Theory of Language Translation (MTH683)
Finals Questions Bank

Mostafa Hassanein

11 Jan 2026

3-1

Regular Definitions

Give a regular definition for non-negative integers without leading zeros. A zero is represented by a single 0.

Solution:

Let:

$P_1 = \langle digit, [0-9] \rangle$

$P_2 = \langle nonzero\_digit, [1-9] \rangle$

$P_3 = \langle number, 0 \mid nonzero\_digit(digit)^* \rangle$

The regular definition is given by the sequence:

$$\langle P_1, P_2, P_3 \rangle.$$

Alternatively:

$digit \rightarrow [0-9]$

$nonzero\_digit \rightarrow [1-9]$

$number \rightarrow 0 \mid nonzero\_digit(digit)^*$

3-2

Action-Augmented Regular Definitions

For the following action-augmented regular definition, give a regular expression describing the language of possible outputs. Assume that all inputs are strings of 0's and 1's only

$5 \rightarrow 0$	printf("c")
$6 \rightarrow 00$	printf("a")
$7 \rightarrow 1$	printf("b")

Solution:

To determine the regular expression describing the language of possible outputs, we must analyze how the lexical analyzer partitions an input string of $\{0, 1\}$ and which actions (outputs) are triggered based on the disambiguation rules defined in the sources.

1. Rule Analysis and Mapping

The definition provides three patterns with associated print actions:

- **Rule 5:** $0 \rightarrow$ Output: `"c"`
- **Rule 6:** $00 \rightarrow$ Output: `"a"`
- **Rule 7:** $1 \rightarrow$ Output: `"b"`

2. Applying Disambiguation Rules

According to the sources, the lexical analyzer resolves ambiguity using two primary strategies:

1. **Longest Match:** The analyzer selects the splitting that results in the longest possible prefix lexeme [1, 2].
2. **Rule Priority:** If two patterns match the same length, the one appearing earlier in the definition is chosen [2].

In this specific set of rules, the **Longest Match** rule is the dominant factor for the characters `'0'`.

- If the analyzer encounters `"00"`, it will always match **Rule 6** (length 2) instead of matching **Rule 5** twice or once (length 1). Thus, the output `"a"` is produced whenever two zeros appear consecutively.
- To produce the output `"c"` (Rule 5), the character `'0'` must not be followable by another `'0'` to form a longer match. This occurs only if the `'0'` is followed by a `'1'` or is at the end of the input string [3, 4].

3. Possible Output Sequences

Based on these mechanics, we can identify the possible "chunks" of output:

- An input of `1` always produces `b`.
- An input of `00` always produces `a`.
- An input of `01` produces `cb` (The `0` is matched as `c` because the next character `1` prevents a `00` match; the `1` is then processed as `b`).
- An input of `0` at the very end of a string produces `c`.

Crucially, the output `c` can never be immediately followed by another `c` or an `a`, because both would imply the existence of a `0` immediately following the first `0`, which the Longest Match rule would have aggregated into an `a` or a different partitioning.

4. Final Regular Expression

The language of possible outputs L_{out} over the alphabet $\Sigma_{out} = \{a, b, c\}$ is described by the following regular expression:

$$(a \mid b \mid cb)^* c?$$

Explanation:

- The term $(a \mid b \mid cb)^*$ allows for any combination of `00` inputs (`a`), `1` inputs (`b`), or `01` inputs (`cb`).
- The trailing $c?$ accounts for the case where the input string ends with a single `0`, which is the only other way to generate the output `c`.

3-3: Fallback DFA with Actions

Input String: *aaabaabbababbb*

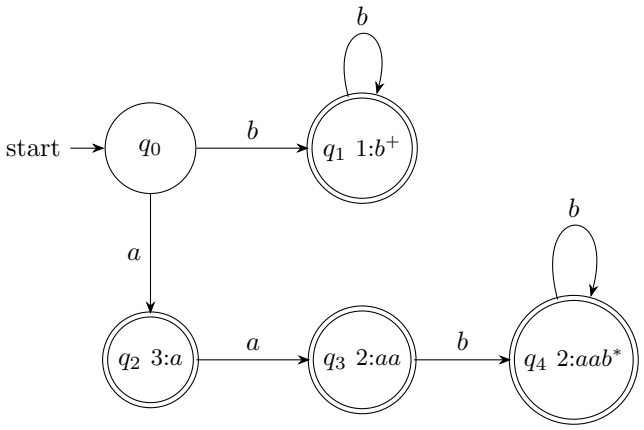
a. First Regular Definition

Rules:

1. $b+ \rightarrow$ `printf("1")`
2. $aab^* \rightarrow$ `printf("2")`
3. $a \rightarrow$ `printf("3")`

Priority: $1 > 2 > 3$

State Diagram



Trace Table

#	Remaining Input	Path (States)	Fallback	Lexeme	Out
1	aaabaabbababbb	$q_0 \xrightarrow{a} q_2 \xrightarrow{a} q_3 \xrightarrow{a}$ Dead	q_3	aa	2
2	abaabbababbb	$q_0 \xrightarrow{a} q_2 \xrightarrow{b}$ Dead	q_2	a	3
3	baabbababbb	$q_0 \xrightarrow{b} q_1 \xrightarrow{a}$ Dead	q_1	b	1
4	aabbababbb	$q_0 \xrightarrow{a} q_2 \xrightarrow{a} q_3 \xrightarrow{b} q_4 \xrightarrow{b} q_4 \xrightarrow{a}$ Dead	q_4	aabb	2
5	ababbb	$q_0 \xrightarrow{a} q_2 \xrightarrow{b}$ Dead	q_2	a	3
6	babbb	$q_0 \xrightarrow{b} q_1 \xrightarrow{a}$ Dead	q_1	b	1
7	abbb	$q_0 \xrightarrow{a} q_2 \xrightarrow{b}$ Dead	q_2	a	3
8	bbb	$q_0 \xrightarrow{b} q_1 \xrightarrow{b} q_1 \xrightarrow{b} q_1$	End	bbb	1

Final Printed Output: 2 3 1 2 3 1 3 1

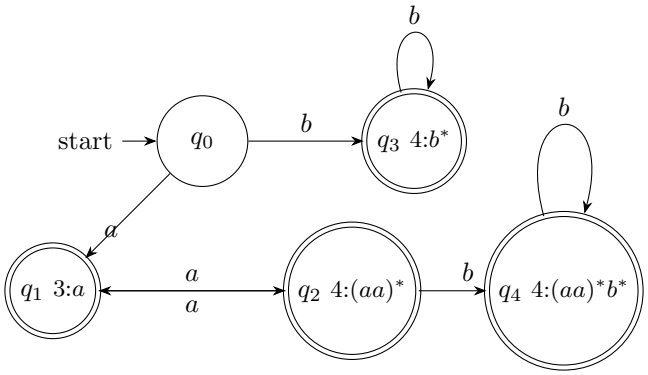
b. Second Regular Definition

Rules:

4. $(aa)^*b^* \rightarrow$ `printf("2")`
3. $a \rightarrow$ `printf("3")`

Priority: $4 > 3$

State Diagram



Trace Table

#	Remaining Input	Path (States)	Fallback	Lexeme	Out
1	aaabaabbababbb	$q_0 \xrightarrow{a} q_1 \xrightarrow{a} q_2 \xrightarrow{a} q_1 \xrightarrow{b}$ Dead	q_2	aa	2
2	abaabbababbb	$q_0 \xrightarrow{a} q_1 \xrightarrow{b}$ Dead	q_1	a	3
3	baabbababbb	$q_0 \xrightarrow{b} q_3 \xrightarrow{a}$ Dead	q_3	b	2
4	aabbababbb	$q_0 \xrightarrow{a} q_1 \xrightarrow{a} q_2 \xrightarrow{b} q_4 \xrightarrow{b} q_4 \xrightarrow{a}$ Dead	q_4	aabb	2
5	ababbb	$q_0 \xrightarrow{a} q_1 \xrightarrow{b}$ Dead	q_1	a	3
6	babbb	$q_0 \xrightarrow{b} q_3 \xrightarrow{a}$ Dead	q_3	b	2
7	abbb	$q_0 \xrightarrow{a} q_1 \xrightarrow{b}$ Dead	q_1	a	3
8	bbb	$q_0 \xrightarrow{b} q_3 \xrightarrow{b} q_3 \xrightarrow{b} q_3$	End	bbb	2

Final Printed Output: 2 3 2 2 3 2 3 2

3-4

Lexical Analyzers

We would like to construct a tokenizer with the following specification.

Given an input stream of decimal digits, the stream should be split into segments and the outputs for consecutive segments should be produced in sequence.

A segment is either a string of two digits, or a string of one digit if only one digit is available.

If the segment is a string d_1d_2 of two digits, then the corresponding output depends on d_2: if $d_2 < 9$, the output is the two-digit decimal string representing $d_1d_2 + 1$; if $d_2 = 9$, the output is d_{10}.

If the segment is a string of one digit d, then the output is d. Here are some illustrative examples.

Input	**Output**	
1235	13	36
9807	99	08
999	90	9
090	00	0

a. Give an action-augmented regular definition with rules describing input patterns and with actions producing the corresponding outputs.

b. Draw the state diagram of the corresponding NFA.

c. Draw the state diagram of the corresponding fallback DFA with actions M.

d. Trace the operation of M on input 2003798, showing (i) the output and (ii) the sequence of states M goes through as it scans the input.

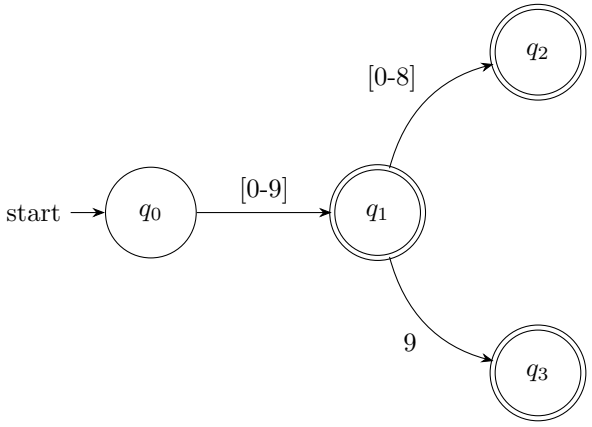
Solution:

We define the patterns based on the digit sets $D = \{0, 1, \ldots, 9\}$ and $D_{<9} = \{0, 1, \ldots, 8\}$. The rules are prioritized by the **Longest Match** principle.

1.	$DD_{<9}$	printf("%02d", val(d1 d2) + 1)
2.	$D9$	printf("%d0", d1)
3.	D	printf("%d", d)

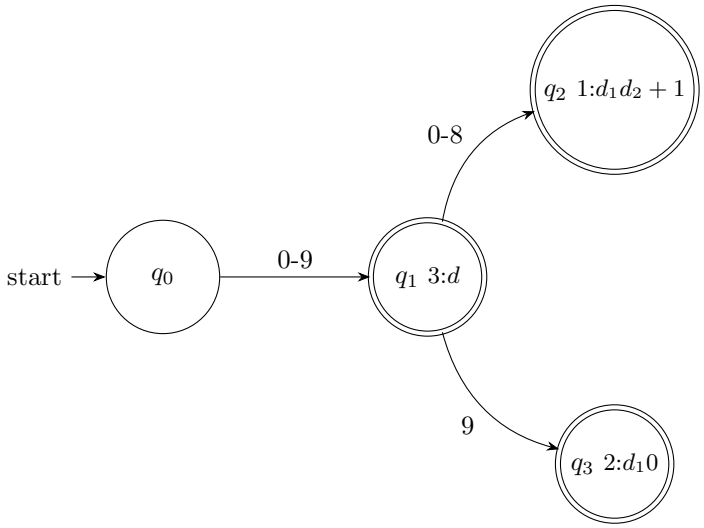
b. State Diagram of the NFA

The NFA transitions to different states to distinguish between the one-digit case and the two-digit cases.



c. Fallback DFA with Actions

In the fallback DFA, every state reached by a digit is an accepting state. If the DFA reaches a dead end (e.g., trying to read a third digit), it falls back to the last seen accepting state.



d. Trace on input: 2003798

(i) Output Trace Table

Lexeme	Input	Path Taken	Rule	Calculation	Output
1	20...	$q_0 \xrightarrow{2} q_1 \xrightarrow{0} q_2$	1	$20 + 1$	**21**
2	03...	$q_0 \xrightarrow{0} q_1 \xrightarrow{3} q_2$	1	$03 + 1$	**04**
3	79...	$q_0 \xrightarrow{7} q_1 \xrightarrow{9} q_3$	2	$7 \rightarrow 70$	**70**
4	8	$q_0 \xrightarrow{8} q_1$	3	8	**8**

(ii) Sequence of States

The DFA moves through the following states (noting that the machine returns to q_0 after each lexeme is finalized):

$$(q_0 \rightarrow q_1 \rightarrow q_2) \rightarrow (q_0 \rightarrow q_1 \rightarrow q_2) \rightarrow (q_0 \rightarrow q_1 \rightarrow q_3) \rightarrow (q_0 \rightarrow q_1)$$

Final Printed Output: 21 04 70 8

3-5

Context-Free Grammars

Give a context-free grammar (CFG) for each of the following languages:
a. $L = \{a^m b^n c^k | k = m + n;\ m, n, k \geq 0\}$ over the alphabet $\Sigma = \{a, b, c\}$.
b. $L = \{a^m b^n | n \neq m\}$ over the alphabet $\Sigma = a, b$.
c. $L = \{w | w$ is a palindrome$\}$ over the alphabet $\Sigma = \{a, b, c\}$.
(Note: A palindrome is a string that reads the same backwards as forwards.)

Solution:

To solve these problems, we utilize the fundamental structure of context-free grammars (CFGs), which are defined by a set of variables, terminals, production rules, and a start variable. The following grammars leverage the recursive nature of CFGs to maintain counts and symmetry across different parts of a string.

a. $L = \{a^m b^n c^k \mid k = m + n;\ m, n, k \geq 0\}$ over $\Sigma = \{a, b, c\}$.

To generate strings where the number of c's equals the sum of a's and b's, we can think of the language as $a^m b^n c^{n+m}$. This can be structurally decomposed into matching the outer a's with the trailing c's, and then matching the inner b's with the remaining c's [1].

Grammar G_a:

$$S \to aSc \mid T$$
$$T \to bTc \mid \epsilon$$

Here, S recursively adds an a to the front and a c to the back, while T does the same for b and c once all a's are generated [1].

b. $L = \{a^m b^n \mid n \neq m\}$ over $\Sigma = \{a, b\}$.

This language consists of two cases: either there are more a's than b's ($m > n$) or more b's than a's ($n > m$). We can use the logic of generating equal numbers ($a^m b^m$) and then ensuring at least one extra symbol exists on the required side [1].

Grammar G_b:

$$S \to M \mid N$$
$$E \to aEb \mid \epsilon$$
$$M \to aM \mid aE \quad (\text{Case: } m > n)$$
$$N \to Nb \mid Eb \quad (\text{Case: } n > m)$$

The variable E generates equal numbers of a's and b's, while M and N force an imbalance by adding extra terminals to one side [1].

c. $L = \{w \mid w \text{ is a palindrome}\}$ over $\Sigma = \{a, b, c\}$.

A palindrome is constructed by ensuring that every symbol added to the front of the string is matched by the same symbol at the end. This symmetry is captured by recursive rules that wrap a smaller palindrome in matching terminal symbols [1].

Grammar G_c:

$$S \to aSa \mid bSb \mid cSc \mid a \mid b \mid c \mid \epsilon$$

The rules aSa, bSb, and cSc maintain the palindrome property, while a, b, c, and ϵ serve as the base cases for odd-length and even-length palindromes, respectively [1].

Building a CFG is like layering an onion; to ensure the outer layers match (like a^m and c^m), you must define rules that grow the string from the center outward, maintaining the relationship between the symbols at the boundaries.

3-6

Parse trees

Consider the grammar:

$$S \to A1B$$
$$A \to 1A \mid 0$$
$$B \to 0B \mid \epsilon$$

Give a parse tree for each of the following strings:
a. 11101
b. 1010
c. 0100

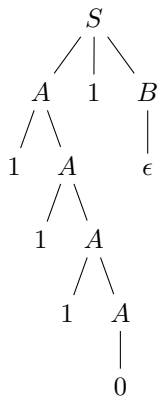
Solution:

The grammar provided is:

$$S \to A1B$$
$$A \to 1A \mid 0$$
$$B \to 0B \mid \epsilon$$

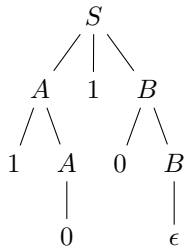
a. Parse tree for "11101"

In this string, the terminal '1' from the S rule is the final digit. The variable A generates the prefix "1110", and B generates the empty string ϵ.



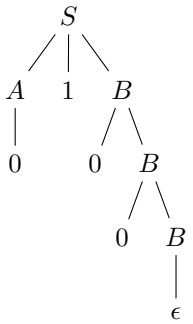
b. Parse tree for "1010"

Here, the '1' from the S rule is the second '1' in the string. A generates "10", and B generates "0".



c. Parse tree for "0100"

In this case, A generates the leading "0", followed by the terminal '1' from the S rule, and B generates the trailing "00".



3-7

Derivations

Consider the following context-free grammar:

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

and the string: $aa + a*$

a. Give a leftmost derivation for the string. Show the sequence of derivation rules applied.

b. Give a rightmost derivation for the string. Show the sequence of derivation rules applied.

c. Give a parse tree for the string.

Solution:

The grammar rules are: (1) $S \to SS+$, (2) $S \to SS*$, (3) $S \to a$.

a. Leftmost Derivation

A leftmost derivation always replaces the variable furthest to the left in the sentential form [1].

$$\begin{aligned} S &\Rightarrow SS* && \text{(Rule 2)} \\ &\Rightarrow SS + S* && \text{(Rule 1 applied to leftmost } S) \\ &\Rightarrow aS + S* && \text{(Rule 3 applied to leftmost } S) \\ &\Rightarrow aa + S* && \text{(Rule 3 applied to leftmost } S) \\ &\Rightarrow aa + a* && \text{(Rule 3 applied to leftmost } S) \end{aligned}$$

Sequence of rules applied: (2), (1), (3), (3), (3).

b. Rightmost Derivation

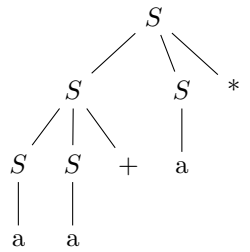
A rightmost derivation always replaces the variable furthest to the right in the sentential form [1].

$$\begin{aligned} S &\Rightarrow SS* && \text{(Rule 2)} \\ &\Rightarrow Sa* && \text{(Rule 3 applied to rightmost } S) \\ &\Rightarrow SS + a* && \text{(Rule 1 applied to rightmost } S) \\ &\Rightarrow Sa + a* && \text{(Rule 3 applied to rightmost } S) \\ &\Rightarrow aa + a* && \text{(Rule 3 applied to rightmost } S) \end{aligned}$$

Sequence of rules applied: (2), (3), (1), (3), (3).

c. Parse Tree

The parse tree illustrates the derivation's hierarchy [1]. The root S branches into the components of the multiplication rule, and the first S sub-tree further branches into the addition components.



4-1

LL(1) Parsing

Consider the following CFG:

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

a. Eliminate left recursion and left factor the grammar.

b. Compute first and follow sets for each non-terminal in the resulting grammar.

c. Build the predictive parsing table.

Solution:

4-2

LL(1) Parsing

Consider the context-free grammar $G = (\{S, L, R\}, \{(,), +, a\}, R, S)$, where R is the set of the following rules.

$$S \to (L) \mid a$$
$$L \to SR$$
$$R \to +SR \mid \epsilon$$

a. Construct the LL(1) predictive parsing table for G.
b. Is G an LL(1) grammar? Why?

Solution:

4-3

LR(0) Automaton and SLR Parsing

Consider the following grammar:

$$S \to Xa$$
$$X \to a \mid aXb$$

a. Construct the LR(0) DFA of the grammar.
b. Construct the SLR parsing table.
c. Use the parsing table to simulate SLR parsing on the string: aaba

Solution:

To solve this problem, we follow the standard procedures for augmenting a grammar, constructing an LR(0) automaton, and building an SLR parsing table.

(a) Construct the LR(0) DFA of the grammar

We augment the grammar by adding a new start symbol S' with the production

$$S' \to S$$

to provide a unique accept condition.

Augmented Grammar G'

$$\begin{array}{ll} (0) & S' \to S \\ (1) & S \to Xa \\ (2) & X \to a \\ (3) & X \to aXb \end{array}$$

We now compute the LR(0) item sets using the **CLOSURE** and **GOTO** functions.

$$I_0 = \text{CLOSURE}(\{S' \to \cdot S\}) = \{S' \to \cdot S,\ S \to \cdot Xa,\ X \to \cdot a,\ X \to \cdot aXb\}$$

$$\begin{aligned} I_1 &= \text{GOTO}(I_0, S) = \{S' \to S\cdot\} \\ I_2 &= \text{GOTO}(I_0, X) = \{S \to X \cdot a\} \\ I_3 &= \text{GOTO}(I_0, a) = \text{CLOSURE}(\{X \to a\cdot,\ X \to a \cdot Xb\}) \\ &\quad = \{X \to a\cdot,\ X \to a \cdot Xb,\ X \to \cdot a,\ X \to \cdot aXb\} \\ I_4 &= \text{GOTO}(I_2, a) = \{S \to Xa\cdot\} \\ I_5 &= \text{GOTO}(I_3, X) = \{X \to aX \cdot b\} \\ I_6 &= \text{GOTO}(I_5, b) = \{X \to aXb\cdot\} \end{aligned}$$

There is also a self-loop:

$$\text{GOTO}(I_3, a) = I_3$$

(b) SLR Parsing Table

The FOLLOW sets are

$$\text{FOLLOW}(S) = \{\$\}, \qquad \text{FOLLOW}(X) = \{a, b\}.$$

State	**ACTION**			**GOTO**	
	a	b	\$	S	X
0	$s3$			1	2
1			acc		
2	$s4$				
3	$s3/r2$	$r2$			5
4			$r1$		
5		$s6$			
6	$r3$	$r3$			

State 3 contains a shift/reduce conflict on input a. We choose **shift**.

(c) Parsing the string *aaba*

Step	**Stack**	**Input**	**Action**
1	0	$aaba\$$	$s3$
2	0 3	$aba\$$	$s3$
3	0 3 3	$ba\$$	$r2: X \to a$
4	0 3 5	$ba\$$	$s6$
5	0 3 5 6	$a\$$	$r3: X \to aXb$
6	0 2	$a\$$	$s4$
7	0 2 4	\$	$r1: S \to Xa$
8	0 1	\$	**accept**

Conclusion

The SLR parser correctly accepts the string `aaba`. During reductions (e.g., Step 5), the parser pops a number of states equal to the length of the right-hand side and uses the GOTO table to push the next state corresponding to the reduced nonterminal.

4-4

LR(0) Automaton and SLR Parsing

Consider the following grammar:

$$S \to aSc \mid Td$$
$$T \to Tb \mid b$$

a. Construct the LR(0) DFA of the grammar.
b. Construct the SLR parsing table.
c. Is this grammar SLR ? Justify your answer.
d. Use the parsing table to simulate SLR parsing on the string: aabdcc

Solution:

To solve this problem, we follow the standard procedure for augmenting a grammar, constructing an LR(0) automaton, and building an SLR parsing table.

(a) LR(0) Automaton

We first augment the grammar by adding a new start symbol S' and the production

$$S' \to S.$$

Augmented Grammar G'

$$\begin{array}{lccl} (0) & S' & \to & S \\ (1) & S & \to & aSc \\ (2) & S & \to & Td \\ (3) & T & \to & Tb \\ (4) & T & \to & b \end{array}$$

The LR(0) item sets are obtained using **CLOSURE** and **GOTO**.

$$
\begin{aligned}
I_0 &= \text{CLOSURE}(\{S' \to \cdot S\}) \\
&= \{S' \to \cdot S,\ S \to \cdot aSc,\ S \to \cdot Td,\ T \to \cdot Tb,\ T \to \cdot b\} \\
I_1 &= \text{GOTO}(I_0, S) = \{S' \to S\cdot\} \quad \text{(accept)} \\
I_2 &= \text{GOTO}(I_0, a) \\
&= \text{CLOSURE}(\{S \to a \cdot Sc\}) \\
&= \{S \to a \cdot Sc,\ S \to \cdot aSc,\ S \to \cdot Td,\ T \to \cdot Tb,\ T \to \cdot b\} \\
I_3 &= \text{GOTO}(I_0, T) = \{S \to T \cdot d,\ T \to T \cdot b\} \\
I_4 &= \text{GOTO}(I_0, b) = \{T \to b\cdot\} \\
I_5 &= \text{GOTO}(I_2, S) = \{S \to aS \cdot c\} \\
I_6 &= \text{GOTO}(I_3, d) = \{S \to Td\cdot\} \\
I_7 &= \text{GOTO}(I_3, b) = \{T \to Tb\cdot\} \\
I_8 &= \text{GOTO}(I_5, c) = \{S \to aSc\cdot\}
\end{aligned}
$$

Some important transitions:

$$\text{GOTO}(I_2, a) = I_2, \qquad \text{GOTO}(I_2, T) = I_3, \qquad \text{GOTO}(I_2, b) = I_4.$$

(b) SLR Parsing Table

The FOLLOW sets are:

$$\text{FOLLOW}(S) = \{c, \$\}, \qquad \text{FOLLOW}(T) = \{d, b\}.$$

State	**ACTION**					**GOTO**	
	a	b	c	d	\$	S	T
0	$s2$	$s4$				1	3
1					acc		
2	$s2$	$s4$				5	3
3		$s7$		$s6$			
4		$r4$		$r4$			
5			$s8$				
6			$r2$		$r2$		
7		$r3$		$r3$			
8			$r1$		$r1$		

(c) Is the Grammar SLR?

Yes. The SLR table contains no shift/reduce or reduce/reduce conflicts. Each table entry has at most one action.

(d) Parsing `aabdcc`

Step	**Stack**	**Input**	**Action**
1	0	$aabdcc\$$	$s2$
2	0 2	$abdcc\$$	$s2$
3	0 2 2	$bdcc\$$	$s4$
4	0 2 2 4	$dcc\$$	$r4 : T \to b,\ \text{GOTO}(2, T) = 3$
5	0 2 2 3	$dcc\$$	$s6$
6	0 2 2 3 6	$cc\$$	$r2 : S \to Td,\ \text{GOTO}(2, S) = 5$
7	0 2 2 5	$cc\$$	$s8$
8	0 2 2 5 8	$c\$$	$r1 : S \to aSc,\ \text{GOTO}(2, S) = 5$
9	0 2 5	$c\$$	$s8$
10	0 2 5 8	$\$$	$r1 : S \to aSc,\ \text{GOTO}(0, S) = 1$
11	0 1	$\$$	**accept**

4-5

Consider the following grammar:

$$S \to T, S \mid \epsilon$$
$$T \to int 0$$

a. Construct the LR(0) DFA of the grammar.
b. Construct the SLR parsing table.
c. Is this grammar SLR ? Justify your answer.
d. Use the parsing table to simulate SLR parsing on the string: int 0 , int 0

Solution:

a. Construct the LR(0) DFA of the grammar.

First, we augment the grammar with $S' \to S$ to create a unique starting point [1]. The augmented grammar G' is: (0) $S' \to S$, (1) $S \to T, S$, (2) $S \to \epsilon$, (3) $T \to int\ 0$.

We compute the sets of items using the **CLOSURE** and **GOTO** functions [1, 2]:

- **State I_0:** CLOSURE($\{S' \to \cdot S\}$)
  - $S' \to \cdot S$
  - $S \to \cdot T, S$
  - $S \to \cdot$ (reduction item for $S \to \epsilon$) [3]
  - $T \to \cdot int\ 0$ (added because dot is before T) [1]
- **State I_1:** GOTO$(I_0, S) = \{S' \to S\cdot\}$
- **State I_2:** GOTO$(I_0, T) = \{S \to T\cdot, S\}$
- **State I_3:** GOTO$(I_0, int) = \{T \to int \cdot 0\}$
- **State I_4:** GOTO$(I_2, ,) =$ CLOSURE$(\{S \to T, \cdot S\})$
  - $S \to T, \cdot S$
  - $S \to \cdot T, S$
  - $S \to \cdot$
  - $T \to \cdot int\ 0$
- **State I_5:** GOTO$(I_3, 0) = \{T \to int\ 0\cdot\}$
- **State I_6:** GOTO$(I_4, S) = \{S \to T, S\cdot\}$

Note: Transitions for T and int from I_4 lead back to I_2 and I_3 respectively.

b. Construct the SLR parsing table.

We need the **FOLLOW** sets to place reduction actions [4]:

- FOLLOW(S) = {\$}
- FOLLOW(T) = {,}

State	ACTION				GOTO	
	int	**0**	**,**	**\$**	**S**	**T**
0	s3			r2	1	2
1				acc		
2			s4			
3		s5				
4	s3			r2	6	2
5			r3			
6				r1		

c. Is this grammar SLR? Justify your answer.

Yes, this grammar is SLR. A grammar is SLR if there are no conflicting actions in its parsing table [4]. In State 0 and State 4, we have both a shift action (on *int*) and a reduce action ($r2$ on \$). Because the terminal *int* is not in FOLLOW(S), these actions occupy different columns, and no shift/reduce or reduce/reduce conflicts occur [4, 5].

d. Use the parsing table to simulate SLR parsing on the string: `int 0 , int 0`

Step	Stack	Input	Action
(1)	0	int 0 , int 0 \$	shift to 3
(2)	0 3	0 , int 0 \$	shift to 5
(3)	0 3 5	, int 0 \$	reduce by $T \rightarrow int\ 0$ ($r3$)
(4)	0 2	, int 0 \$	shift to 4
(5)	0 2 4	int 0 \$	shift to 3
(6)	0 2 4 3	0 \$	shift to 5
(7)	0 2 4 3 5	\$	**ERROR**

Justification of Error: The simulation results in an error at step 7 because $ACTION[5, \$]$ is empty. In this grammar, every T (*int* 0) must be followed by a comma to satisfy $S \rightarrow T, S$, or the string must be empty (ϵ) to satisfy $S \rightarrow \epsilon$ [3, 6]. The provided string `int 0 , int 0` is missing a trailing comma and is therefore not in the language defined by the grammar.

4-6

LR(0) Automaton and SLR Parsing

Consider the following grammar:

$$S \to AaAb \mid BbBa$$
$$A \to \epsilon$$
$$B \to \epsilon$$

Is it LL(1)? Is it SLR?

Solution:

Is it LL(1)?

Yes, the grammar is LL(1).

Why it's LL(1): A top-down parser can easily distinguish between starting with A or B based on whether the first token it sees is a or b.

Is it SLR?

No, the grammar is not SLR.

Why it fails SLR(1): The SLR parser will reach a state where it needs to decide whether to reduce $A \to \epsilon$ or $B \to \epsilon$. Because both a and b can follow these non-terminals in different parts of the grammar, the FOLLOW sets overlap, creating a conflict the SLR parser cannot resolve.

4-7

Consider the following grammar:

$$S \to SA \mid A$$
$$A \to a$$

Show that the grammar is SLR but not LL(1).

Solution:

The Grammar is Not LL(1)

The grammar cannot be LL(1) if it is left-recursive.

In the production $S \to SA$, the non-terminal S appears as the very first symbol on the right-hand side. This creates an infinite loop for a top-down parser: to parse S, it first tries to parse S, which requires it to parse S again, and so on.

Formal LL(1) Conflict: For a grammar to be LL(1), the FIRST sets of the productions for a non-terminal must be disjoint.

$FIRST(SA) = FIRST(S) = \{a\}$

$FIRST(A) = \{a\}$

Since the FIRST sets for both productions of S overlap $a \in \{a\} \cap \{a\}$, the parser cannot decide which production to choose based on one token of lookahead.

The Grammar is SLR

To prove a grammar is SLR, we must construct the SLR parsing table and verify that there are no conflicts.

4-8

Consider the following grammar:

$$A \to BCD$$
$$B \to Bb \mid \epsilon$$
$$C \to cC \mid \epsilon$$
$$D \to d$$

Is this grammar LR(0)?

Solution:

The Augmented Grammar
First, we add a new starting production to create a clear "Accept" state:

$$0.A' \to A$$
$$1.A \to BCD$$
$$2.B \to Bb$$
$$3.B \to \epsilon$$
$$4.C \to cC$$
$$5.C \to \epsilon$$
$$6.D \to d$$

Constructing State I_0
$I_0 = \{A' \to \cdot A, A \to \cdot BCD, B \to \cdot Bb, B \to \cdot\}$

Identifying the LR(0) Conflict
An LR(0) grammar cannot have any state that contains a completed reduction alongside another item.

In State I_0, we have a clear Shift/Reduce conflict:

Reduce Item: $B \to \cdot$

Shift Items: $A \to \cdot BCD$ (requires a shift on B) and $B \to \cdot Bb$ (requires a shift on B).

In an LR(0) parser, if a state contains $B \to \cdot$, the parser is told to "Reduce by $B \to \epsilon$" regardless of the next input token. However, in the same state, the parser is also told to "Shift" if it sees symbols that could start B. This ambiguity makes the grammar not LR(0).

4-9

SLR and LALR Parsing

Show that the following grammar is LALR but not SLR.

$$S \to Aa \mid bAc \mid dc \mid bda$$
$$A \to d$$

Trace the operation of an LALR parser on input sentence

Solution:

To solve this problem, we will apply the concepts of core-equivalence and lookahead propagation that we've discussed. This specific grammar is a classic example used to show how SLR's reliance on global FOLLOW sets can lead to conflicts that more advanced parsers, like LALR, can resolve by tracking context.

1. Why the grammar is NOT SLR

To determine if the grammar is SLR, we first compute the FOLLOW sets for the non-terminals:

- FOLLOW(S) = {\$}
- FOLLOW(A) = $\{a, c\}$ (from $S \to Aa$ and $S \to bAc$)

Now, let's look at the LR(0) item sets where conflicts arise:

State I_2 (reached from I_0 on terminal d):

$$S \to d.c$$
$$A \to d.$$

In SLR, for the item $A \to d.$, we call for a reduction if the next input symbol is in FOLLOW(A), which is $\{a, c\}$. However, the item $S \to d.c$ calls for a shift on terminal c. Because $c \in$ FOLLOW(A), there is a **shift–reduce conflict** on input c.

State I_3 (reached from I_1 on terminal d, where $I_1 = \mathbf{GOTO}(I_0, b)$):

$$S \to bd.a$$
$$A \to d.$$

Similarly, FOLLOW(A) includes a. The item $S \to bd.a$ calls for a shift on a, while $A \to d.$ calls for a reduction on a. This is a second **shift–reduce conflict**. Because of these conflicts, the grammar is not SLR.

2. Why the grammar IS LALR

LALR resolves these conflicts by using specific lookaheads rather than the broad FOLLOW set. Let us examine the LR(1) items for these states.

In State I_2 (prefix d): The item $A \to .d$ is generated by $S \to .Aa$. The lookahead for this A production is

$$\text{FIRST}(a\$) = \{a\}.$$

$$\begin{aligned} S &\to d.c, \quad \$ \\ A &\to d., \quad a \end{aligned}$$

The parser only reduces $A \to d$ if the next symbol is a. If the next symbol is c, it correctly chooses to shift for $S \to dc$. **No conflict.**

In State I_3 (prefix bd): The item $A \to .d$ is generated by $S \to b.Ac$. The lookahead for this A production is

$$\text{FIRST}(c\$) = \{c\}.$$

$$\begin{aligned} S &\to bd.a, \quad \$ \\ A &\to d., \quad c \end{aligned}$$

The parser only reduces $A \to d$ if the next symbol is c. If the next symbol is a, it correctly chooses to shift for $S \to bda$. **No conflict.**

Since the LR(1) automaton has no conflicts, the LALR automaton (which merges states with the same core) also has no conflicts in this case. Therefore, the grammar is **LALR**.

3. Tracing the LALR Parser on "bdc"

We trace the operation on the string **"bdc"** to show how the parser distinguishes between the different roles of the terminal d.

Stack	**Input**	**Action**	**Explanation**
0	bdc$	shift I_1	Shift b.
0 b 1	dc$	shift I_3	Shift d.
0 b 1 d 3	c$	reduce $A \to d$	**Lookahead is** c, so reduce $A \to d$ (Rule 5).
0 b 1 A 4	c$	shift 5	GOTO(1,A) leads to state 4. Shift c.
0 b 1 A 4 c 5	$	reduce $S \to bAc$	Reduce by Rule 2.
0 S 6	$	acc	Accept string.

4-10

LALR and LR(1) Parsing

Show that the following grammar is LR(1) but not LALR.

$$S \to Aa \mid bAc \mid Bc \mid bBa$$
$$A \to d$$
$$B \to d$$

Solution:

To solve this problem, we must look at how Canonical LR(1) and LALR handle context differently.

As we discussed, LALR is more efficient because it merges states with the same "core," but this merger can sometimes introduce reduce/reduce conflicts that were not present in the original LR(1) automaton.

Consider the augmented grammar G':

(0) $S' \to S$
(1) $S \to Aa$
(2) $S \to bAc$
(3) $S \to Bc$
(4) $S \to bBa$
(5) $A \to d$
(6) $B \to d$

1. Showing the Grammar is LR(1)

In a Canonical LR(1) parser, states are distinguished by their lookaheads. Let us examine the two critical paths for the terminal d.

Path 1: After reading prefix d The parser is at the start of either $S \to Aa$ or $S \to Bc$.

- For $S \to .Aa,\ \$$, the closure gives $A \to .d,\ a$.
- For $S \to .Bc,\ \$$, the closure gives $B \to .d,\ c$.

This results in an LR(1) state (call it I_x):

$$I_x = \{[A \to d., a],\ [B \to d., c]\}.$$

In this state, if the lookahead is a, the parser reduces d to A; if it is c, it reduces d to B. There is no conflict.

Path 2: After reading prefix bd The parser is at the start of $S \to b.Ac$ or $S \to b.Ba$.

- For $S \to b.Ac,\ \$$, the closure gives $A \to .d,\ c$.
- For $S \to b.Ba,\ \$$, the closure gives $B \to .d,\ a$.

This produces a different LR(1) state (call it I_y):

$$I_y = \{[A \to d., c],\ [B \to d., a]\}.$$

Again, the lookaheads clearly distinguish which reduction to take, so no conflict exists in LR(1).

2. Showing the Grammar is *Not* LALR

The LALR construction merges states that are core-equivalent, meaning they have the same LR(0) items regardless of lookaheads.

States I_x and I_y share the same core:

$$\{A \to d.,\ B \to d.\}.$$

When merged into a single LALR state I_{xy}, the lookaheads are unioned:

$$I_{xy} = \{[A \to d., \{a, c\}],\ [B \to d., \{a, c\}]\}.$$

Now, if the parser is in state I_{xy} and the next input symbol is a, it has two possible reductions:

1. Reduce by $A \to d$ (since $a \in \text{Lookahead}(A \to d)$),
2. Reduce by $B \to d$ (since $a \in \text{Lookahead}(B \to d)$).

This is a reduce/reduce conflict. Therefore, although the grammar is LR(1), it is not LALR.

4-11

Canonical LR(1) Parsing

Consider the following grammar:

$$S \to Xa$$
$$X \to a \mid aXb$$

a. Compute the LR(1) item set and construct the LR(1) DFA of the grammar.

b. Construct the canonical LR(1) parsing table.

c. Use the parsing table to simulate canonical LR(1) parsing on the string: aaba

Solution:

To solve this problem, we follow the formal procedures for augmenting the grammar, constructing the canonical collection of LR(1) item sets, building the parsing table, and tracing the input string [1, 2].

The grammar is:

1. $S \to Xa$
2. $X \to a$
3. $X \to aXb$

First, we augment the grammar with a new start symbol S' and an end-marker \$ [3, 4]:
(0) $S' \to S, \$$

a. LR(1) Item Sets and DFA

We compute the sets of items using the CLOSURE and GOTO functions [3, 5].
I_0 **(Initial State):**
CLOSURE($\{[S' \to .S, \$]\}$):

1. $[S' \to .S, \$]$
2. $[S \to .Xa, \$]$ (Since FIRST($\epsilon a\$$) = $\{a\}$ – *Correction: lookahead is 'a'*)
3. $[X \to .a, a]$ (Since FIRST($a\$$) = $\{a\}$)
4. $[X \to .aXb, a]$ (Since FIRST($a\$$) = $\{a\}$)

$I_0 = \{[S' \to .S, \$], [S \to .Xa, \$], [X \to .a, a], [X \to .aXb, a]\}$

$I_1 = \mathbf{GOTO}(I_0, S) = \{[S' \to S., \$]\}$
$I_2 = \mathbf{GOTO}(I_0, X) = \{[S \to X.a, \$]\}$

$I_3 = \mathbf{GOTO}(I_0, a)$**:**
Core: $[X \to a., a]$ and $[X \to a.Xb, a]$
CLOSURE of $[X \to a.Xb, a]$ adds X productions with lookahead FIRST$(ba) = \{b\}$:
$I_3 = \{[X \to a., a], [X \to a.Xb, a], [X \to .a, b], [X \to .aXb, b]\}$
$I_4 = \mathbf{GOTO}(I_2, a) = \{[S \to Xa., \$]\}$
$I_5 = \mathbf{GOTO}(I_3, X) = \{[X \to aX.b, a]\}$
$I_6 = \mathbf{GOTO}(I_3, a)$**:**
Core: $[X \to a., b]$ and $[X \to a.Xb, b]$
CLOSURE adds X productions with lookahead FIRST$(bb) = \{b\}$:
$I_6 = \{[X \to a., b], [X \to a.Xb, b], [X \to .a, b], [X \to .aXb, b]\}$
$I_7 = \mathbf{GOTO}(I_5, b) = \{[X \to aXb., a]\}$
$I_8 = \mathbf{GOTO}(I_6, X) = \{[X \to aX.b, b]\}$
$I_9 = \mathbf{GOTO}(I_8, b) = \{[X \to aXb., b]\}$
Transitions: $GOTO(I_6, a) = I_6$.

b. Canonical LR(1) Parsing Table

We fill the table based on the items in each state [2, 6]. For the conflict in State 3, we follow the standard convention of shifting over reducing.

State	**ACTION**			**GOTO**	
	a	**b**	**$**	**S**	**X**
0	s3			1	2
1			acc		
2	s4				
3	s6 / r2*				5
4			r1		
5		s7			
6	s6	r2			8
7	r3				
8		s9			
9		r3			

**Note: Conflict in State 3. We use s6 for the simulation [7].*

c. Simulation on string "aaba"

Stack	**Input**	**Action**	**Explanation**
0	aaba$	s3	Shift 'a', go to state 3.
0 a 3	aba$	s6	Shift 'a' (resolving conflict), go to state 6.
0 a 3 a 6	ba$	r2	Reduce by $X \to a$. Pop 6. GOTO$(3, X) = 5$.
0 a 3 X 5	ba$	s7	Shift 'b', go to state 7.
0 a 3 X 5 b 7	a$	r3	Reduce by $X \to aXb$. Pop 7, 5, 3. GOTO$(0, X) = 2$.
0 X 2	a$	s4	Shift 'a', go to state 4.
0 X 2 a 4	$	r1	Reduce by $S \to Xa$. Pop 4, 2. GOTO$(0, S) = 1$.
0 S 1	$	acc	Accept string.

4-12

LALR Parsing

Consider the context-free grammar $G = (\{S, T\}, \{0, 1\}, R, S)$, where R is the set of the following rules.

$$S \to 0T$$
$$T \to 1ST$$
$$T \to 0$$

The LALR automaton of G consists of the following states.

$$I_0 : S' \to .S, \$$$
$$S \to .0T, \$$$

$$I_1 : S' \to S., \$$$

$$I_2 : S \to 0.T, \$/1/0$$
$$T \to .1ST, \$/1/0$$
$$T \to .0, \$/1/0$$

$$I_3 : S \to 0T., \$/1/0$$

$$I_4 : T \to 1.ST, \$/1/0$$
$$S \to .0T, 1/0$$

$$I_5 : T \to 0., \$/1/0$$

$$I_6 : T \to 1S.T, \$/1/0$$
$$T \to .1ST, \$/1/0$$
$$T \to .0, \$/1/0$$

$$I_7 : T \to 1ST., \$/1/0$$

a. With δ being the transition function of the LALR automaton, which states are $\delta(I_2, 0)$ and $\delta(I_6, 1)$?

b. Construct the LALR parsing table for G.

c. Trace the operation of the LR parsing algorithm on input 01000 using the LALR parsing table from part (2) above. (Show the sequence of stack, input, and chosen action.)

Solution:

Using the provided grammar and LALR automaton states, we will determine the transitions, construct the parsing table, and trace the input string.

The grammar rules are:

$$(1)\ S \to 0T \qquad (2)\ T \to 1ST \qquad (3)\ T \to 0$$

a. Transition Functions

The transition function δ (or GOTO) for an LALR automaton is determined by moving the dot over the specified symbol in the items of a state and finding the resulting set of items.

- $\delta(I_2, 0)$**:** In state I_2, we have the item $[T \to .0, \$/1/0]$. Moving the dot over 0 results in the item $[T \to 0., \$/1/0]$. This set matches state $\mathbf{I_5}$.
- $\delta(I_6, 1)$**:** In state I_6, we have the item $[T \to .1ST, \$/1/0]$. Moving the dot over 1 results in the item $[T \to 1.ST, \$/1/0]$. This core matches state $\mathbf{I_4}$.

b. LALR Parsing Table

The ACTION table is filled by identifying shifts based on terminal transitions and reductions based on completed items ($A \to \alpha.$) and their lookaheads. The GOTO table is filled using transitions on non-terminals.

State	**ACTION**			**GOTO**	
	0	**1**	**$**	**S**	**T**
0	s2			1	
1			acc		
2	s5	s4			3
3	r1	r1	r1		
4	s2			6	
5	r3	r3	r3		
6	s5	s4			7
7	r2	r2	r2		

Table Logic:

- **State 0:** Transitions on '0' lead to I_2 (s2). Transition on 'S' leads to I_1.
- **State 2:** Transitions on '0' lead to I_5 (s5) and on '1' to I_4 (s4). Transition on 'T' leads to I_3.
- **State 3:** Contains $[S \to 0T., \$/1/0]$, calling for reduction by rule 1 on all lookaheads.
- **State 4:** Transition on '0' leads to I_2 (s2). Transition on 'S' leads to I_6.

- **State 5:** Contains $[T \to 0., \$/1/0]$, calling for reduction by rule 3.
- **State 6:** Transitions on '0' lead to I_5 (s5) and on '1' to I_4 (s4). Transition on 'T' leads to I_7.
- **State 7:** Contains $[T \to 1ST., \$/1/0]$, calling for reduction by rule 2.

c. Trace on input "01000"

The simulation follows the standard LR parsing algorithm.

Stack	**Input**	**Action**	**Explanation**
0	01000$	s2	Shift 0, go to state 2.
0 0 2	1000$	s4	Shift 1, go to state 4.
0 0 2 1 4	000$	s2	Shift 0, go to state 2.
0 0 2 1 4 0 2	00$	s5	Shift 0, go to state 5.
0 0 2 1 4 0 2 0 5	0$	r3	$T \to 0$. Pop $(0, 5)$. GOTO$(2, T) = 3$.
0 0 2 1 4 0 2 T 3	0$	r1	$S \to 0T$. Pop $(0, 2, T, 3)$. GOTO$(4, S) = 6$.
0 0 2 1 4 S 6	0$	s5	Shift 0, go to state 5.
0 0 2 1 4 S 6 0 5	$	r3	$T \to 0$. Pop $(0, 5)$. GOTO$(6, T) = 7$.
0 0 2 1 4 S 6 T 7	$	r2	$T \to 1ST$. Pop $(1, 4, S, 6, T, 7)$. GOTO$(2, T) = 3$.
0 0 2 T 3	$	r1	$S \to 0T$. Pop $(0, 2, T, 3)$. GOTO$(0, S) = 1$.
0 S 1	$	acc	String accepted.