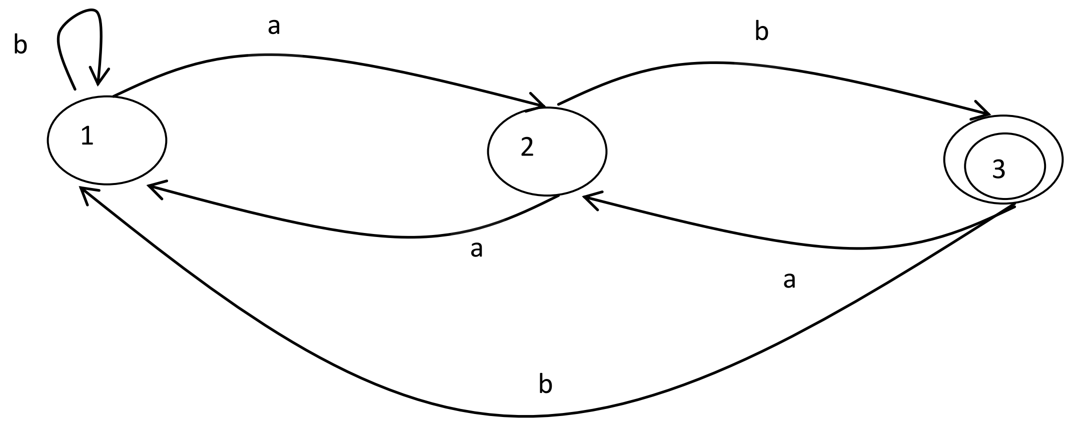
1) The regular language equivalent to `(a+b+c)*a(a+b+c)*`

```
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. In class, we discussed language that produces a set of strings, which is the super set of strings produced by the given regular expression. Note that this would not be an *equivalent* regular language.

2)



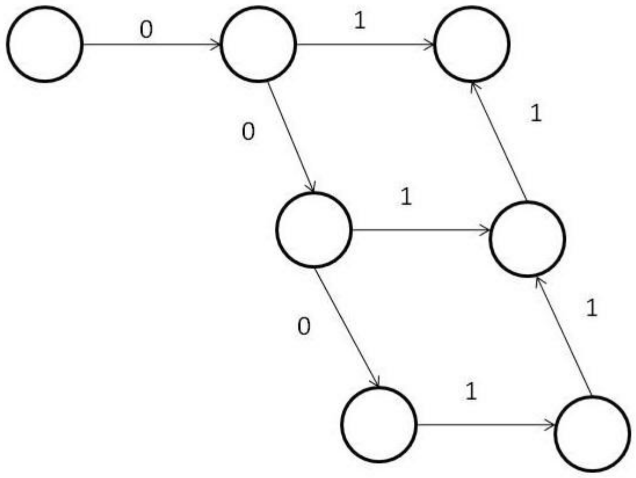
The regular language represented by this FA is:

```
1. (a+b)*
2. (a*+ab)(a+b)*ab
3. a*+(ab)*+(aab)*+(aaa*b)*
4. (a+b)*ab
```

Ans: None of these. In class, we discussed if this FA would represent a language accepting strings ending with `'ab'` (options 2 and 4). It turned out that we got counter-examples with those options. If we want to get an FA that accepts a string ending with **ab,** one modification to this FA would be to draw a an edge from Node 2 to Node 2 on 'a' transition (instead of Node 2 to Node 1 as it is now).

3) Let S_i be the string consisting of i 0's followed by i 1's. Define the language L_n=`{Si|1≤i≤n}`. For example, L_3=`{01,0011,000111}`. Can you represent L_n as a FA? If so, what is the smallest number of states needed for a DFA that recognizes L_n?

Ans: No, L_n can't be represented as a FA since it involves counting. However, if the value of n is given, we can draw an FA for L_n. In this case, we would represent L_n with a minimum of 2n + 1 states as shown below for L_3:



4) Given the following lexical specification

`a(ba)*`

`b*(ab)*`

abd

d+

`i)` dddabbabab is tokenized as ddd / a / bbabab
`ii)` ababddababa is tokenized as ab / abd / d / ababa

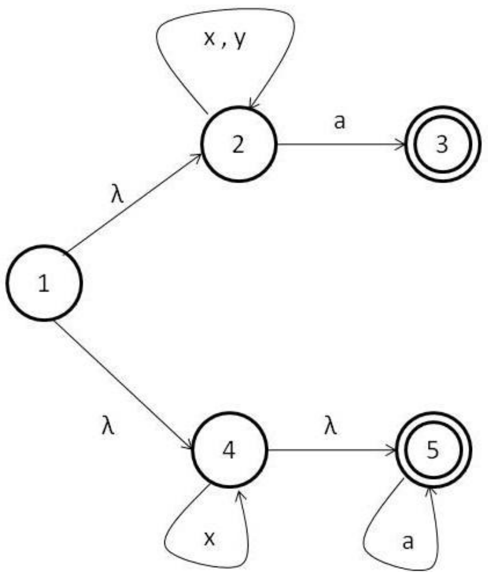
a) i and ii correct b) only i correct c) only ii correct d) neither i nor ii correct

Ans: d. Neither i nor ii is correct. According to the rules of tokenization, as big a contiguous chunk as possible is matched while evaluating regular expressions in an order starting from the first one. So, according to this rule, dddabbabab would be tokenized as ddd / ab / babab and ababddababa would be tokenized as abab / dd / ababa

5) a) Draw an NFA for the regular expression ((x+y)* a) + (x* a*)
b) Give the equivalent DFA for the NFA: draw state transition table

c) Reduce the DFA from step2 if possible.

Ans: a) Several NFAs possible. One NFA is shown below.



b) One way of converting the NFA to DFA is through state transition diagram. Initial set of states reachable from State 1 through lambda transitions: 1, 2, 4, 5. So, create a state called 1245 and look for transitions from each of the constituent states. E.g. from 1245, look for a transition on a from each of the constituent states 1, 2, 4, and 5. We see that states 3 (1->3, 2->3), and 5 (1->4->5) are reachable on a transition from State 1245. Mark the state 1245 as a final state if at least one of the constituent states is a final state. For every new state seen (e.g. States 35, 245, and 2) , create a row in transition table.

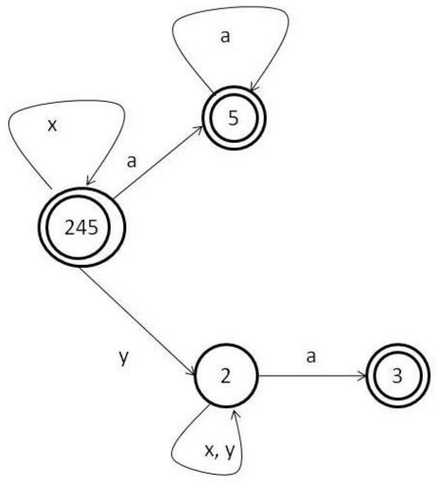
State	a	x	y	IsFinal
1,2,4,5	3,5	2,4,5	2	Yes
3,5	5	-	-	Yes
2,4,5	3,5	2,4,5	2	Yes
2	3	2	2	No
5	5	-	-	Yes
3	-	-	-	Yes

c) We can see that rows 1 and 3 have same transitions. So, they can be merged into one state. Similarly, Rows 2 and 5 can be merged into one state. Note that we can merge states only if they have same transitions and they are final states.

After merging, we get:

State	a	x	y	IsFinal
245	5	245	2	Yes
5	5	-	-	Yes
2	3	2	2	No
3	-	-	-	Yes

Graphical representation of this is a DFA:



6) 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}

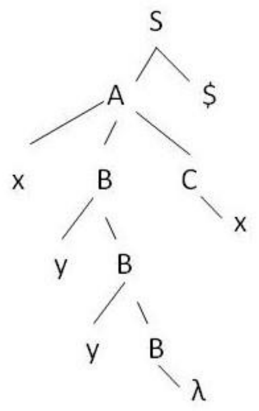
b) All strings generated by this grammar are through `S->A$.`

`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:



d) 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 $.
(reference slides 7, 8, week 5) 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}
```

e) Predict sets (slide 9, week 5):

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

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

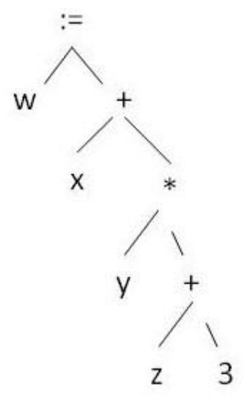
	x	y	$
S	1		
A	2,3		
B	5	4	5
C	6		

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->x`BC), 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.

7) a) Draw an AST for the assignment statement `w := x + y * (z + 3)`
b) Give one advantage to generating ASTs before producing code, rather than producing code directly.
c) Give one possible three address code that would be generated for the above tree. Use the following instructions: LD A, T loads from variable A into temporary T. OP T1, T2, T3 performs T3 = T1 OP T2. ST T, A stores from variable A into temporary T. OP are ADD, MUL.

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.

c)

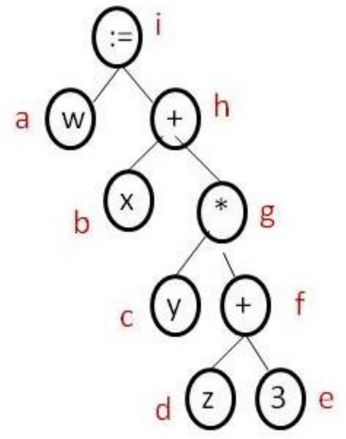
```
LD x, T1
LD y, T2
LD z, T3
ADD T3, 3, T4
MUL T2, T4, T5
ADD T1, T5, T6
ST T6, w
```

Comments: the above code generation scheme is done by manual inspection of AST. If we were to generate this code automatically using semantic routines (or even by just traversing AST --- Yes, this is possible.), then we would get the following code:

```
LD x, T6
LD y, T4
LD z, T2
ADD T2, 3, T1
MUL T4, T1, T3
ADD T6, T4, T5
ST T5, w
```

Note that in the code generation, we defer: i) loads until we know whether a variable is used as a source or a destination for an assignment (l value or rvalue. Slide 24, week 6),

ii) generating the temporary for the result of an expression before generating any code for the expression, including any loads that might need to happen (Slide 25, week 6). The AST nodes are traversed in post-order, visiting the left child before the right child i.e., in alphabetical order of the AST shown below with nodes marked (Slide 19, week 6):



Node a:
Temp: w
Type: l-value
Code: --

Node b:
Temp: x
Type: l-value
Code: --

Node c:
Temp: y
Type: l-value
Code: --

Node d:
Temp: z
Type: l-value
Code: --

Node e:
Temp: 3
Type: constant

Code: --

Node f:
Temp: T1
Type: r-value
Code:
```
LD z, T2
ADD T2, 3, T1
```

Node g:
Temp: T3
Type: r-value
Code:
```
LD y,
T4 LD z,
T2 ADD T2, 3, T1
MUL T4, T1, T3
```

Node h:
Temp: T5
Type: r-value
Code:
```
LD x, T6
LD y, T4
LD z, T2
ADD T2, 3, T1
MUL T4, T1, T3
ADD T6, T4, T5
```

Node i:
Temp: N/A
Type: N/A
Code:
```
LD x, T6
LD y, T4
LD z, T2
ADD T2, 3, T1
MUL T4, T1, T3
ADD T6, T4, T5
ST T5, w
```

Note that node i represents a complete statement (:=), and hence does not have a temporary holding its value.