Ali Taheri – 40012323

Q1:

A. The Chomsky hierarchy consists of four types of languages: Regular Languages (RL), Context-Free Languages (CFL), Context-Sensitive Languages (CSL), and Recursively Enumerable Languages (REL). The languages LR(0), LL(1), and LR(1) are all subsets of Context-Free Languages (CFL) and can be parsed using different types of parsers.

B. In the process of constructing an LALR(1) parser from an LR(1) parser, we merge states with identical core structures, combining only their lookahead sets. The original LR(1) grammar's shifts and reduces remain unchanged. Notably, if the original LR(1) grammar is conflict-free, merging lookaheads in the LALR(1) construction cannot introduce new conflicts. This is because LALR(1) maintains the same state transitions (shifts) and reduction conditions as the original LR(1) parser. The sole distinction lies in LALR(1)'s use of combined lookahead sets to determine the appropriate action. Since the original LR(1) grammar lacked conflicts, the combined lookaheads cannot create scenarios where multiple actions are valid for the same input symbol.

C. Turing machines, despite their theoretical significance in defining the limits of computation, are not suitable for practical use in real-world parsing applications. This is due to their inherent slowness compared to widely used algorithms like LL, LR, and LALR. The process of designing and implementing a Turing machine for a real-world programming language would be incredibly complex, requiring detailed manipulation of tape symbols and state transitions. In contrast, standard parsing algorithms, like LL and LR, are specifically tailored for grammar parsing, resulting in simpler implementations and enhanced performance. Furthermore, Turing machines are computationally less efficient than their counterparts, rendering them unsuitable for the demands of modern software development.

Q2:

A. This statement is **incorrect,** LR(2) ⊂ LR(1): This is false. LR(2) is not a subset of LR(1). LR(2) uses a two-symbol lookahead, while LR(1) uses a single symbol lookahead. Therefore, LR(2) is a superset of LR(1) and can parse languages that LR(1) cannot.
For example:

S → xXd
S → yYd
X → z
Y → z

B. This statement is **correct** because there exists an algorithm for converting an LR(K) grammar to an LR(1) grammar.

C. This statement is True

S → Xy

S → bXc

S → ac

S → bay

X → a

Q3. The combination of states in LR(1), also known as CLR(1), results in a greater number of states compared to LALR(1). Therefore, the number of states in LALR(1) is less than LR(1), so n1 > n3. On the other hand, SLR(1) and LALR(1) have the same number of states, meaning n2 = n3.

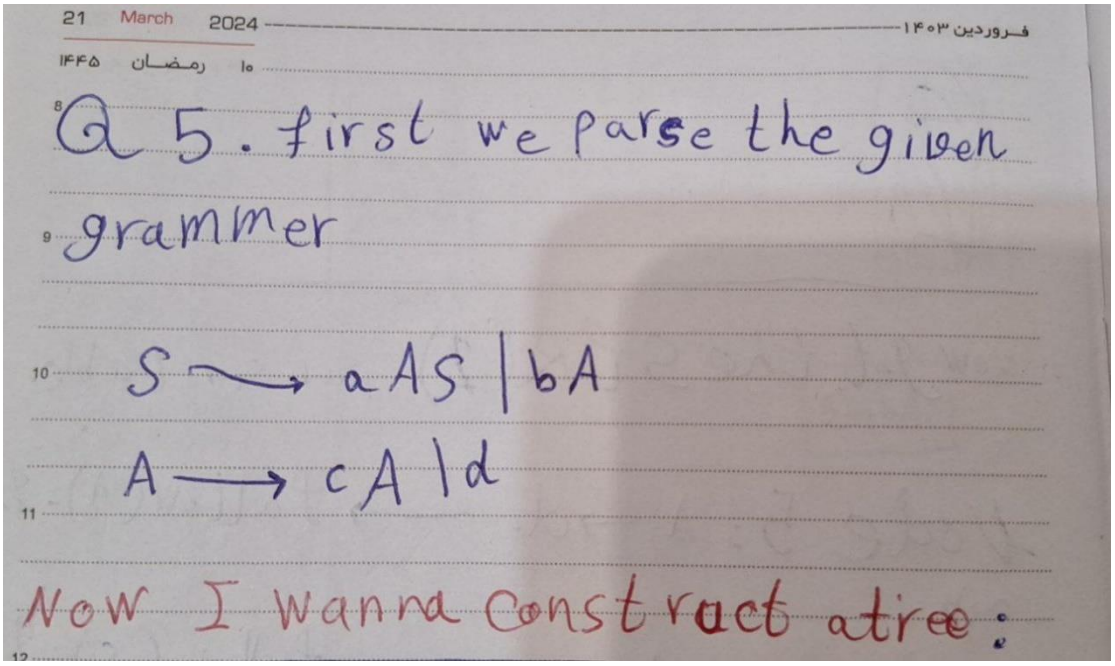
Thus : n1>n2=n3

Q4. In a bottom-up parser, each time a reduction is performed, a number of symbols in the input string are replaced by a nondistinct symbol (nonterminal). If n

is the number of symbols in the input string, we can expect a maximum of n reductions to be applied, as each reduction removes at least one symbol from the string. Additionally, if we want to create a nondistinct symbol from among n symbols, we need n-1 reductions, as creating a nondistinct symbol requires removing an existing symbol, resulting in one less reduction than the number of symbols. Therefore, the minimum number of reductions is equal to the sum of the maximum number of symbols reduced and the minimum possible number of reductions from them, which is one less than the maximum: n + (n-1) = 2n-1.

Q5.

I'll write the answer of this question in paper



۵ یکشنبه

24 March 2024 — ۵ فروردین ۱۴۰۳

۱۴۴۵ رمضان ۱۳

(0)
S' → .S
S → .aAS
S → .bA

(1)
S' → S.

(2)
S → a.AS
A → .cA
A → .d

(3)
S → b.A
A → .cA
A → .d

(4)
A → c.A
A → .cA
A → .d

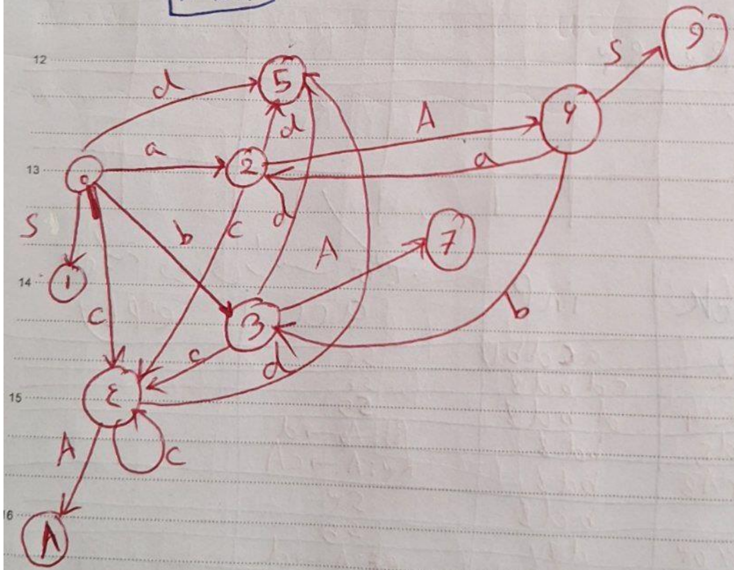
(5)
A → d.

(6)
S → aA.S
S → .aAS
S → .bA

(7)
S → bA.

(8)
A → cA.

(9)
S → aAS.



Now get the SLR(1) parser table (↑ one)

Node 5: A → d. ⟶ fallow(A) = $, a, b

a, b

Node 7: S → bA. ⟶ fallow(S) = $

Node 8: S → cA. ⟶ fallow(A) = $, a, b

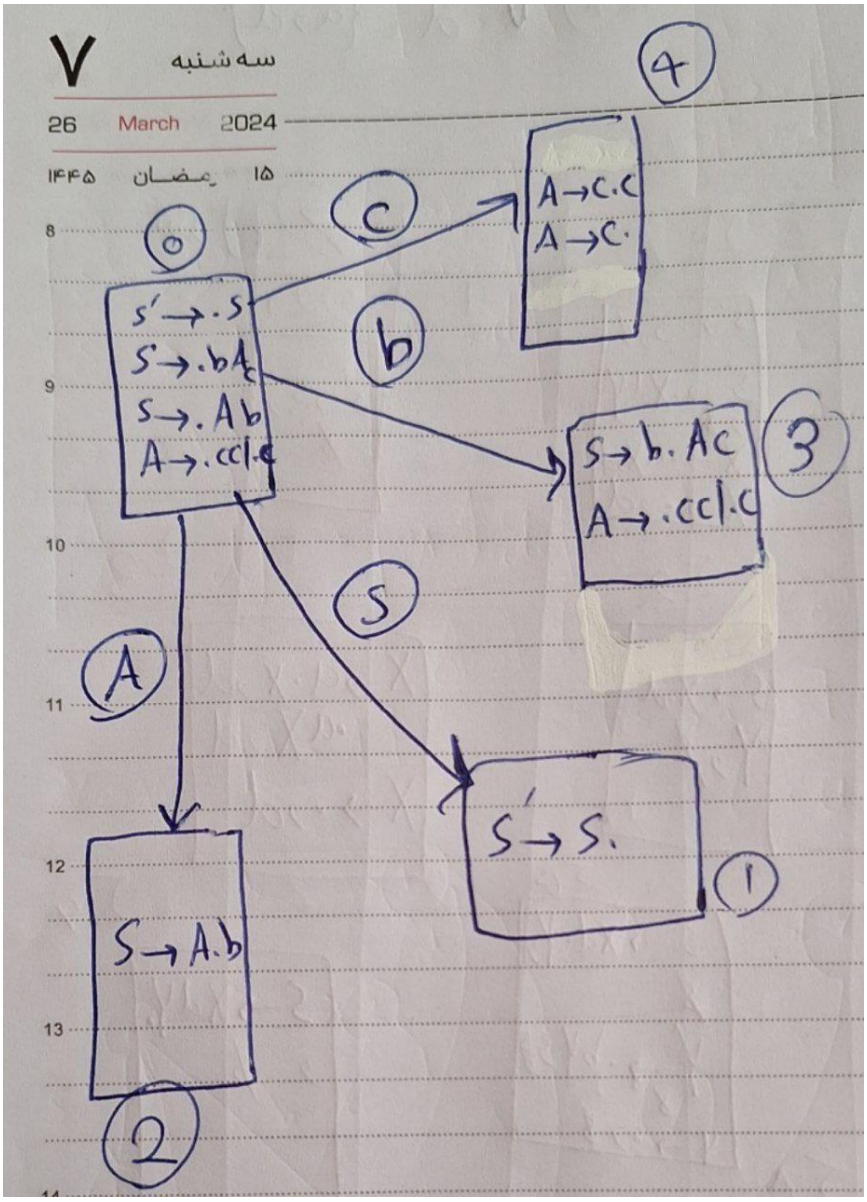
Node 9: S → aAS. ⟶ fallow(S) = $

State	Action					goto	
	a	b	c	d	$	S	A
0	S_2	S_3	S_4	S_5		1	
1					accept		
2			S_4	S_5			**6**
3			S_4	S_5			**7**
4			S_4	S_5			**8**
5	R_4	R_4			R_4		
6	S_2	S_3				9	
7					R_2		
8	R_3	R_3			R_3		
9					R_1		

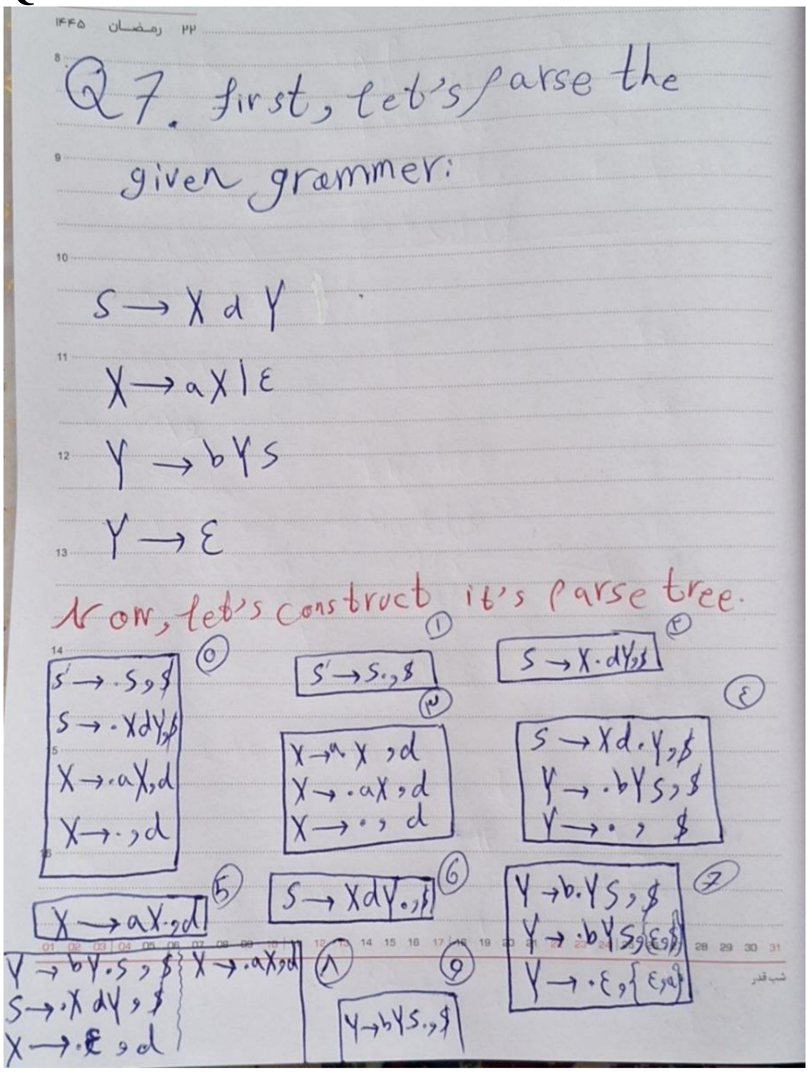
Now, let's examine the input string.

Stack	input	Action
0	acdbd$	**S_2**
0a2	cdbd$	**S_4**
0a2c4	dbd$	**S_5**
0a2c4d5	bd$	**R_4:** A → d
0a2c4A8	bd$	**R_3:** A → cA
0a2A6	bd$	**S_3**
0a2A6b3	d$	**S_5**
0a2A6b3d5	$	**R_4:** A → d
0a2A6b3A7	$	**R_2:** S → bA
0a2A6S9	$	**R_1:** S → aAS
0S1	$	**accept**

Q6. in the DFA , State 4 has the rule A → c. and also Follow(A) includes b and c, in our SLR(1) parser, we have a Reduce action for State 4 in the column of terminal c. On the other hand, in State 4, the rule A → c.c is present as well, which leads us to have a Shift action in the same place. So, we end up with a Shift-Reduce conflict in the row for State 4 and the column c. Note that we didn't extend the DFA here since we could reach the answer with State 4 alone.



Q7.



۲۲ رمضان ۱۴۴۵

Q7. first, let's parse the given grammer:

S → X d Y

X → aX | ε

Y → bYS

Y → ε

Now, let's construct it's parse tree.

(0)
S' → .S, $
S → .XdY, $
X → .aX, d
X → ., d

(1)
S' → S., $

(2)
S → X.dY, $

(3)
X → a.X, d
X → .aX, d
X → ., d

(4)
S → Xd.Y, $
Y → .bYS, $
Y → ., $

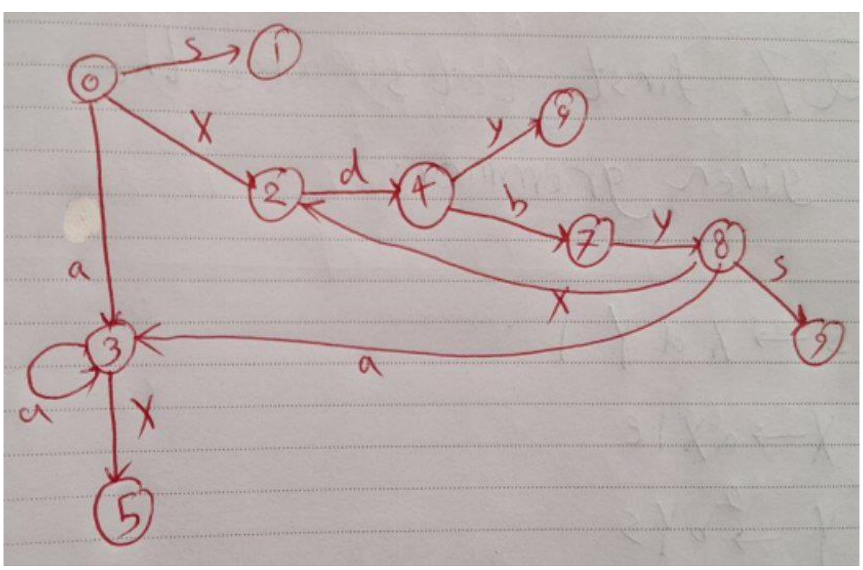
(5)
X → aX., d

(6)
S → XdY., $

(7)
Y → b.YS, $
Y → .bYS, {ε, a}
Y → .ε, {ε, a}

(8)
Y → bY.S, $
S → .XdY, $
X → .ε, d
X → .aX, d

(9)
Y → bYS., $



Let's construct the LR(1) parser table.

We know ⇒ follow(S) = {a, d, $} and follow(X) = {d} and follow(Y) = {a,d,$}

State	Action				goto		
	a	b	d	$	S	X	**Y**
0	S_3		R_3		1	2	
1				Accept			
2			S_4				
3	S_3		R_3			5	
4	R_5	S_7	R_5	R_5			**6**
5			R_2				
6	R_1		R_1	R_1			
7	R_5	S_7	R_5	R_5			**8**
8	S_3		R_3		9	2	
9	R_4		R_4	R_4			

Now, let's examine the input string.

Stack	input	Action
0	aadbbadd$	**S_3**
0a3	adbbadd$	**S_3**
0a3a3	dbbadd$	**R_3:** X → ϵ
0a3a3X5	dbbadd$	**R_2:** X → aX
0a3X5	dbbadd$	**R_2:** X → aX
0X2	dbbadd$	**S_4**
0X2d4	bbadd$	**S_7**
0X2d4b7	badd$	**S_7**
0X2d4b7b7	add$	**R_5:** Y → ϵ
0X2d4b7b7Y8	add$	**S_3**
0X2d4b7b7Y8a3	dd$	**R_3:** X → ϵ
0X2d4b7b7Y8a3X5	dd$	**R_2:** X → aX
0X2d4b7b7Y8X2	dd$	**S_4**

0X2d4b7b7Y8X2d4	d$	**R_5:** Y → ϵ
0X2d4b7b7Y8X2d4Y6	d$	**R_1:** S → XdY
0X2d4b7b7Y8S9	d$	**R_4:** Y → bYS
0X2d4b7Y8	d$	**R_3:** X → ϵ
0X2d4b7Y8X2	d$	**S_4**
0X2d4b7Y8X2d4	$	**R_5:** Y → ϵ
0X2d4b7Y8X2d4Y6	$	**R_1:** S → XdY
0X2d4b7Y8S9	$	**R_4:** Y → bYS
0X2d4Y6	$	**R_1:** S → XdY
0S1	$	**Accept**

Q8.

Part a.

Right sentential	Handle	Reducing production
aaa*a++	a	S → a
aaS*a++	A	S → a
aSS*a++	SS*	S → SS ∗
aSa++	A	S → a
aSS++	SS+	S → *SS* +
aS+	A	S → a
SS+	SS+	S → *SS* +

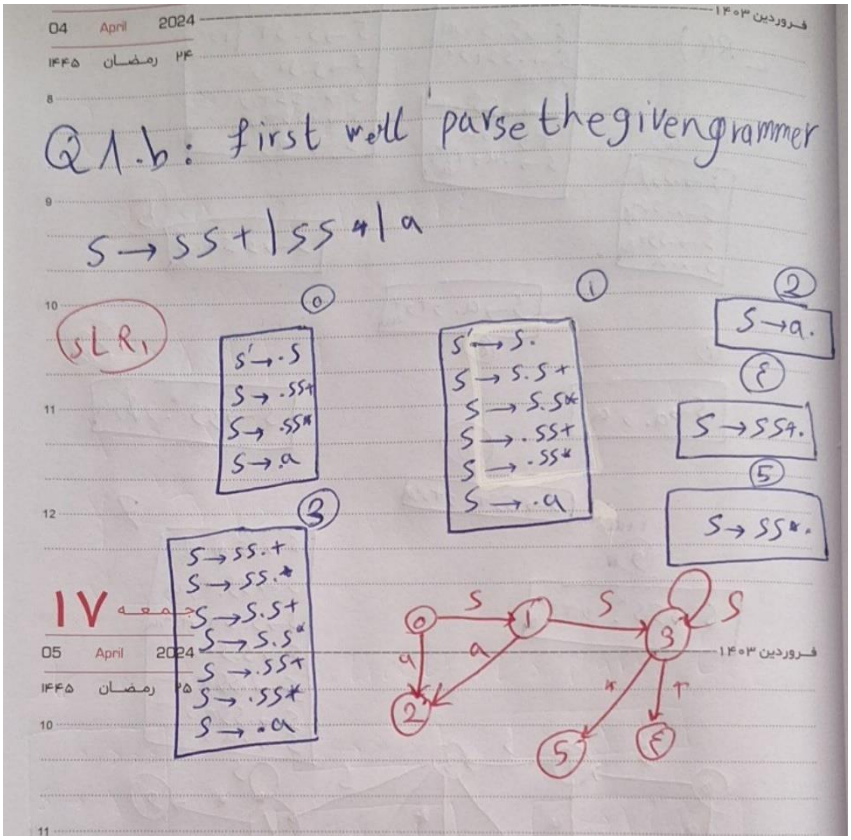
Right sentential	Handle	Reducing production
SSS+a*+	SS+	$S \rightarrow SS+$
SSa*+	a	$S \rightarrow a$
SSS*+	SS*	$S \rightarrow SS*$
SS+	SS+	$S \rightarrow SS+$

Part b.

Let's start by parsing the given grammar.

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

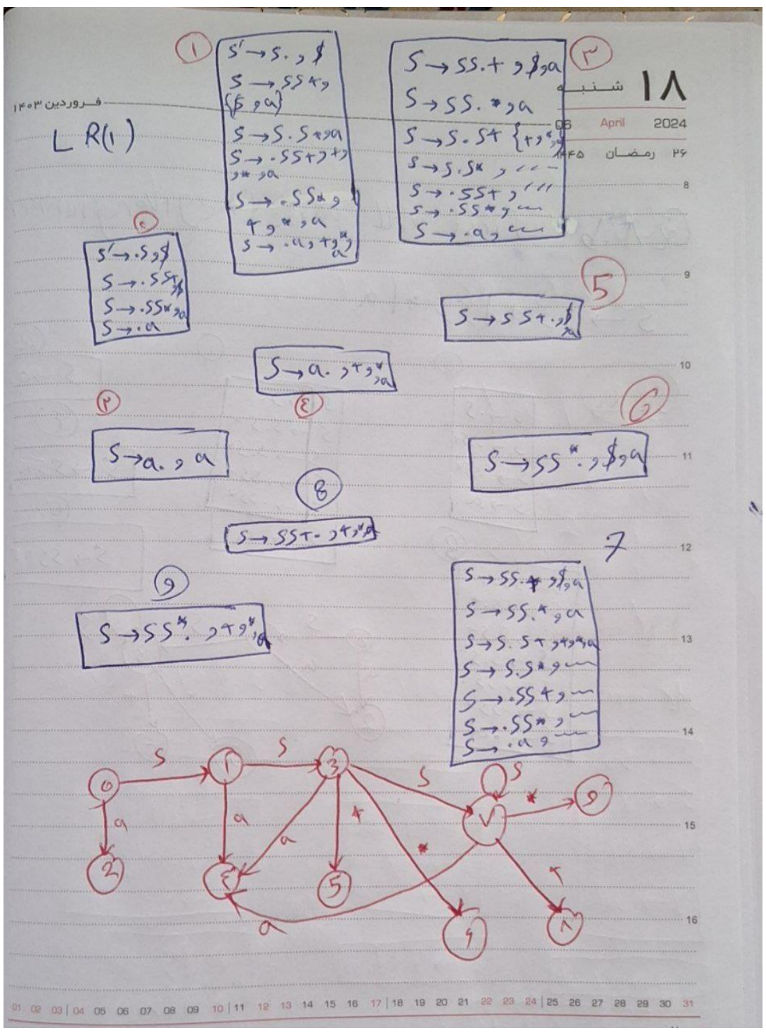
slr(1):



let's construct the SLR(1) parser table.

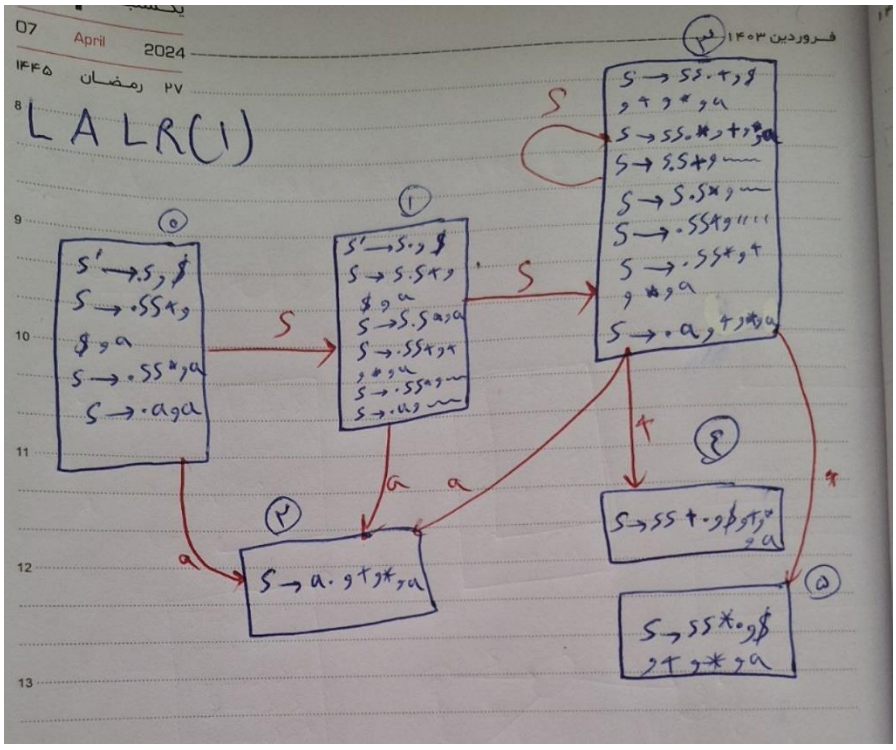
State	Action				goto
	*	+	a	$	**S**
0			S_2		**1**
1			S_2	a	**3**
2	R_3	R_3	R_3	R_3	
3	S_5	S_4	S_2		**3**
4	R_1	R_1	R_1	R_1	
5	R_2	R_2	R_2	R_2	

LR(1)



Let’s construct an LR(1) parsing table:

State	Action				goto
	*	+	a	$	S
0			S_2		**1**
1			S_4	Accept	**3**
2			R_3		
3	S_6	S_5	S_4		**7**
4	R_3	R_3	R_3		
5			R_1	R_1	
6			R_2	R_2	
7	S_9	S_8	S_4		**7**
8	R_1	R_1	R_1		
9	R_2	R_2	R_2		



Let's construct an LALR(1) parsing table

State	Action				goto
	*	+	a	$	S
0			S_2		**1**
1			S_2	Accept	**3**
2	R_3	R_3	R_3		
3	S_5	S_4	S_2		**3**
4	R_1	R_1	R_1	R_1	
5	R_2	R_2	R_2	R_2	

Part c

Let's check the input string:

Stack	input	Action
0	aa*aa+a+*$	**S_2**
0a2	a*aa+a+*$	**R_3:** $S \rightarrow a$
0S1	a*aa+a+*$	**S_2**
0S1a2	*aa+a+*$	**R_3:** $S \rightarrow a$
0S1S3	*aa+a+*$	**S_5**
0S1S3*5	aa+a+*$	**R_2:** $S \rightarrow SS*$
0S1	aa+a+*$	**S_2**
0S1a2	a+a+*$	**R_3:** $S \rightarrow a$
0S1S3	a+a+*$	**S_2**
0S1S3a2	+a+*$	**R_3:** $S \rightarrow a$
0S1S3S3	+a+*$	**S_4**
0S1S3S3+4	a+*$	**R_1:** $S \rightarrow SS+$
0S1S3	a+*$	**S_2**

0S1S3a2	+*$	**R_3:** S → a
0S1S3S3	+*$	**S_4**
0S1S3S3+4	*$	**R_1:** S → SS +
0S1S3	*$	**S_5**
0S1S3*5	$	**R_2:** S → SS ∗
0S1	$	**Accept**

Q9.

For the grammar rules Ai→aiAi and Ai→ai, the total number of rules is 2(n-2)(n-2) as the values for j and i range from 1 to n-2.

Similarly, considering the grammar rule S→AiBi, we can establish that it comprises n-2 rules, given that the possible values for i range from 1 to n-2.

Combining the above findings, we can determine that the total number of rules in the grammar is:

2(n-2)(n-2) + (n-2)

= 2(n-2)^2 + (n-2)

= n-2 + 2n^2 - 8n + 8

= 2n^2 - 7n + 6

For a grammar to be classified as SLR(1), it must adhere to the conditions of having no shift-reduce conflicts and no reduce-reduce conflicts. In the case of this grammar, it exhibits ambiguities that impede the creation of a valid SLR(1) parsing table. These ambiguities stem from the uncertainty surrounding the selection of Ai and Bi in each state. Consequently, conflicts arise during the parsing table construction process, preventing the generation of a valid SLR(1) parsing table. Hence, this grammar does not meet the criteria for being recognized as SLR(1).

Q10.

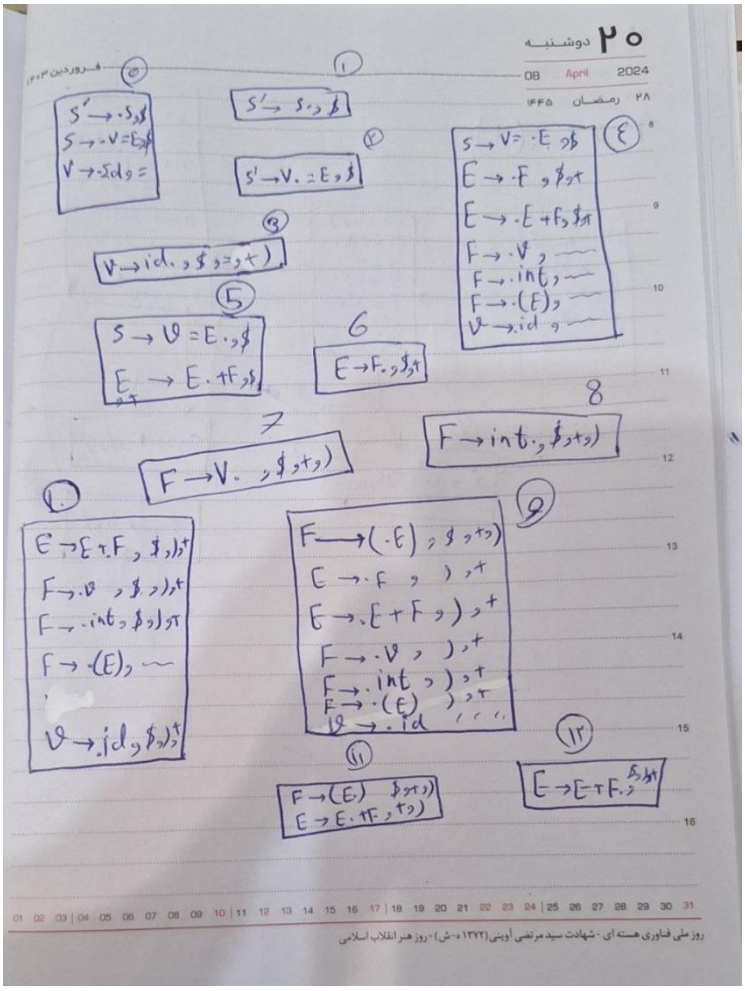
First, we analyze the given grammar.

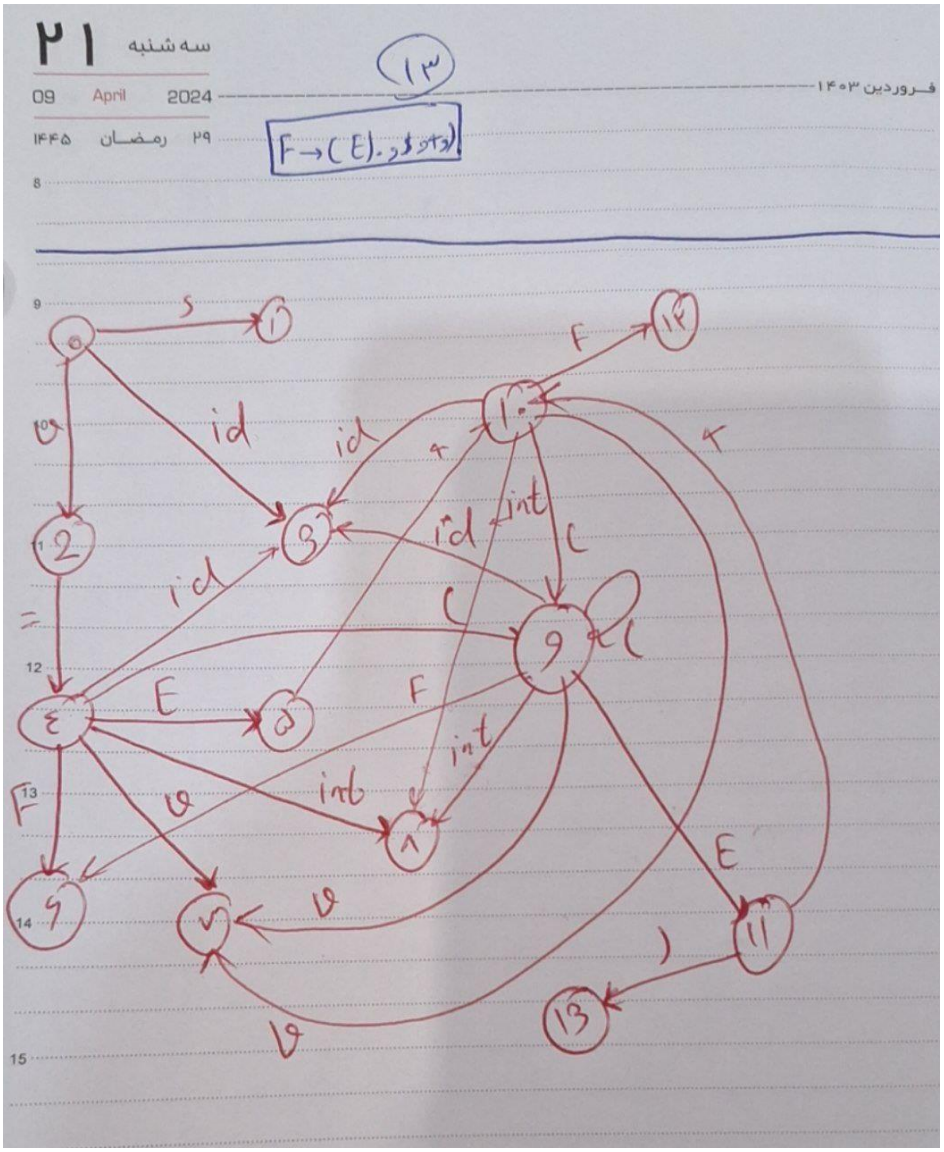
$$S \rightarrow V = E$$

$$E \rightarrow F \mid E + F$$

$$F \rightarrow V|\text{int}|(E)$$

$$V \rightarrow \text{id}$$





Let's create an LALR(1) parse table:

State	Action							goto			
	=	+	Int	(	)	Id	$	S	E	F	**V**
0						S_3		1			**2**
1							Accept				
2	S_4										
3	R_7	R_7			R_7		R_7				

4			S_8	S_9		S_3			5	6	**7**
5		S_{10}					R_1				
6		R_2			R_2		R_2				
7		R_4			R_4		R_4				
8		R_5			R_5		R_5				
9			S_8	S_9		S_3			11	6	**7**
10			S_8	S_9		S_3				12	**7**
11		S_{10}			S_{13}						
12		R_3			R_3		R_3				
13		R_6			R_6		R_6				

No conflict is evident in the table.