1. Write a sequence of lex-style regular expressions for each of the following sets

\[ A = \{ w \mid w \text{ is over the alphabet \{a,b\} and all a’s occur before any b’s} \} \]
\[ a^*b^* \]

\[ B = \{ x \mid x \text{ is an alphanumeric string that starts with an alphabetic character} \} \]
\[ [a-zA-Z]([a-zA-Z0-9]*) \]

\[ C = \{ \text{list} \mid \text{list consists of one or more strings from B separated by commas} \} \]
\[ [a-zA-Z]([a-zA-Z0-9]*),[a-zA-Z]([a-zA-Z0-9]*)* \]

\[ D = \{ y \mid y \text{ is a numeric string consisting of 1 or more digits followed by an optional decimal point and 0 or more additional digits} \} \]
\[ ([0-9])+(\.(0-9)*)? \]

2. Consider the language
\[ L = \{ a^i b^j \mid i \geq 0, j > i \} \]
The following is a grammar for L. Show this is an ambiguous grammar.

\[ S \rightarrow aSb \mid Sb \mid b \]
Using Leftmost Derivations
\[ S \Rightarrow aSb \Rightarrow aSbb \Rightarrow abbb \]
\[ S \Rightarrow Sb \Rightarrow aSbb \Rightarrow abbb \]

3. Write an unambiguous grammar that leads to correct parse trees for the language consisting of expressions involving the operand ID and the operators described below.

<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>ASSOCIATIVITY</th>
<th>PRECEDENCE</th>
<th>BINARY/UNARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>@, !</td>
<td>left to right</td>
<td>High (3)</td>
<td>Binary</td>
</tr>
<tr>
<td>^</td>
<td>right to left</td>
<td>Medium (2)</td>
<td>Unary</td>
</tr>
<tr>
<td>&amp;</td>
<td>right to left</td>
<td>Low (1)</td>
<td>Binary</td>
</tr>
</tbody>
</table>

Parentheses are also allowed, with their usual interpretation.

\[ \text{Expr1} \rightarrow \text{Expr2} \& \text{Expr1} \mid \text{Expr2} \]
\[ \text{Expr2} \rightarrow ^\text{Expr2} \mid \text{Expr3} \]
\[ \text{Expr3} \rightarrow \text{Expr3} @ \text{Expr4} \mid \text{Expr3} ! \text{Expr4} \mid \text{Expr4} \]
\[ \text{Expr4} \rightarrow \text{ID} \mid (\text{Expr1}) \]

Present a parse tree, using your grammar, for the string

\[ ^\text{ID} \& ^\text{ID} \]
\[ \text{Expr1} \Rightarrow \text{Expr2} \& \text{Expr1} \Rightarrow ^\text{Expr2} \& \text{Expr1} \Rightarrow ^\text{Expr3} \& \text{Expr1} \Rightarrow ^\text{Expr3} ! \text{Expr4} \& \text{Expr1} \]
\[ ^\text{Expr3} @ \text{Expr4} \mid \text{Expr4} \& \text{Expr1} \Rightarrow ^\text{Expr4} @ \text{Expr4} \mid \text{Expr4} \& \text{Expr1} \]
\[ ^\text{ID} @ \text{Expr4} \mid \text{Expr4} \& \text{Expr1} \Rightarrow ^\text{ID} @ \text{ID} \mid \text{Expr4} \& \text{Expr1} \Rightarrow ^\text{ID} @ \text{ID} ! \text{ID} \& \text{Expr1} \]
\[ ^\text{ID} @ \text{ID} \mid \text{ID} \& \text{Expr2} \Rightarrow ^\text{ID} @ \text{ID} \mid \text{ID} \& ^\text{Expr2} \Rightarrow ^\text{ID} @ \text{ID} \mid \text{ID} \& ^\text{Expr3} \]
\[ ^\text{ID} @ \text{ID} \mid \text{ID} \& ^\text{Expr4} \Rightarrow ^\text{ID} @ \text{ID} \mid \text{ID} \& ^\text{ID} \]

I will show parse tree in class
4. Consider the Pascal style FOR statement, which has the following description:

\[
\text{for_stmt} \rightarrow \text{FOR index := expression TO expression DO statement} \\
| \text{FOR index := expression DOWNTO expression DO statement}
\]

Assume procedures have already been written to do a recursive descent parse of expressions, expression( ) and of statements, statement( ). Write the procedure, for_statement( ), needed to do a recursive descent parse of a FOR statement. Assume token are returned by a procedure token( ) which sets a global variable SY. Assume SY = FORSY at start. Assume SY = IDENT on an identifier, SY = ASSIGN on ":=". TOSY on "TO", DOWNSY on "DOWNTO" and DOSY on "DO".

```c
void for_statement( ) {
    token( );
    if (sy == IDENT) {
        token( );
        if (sy == ASSIGN) {
            token( );
            expression( );
            if (sy in [TOSY,DOWNTOSY]) {
                token( );
                expression( );
                if (sy == DOSY) {
                    token( );
                    statement( );
                } else error( );
            }
            else error( );
        }
        else error( );
    }
    else error( );
}
```

5. Consider a grammar $G = (\{\text{Stmt, Exp, Var}\}, \{\text{WHILE, DO, BASIC, =, >, ID, [ , ]}\}, \text{Stmt, P})$, where $P$ is:

   1. $\text{Stmt} \rightarrow \text{WHILE Exp DO Stmt | BASIC}$
   2. $\text{Exp} \rightarrow \text{Var Test}$
   3. $\text{Test} \rightarrow = \text{Var} | > \text{Var}$
   4. $\text{Var} \rightarrow \text{ID [ NUM ] | ID}$.

Compute the FIRST and FOLLOW sets for this grammar's non-terminals. Produce the LL(1) parsing table based on these.

$\text{FIRST(Stmt)} = \{\text{WHILE, BASIC}\}$

$\text{FOLLOW(Stmt)} = \{$$

$\text{FIRST(Exp)} = \{\text{ID}\}$

$\text{FOLLOW(Exp)} = \{\text{DO}\}$

$\text{FIRST(Test)} = \{\text{=}", ", \text{>"}\}$

$\text{FOLLOW(Test)} = \{\text{DO}\}$

$\text{First(Var)} = \{\text{ID}\}$

$\text{FOLLOW(Var)} = \{\text{DO}\}$

<table>
<thead>
<tr>
<th></th>
<th>WHILE</th>
<th>DO</th>
<th>BASIC</th>
<th>=</th>
<th>&gt;</th>
<th>ID</th>
<th>[</th>
<th>]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stmt</td>
<td>1a</td>
<td>1b</td>
<td></td>
<td></td>
<td></td>
<td>ID</td>
<td>[</td>
<td>]</td>
</tr>
<tr>
<td>Exp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Test</td>
<td></td>
<td></td>
<td></td>
<td>3a</td>
<td>3b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Var</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4a, 4b</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

As you can see this is NOT LL91). However, just using left factoring will resolve this. Why and how?
6. The following grammar contains occurrences of left recursion. Rewrite it so that there is no left recursion. Once this is done, use left factoring to remove right hand sides that have common prefixes.

```
stmt \rightarrow stmt SEMICOLON stmt
    \mid stmt QMARK stmt COLON stmt
    \mid stmt QMARK stmt
    \mid stmt
simple_stmt \rightarrow VAR EQUALS VAR PLUS VAR
    \mid VAR EQUALS VAR TIMES VAR
```

Here SEMICOLON, QMARK, COLON, VAR, EQUALS, PLUS and TIMES are terminals.

**Remove left recursion**
```
stmt \rightarrow simple_stmt rest
rest \rightarrow SEMICOLON simple_stmt rest
    \mid QMARK stmt COLON stmt rest
    \mid QMARK stmt rest
    \mid \epsilon
simple_stmt \rightarrow VAR EQUALS VAR PLUS VAR
    \mid VAR EQUALS VAR TIMES VAR
```

**Perform left factoring**
```
rest \rightarrow SEMICOLON simple_stmt rest
    \mid QMARK stmt qmend
    \mid \epsilon
qmend \rightarrow COLON simple_stmt rest
    \mid rest
```

```
simple_stmt \rightarrow VAR EQUALS VAR opPart
oppart \rightarrow PLUS VAR
    \mid TIMES VAR
```

Now, rewrite the original grammar in the notations of Extended BNF.

```
stmt \rightarrow simple_stmt \{(SEMICOLON | QMARK [simple_stmt COLON] ) simple_stmt\}
simple_stmt \rightarrow VAR EQUALS VAR [PLUS | TIMES] VAR
```
7. Consider the following grammar for statements.

\[
\begin{align*}
\text{Stmt} & \rightarrow \text{WHILE Exp DO Stmt | Exp} \\
\text{Exp} & \rightarrow \text{Var Test} \\
\text{Test} & \rightarrow = \text{Var | > Var} \\
\text{Var} & \rightarrow \text{ID [NUM] | ID}
\end{align*}
\]

Show the contents of the stack at every step of a top-down predictive parse of the string. Be sure to show what remains of the input string at every stage as well.

WHILE ID > ID[NUM] DO ID = ID $

The stack starts with two items. On the bottom is a $, signifying end of input and on the top is the non-terminal Stmt.

\[
\begin{align*}
\text{Stack} = \text{Stmt }$ & \quad \text{Input} = \text{WHILE ID > ID[NUM] DO ID = ID }$
\text{Stack} = \text{WHILE Exp DO Stmt }$ & \quad \text{Input} = \text{WHILE ID > ID[NUM] DO ID = ID }$
\text{Stack} = \text{Exp DO Stmt }$ & \quad \text{Input} = \text{ID > ID[NUM] DO ID = ID }$
\text{Stack} = \text{Var Test DO Stmt }$ & \quad \text{Input} = \text{ID > ID[NUM] DO ID = ID }$
\text{Stack} = \text{ID Test DO Stmt }$ & \quad \text{Input} = \text{ID > ID[NUM] DO ID = ID }$
\text{Stack} = \text{Test DO Stmt }$ & \quad \text{Input} = \text{ID[NUM] DO ID = ID }$
\text{Stack} = \text{> Var DO Stmt }$ & \quad \text{Input} = \text{ID[NUM] DO ID = ID }$
\text{Stack} = \text{Var DO Stmt }$ & \quad \text{Input} = \text{ID[NUM] DO ID = ID }$
\text{Stack} = \text{ID[NUM] DO Stmt }$ & \quad \text{Input} = \text{ID[NUM] DO ID = ID }$
\text{Stack} = \text{[NUM] DO Stmt }$ & \quad \text{Input} = \text{ID[NUM] DO ID = ID }$
\text{Stack} = \text{ID DO Stmt }$ & \quad \text{Input} = \text{DO ID = ID }$
\text{Stack} = \text{DO Stmt }$ & \quad \text{Input} = \text{DO ID = ID }$
\text{Stack} = \text{Stmt }$ & \quad \text{Input} = \text{ID = ID }$
\text{Stack} = \text{Exp }$ & \quad \text{Input} = \text{ID = ID }$
\text{Stack} = \text{Var Test }$ & \quad \text{Input} = \text{ID = ID }$
\text{Stack} = \text{ID Test }$ & \quad \text{Input} = \text{ID = ID }$
\text{Stack} = \text{Test }$ & \quad \text{Input} = \text{ID }$
\text{Stack} = \text{Var }$ & \quad \text{Input} = \text{ID }$
\text{Stack} = \text{ID }$ & \quad \text{Input} = \text{ID }$
\text{Stack} = \text{$ }$ & \quad \text{Input} = \text{ }$
\text{Stack} = \text{=$ }$ & \quad \text{Input} = \text{ }$
\end{align*}
\]