1) The regular language equivalent to `(a|b|c)*a(a|b|c)*`

```
1. (c|b|a)*(c|b|a)*
2. (a|b|c)*(ab|bc|a)(a|b|c)*
3. (c|b|a)*a(c|b|a)*
4. (a|b|c)*(a|b|c)(a|b|c)*
```

Ans: Only 3. *equivalence* of two languages implies that the languages have exactly the same set of strings.

2) LL Parsers: Consider the Grammar:

```
1.S-> A$
2.A-> xBC
3.A->CB
4.B->yB
5.B->λ
6.C->x
```

a) What are the terminals and non-terminals of this language?
b) Describe the strings generated by this language with the help of a regular expression
c) What sequence of productions are applied to derive the string xyyx$? Draw the parse tree.
d) Compute the first and follow sets for all non-terminals.
e) Compute the predict set for each productions
f) Is this grammar LL(1)? If not, why not?

Ans: a) Terminals = {x,y,$} Non-Terminals = {S, A, B, C}

b) All strings generated by this grammar are through `S->A$.`

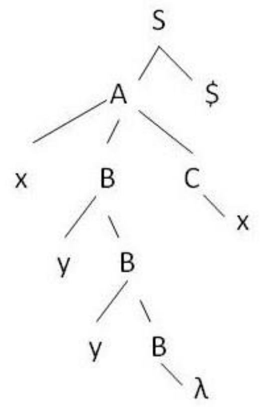
`B->yB` and `B->λ` tell us that B generates all string containing zero or more `y (y*).` `A->xBC` and `C->x` tell us that all strings start and end with x when we apply this production `(xy*x).` `A->CB` gives us `xy*.` So, the regular expression is `xy*x + xy*`

c) `xyyx$` is derived through:

```
S->A$,
->xBC$,           (applying A->xBC)
```

```
->xyBC$           (applying B->yB)
->xyyBC$          (applying B->yB)
->xyyC$           (applying B-> λ)
->xyyx$           (applying C->x)
```

Parse tree:



a) First sets: Recall that first sets can contain λ (slide 7, week 5)

```
First(S) = {x}            First(S) ⊇ First(A)
First(A) = {x}            First(A) ⊇ First(xBC) = {x} and
                          First(A) ⊇ First(CB) = First(C) = {x}
First(B) = {y,λ}          First(B) ⊇{λ} and
                          First(B) ⊇ First(yB) = {y}
First(C) = {x}
```

Follow sets: Recall follow sets can't have λ . But can have $.
and an additional rule: if A->xB and B derives λ, then follow(B) ⊇ follow (A).

```
Follow(S) = {} (always)
Follow(A) = {$}
Follow(B) = {x,$}   Follow(B) ⊇ Follow(A) and
                    Follow(B) ⊇ First(C)

Follow(C) = {y,$}   Follow(C) ⊇ Follow(A) and

                    Follow(C) ⊇ First(B)-λ = {y}
```

b) Predict sets:

```
Predict(1) = {x}
Predict(2) = {x}
Predict(3) = {x}
Predict(4) = {y}
Predict(5) = {x,$}
Predict(6) = {x}
```

c) This grammar is not LL(1) because of the conflict (presence of multiple rules in a cell) in the parse table as indicated:

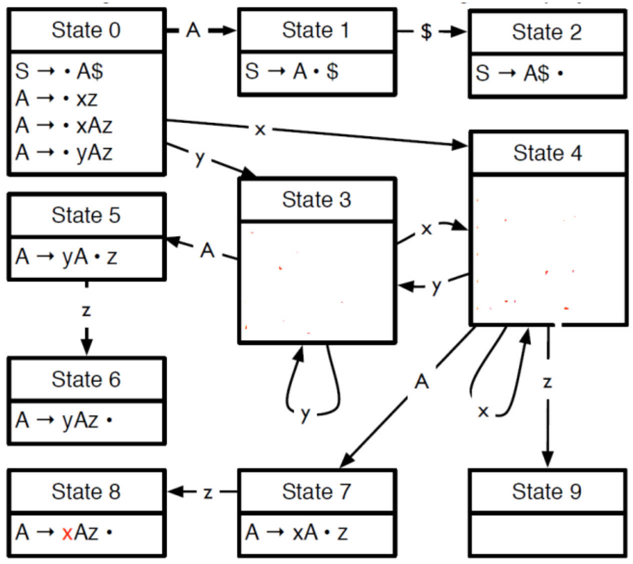
	x	y	$
S	1		
A	2,3		
B	5	4	5
C	6		

This table tells us which production to apply (left most derivation) based on the next lookup symbol. E.g. if the next lookup symbol is x, we expand using either rule 2 (A->xBC), or rule 3 (A->CB) when the left-most non terminal is A. In this situation, when the left-most non-terminal is A, because we can apply either rule 2 or rule 3 as indicated by the parse table, there is a conflict.

How is this table constructed? based on Predict sets. (P1) = {x} => mark 1 in the cell indicated by row= LHS(P1) and column= x. The table will have one row for every non-terminal and one column for every terminal.

3) LR(0) Parsers:

a. Fill in the missing information (states 3, 4, and 9)



State 3: { A->y.Az, A->.xz, A->.xAz, A->.yAz} (we have to add more items because . appears before a non-terminal in A->y.Az)

State 4: {A->x.z, A->x.Az, A->.xz, A->.xAz, A->.yAz} (after adding A->x.z and A->x.Az, we have to add more items because . appears before a non-terminal in A->x.Az)

State 9: {A->xz.}

b. What are the shift states and what are the reduce states?
   State 9, 8, 6, and 2 are reduce states. 2 is generally called accept state.
   All others are Shift states

c. Is this grammar an LR(0) grammar?
   Yes, because there are no SR/RR conflicts.

d. For the following sub-questions, use the CFSM you built in the previous question. Each question will provide the state of the parser in mid-parse, giving the state stack (most recent state on the right) and the next token. For each question, give the action the parser will take next, using the format “Shift X” for shift actions (where X is the state being shifted to) and “Reduce R, goto X” for reduce actions (where R is the rule being reduced, and X is the state the parser winds up in after finishing the reduction). Also provide the new state stack.

   i. State stack: 0 3 3 4. Next token: x
      Top of the stack contains 4. 4 is a shift state. State 4, on transition using x (next token) goes to State 4. So, Parser action is “Shift 4”. Parse stack state: 0 3 3 4 4

   ii. State stack: 0 1 2. Next token: none
      Accept

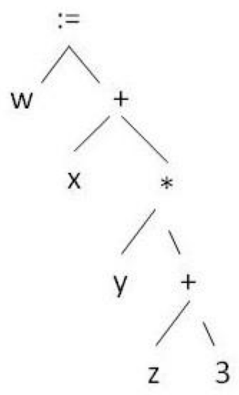
   iii. State stack: 0 3 4 9. Next token: z
      9 is a reduce state. We have to replace a set of symbols that are on top of the stack. These symbols correspond to the RHS of the production using which you are trying to reduce (A->xz). There are 2 symbols in this production, so, pop 2 symbols off the stack: 0 3. Now, top of the stack contains 3 i.e. you are in state 3. You are trying to replace xz with A. So, if you were to see an A in state 3, what would you do? Goto state 5. So, the answer:
      “Reduce A->xz, Goto 5”
      Parse stack: 0 3 5

   iv. State stack: 0 4 4 7 8. Next token: z
      “Reduce A->xAz, Goto 7”
      Parse stack: 0 4 7

v. State stack: 0 4 3. Next token: z
"Error"

4) a) Draw an AST for the assignment statement `w := x + y * (z + 3)`
b) Give one advantage to generating ASTs before producing target code, rather than producing target code directly.

Ans: a)



b) If our language allowed assignment of integer variables with only integer variables or integer valued results of expressions (no type conversions), then we could traverse the AST and check for type mismatch. This would be impossible to do if code was produced directly using semantic routines.