : CONST
    | VAR
    | '(' expr ')'
    ;
```

   (a) (15 points) Alter to the grammar so that expressions can involve named function calls. A named function call has the form $f(\cdots)$, where f is a name (VAR) and $\cdots$ is a comma-separated list of zero or more expressions (the *arguments*). Each argument can be an arbitrary expression. For example, the following are grammatically correct function call expressions:

```
foo()
foo(bar)
bar(bat,fat)
bat(bat+foo(6),bat-foo*bar)
...
```

   Your altered grammar should still be unambiguous.

   (b) (10 points) Add semantic actions to the *original, unaltered* grammar above so that the root of the parse tree contains as its attribute the *sum* of all constant values appearing in the expression. You can assume that all constant values are integers. Also assume that the lexical scanner sets the attribute of each `CONST` token to its value. If there are no constants appearing in the expression, then the root attribute should be 0.

   For example, if the input is "`3*(42+z)+6`", then the root should have attribute value 51; if the input is "`x-y+z`", then the root should have attribute 0. Make your actions as simple as possible, and do not use any library routines. Any omitted actions are assumed to be the default action `{$$ = $1;}`.

Answer:

(a) We add four productions to the grammar above, starting with factor (the expr and term productions are unchanged):

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

    ;

args
    : expr
    | args ',' expr
    ;
```

(b) Here is a correct answer:

```
expr
    : term
    | expr '+' term     { $$ = $1 + $3; }
    | expr '-' term     { $$ = $1 + $3; }
    ;
term
    : factor
    | term '*' factor   { $$ = $1 + $3; }
    | term '/' factor   { $$ = $1 + $3; }
    ;
factor
    : CONST
    | VAR               { $$ = 0; }
    | '(' expr ')'      { $$ = $2; }
    ;
```

Any missing actions are default actions (`$$ = $1;`). The first four actions are identical.