COP3402 Spring 2011 Sample Problems

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

A = { w | w is over the alphabet {a,b} and all a's occur before any b's }
a*b*

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

C = { list | 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 | y 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

$L = \{ a^i b^j \mid i>=0, j>i \}$

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.

OPERATOR	ASSOCIATIVITY	PRECEDENCE	BINARY/UNARY
@, !	left to right	High (3)	Binary
^	right to left	Medium (2)	Unary
&	right to left	Low (1)	Binary

Parentheses are also allowed, with their usual interpretation.

Expr1 → Expr2 & Expr1 | Expr2
Expr2 → ^Expr2 | Expr3
Expr3 → Expr3 @ Expr4 | Expr3 ! Expr4 | Expr4
Expr4 → ID | (Expr1)

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

^ ID @ ID ! ID & ^ ID
Expr1 ⇒ Expr2 & Expr1 ⇒ ^Expr2 & Expr1 ⇒ ^Expr3 & Expr1 ⇒ ^Expr3 ! Expr4 & Expr1
⇒ ^Expr3 @ Expr4 ! Expr4 & Expr1 ⇒ ^Expr4 @ Expr4 ! Expr4 & Expr1
⇒ ^ID @ Expr4 ! Expr4 & Expr1 ⇒ ^ID @ ID ! Expr4 & Expr1 ⇒ ^ID @ ID ! ID & Expr1
⇒ ^ID @ ID ! ID & Expr2 ⇒ ^ID @ ID ! ID & ^Expr2 ⇒ ^ID @ ID ! ID & ^Expr3
⇒ ^ID @ ID ! ID & ^Expr4 ⇒ ^ID @ ID ! ID & ^ID
I will show parse tree in class

4. Consider the Pascal style FOR statement, which has the following description:

for_stmt → FOR index := expression TO expression DO statement

| 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".

```
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 = ({Stmt, Exp, Var}, {WHILE, DO, BASIC, = , > , ID, [,] }, Stmt, P), where P is:

1. Stmt → WHILE Exp DO Stmt | BASIC
2. Exp → Var Test
3. Test → = Var | > Var
4. Var → ID [NUM] | ID.

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

FIRST(Stmt) = {WHILE, BASIC}

FIRST(Exp) = {ID}

FIRST(Test) = {'=", ">"}

First(Var) = {ID}

FOLLOW(Stmt) = {$}

FOLLOW(Exp) = {DO}

FOLLOW(Test) = {DO}

FOLLOW(Var) = {DO}

	WHILE	DO	BASIC	=	>	ID	[	]
Stmt	1a		1b					
Exp						2		
Test				3a	3b			
Var						4a, 4b		

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         →   stmt SEMICOLON simple_stmt
             |   stmt QMARK simple_stmt COLON simple_stmt
             |   stmt QMARK simple_stmt
             |   simple_stmt
simple_stmt  →   VAR EQUALS VAR PLUS VAR
             |   VAR EQUALS VAR TIMES VAR
```

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

Remove left recursion

```
stmt         →  simple_stmt rest
rest         →  SEMICOLON simple_stmt rest
             |  QMARK simple_stmt COLON simple_stmt rest
             |  QMARK simple_stmt rest
             |  ε
simple_stmt  →  VAR EQUALS VAR PLUS VAR
             |  VAR EQUALS VAR TIMES VAR
```

Perform left factoring

```
rest         →  SEMICOLON simple_stmt rest
             |  QMARK simple_stmt qmend
             |  ε
qmend        →  COLON simple_stmt rest
             |  rest


simple_stmt  →  VAR EQUALS VAR opPart
oppart       →  PLUS VAR
             |  TIMES VAR
```

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

```
stmt         →  simple_stmt {(SEMICOLON | QMARK [simple_stmt COLON] ) simple_stmt}
simple_stmt  →  VAR EQUALS VAR [PLUS | TIMES] VAR
```

7. Consider the following grammar for statements.

Stmt	→	WHILE Exp DO Stmt | Exp
Exp	→	Var Test
Test	→	= Var | > Var
Var	→	ID [NUM] | ID

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.

Stack = Stmt $	Input = WHILE ID > ID[NUM] DO ID = ID $
Stack = WHILE Exp DO Stmt $	Input = WHILE ID > ID[NUM] DO ID = ID $
Stack = Exp DO Stmt $	Input = ID > ID[NUM] DO ID = ID $
Stack = Var Test DO Stmt $	Input = ID > ID[NUM] DO ID = ID $
Stack = ID Test DO Stmt $	Input = ID > ID[NUM] DO ID = ID $
Stack = Test DO Stmt $	Input = > ID[NUM] DO ID = ID $
Stack = > Var DO Stmt $	Input = > ID[NUM] DO ID = ID $
Stack = Var DO Stmt $	Input = ID[NUM] DO ID = ID $
Stack = ID[NUM] DO Stmt $	Input = ID[NUM] DO ID = ID $
Stack = [NUM] DO Stmt $	Input = [NUM] DO ID = ID $
Stack = NUM] DO Stmt $	Input = NUM] DO ID = ID $
Stack =] DO Stmt $	Input =] DO ID = ID $
Stack = DO Stmt $	Input =DO ID = ID $
Stack = Stmt $	Input = ID = ID $
Stack = Exp $	Input = ID = ID $
Stack = Var Test $	Input = ID = ID $
Stack = ID Test $	Input = ID = ID $
Stack = Test $	Input = = ID $
Stack = = Var $	Input = = ID $
Stack = Var $	Input = ID $
Stack = ID $	Input = ID $
Stack = $	Input = $
Stack =	Input =