Q1: Ambiguous Grammar? [10]

$S \rightarrow S\,S\,* \mid S\,S\,+ \mid a$

a. Postfix expression consisting of terminals a, plus sign and multiplication sign
b. No, this grammar unambiguous. This is because the precedence order of + and * is fixed leading to no ambiguity.
c. SLR parse table:
   Augmenting:
   1. S' -> S
   2. S -> SS*
   3. S -> SS+
   4. S -> a

   Non terminals : {S} Terminals: {a, +, *}
   Follow(S) = {a, + , * , $}
   Items:
   1. I_0 -> { (S' -> S) }
      S' -> .S
      S -> .SS*
      S -> .SS+
      S -> .a
   2. I_1 : goto(I_0, S)
      S' -> S.
      S -> S.S*
      S -> S.S+
      S -> .SS*
      S -> .SS+
      S -> .a
   3. I_2 : goto(I_1, S)
      S -> SS.*
      S -> SS.+
      S -> S.S*
      S -> S.S+
      S -> .SS*
      S -> .SS+
      S -> .a
   4. I_3 : goto(I_1, a)
      S -> a.
   5. I_4 : goto(I_2, +)
      S -> SS+.
   6. I_5 : goto(I_2, *)
      S -> SS*.

   goto(I_0, a) = I_3 , goto(I_1, a) = I_3, goto(I_2, a) = I_3 , goto(I_2, +) = I_4 , , goto(I_2, *) = I_5, goto(I_0, S) = I_1 , goto(I_1, S) = I_2 , goto(I_2, S) = I_1, goto(I_4, $) = accept , goto($I_5$, $) = accept

SLR Parse Table:

	+	*	a	$	S
0			s3		s1
1			s3	Accept	s2
2	s4	s5	s3		s1
3	r4	r4	r4	r4	
4	r3	r3	r3	r3	
5	r2	r2	r2	r2	

Q2: Parse Table [10]

Augmenting:

1. S'-> S
2. S -> Ma
3. S -> bMc
4. S -> dc
5. S -> bMa
6. M -> d
7. M -> ϵ; *# ϵ; is the empty string*

First(S) = { b, d, a, ϵ}
First(M) = { d , ϵ }
Follow(S') = { $ }
Follow(S) = { $ }
Follow(M) = { a, c }
Items:

1. I_0 : { (S' -> S) }
   S'-> .S
   S -> .Ma
   S -> .bMc
   S -> .dc
   S -> .bMa
   M -> .d
   M -> .
2. I_1 : goto(I_0, S)
   S' -> S.
3. I_2 : goto(I_0, M)
   S -> M.a
4. I_3 : goto(I_0, b)
   S -> b.Mc
   S -> b.Ma
   M -> .d
   M -> .
5. I_4 : goto(I_0, d)
   S -> d.c
   M -> d.

6. I_5 : goto(I_2, a)
   S -> Ma.
7. I_6 : goto(I_3, M)
   S -> bM.c
   S -> bM.a
8. I_7 : goto(I_3, d)
   M -> d.
9. I_8 : goto(I_4, c)
   S -> dc.
10. I_9 : goto(I_6, c)
    S -> bMc.
11. I_{10} : goto(I_6, a)
    S -> bMa.

Reduce cases:

[I_4, a] => M -> d (r6)
[I_4, c] => M -> d (r6)
[I_5, $] => S -> Ma (r2)
[I_7, a] => M -> d (r6)
[I_7, c] => M -> d (r6)
[I_8, $] => S -> dc (r4)
[I_9, $] => S -> bMc (r3)
[I_{10}, $] => S -> bMa (r5)
[I_0, a] => M -> ϵ (r7)
[I_0, c] => M -> ϵ (r7)
[I_3, a] => M -> ϵ (r7)
[I_3, c] => M -> ϵ (r7)

Accept cases:

[I_1, $] => accept

SLR Parse Table:

	a	**b**	**c**	**d**	**$**	**S**	**M**
0	r7	s3	r7	s4		s1	s2
1					accept		
2	s5						
3	r7		r7	s7			s6
4	r6		r6, s8				
5					r2		
6	s10		s9				
7	r6		r6				
8					r4		
9					r3		
10					r5		

a. There is only one conflict state. On state 4, when the input is c, we get a shift/reduce conflict between r6 (M -> d) and s8. Thus we either remain at s4(reduce) or move to s8(shift).

b. Consider string dc :

Stack	Input	Action
0	dc$	Shift 4
0 d 4	c$	Shift/reduce conflict (shift 8/ M->d)

c. Consider string ba :

Stack	Input	Action
0	ba$	shift 3
0 b 3	a$	reduce M -> ϵ
0 b 3 M 6	a$	shift 10
0 b 3 M 6 a 10	$	reduce S -> bMa
0 S 1	$	accept

Q3: SLR vs CLR vs LALR [10]

1. S’ -> S
2. S -> id [E] := E
3. E -> E + T
4. E -> T
5. T -> T * F
6. T -> F
7. F -> (E)
8. F -> id

Items:

1. I_0 = { (S’ -> S) }
   S’ -> .S
   S -> . id [E] := E
2. I_1 : goto(I_0, S)
   S’ -> S.
3. I_2: goto(I_0, id)
   S -> id. [E] := E
4. I_3: goto(I_2, [)
   S -> id [. E] := E
   E -> .E + T
   E -> .T
   T -> .T * F
   T -> .F
   F -> .(E)
   F -> .id
5. I_4: goto(I_3, E)
   S -> id [E.] := E
   E -> E. + T
6. I_5: goto(I_3, T)
   E -> T.
   T -> T. * F
7. I_6: goto(I_3, F)
   T -> F.
8. I_7: goto(I_3, ( )
   F -> (.E)
   E -> .E + T
   E -> .T
   T -> .T * F
   T -> .F
   F -> .(E)
   F -> .id
9. I_8: goto(I_3, id)
   F -> id.
10. I_9: goto(I_4,])
    S -> id [E]. := E

11. I_{10}: goto(I_4, +)
    E -> E + . T
    T -> .T * F
    T -> .F
    F -> .(E)
    F -> .id
12. I_{11}: goto(I_5, *)
    T -> T * . F
    F -> .(E)
    F -> .id
13. I_{12}: goto(I_7, E)
    F -> (E.)
    E -> E. + T
14. I_{13}: goto(I_9, :=)
    S -> id [E] := .E
    E -> .E + T
    E -> .T
    T -> .T * F
    T -> .F
    F -> .(E)
    F -> .id
15. I_{14}: goto(I_{10}, T)
    E -> E + T.
    T -> T. * F
16. I_{15}: goto(I_{11}, F)
    T -> T * F.
17. I_{16}: goto(I_{12},))
    F -> (E).
18. I_{17}: goto(I_{13}, E)
    S -> id [E] := E.
    E -> E. + T

goto(I_7, T) = I_5, goto(I_7, F) = I_6, goto(I_7, ( ) = I_7, goto(I_7, id) = I_8, goto(I_{10}, F) = I_6, goto(I_{10}, ( ) = I_7, goto(I_{10}, id) = I_8, goto(I_{11}, ( ) = I_7, goto(I_{11}, id) = I_8, goto(I_{12}, +) = I_{10}, goto(I_{13}, T) = I_5, goto(I_{13}, F) = I_6, goto(I_{13}, ( ) = I_7, goto(I_{13}, id) = I_8, goto(I_{14} , *) = I_{11}, goto(I_{14}, +) = I_{10}

follow(S') = {$}, follow(S) = { $}, follow(E) = {], +,), $}, follow(T) = {], $, +, *,)}, follow(F) = {], +, *,), $}

first(S') = {id}, first(S) = {id}, first(E) = { id, (}, first(T) = { id, (}, first(F) = { id, (}

SLR Parse table:

	Id	[	]	:=	+	*	(	)	$	S	E	T	F
0	S2									S1			
1									Accept				
2		S3											
3	S8						S7				S4	S5	S6
4			S9		S10								
5			R4		R4	S11		R4	R4				
6			R6		R6	R6		R6	R6				
7	S8						S7				S12	S5	S6
8			R8		R8	R8		R8	R8				
9				S13									
10	S8						S7					S14	S6
11	S8						S7						S15
12					S10			S16					
13	S8						S7				S17	S5	S6
14			R3		R3	S11		R3	R3				
15			R5		R5	R5		R5	R5				
16			R7		R7	R7		R7	R7				
17					S10				R2				

CLR Parse table:

Items:

1. I0: {[S' -> .S, $]}
   S' -> .S, $
   S -> .id [E] := E, $
2. I1: goto(0, S)
   S' -> S., $
3. I2: goto(0, id)
   S -> id.[E] := E, $
4. I3: goto(2, [)
   S -> id [.E] := E, $
   E -> .E + T,]/+
   E -> .T,]/+
   T -> .T * F,]/+/*
   T -> .F,]/+/*
   F -> .(E),]/+/*
   F -> .id,]/+/*
5. I4: goto(3, E)
   S -> id [E.] := E, $
   E -> E.+ T,]/+
6. I5: goto(3, T)
   E -> T.,]/+
   T -> T.* F,]/+/*
7. I6: goto(3, F)
   T -> F.,]/+/*

8. I7: goto(3, ()
   F -> (.E),]/+/*
   E -> .E + T,)/+
   E -> .T,)/+
   T -> .T * F,)/+/*
   T -> .F,)/+/*
   F -> .(E),)/+/*
   F -> .id,)/+/*
9. I7: goto(3, id)
   F -> id.,]/+/*
10. I9: goto(4,])
    S -> id [E].:= E, $
11. I10: goto(4, +)
    E -> E +.T,]/+
    T -> .T * F,]/+/*
    T -> .F,]/+/*
    F -> .(E),]/+/*
    F -> .id,]/+/*
12. I11: goto(5, *)
    T -> T *.F,]/+/*
    F -> .(E),]/+/*
    F -> .id,]/+/*
13. I12: goto(7, E)
    F -> (E.),]/+/*
    E -> E.+ T,)/+
14. I13: goto(7, T)
    E -> T.,)/+
    T -> T.* F,)/+/*
15. I14: goto(7, F)
    T -> F.,)/+/*
16. I15: goto(7, ()
    F -> (.E),)/+/*
    E -> .E + T,)/+
    E -> .T,)/+
    T -> .T * F,)/+/*
    T -> .F,)/+/*
    F -> .(E),)/+/*
    F -> .id,)/+/*
17. I16: goto(7, id)
    F -> id.,)/+/*
18. I17: goto(9, :=)
    S -> id [E] :=.E, $
    E -> .E + T, $/+
    E -> .T, $/+
    T -> .T * F, $/+/*
    T -> .F, $/+/*
    F -> .(E), $/+/*
    F -> .id, $/+/*

19. I18: goto(10, T)
    E -> E + T.,]/+
    T -> T.* F,]/+/*
20. I19: goto(11, F)
    T -> T * F.,]/+/*
21. I20: goto(12,))
    F -> (E).,]/+/*
22. I21: goto(12, +)
    E -> E +.T,)/+
    T -> .T * F,)/+/*
    T -> .F,)/+/*
    F -> .(E),)/+/*
    F -> .id,)/+/*
23. I22: goto(13, *)
    T -> T *.F,)/+/*
    F -> .(E),)/+/*
    F -> .id,)/+/*
24. I23: goto(15, E)
    F -> (E.),)/+/*
    E -> E.+ T,)/+
25. I24: goto(17, E)
    S -> id [E] := E., $
    E -> E.+ T, $/+
26. I25: goto(17, T)
    E -> T., $/+
    T -> T.* F, $/+/*
27. I26: goto(17, F)
    T -> F., $/+/*
28. I27: goto(17, ()
    F -> (.E), $/+/*
    E -> .E + T,)/+
    E -> .T,)/+
    T -> .T * F,)/+/*
    T -> .F,)/+/*
    F -> .(E),)/+/*
    F -> .id,)/+/*
29. I28: goto(17, id)
    F -> id., $/+/*
30. I29: goto(21, T)
    E -> E + T.,)/+
    T -> T.* F,)/+/*
31. I30: goto(22,F)
    T -> T * F.,)/+/*
32. I31: goto(23,))
    F -> (E).,)/+/*
33. I32: goto(24, +)
    E -> E +.T, $/+
    T -> .T * F, $/+/*

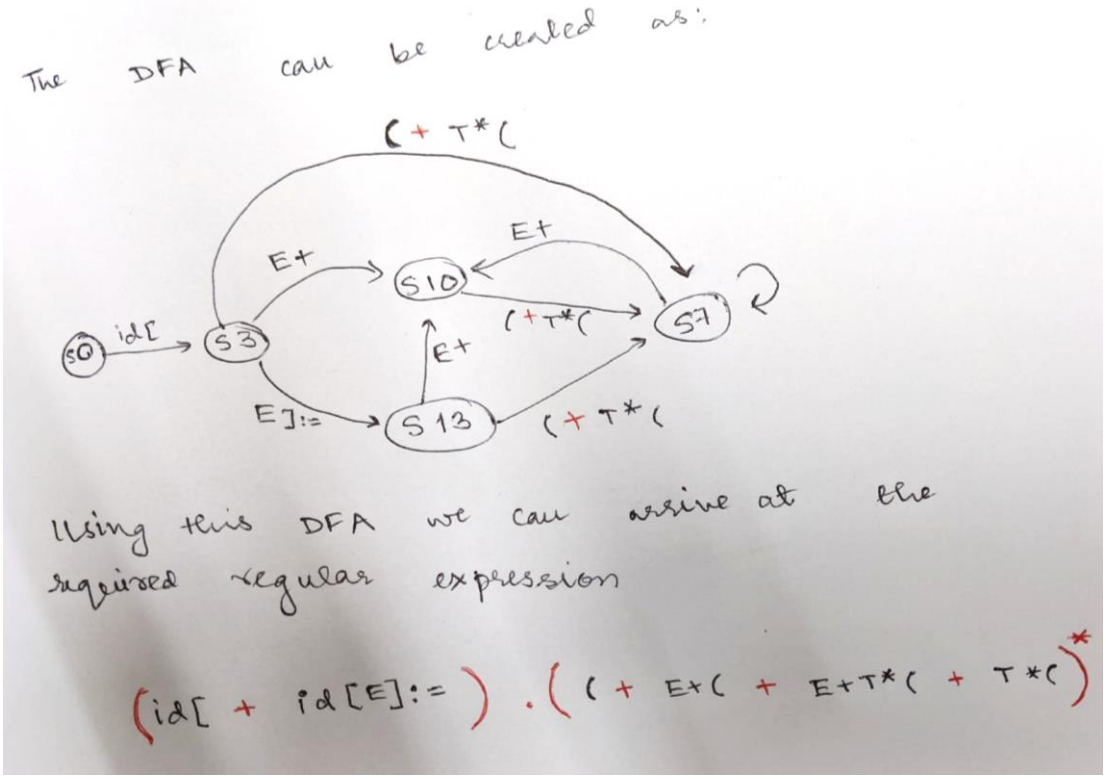
T -> .F, $/+/*
F -> .(E), $/+/*
F -> .id, $/+/*

34. I33: goto(25, *)
    T -> T *.F, $/+/*
    F -> .(E), $/+/*
    F -> .id, $/+/*
35. I34: goto(27, E)
    F -> (E.), $/+/*
    E -> E.+ T,)/+
36. I35: goto(32, T)
    E -> E + T., $/+
    T -> T.* F, $/+/*
37. I36: goto(33, F)
    T -> T * F., $/+/*
38. I37: goto(34,))
    F -> (E)., $/+/*

	Id	**[**	**]**	**:=**	**+**	*****	**(**	**)**	**$**	**S**	**E**	**T**	**F**
0	S2									1			
1									Accept				
2		S3											
3	S8						S7				S4	S5	S6
4			S9		S10								
5			R3		R3								
6			R5		R5	R5							
7	S16						S15				S12	S13	S14
8			R7		R7	R7							
9				S17									
10	S8						S7					s18	s6
11	S8						S7						s19
12					S21			S20					
13					R3	S22		R3					
14					R5	R5		R5					
15	S16						S15				s23	s13	s14
16					R7	R7		R7					
17	S28						S27				s24	s25	s26
18			R2		R2	S11							
19			R4		R4	R4							
20			R6		R6	R6							
21	S16						S15					s29	s14
22	S16						S15						S30
23					S21			S31					
24					S32				R1				
25					R3	S33			R3				
26					R5	R5			R5				
27	S16						S15				S34	S13	S14
28					R7	R7			R7				
29					R2	S22		R2					

30					R4	R4		R4					
31					R6	R6		R6					
32	S28						S27					S35	S26
33	S28						S27						S36
34					S21			S37					
35					R2	S33			R2				
36					R4	R4			R4				
37					R6	R6			R6				

a. State 7 is a self loop state. On input '(' it goes to itself. The regular expression denoting all the viable prefixes is:



b. We can take any string that doesn't belong to the grammar. As we know that a string belonging to the language of grammar will always be accepted by any parser. But the parsers behave differently for strings that are not part of the grammar. This occurs primarily because the reductions used in SLR and CLR parsers are different. In CLR only a subset of the follow set is used for reducing the expression whereas in CLR the entire follow set is used.

Take for example the string id[id*) :

SLR Parser:

Stack	Input	Action
0	Id[id*)$	shift 2
0 id 2	[id*)$	Shift 3
0 id 2 [3	id*)$	shift 8
0 id 2 [3 id 8	*)$	Reduce 8
0 id 2 [3 F	)$	S6
…	…	…

CLR Parser:

Stack	Input	Action
0	Id[id*)$	shift 2
0 id 2	[id*)$	Shift 3
0 id 2 [3	id*)$	shift 8
0 id 2 [3	*)$	Error

Here CLR gives error at step 4 when taking * as input whereas in SLR the error is encountered after step 4.

c. In LALR parser we take the union of lookahead sets leading to lesser no of states compared to CLR parser. For this grammar the SLR parser and the LALR parser are exactly the same. This can be seen by simply looking at the CLR table and merging the lookahead sets. Thus, for this question as well we can take the same input string and parsing for LALR is similar to that of SLR parser given above whereas CLR gives error pretty early. The reason behind this is that no. of reduce actions increase in LALR as we are taking union of all the lookahead sets from CLR. Thus, for an input symbol CLR may give error directly but in LALR reduction may happen.