INTRODUCTION TO COMPILER

Topic _____ Date 15/Sep 2021 Page No. : _____

COMPILER:- A compiler is a converts high-level language into assembly language.

Step-1:- HLL → Removal of Pre-processors.

↓ pure HLL

Step-2:- Now, pure HLL will go to the compiler.

↓ Assembly lang.

Step-3:- Assembler

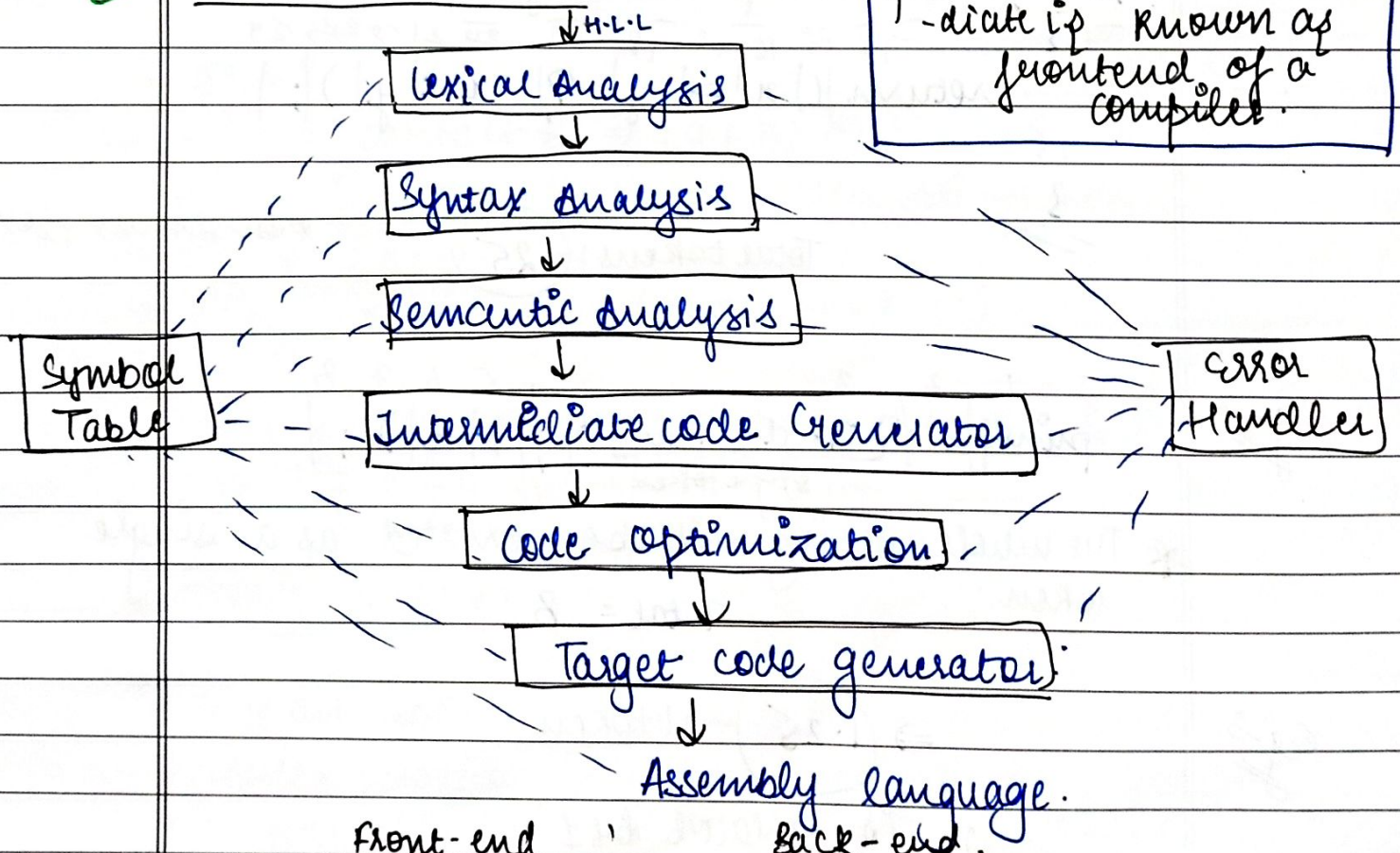
↓ Relocatable code.

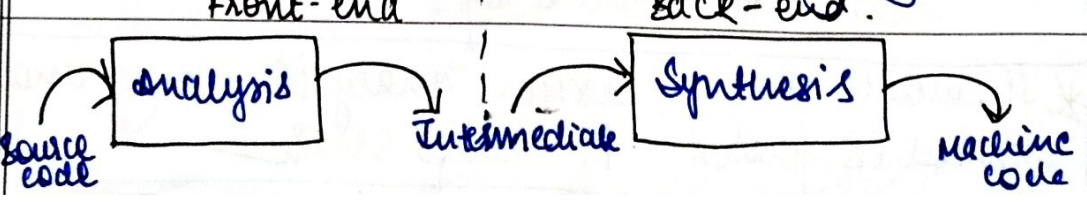
Step-4:- Linker/loader (connects secondary memory add. to primary memory)

↓ machine code.

PHASES OF COMPILER:-

from lexical to intermediate is known as frontend of a compiler.





Topic _____ Date 16/Sep/2021 Page No. :_____

(1.) LEXICAL ANALYSIS:- (Tokenization of program will happen).

Let we have a C program.

Eg-1

```
int max (x,y)
  int x, y;
/* find max. of x & y */
{ return (x>y ? x:y);
}
```

lexical analysis is the first phase of the compiler also known as scanner. It converts H.L input program into sequence of Tokens.

→ create tokens:-

Calculate the no. of tokens generated after lexical analysis!

1	2	3	4	5	6	7
int	max	(	x	,	y	)

8	9	10	11	12
int	x	,	y	;

/* find max. of x array */ → Remove. // comment

13	14	15	16	17	18	19	20	21	22	23	24
return	(	x	>	y	?	x	:	y	)	;	

25
}

Total tokens: 25

Eg-2

1	2	3	4	5	6	7	8
printf	(	"i = %d Hay"	,	&	x	)	;

single token

* The whole string will be treated as a single token.

Total = 8

Eg-3

⇒ 1.25 → 1 token

no. is replaced by 1.

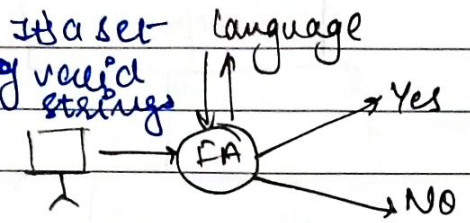
M/R

* If, while doing lexical analysis any undefined symbol is found, it shows error.

Topic _____ Date __/__/____ Page No. :_____

→ lex is a tool used in linux for lexical analysis.
→ yacc (from frontend we have this tool).
↳ yet another compiler compiler

17/Sep/2021



* INPUT ALPHABETS:-

Eg) $\Sigma = \{a, b\}$
L_1 = Set of all strings with length "2".

$L_1 = \{ab, aa, bb, ba\}$

Eg) $\Sigma = \{a-b, A-B, 0-9,$ special characters?
↳ Input Alphabets in C.

→ Σ^1 → length 1 $= \{a, b\}$
→ $\Sigma^2 = \Sigma^1$ concat $\Sigma^1 \Rightarrow \{a,b\} \cup \{a,b\} \cup \cdots$ so on.
→ $\Sigma^0 = \{\in\}$ ↳ concat operation.
→ Σ^* → Kleen closure $\Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cdots$
→ Σ^+ → positive closure. $= \Sigma^1 \cup \Sigma^2 \cup \cdots$ (we don't have $\in^0$)

if any unknown ~~type~~ input alphabet is detected, then it will ~~throw~~ throw an error.

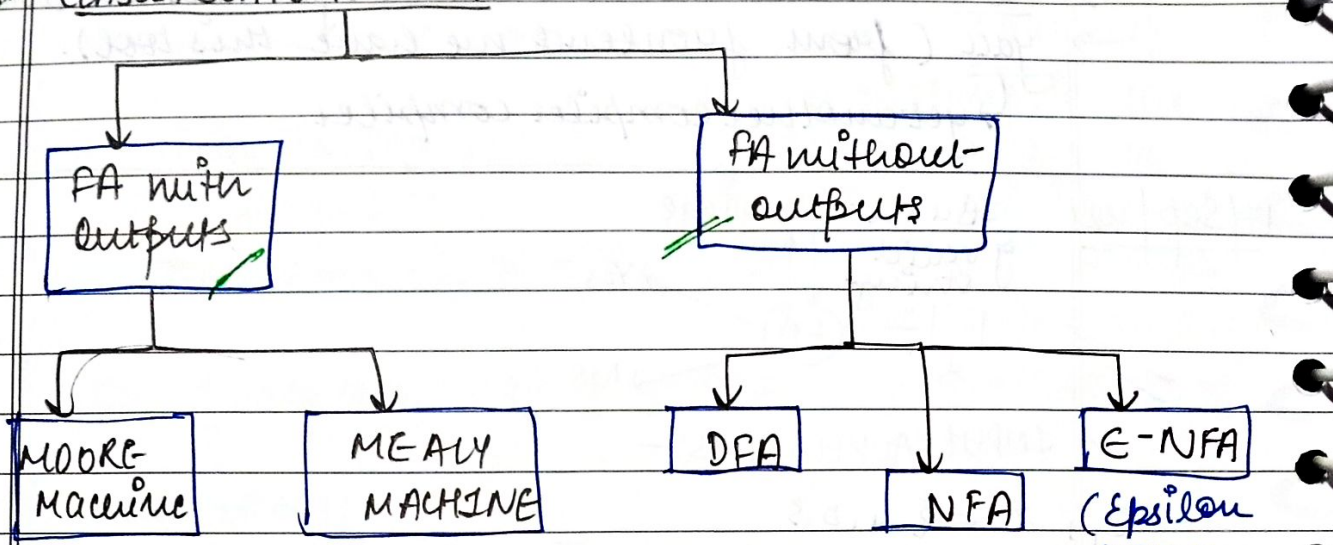
* FINITE AUTOMATA:-

- It is an abstract machine, which is useful to check valid strings from our program. (H.L.L)

MR

Topic _____ Date __/__/____ Page No. :_____

CLASSIFICATION OF FA :-



DFA :- (in this we have only one path)

→ Associated tuples :- parameters or Attributes

5-TUPLES

① Q ④ q_0
② Σ ⑤ δ
③ F

① Q : (Set of all states)
② Σ : (Input Alphabet)
③ F : (Set of final states)
④ q_0 : (Initial state)
⑤ δ : (Transaction from one state to another).

"$Q \rightarrow \Sigma \Rightarrow Q$"
(one state, read input, another state)

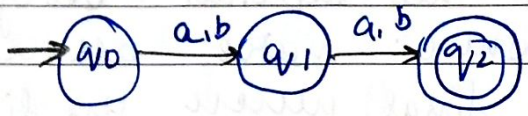
PROBLEM :- Create a DFA which accepts all strings over the Input Alphabet (Σ) $\Sigma = \{a, b\}$ with length "2".

SOLUTION :- Given :- $\Sigma = \{a, b\}$

$L = \{aa, bb, ab, ba\}$

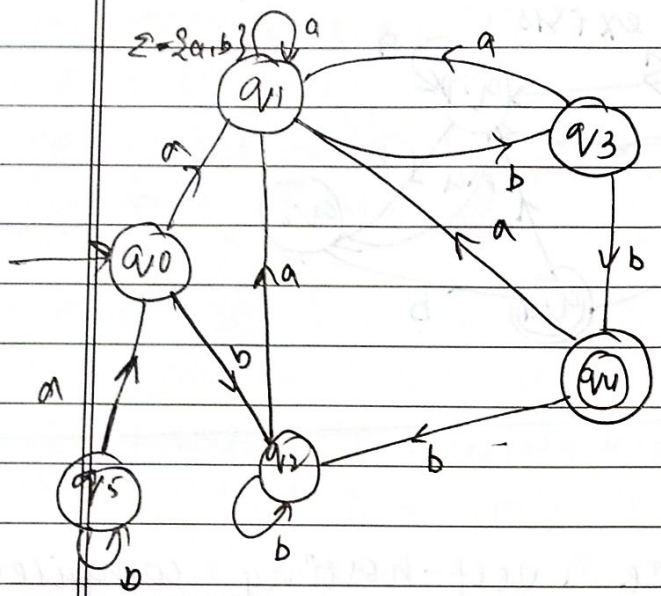
M R

Topic _____ Date 20/Sept/2021 Page No. : _____



DFA MINIMIZATION

(10 marks)



Step-1 :- Try to find all unreachable states in DFA & remove (delete) them & all its corresponding transactions. i.e. q_5.

Step-2 :- Draw state transaction diagram (STD) of the given DFA:

	a	b
q_0	q_1	q_2
q_1	q_1	q_3
q_2	q_1	q_2
q_3	q_1	q_4
q_4	q_1	q_2

Step-3 :- write all the states in a form of { set of all non-final states } { set of all final states }.

$\Rightarrow$ $\{q_0, q_1, q_2, q_3\}$ $\{q_4\}$ (0-equivalent)

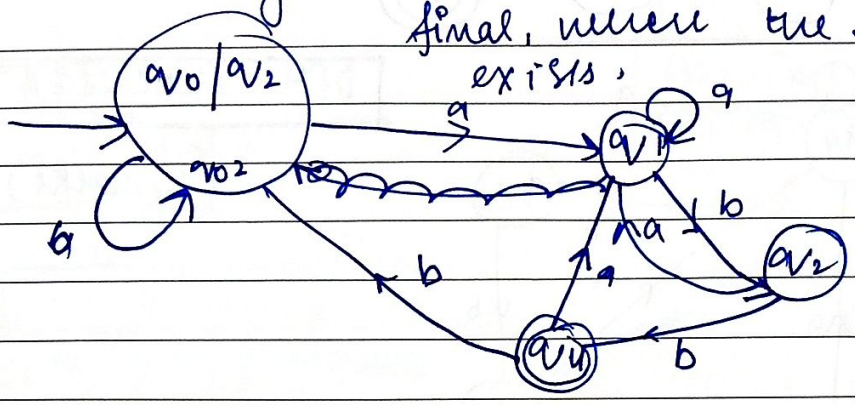
a a b c

On checking $\{q_0, q_1, q_2\}$ $\{q_3\}$ $\{q_4\}$ (1-equivalent)

$\{q_0, q_2\}$ $\{q_1\}$ $\{q_3\}$ $\{q_4\}$ (2-equivalent)

Topic _____ Date __/__/____ Page No. :_____

Step-4:- We make all separate set of states as a single state. make all the states as final, where the final state exists.

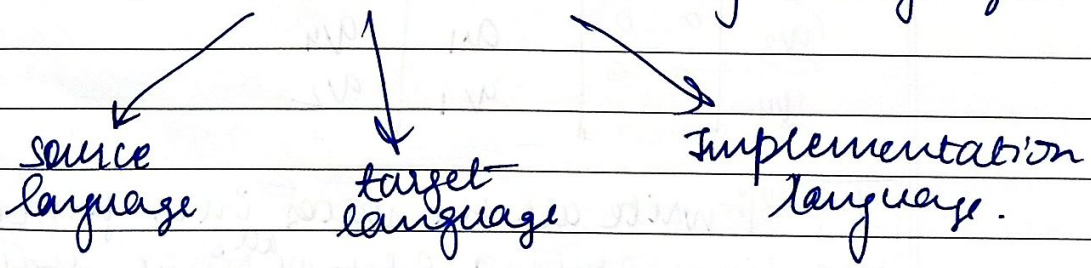


21/sep/2021

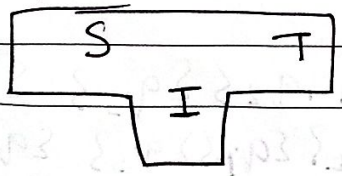
* **BOOTSTRAPPING**:-

- It is used to produce a self-hosting compiler. Self-hosting compiler is a type of compiler that can compile its own source code.
- Bootstrap compiler is used to compile the compiler & then you can use this compiled compiler to compile everything else as well as further version of itself.

A compiler can be characterized by 3 languages.



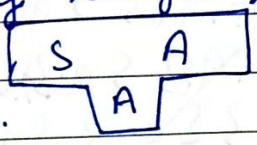
The T-diagram shows a compiler $S C_I^T$ for source S, Target T, implemented in I.



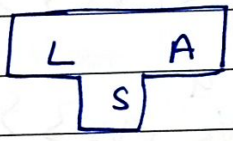
MR

Topic _____ Date __/__/____ Page No. :_____

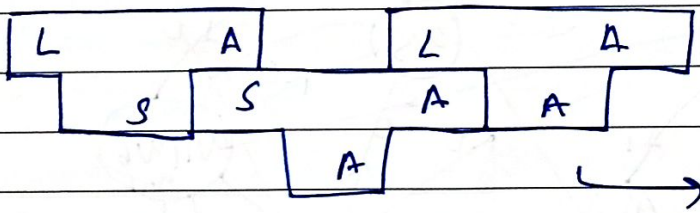
(1) Create a compiler ${}^{S}C_{A}{}^{A}$ for subset, S of the desired lang. L using language "A" and that compiler runs on machine A.



(2) Create a compiler ${}^{L}C_{S}{}^{A}$ for language L written in a subset of L.

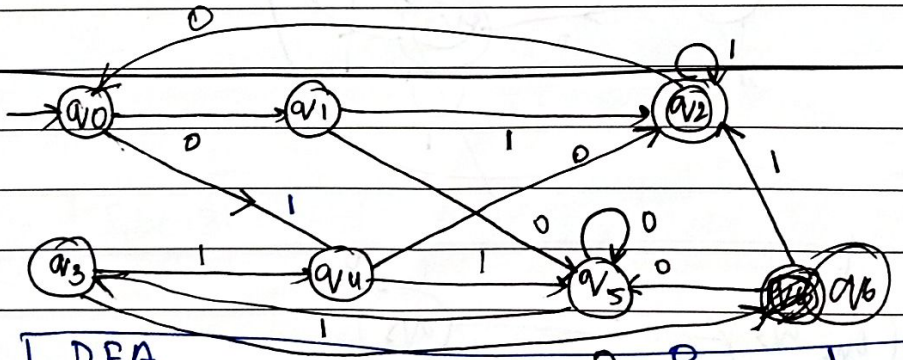


(3) Compile ${}^{L}C_{S}{}^{A}$ using the compiler ${}^{S}C_{A}{}^{A}$ to obtain ${}^{L}C_{A}$. ${}^{L}C_{A}{}^{A}$ is a compiler of language L, which runs a machine A and produces code for machine A.



↳ Bootstrapping.

22/Sep/2021



Minimize the DFA.

DFA		
q_0	q_1	q_4
q_1	q_3	q_5
q_2	q_0	q_2
q_3	q_6	q_4
q_4	q_2	q_5
q_5	q_3	q_5
q_6	q_5	q_2.

MR

Topic _____ Date ___/___/___ Page No. :_____

= {non-final set} {final set}

= {$q_0, q_1, q_3, q_4, q_5, q_6$} {$q_2$}

= {q_0, q_3, q_4, q_5, q_6} {q_1} {q_2}

= {q_3, q_4, q_5, q_6} {q_0} {q_1} {q_2}

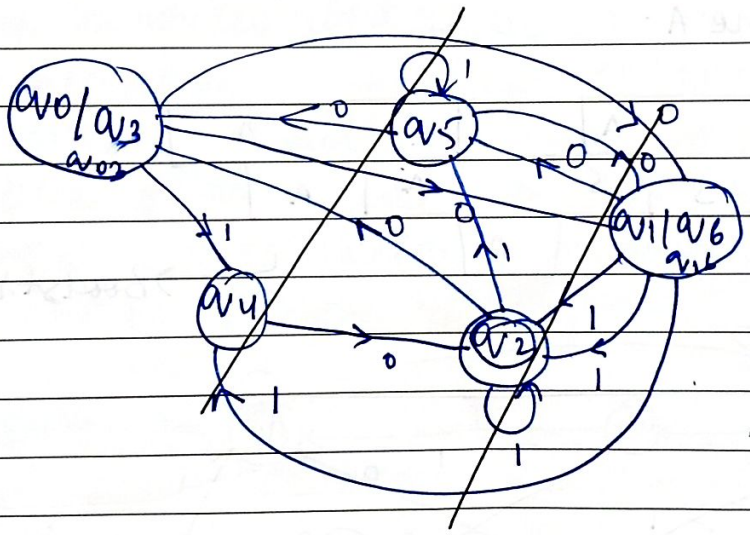
= {q_3, q_5, q_6} {q_4} {q_0} {q_1} {q_2}

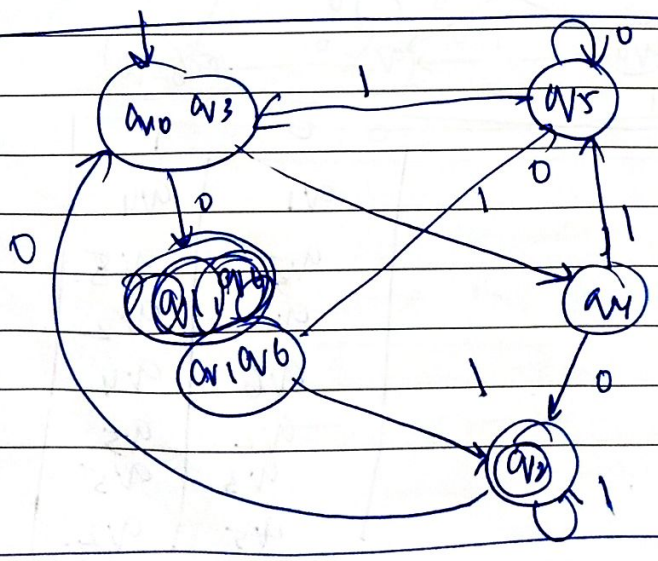
= {q_6} {q_3, q_5} {q_4}

= {q_0, q_3, q_5} {q_1, q_6} {q_4} {q_2}

	a	b
q_0	a	a
q_1	a	b
q_3	a	a
q_4	b	a
q_5	a	a
q_6	a	b

	b	c
q_0	b	c
q_3	b	c
q_5	a	a





Topic _____ Date 24/Sep/2021 Page No. : _____

GRAMMERS:-

→ 4 tuple

language

Input symbols (Strings)

- → variables → V
- → Terminals → T
- → Production → P
- → Start Symbol → S

eg:- "start" S → AA "or"

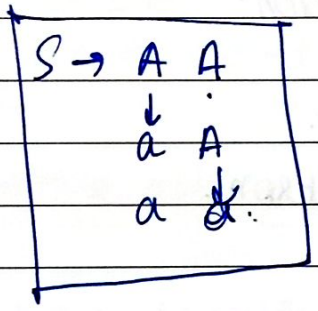
A → a b → production

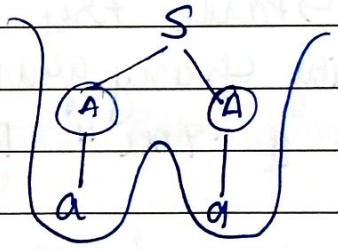
T T

$\Sigma = \{a, b\}$

L = {set of all strings of length 2}

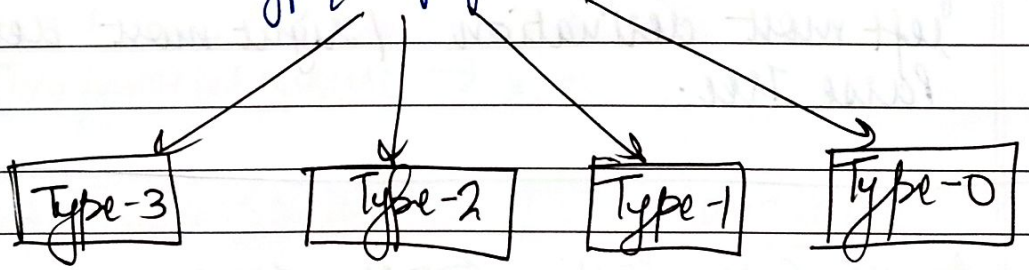
$L = \{aa, bb, ab, ba\}$





CLASSIFICATION OF GRAMMAR:-

There are 4 types of grammars:-



Regular grammar | Context free grammar | Context sensitive grammar | Recursive Grammar.

RL → Regular Lang.

LBA → Linear Bounded
PDA → Pushdown Automata

27/sep/2024 Topic Date / / Page No. :

UNIT-1 SUMMARY

1. Introduction to compiler / compiler overview.
2. Compiler Phases (From : lexical to target code generation)
3. Bootstrapping (concept)
4. Finite Automata (including it's classification)
5. Formal Grammar
6. Minimization of DFA.
7. Context free Grammar.
8. String Derivation
   - → left most derivation.
   - → Right most Derivation.
   - → Parse tree.
9. Ambiguity of any grammar.
10. "LEX" & "YACC" Tool Intro.

NOTE :- In order to derive a string from a given grammar, if we have more than one left most derivation / right most derivation / Parse Tree.

---

* Regular Expressions :- ~~Exact~~ It's a mathematical representation of a given language

→ for regular languages we have this regular expression as its corresponding mathematical representation.

MR

Topic _____ Date __/__/____ Page No. :_____

* Basic operations on Regular expressions:-

1. Union (+)
2. Concatenation (.)
3. Kleen Closure (*)
4. Positive closure (_+)

* Basic primitives of Regular expressions:-

Let's assume it is represented by 'r', then:-

(i.) $r = \phi = \{\}$

(ii.) $r = \in$

(iii) $a \in \Sigma$, $\Sigma = \{a, b\}$

* Operations applied on Regular expression:-

If we 2 regular exp. as r_1 & r_2, then:-

(i.) union → $r_1 + r_2$

(ii.) concat → $r_1 . r_2$

(iii) Kleen Closure → r_1^*

(iv.) positive closure → r_1^+

* Regular expression	Language
ϕ	$\{\}$
$\in$	$\{\in\}$
a	$\{a\}$
a^*	$\{\in, a, aa, aaa \ldots\}$
$a^+ \mid a.a^+$	$\{a, aa \ldots\}$ ↓ at least one should be there.
$(a+b)^*$	$\{\in, a, b, ab, ba, \ldots\}$

M/R

Topic _____ Date ___/___/_____ Page No. :_____

$b(a)^* = \{b, ba, baa, baaa \cdots\}$

$(ab+ba)^* = \{\epsilon, ab, ba, abba, baab \cdots\}$

Ques ⇒ Represent this lang. to a regular expression:- Set of all strings of length exactly 2 on input symbols $\Sigma = \{a, b\}$.

⇒ $L \to \{aa, bb, ab, ba\}$

$aa + bb + ab + ba$

$a(a+b) + b(a+b)$

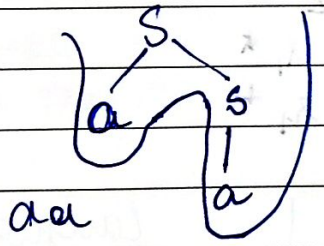
$(a+b).(a+b)$

29/Sep/2021 EXAMPLES OF UNIT-1

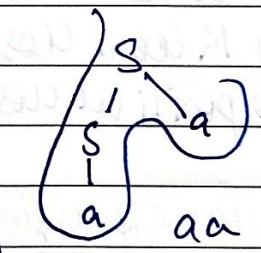
(1.) Questions on ambiguity.

⇒ (i.) $S \to aS \mid Sa \mid a$ - find out whether this grammar is ambiguous or not.

⇒



(i.)

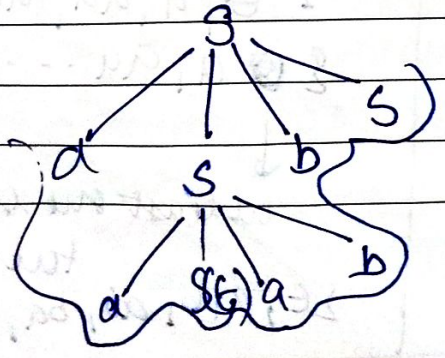


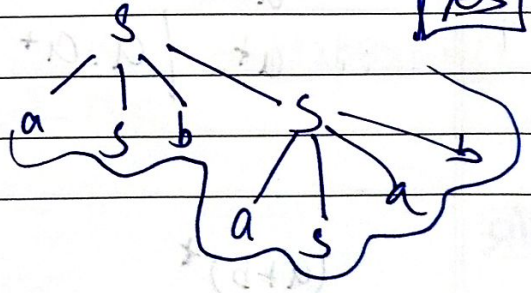
(ii.)

Yes

(ii.) $S \to aSbS \mid bSaS \mid \epsilon$

abab





Yes

MR

Topic _____ Date __/__/____ Page No. :_____

(iii) $R \to R+R \mid RR \mid R^* \mid a \mid b \mid c$ **Yes**

$a+bc$

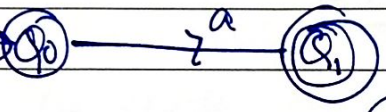
a. Questions on constructing a DFA.

(i) construct a minimal DFA, which accept set of all strings over input alphabets a, b, where each string starts with small a.

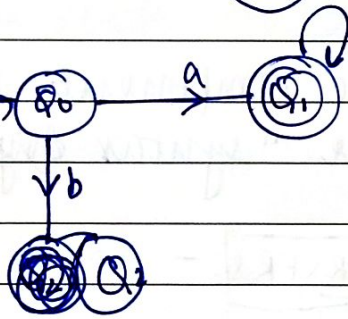
$\Sigma = \{a, b\}$

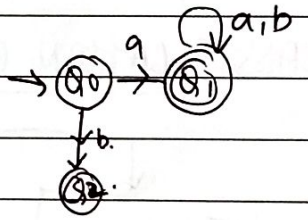
$L = \{a, ab, abb, \ldots\}$

• NFA:-



• DFA:-





(ii) each string containing 'a'

$\Sigma = \{a, b\}$

$L = \{a, ab, ba, aab, \ldots\}$

(iii) ending with small a.

(iv) starts with ab.

MR

Topic _____ Date __/__/____ Page No. _____

~~(iv) string starts with 'a'.~~

(v) starts with a & ends with b.

⇒ $a(a+b)^*b$.

(vi) starts & end with diff. symbol.

⇒ $a(a+b)^*b + b(a+b)^*a$

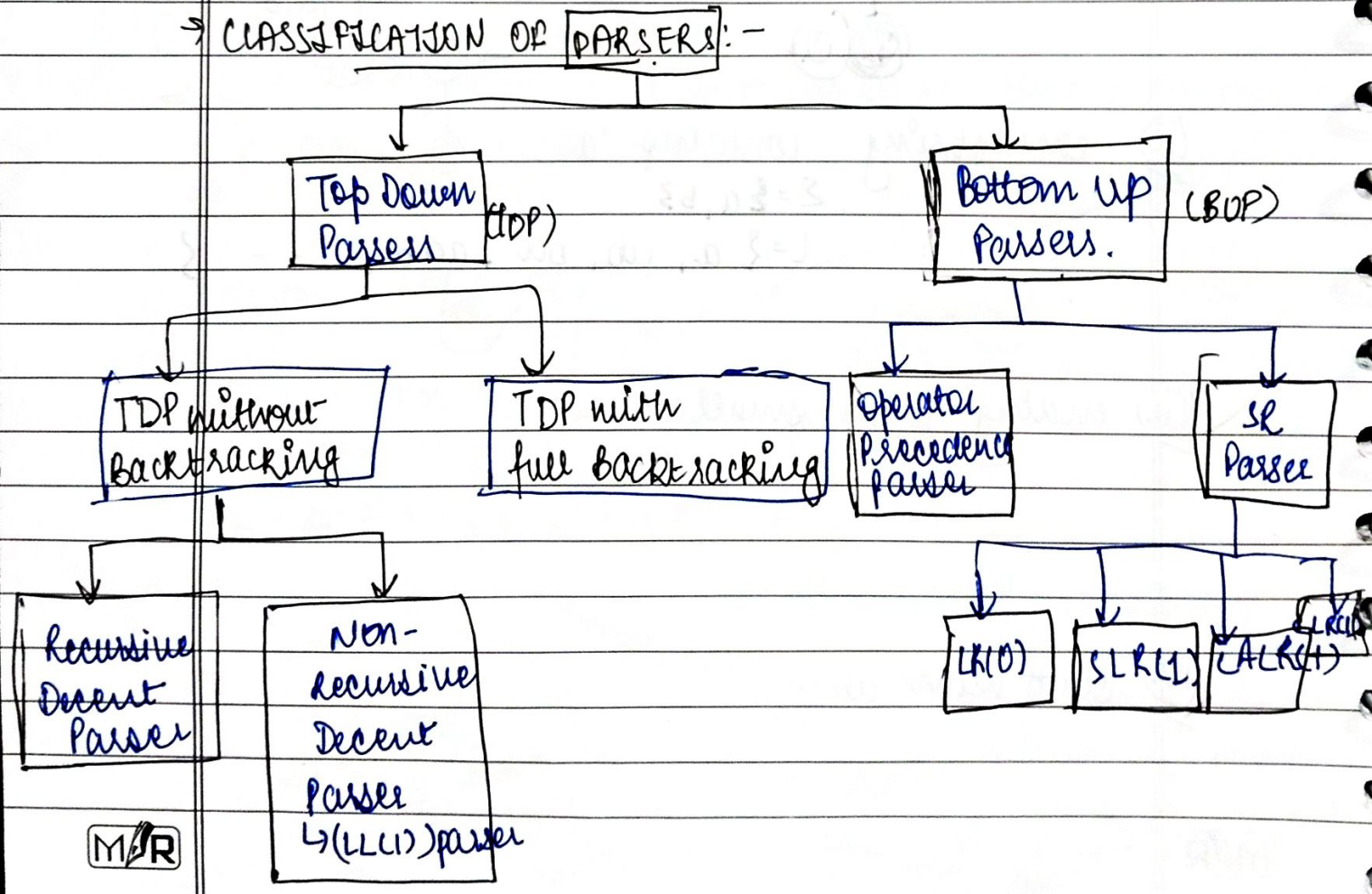
1/Oct/2024

UNIT-2

→ PARSERS

Parsers are the basic component which is used in the compiler phase "syntax ~~Analysis~~ Analysis".

→ CLASSIFICATION OF PARSERS:-



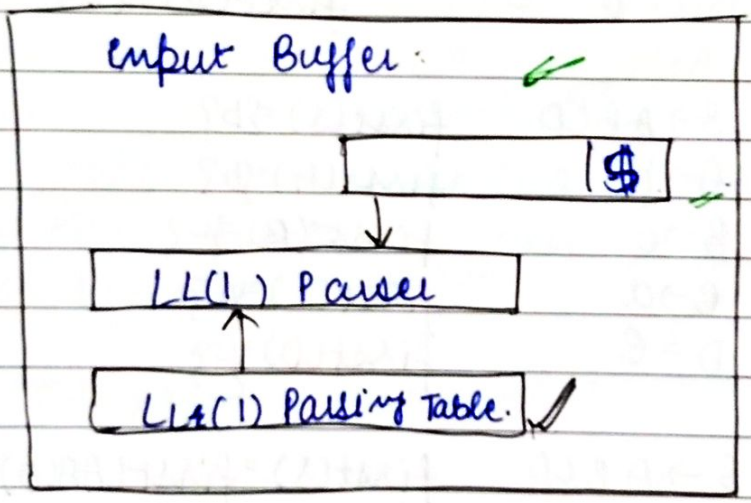
MJR

4/Oct/2021

Topic _____ Date __/__/____ Page No. : _____

NOTE:
- In all top-down parsers we follow leftmost derivation. ("It follows "what to use" approach).
- On other hand bottom-up parsers follows rightmost derivation in reverse order. (It follows "when to reduce" Approach).

* LL(1) PARSER:-
- L: Left to Right
- L: Left most derivation
- (1): no. of alphabets to be scanned. It is also known as look ahead symbol.



- Input buffer:- string to be scanned.
- Stack:- The data structure, which is used by LL(1) parser to create its parsing table.
- $:- end to string & or top of the stack.

- In order to create LL(1) parsing table we need to know the concept of: First( ) / Follow( ) /

M/R

RCFP.

Topic _____ Date __/__/____ Page No. :_____

→ CONCEPT OF FIRST() & FOLLOW():-

(1.) If we have a variable in our grammar from the variable, if we try to derive all strings, then whatever is coming at the first place is known as FIRST() of that variable.

(2.) Eg-1
S → aABCD. first(S) = first(aABCD) = first(a) = {a}
A → b. first(A) = {b}
B → c first(B) = {c}
C → d first(C) = {d}
D → e. first(D) = {e}

(3.) Eg-2
S → ABCD first(S) = {b}
A → b first(A) = {b}
B → c first(B) = {c}
C → d first(C) = {d}
D → e first(D) = {e}

(3.) Eg3
S → ABCD first(S) = first(ABCD) = {b, c}
A → b | ε ; b, first(A) = {b, ε}.
B → c first(B) = {c}.
C → d first(C) = {d}.
D → e. first(D) = {e}.

b and c for check for both.

→ first(bBCD) / first(BCD)

if B → c | ε
C → d | ε → first(S) = {b, c, d, e, $ε}
D → e | ε

M R

Topic _____ Date __/__/__ Page No. _____

→ CONCEPT OF FOLLOW( ): -

- String is always in the format of S$ where 'S' is string '$' is end string.
- Imp • follow( ) never contains 'ϵ', because if nothing is followed by variable, then it must be '$' at the end.
- STEPS TO FIND OUT FOLLOW:-

(1.) Follow( ) of start symbol (S) always consists atleast- $.

(2.) Look other variables including start symbol to the R.H.S of the production

(3.) Evaluate follow( ) of each variable by finding the first fun. of the upcoming string.

(4.) If variable is in the end, then follow( ) of that variable will be the follow( ) of the L.H.S.

Eg→1→ S→ ABCD
A→Ab | ϵ
B→ C
C→ d | ϵ
D→ e | ϵ

⇒ follow(S) = {$}.
follow(A) = first(BCD) → first(B) = {b,c}
follow(B) = first(CD) = first(C) = {d}
follow(C) = first(D) = {e}
follow(D) = & follow(S)

M/R

Topic _____ Date __/__/____ Page No. :_____

Q → S → ABCDE
A → a | ∈
B → b | ∈
C → c
D → d | ∈
E → e | ∈

first(A) =
first(A) = {a, b, c}
first(B) = {b, c}
first(C) =

find First & Follow.

⇒

Important

* EXAMPLES ON FIRST() & FOLLOW():-

(1)
S → ABCDE
A → a | ∈
B → b | ∈
C → c
D → d | ∈
E → e | ∈

⇒ first(S) = first(ABCD) = first(A) = {a, ∈}
first(A) = {a, ∈}
first(B) = {b, ∈}
first(C) = {c}
first(D) =
first(E) =

first(aBCD)
(BCD)

MR

Topic _____ Date __/__/____ Page No. :_____

first(S) = first(ABCDE) = first(A) = {a, b, c}
first(A) = first(aBCDE), first(BCDE) = {a, b, ∈}
first(B) = first(bBCDE), first(CDE) = {b, ∈}
first(C) = {c}
first(D) = first(dBCDE), first(E) = {d, ∈, ∈}
first(E) = first(EBCDE), first(∈) = {e, ∈}

follow(S) = {$}
follow(A) = {b, c}
follow(B) = {c}
follow(C) = {d, e, $}
follow(D) = {e, $}
follow(E) = {$}

---

(2)

S → BB | Cd
B → aB | ∈
C → cC | ∈

first(S) = first(BB), first(Cd) = {a, b, ∈, c}
first(B) = first(aB), first(∈) = {a, ∈}
first(C) = first(cC), first(∈) = {c, ∈}

follow(S) = {$}
follow(B) = {b}
follow(C) = {d}

↳ follow(C) = follow of (∈)

Topic _____ Date ___/___/_____ Page No. :_____

Q→ E → TE'
E' → +TE' | ∈
T → FT'
T' → *FT' | ∈
F → id | (E)

starting symbol

⇒ first(E) = first(TE') = first(T) = {id, (} ✓
first(E') = first(+TE') | first(∈) = {+, ∈} ✓
first(T) = first(FT') = first(F) = {id, (} ✓
first(T') = first(*FT') | first(∈) = {*, ∈} ✓
first(F) = first(id) | first(E) = {id, (} ✓

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

Topic _____ Date 6/Oct/2024 Page No. :

* LL(1) PARSER & ITS CORESPONDENCY PARSING TREE

We will explain this concept through an example.

(10)

$E \to TE'$	first(T)
$E' \to +TE' / \epsilon$	first
$T \to FT'$	
$T' \to *FT' / \epsilon$	first $T' \to (*, \epsilon)$
$F \to id / (E)$	first(F) = (id, ( )

Find out whether the above given grammar is LL(1) or not ?.

⇒ • **Step-1:-** Firstly we need to calculate first() & follow() of the given grammar.

Production	first()	follow()
$E \to TE'$	{id, (}	{$,)}
$E' \to TE' / \epsilon$	{+, ϵ}	{$,)}
$T \to FT'$	{id, (}	{+, $,)}
$T' \to *FT' / \epsilon$	{*, ϵ}	{+, $,)}
$F \to id / (E)$	{id, (}	{*, +, $,)}

• **Step-2:-** After calculating first() & follow() now we'll draw LL(1) parsing table.

	+	*	id	(	)	$
E			$E \to TE'$	$E \to TE'$		
E'	$E' \to +TE'$				$E' \to \epsilon$	$E' \to \epsilon$
T			$T \to FT'$	$T \to FT'$		
T'	$T' \to \epsilon$	$T' \to *FT'$			$T' \to \epsilon$	$T' \to \epsilon$
F			$F \to id$	$F \to (E)$		

M/R

Topic _____ Date __/__/____ Page No. :_____

- In case of normal production we take first(). But in case of ε we check for follow().
- For adding the values in table we check the corresponding first() or follow() (in case of ε) values & add the production there.
- Since, bcoz we only have single entry in whole LL(1) parsing tree, ∴ this grammar is LL(1) grammar.

(2.) Find out whether this grammar is LL(1) or not.
$S \rightarrow aSbS \mid bSaS \mid \epsilon$.

⇒ Step-1:-

	first()	follow()
$S \rightarrow aSbS \mid bSaS \mid \epsilon$	{a, b, ε}	{b, a, $}

Step-2:-

	a	b	$
S	$S \rightarrow aSbS$ $S \rightarrow \epsilon$	$S \rightarrow bSaS$ $S \rightarrow \epsilon$	$S \rightarrow \epsilon$.

M/R

Topic _____ Date 7/[illegible]/2021 Page No. : _____

* FIND OUT WHETHER THE GIVEN GRAMARS ARE LL(1) or not

(1) $S \to aABb$
$A \to c \mid \epsilon$
$B \to d \mid \epsilon$

(2.) $S \to A \mid a$
$A \to a$

(3.) $S \to aAb \mid \epsilon$

(4.) $S \to aB \mid \epsilon$

(5.) $C \to cS \mid \epsilon$

(6.) $S \to AB$
$A \to a \mid \epsilon$
$B \to b \mid \epsilon$

(7.) $S \to aSA \mid \epsilon$
~~$A \to c \mid \epsilon$~~

(1.)

	first()	follow()
$S \to aABb$	$\{a\}$	$\{\$\}$
$A \to c \mid \epsilon$	$\{c, \epsilon\}$	$\{b, d\}$
$B \to d \mid \epsilon$	$\{d, \epsilon\}$	$\{b\}$

	a	b	c	d	$
S	$S \to aABb$				
A		$A \to \epsilon$	$A \to c$	$A \to \epsilon$	
B		$B \to \epsilon$		$B \to d$	

Yes

(2.)

	first()	follow()
$S \to A \mid a$	$\{a\}$	$\{\$\}$
$A \to a$	$\{a\}$	$\{\$\}$

	a	$
S	$S \to A$ / $S \to a$	
A	$A \to a$	

NO.

(3.)

	first()	follow()
$S \to aB \mid \epsilon$	$\{a, \epsilon\}$	$\{c, \$\}$
$B \to bC \mid \epsilon$	$\{b, \epsilon\}$	$\{\$\}$
$C \to cS \mid \epsilon$	$\{c, \epsilon\}$	$\{b, c\}$

	a	b	c	$
S	$S \to aB$		$S \to \epsilon$	$S \to \epsilon$
B		$B \to bC$		$B \to \epsilon$
C		$C \to \epsilon$	$C \to cS$ / $C \to \epsilon$	

NO.

Topic _____ Date __/__/__ Page No. _____

4. $S \to AB$, $A \to a \mid \epsilon$, $B \to b \mid \epsilon$

	first()	follow()
S	{a, b}	{$}
A	{a, ε}	{b, $}
B	{b, ε}	{$}

	a	b	$
S	S→AB	S→AB	
A	A→a	A→ε	A→ε
B	B→b		B→ε

Yes

5. $S \to aSA \mid \epsilon$, $A \to c \mid \epsilon$

	first()	follow()
S	{a, ε}	{$, a}
A	{c, ε}	{$}

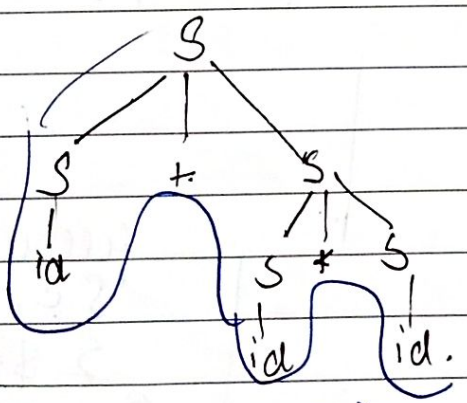
	a	c	$
S	S→aSA	S→ε	S→ε
A		A→c	A→ε

8/Oct/2021

Ambiguous Grammar

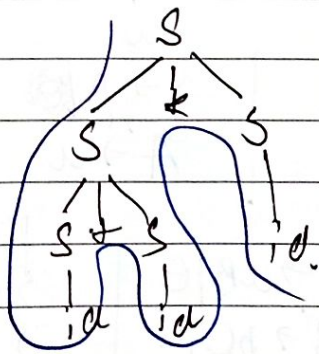
$S \to S+S \mid S*S \mid id$

⇒ (id + id * id)



id + (id * id)

Here, the answer is correct.



(id + id) * id

Triggers a wrong answer

Topic _____ Date __/__/____ Page No. :_____

The reason is :-

→ whenever we calculating an expression we must check for priority of the operators.

→ But if the priorities are same, then we need to look for the associativity.

* How to convert ambiguous grammar to non-ambiguous grammar :-

$S \rightarrow S+S \mid S*S \mid id.$

↳ $S \rightarrow S+S \mid A$

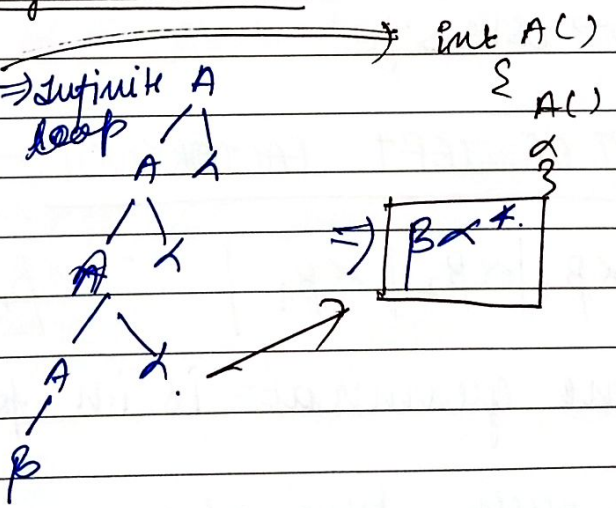
$A \rightarrow S*S \mid B$ → (UA).

$B \rightarrow id.$

* Concept of left Recursion :-

$A \rightarrow A\alpha \mid B$ ⇒ Infinite loop

∵ these symbols are same ∴ it is a left Recursion.



$A \rightarrow A\alpha \mid B$

* Solution to convert left Recursive grammar into its equivalent right recursive grammar.

Lets assume the given grammar is in a form of :-

$A \rightarrow A\alpha \mid B$

MR

Topic _____ Date ___/___/_____ Page No. : _____

then its corresponding equivalent RR grammar will be :-

$$A \to \beta A'$$
$$A' \to \alpha A' \mid \epsilon$$

* Examples :-

(1) "$E \to E + T \mid T$"

$A \to A \alpha \mid \beta$

$E \to T E'$
$E' \to +TE' \mid \epsilon$

$E \to T E'$
$E' \to +TE' \mid \epsilon$

(2) $S \to SOS1S \mid 01$

$S \to 01S'$
$S' \to OS1SS' \mid \epsilon$

* CONCEPT OF LEFT FACTORING :-

$A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3 \mid \cdots \alpha\beta_n$

If some grammar is in form of :-

Then such type of grammar are known as non-deterministic.

→ Conversion of non-deterministic grammar into deterministic grammar :-

MR

Topic _____ Date ___/___/_____ Page No. :_____

If $A \to \alpha\beta_1 | \alpha\beta_2 | \alpha\beta_3 | \cdots \alpha\beta_n$.

Then non-d. will be :-

$A \to \alpha A'$

$A' \to \beta_1 | \beta_2 | \beta_3 \cdots \beta_n$.

→ example :- $S \to aSSbS | aSaSb | abb$.

$\Rightarrow$ $S \to a$

21/Oct/2021

→ RECURSIVE DESENT PARSER :-

To create RDP, create methods for each left hand side of the production, in C program.

Eg :- $E \to iE'$

$E' \to +iE' | \epsilon$

* Steps to create RDP :-

```
E()
{ if (l = 'i')
    { match ('i');
      E'();
    }
}
```

$E \to iE'$

```
match ('i');
E'();
}
```

```
E'()
{ if (l = '+')
    { match ('+');
```

```
else
  return;
}
```

$E' \to +iE'$ $E' \to \epsilon$

```
match (char t)
{ if ( l==t)
    l=getchar();
  else
    printf ("error");
}
```

∴ l = lookahead symbol.

```
main()
{
  E()
  if (l==$)
  printf("parsing success");
}
```

main ← function

NOTE:- The data structure that is used in this KDP is recursive stack.

→ BOTTOM-UP PARSER:-

(1.) OPERATOR PRECEDENCE PARSER:-

① It's a bottom-up parser.
② It is basically used for mathematical operations.

Before, exploring it further we need to understand about the operator grammar.

* OPERATOR GRAMMAR:- There are 2 basic rules to understand whether the grammar is operator or not.

MR

Topic _____ Date ___/___/_____ Page No. : _____

- **Rule-1** :- In operator grammar "no 2 consecutive variables can appear in a production".

  Eg:- 1. $E \rightarrow AB\,a \mid \epsilon$ ✗ ∵ it has 2 variable A & B.
  $A \rightarrow a$
  $B \rightarrow b$

  2. $E \rightarrow a\,Ba \mid \epsilon$ ✓ There are no such productions in which 2 variables are present - Hence, Satisfies Rule-1.
  $B \rightarrow b$

- **Rule-2** :-
  In operator grammar no null productions are allowed; i.e. no 'ϵ' production are allowed.

NOTE:- Operator precedence parser is the only parser which can pass an ambiguous grammar.

25/Oct/2021

EXAMPLES:-

(1.) $E \rightarrow E+E \mid E*E \mid id$ (Yes)

(2.) $E \rightarrow E\,A\,E \mid id$ (NO)
$A \rightarrow + \mid *$ ∵ 2 consecutive variables

⟹ Correction: $E \rightarrow E+E \mid E*E \mid id$.

MJR

Topic _____ Date __/__/____ Page No. :_____

→ HOW TO CONSTRUCT OPERATOR RELATION TABLE:-

E → E + E | E * E | id.

(Steps to create table for this operator grammar)

	id	+	*	$
id	error	.>	.>	.>
+	<.	.>	<.	.>
*	<.	.>	.>	.>
$	<.	<.	<.	error.

In order to fill the mentioned table we need to apply few rules:-

Rule-1 :-

Identifiers are the highest priority / higher precedence as compared to any other operator.

Rule-2 :-

$ is having the least precedence as compared to any other operand.

Topic _____ Date __/__/____ Page No. :_____

→ How passing works? :-

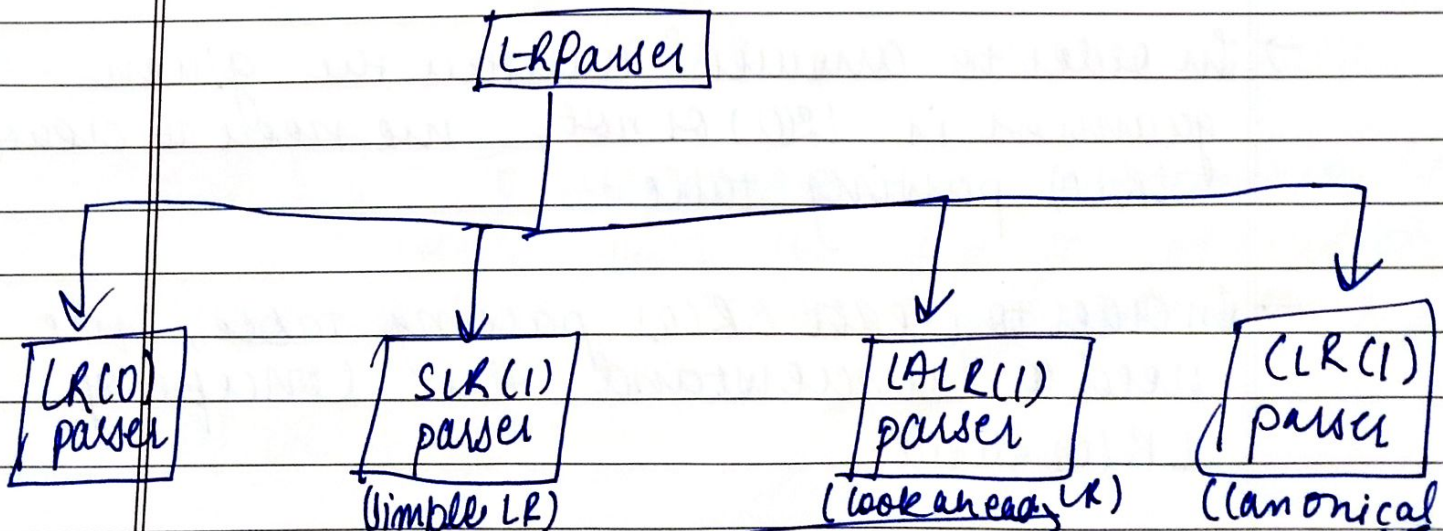
1. We use stack for the scanning.
2. Initially stack is having dollar '$' at top of it.
3. When precedence is less than the next character, then perform operation = 'PUSH'.
4. When precedence is higher than the next character, then perform operation = 'POP'.

NOTE:- If we have 'n' operators, then the size of table is order of $\Rightarrow O(n^2)$.

---

* L-R PARSERS:-

These are the bottom-up parsers.



Increasing order of processing power.

26/Oct/2021

Topic _____ Date __/__/____ Page No. : _____

* Their corresponding Passing Table:-

Concept:-

LR(0):
SLR(1):
→ LR(0) Items

LALR(1):
CLR(1):
→ LR(1) Items (LR(0) Items + Lookahead)

→ In regards to LR Parsers, the main question is whether the grammar is: LR(0) / SLR(1) / LALR(1) / CLR(1) ?

→ In order to answer such question, we need to create their corresponding parsing table.

* LR(0) PARSERS:-

→ In order to answer whether the given grammar is LR(0) or not, we need to create LR(0) parsing table.

→ In order to create LR(0) parsing table, we need to understand the concept of LR(0) Items.

M/R

Topic _____ Date __/__/____ Page No. :_____

- **STEPS TO FIND LR(0) Items :-**

Grammar :- "$S \to AA$
$A \to aA / b$"

→ **Step-1** :- In the given grammar, we need to add a dummy production, named as "Augmented Production" and it basically produces the start symbol and denoted as "S'", if the start symbol is "S".

$S' \to S$
$S \to AA$
$A \to aA / b$

→ **Step-2** :- Now we need to start with the augmented production and place a LR(0) item sign as '.'

$S' \to .S$
$S \to AA$
$A \to aA / b$

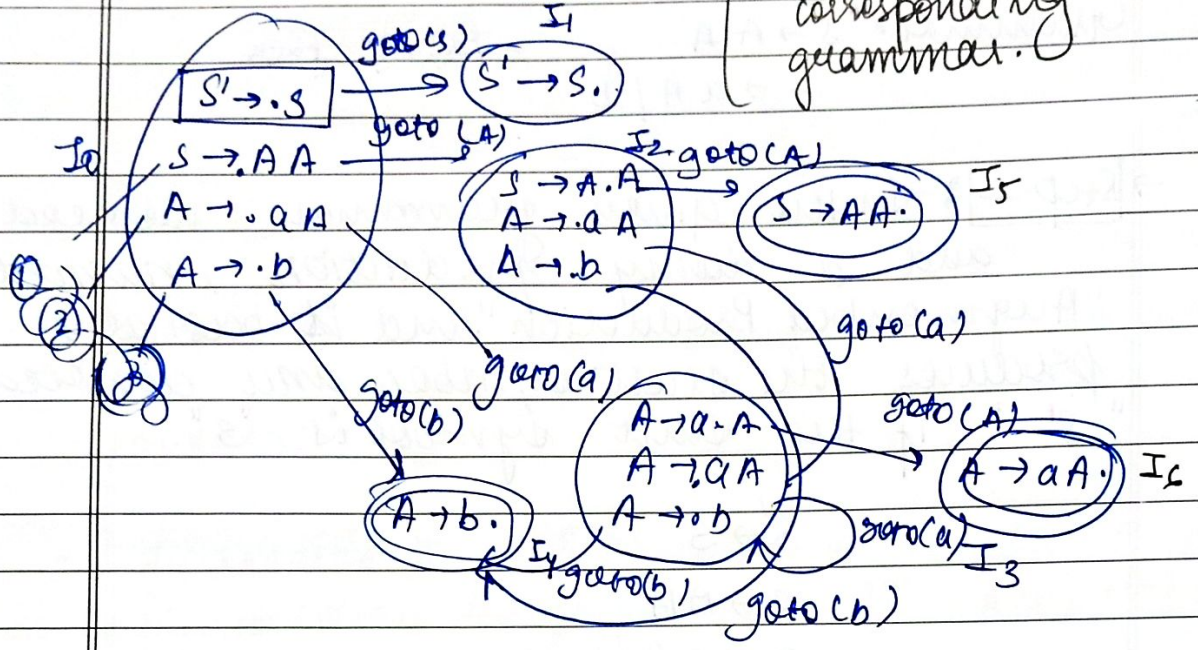
→ **Step-3** :- To understand the functionality of LR(0) items sign are specific operation "closure & goto" needs to be understand.

- **CLOSURE & GOTO :-**

Topic _____ Date __/__/____ Page No. :_____

→ ~~Closure Operation~~ :-

[LR(0) items of the corresponding grammar.]



goto & closure will work parallely, first we will take out the closure, then we will take out goto, & the process will continue till the final state is not reached.

Q → $S \to aA \mid aB$
$A \to bA \mid c$
$B \to bB \mid c$

⇒ $S' \to S$
$S \to aA \mid aB$
$A \to bA \mid c$
$B \to bB \mid c$

$\Rightarrow$

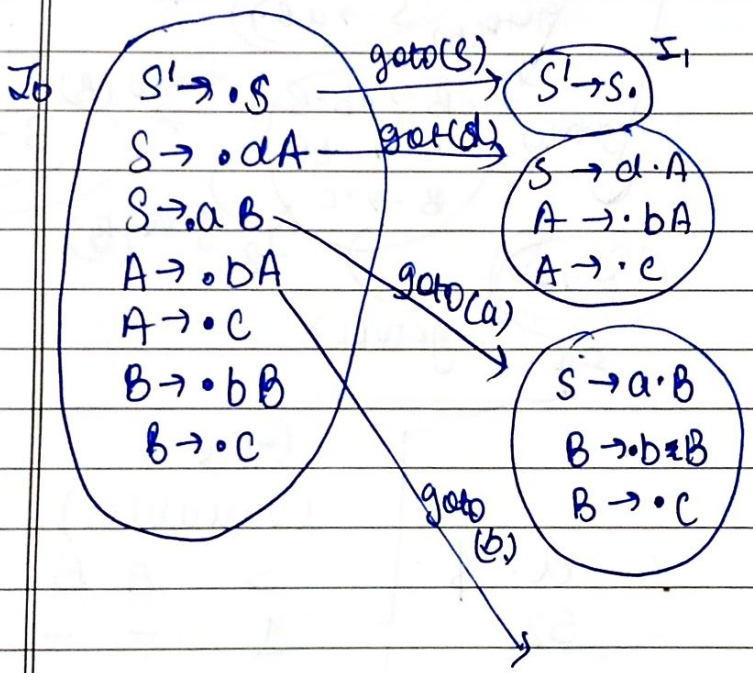
$S' \to .S$
$S \to aA \mid aB$
$A \to bA \mid c$
$B \to bB \mid c$

Topic _____ Date ___/___/_____ Page No. : _____

$S' \to .S$
$S \to dA \mid aB$
$A \to bA \mid c$
$B \to bB \mid c$

$\longrightarrow$

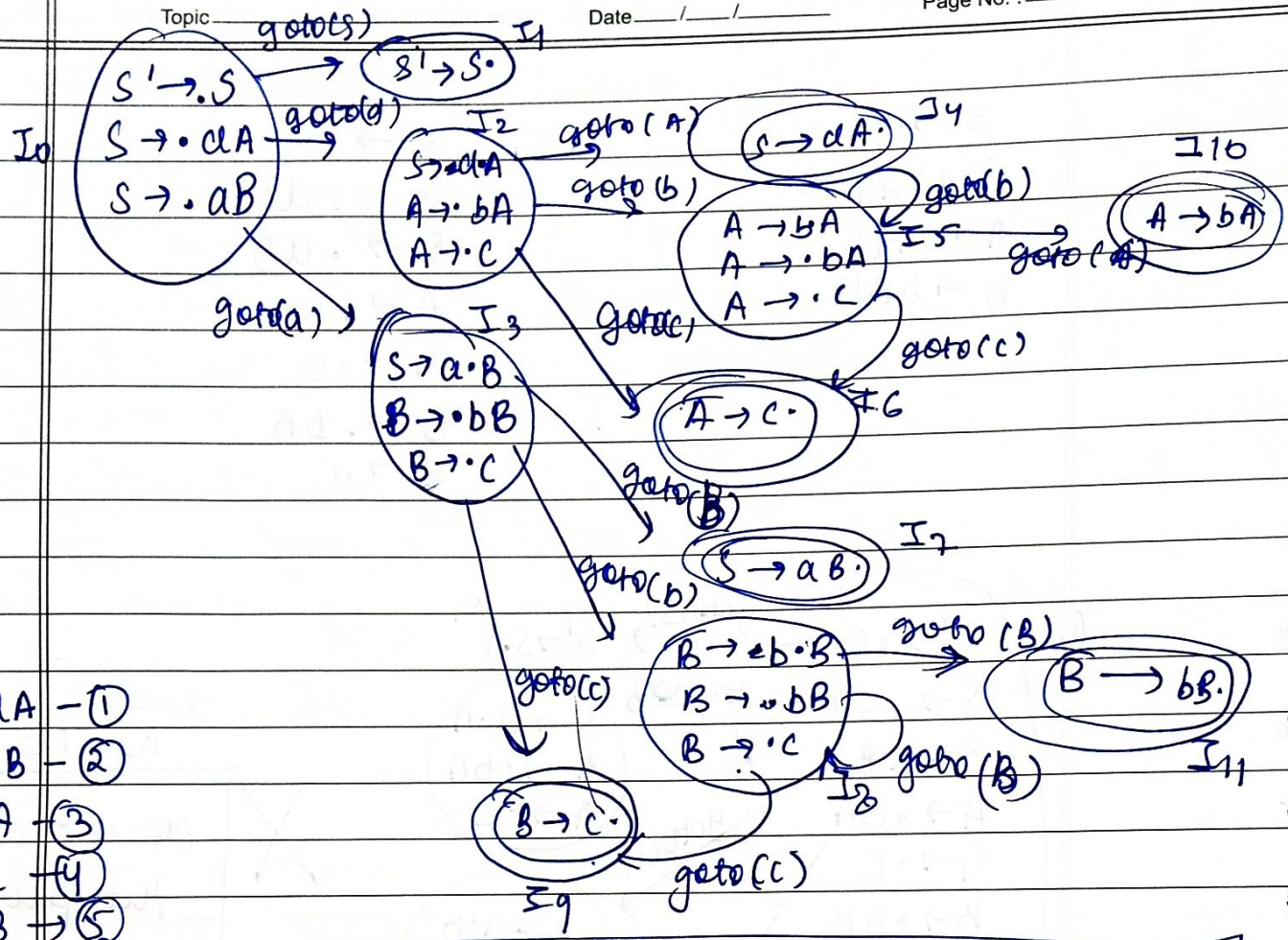
$S' \to \cdot S$
$S \to . dA$
$S \to . aB$
$A \to . bA$
$A \to . C$
$B \to . bB$
$B \to . C$



NOTE

Donot start the problem with all the productions

Topic _____ Date ___/___/_____ Page No. :_____



S → .dA — (1)
S → .aB — (2)
A → .bA — (3)
A → .c — (4)
B → .bB — (5)
B → .c — (6)

States	Action (Terminals)					Goto (Variables)		
	a	b	c	d	$	S	A	B
0	S3			S2		1	–	–
1					"accept"			
2		S5	S6				4	
3		S8	S9					7
4	r1	r1	r1	r1	r1			
5		S5	S6				10	
6	r4	r4	r4	r4	r4			
7	r2	r2	r2	r2	r2			
8		S8	S9					11
9	r6	r6	r6	r6	r6			
10	r3	r3	r3	r3	r3			
11	r5	r5	r5	r5	r5			

Topic _____ Date __/__/____ Page No. :_____

* Shift & Reduce Moves:-

(1.) Shift Move:-

(i) $A \rightarrow \cdot aA$ ✓

(ii) $A \rightarrow a \cdot A$ ✗ because . is before a variable which is not acceptable.

(iii) $A \rightarrow aA \cdot$ ✗ because LR(0) item sign (.) is at the end of the production.

(iv) $A \rightarrow aB \cdot CD$ ✓

(v.) $C \rightarrow aB \cdot$ ✗

(vi.) $S \rightarrow S \cdot Sab$ ✗

(vii.) $S \rightarrow \epsilon$

↳ represents $S \rightarrow \cdot$ → means at the end of production

⇒ ∴ it is not a shift move.

(2.) Reduce Move:-

LR(0) item symbol (.) at the end of the production, in case of reducing

Eg:-

(i.) $S \rightarrow aAB \cdot$ ✓

(ii.) $S \rightarrow \cdot$ ✓

(iii.) $A \rightarrow aA \cdot$ ✓

(iv.) $A \rightarrow a \cdot A$ ✗

MR

Topic _____ Date __/__/____ Page No. : _____

After the creation of LR(0) items it may be the possibility that we have various shift & reduce moves in all the existing sticks ($I_0, I_1, I_2 \ldots$)

Now we need to identify the conflicts of shift & reduce moves.

* Conflicts of SR → (shift - Reduce).

There are 2 types of conflicts in the internal states of LR(0) items:-

1. Shift - Reduce conflict (S-R conflict)
2. Reduce - Reduce conflict - (R-R conflict)

(i.) $S \to S.$ → R → because LR(0) item sign is at the end.
$S \to a.A$ → S → S-R conflict
$A \to .a$ → neither shift nor conflict.
→ because LR(0) item sign is at the beginning.

(ii.) $S \to S.$ → R
$S \to a.A$ → There is no conflict.

(iii.) ~~$S \to aA.$~~ ~~$A \to aA.$~~ $S \to aA.$ → R
$A \to a.$ → R
→ R-R conflict.

Topic _____ Date __/__/____ Page No. :_____

→ **Step-5**: Now create LR(0) Parsing Table.

States	ACTION (Terminals) a	b	$	GOTO (variables) S	A
0	S3	S4		1	2
1			"accept"		
2	S3	S4			5
3	S3	S4			6
4	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		

Reduce se shift ki mejarahai.

Check the goto variables of A & S from every state

S → AA — (1)
A → aA — (2)
A → b — (3)

(1) Entry of variables.

(2) We need to make an entry of "accept" under '$' within the state where we get our augmented production.
(In our eg. it is state I1)

(3) Now, we need to make the entries for shift move.

(4) Entry for reduce now, in which need to look the final states & make their final entries in the parsing table as per the production no.

MR

Topic _____ Date 27/oct/2024 Page No. _____

Step-6:- • In order to answer whether the given grammar is LR(0) or not, we need to observe LR(0) Parsing Table.

• The given grammar is not LR(0) if it follows the below 2 conditions:-

(1.) There should not be any S-R conflict in any of the cell of given parsing table.

(2.) There should not be any R-R conflict in any of the cell of given parsing table.

Now, in the above LR(0) parsing Table, there's no conflict of SR and RR. Hence, this grammar is LR(0) grammar.

* A SHORTCUT TO FIND WHETHER THE GRAMMAR IS LR(0) OR NOT:-

• In order to answer quickly whether the grammar is LR(0) or not; we only need to follow from step-1 to 3 (till the point of creating LR(0) items.

• Now just observe each & every internal state of LR(0) items and try to find out, any of the conflict of S-R or R-R. If you find any of these conflicts in any of the states, then declare that this is not LR(0) grammar.

But this method is not recommended.

MR

Topic _____ Date 28/8/2021 Page No. :

* SLR(1) :-

→ After the discussion of LR(0), we can simply proceed for SLR(1). Now, again we have a question whether the grammar is SLR(1) or not?

→ In order to answer if grammar is SLR(1) we need to remember these 2 rules:-

- **Rule-1**:- If a grammar is LR(0), then it will definitely be SLR(1) grammar. In other words, if there is no conflict of S-R or R-R in LR(0) parsing table, then obviously the grammar will be LR(0) & as well as SLR(1).

- **Rule-2**:- If a grammar is not SLR(0), then obviously we have some conflicts in corresponding LR(0) parsing table. In that case now we manually need to check, whether the grammar is SLR(1) or not.

* HOW TO CHECH WHETHER THE GRAMMAR IS SLR(1) OR NOT :-

→ The parsing table for SLR(1) is same as LR(0) parsing table. But the only difference is the placement of reduce move.

MR

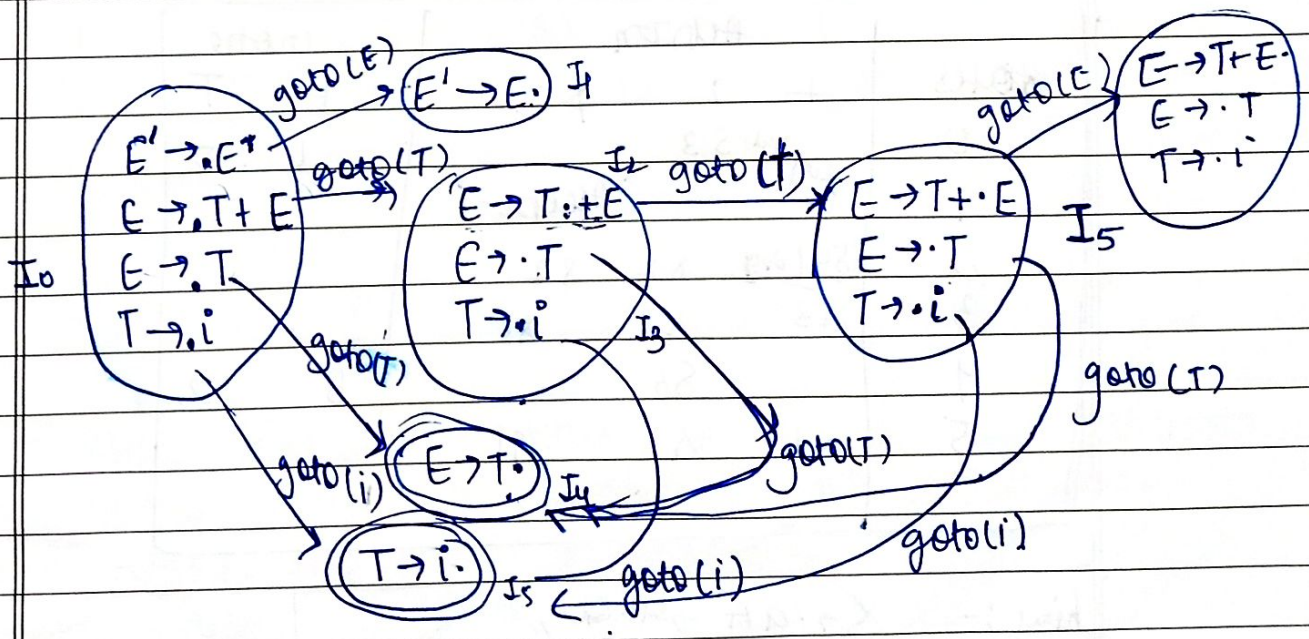
Topic _____ Date ___/___/_____ Page No. :_____

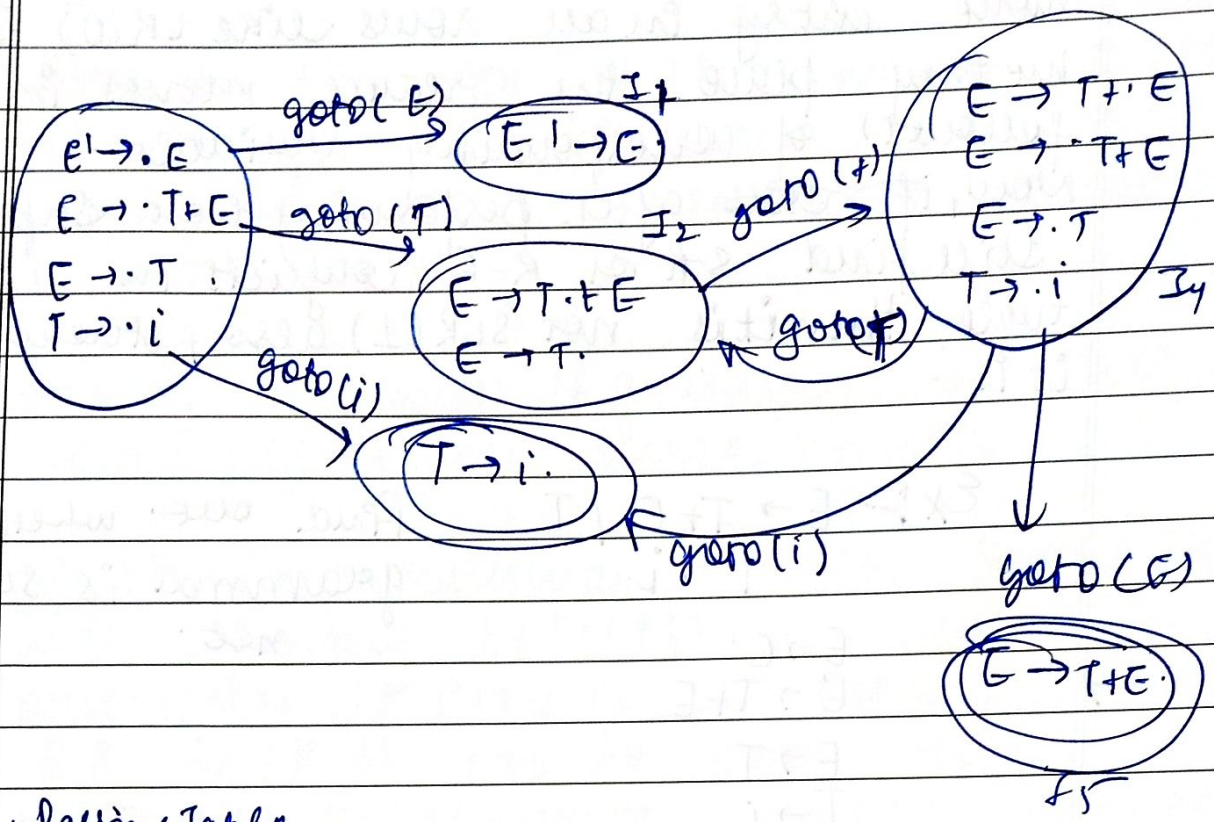
→ In case of SLR(1) we don't write the reduce move entry in all rows like LR(0). In SLR(1) we only place the reduce moves in the follow() of corresponding variable.
Now, if after performing these steps if we still find S-R or R-R conflicts in parsing table, then it is not SLR(1) also, otherwise it is.

Ex:- $E \to T+E \mid T$
$T \to i$

Find out whether grammar is SLR(1) or not.

⇒
$E \to E'$
$E \to T+E$
$E \to T$
$T \to i$





* Passing Table

States	Action			goto	
	+	i	$	E	T
0		shift S3		1	2
1			"Accept"		
2	S4/r2	r2	r2		
3	r3	r3	r3		
4		S3		5	2
5	r1	r1	r1		

Hint :-

[Take all the production collectively]

Topic _____ Date __/__/____ Page No. : _____

* This grammar is not LR(0) grammar because we have a conflict of SR in the 3rd row

* Now we need to follow the (rule-2) & need to check whether this grammar SLR-(1) manually by calculating the follow of the reduced move in the corresponding cell.

State	Action			Goto	
	+	i	$	t	T
0		S3		1	2
1			"Accept"		
2	S4		r2		
3	r3	r3	r3		
4		S3		5	2
5	r1	r1	r1		

Since reduced move ⊗ corresponding production is E → T, whose follow() is $ so we only make the entry of reduce (r2) under $.

If still following rule 2 we find conflicts in parsing table, then grammar is not SLR(1).

MR

Topic _____ Date 8/Nov/2021. Page No. :

* CONCEPT OF LALR(1) & CLR(1)

→ In LALR(1) & CLR(1) the ultimate goal is still creating the corresponding parsing tables. We need to create LALR(1) parsing table of CLR(1) p.t for grammar, and declare whether the grammar is LALR(1) or CLR(1).

→ To create parsing table of LALR(1) & CLR(1) we need to create LR(1) items, which is combination of LR(0) items + lookahead symbols.

Firstly, we need to create parsing table of CLR(1) & then we move to LALR(1) parsing table.

Ques: Create CLR(1) parsing table of the given grammar & find out whether its CLR(1) or not.

$S \rightarrow AA$

$A \rightarrow aA/b$

Sol":- In order to solve such problems, we need to create CLR(1) parsing table.

* STEPS TO CREATE CLR(1) Parsing Table:-

Rule-1:- Look-ahead symbol of the agumented production will always be dollar ($).

MR

Topic _____ Date __/__/____ Page No. :_____

$S' \rightarrow \cdot S, \$$ → look-ahead symbol.

Rule-2 :- Simply follow CLR(0) items creation but while closure operation look-ahead will be the first of next whole string.

Eg → $S \rightarrow AA$
$A \rightarrow aA \mid b$

on applying closure on augmented production

$S' \rightarrow \cdot S$
$S \rightarrow \cdot AA, \$$
$A \rightarrow \cdot aA, a/b$
$A \rightarrow \cdot b, a/b$

I_0

due to this S, this production will be added.

will be the first

Rule-3 :- During the goto operation look-ahead symbol doesn't change.

$S' \rightarrow \cdot S$
$S \rightarrow \cdot A(A, \$)$
$A \rightarrow \cdot aA, a/b$
$A \rightarrow \cdot b, a/b$

goto(A)

I_2.

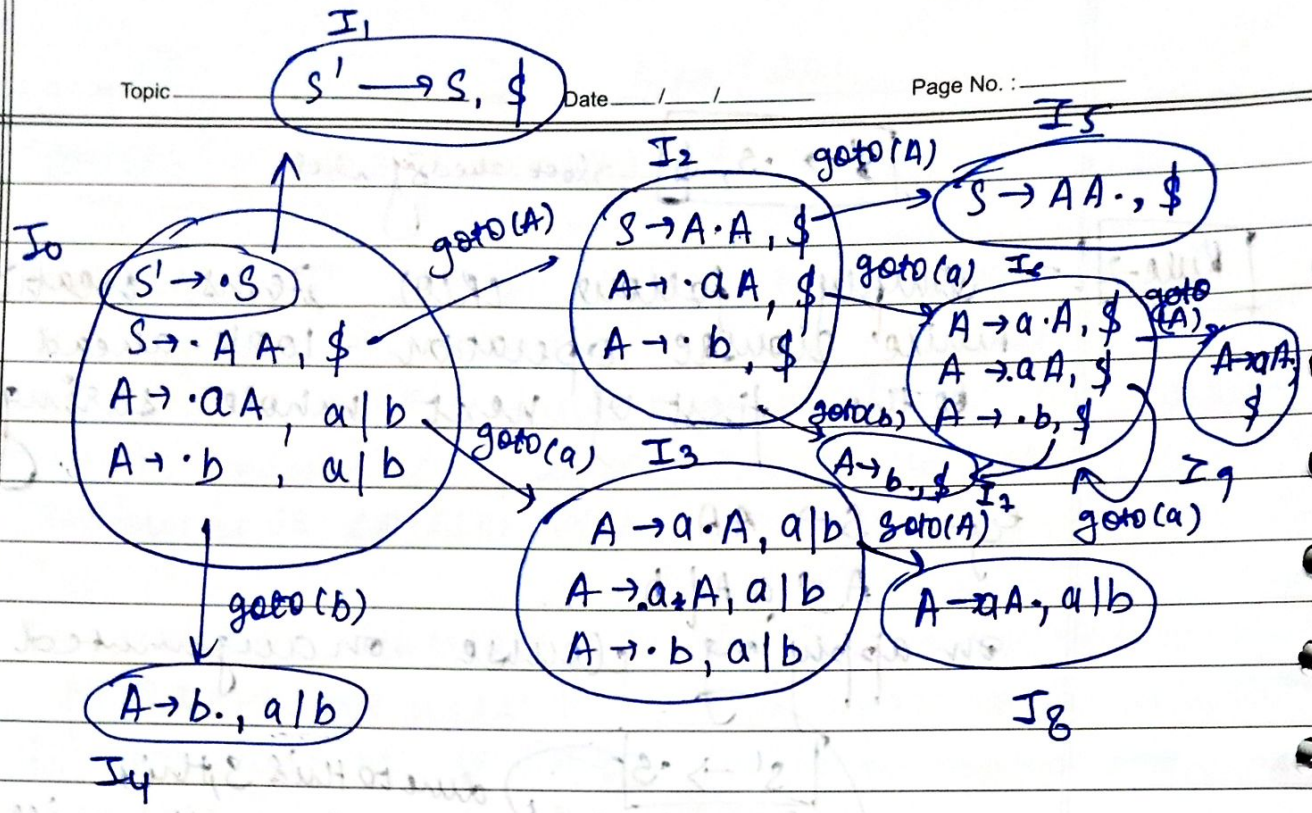
$S \rightarrow A \cdot A, \$$
$A \rightarrow \cdot aA, \$$
$A \rightarrow \cdot b, \$$

here the look-ahead symbol will remain same.

Rule-4 :- In LR(1) parsing table reduce moves are placed under look-ahead symbols.

MR

Topic _____ Date ___/___/___ Page No. : _____



States	Action			goto	
	a	b	$	A	S
0	S3	S4		2	1
1			"accept"		
2	S6	S7		5	
3	S3	S4		8	
4	r3	r3			
5			r1		
6	S6	S7		9	
7			r3		
8	r2	r2			
9			r2		

Topic: MAJOR PROBLEM Date: __/__/____ Page No. :

Ques → $S \rightarrow AaAb \mid BbBa$

$A \rightarrow \epsilon$

$B \rightarrow \epsilon$

Find out the given grammar is LL(1), LR(0), SLR(1), CLR(1).

⇒ **I Checking for LL(1) :-**

Step 1:- We calculate first & follow.

Production.	first()	follow()
$S \rightarrow AaAb \mid BbBa$	$\{a, b\}$	$\{\$\}$
$A \rightarrow \epsilon$	$\{\epsilon\}$	$\{a, b\}$
$B \rightarrow \epsilon$	$\{\epsilon\}$	$\{a, b\}$

Step 2: Parsing table :-

	a	b	$
S	$S \rightarrow AaAb$	$S \rightarrow BbBa$	
A	$A \rightarrow \epsilon$	$A \rightarrow \epsilon$	
B	$B \rightarrow \epsilon$	$B \rightarrow \epsilon$	

Since, we have only single entry in whole LL(1) parsing tree,

∴ This grammar is LL(1) grammar.

II Checking for LR(0) :-

Topic _____ Date __/__/____ Page No. _____

Step-1:- $S' \to S$

$S \to AaAb \mid BbBa$

$A \to \epsilon$

$B \to \epsilon$

Step-2:- $S' \to \cdot S$

$S \to AaAb \mid BbBa$

$A \to \epsilon$

$B \to \epsilon$

Step-3:-

I_0:
1. $S' \to \cdot S$
2. $S \to \cdot AaAb$
3. $S \to \cdot BbBa$
4. $A \to \cdot$
5. $B \to \cdot$

goto(S) → I_1: $S' \to S.$

goto(A) → I_2: $S \to A \cdot aAb$

goto(a) → I_4: $S \to Aa \cdot Ab$, $A \to \cdot$

goto(A) → I_6: $S \to AaA \cdot b$

goto(b) → I_8: $S \to AaAb.$

goto(B) → I_3: $S \to B \cdot bBa$, $B \to \cdot$

goto(b) → I_5: $S \to Bb \cdot Ba$, $B \to \cdot$

goto(B) → I_7: $S \to BbB \cdot a$

goto(a) → I_9: $S \to BbBa.$

Parsing Table.

states	a	b	$	S	A	B
0	r3, r4	r3, r4	r3, r4	1	2	3
1						
2	s4					
3		s5				
4	r3	r3	r3		6	
5	r4	r4	r4			7
6		s8				
7	s9					
8	r1	r1	r1			
9	r2	r2	r2			

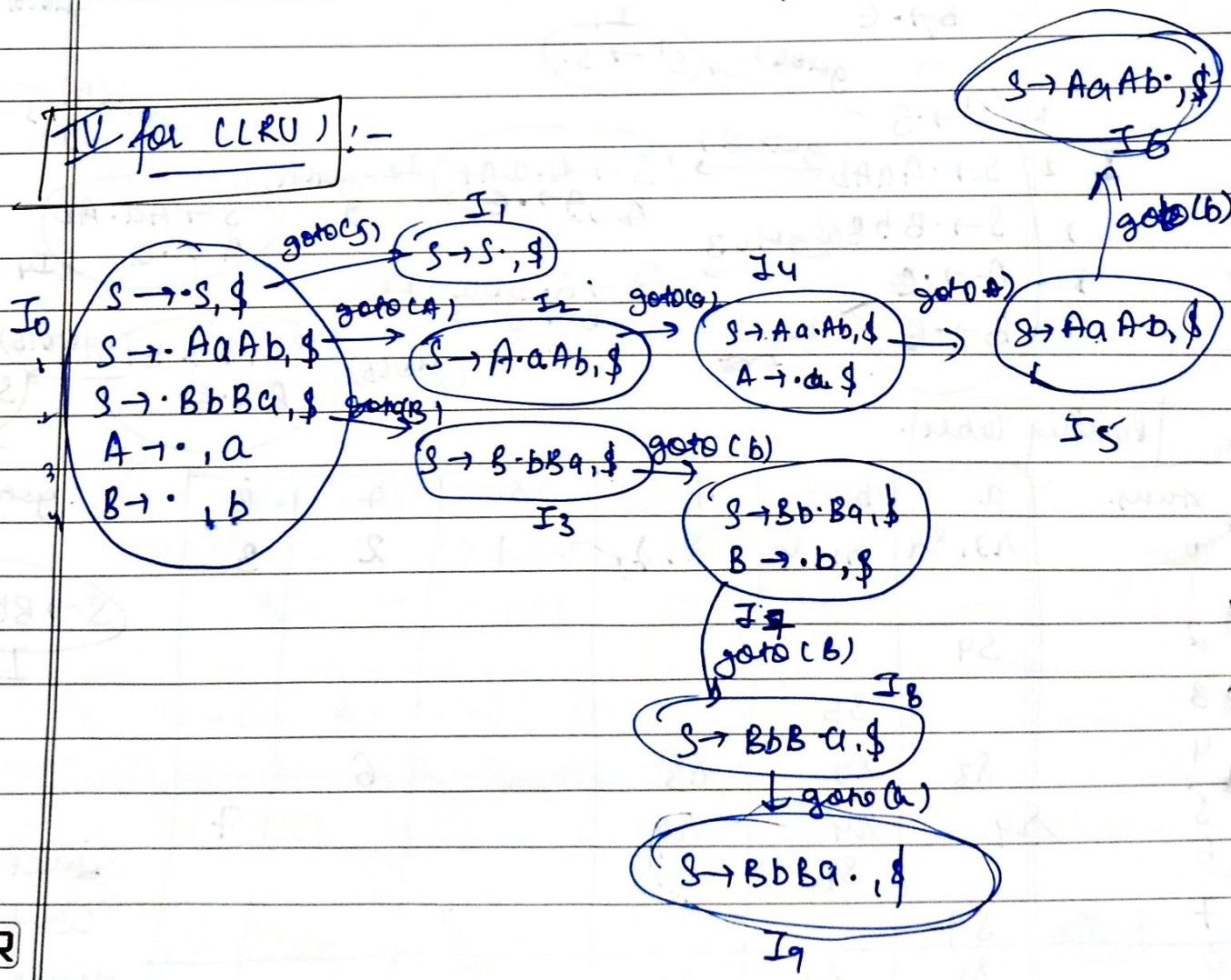
Since, R-R conflict hence, not LR(0).

Topic _____ Date ___/___/___ Page No. : _____

III. for SLR(1) :-

State	a	b	c	S	A	B
0						
1						
2						
3						
4						
5						
6						
7						
8						
9						

IV. for CLR(1) :-



Topic _____ Date ___/___/___ Page No. _____

States	Action. a	b	$	(Goto) S	A	B
0	[illegible]			1	2	3
1			"accept"			
2	S4					
3		S7				
4	[illegible]	[illegible]			5	
5		S6				
6			r1			
7	[illegible]	[illegible]				B
8	S9					
9			r2			

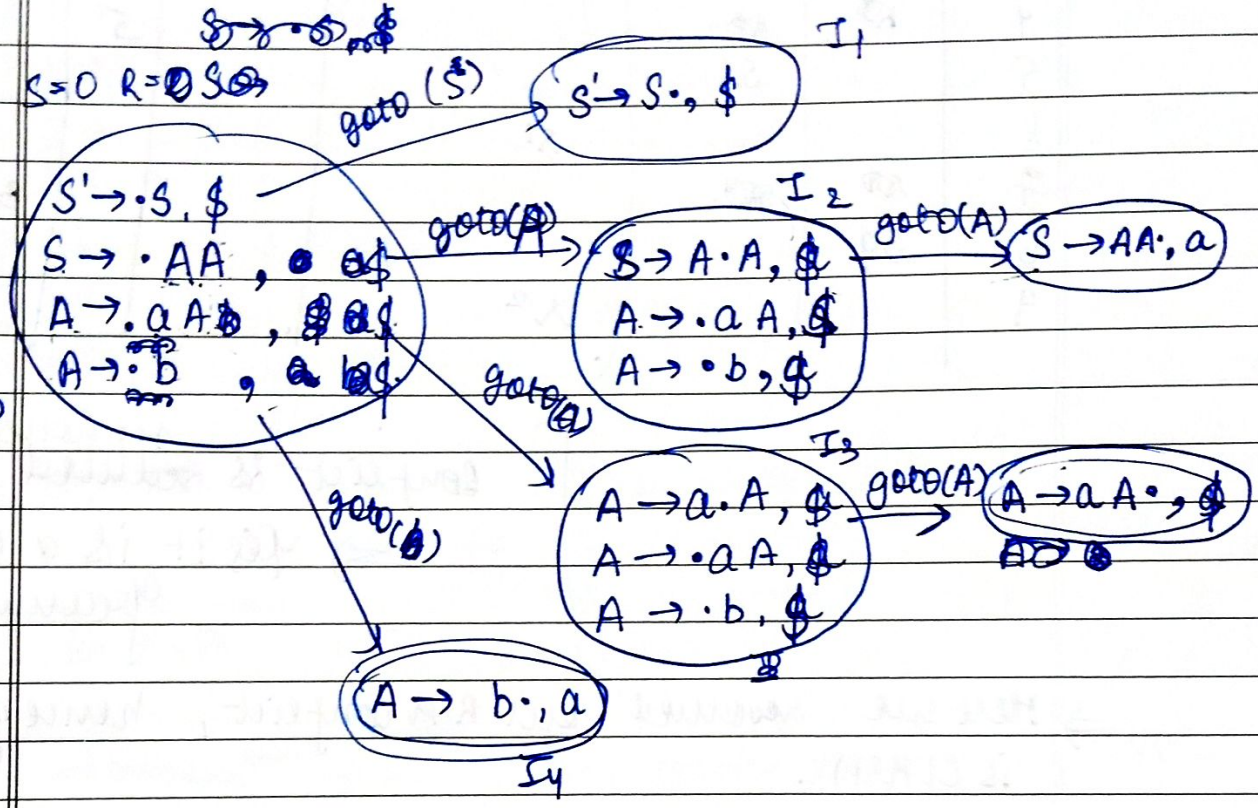
Conflict is removed.

⇒ yes it is a CLR(1) grammar

⇒ Here we resolved our R-R conflict, hence grammar is CLR(1).

Topic Date Page No.

Ques → $S \to AA$
$A \to aA/b$

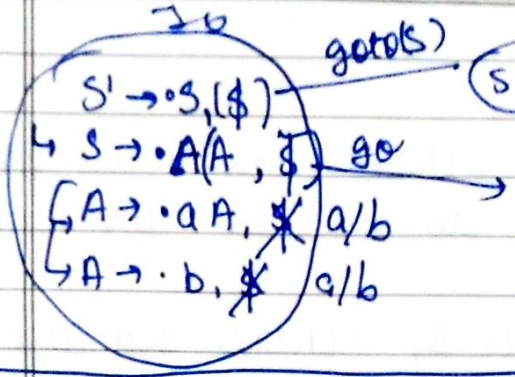


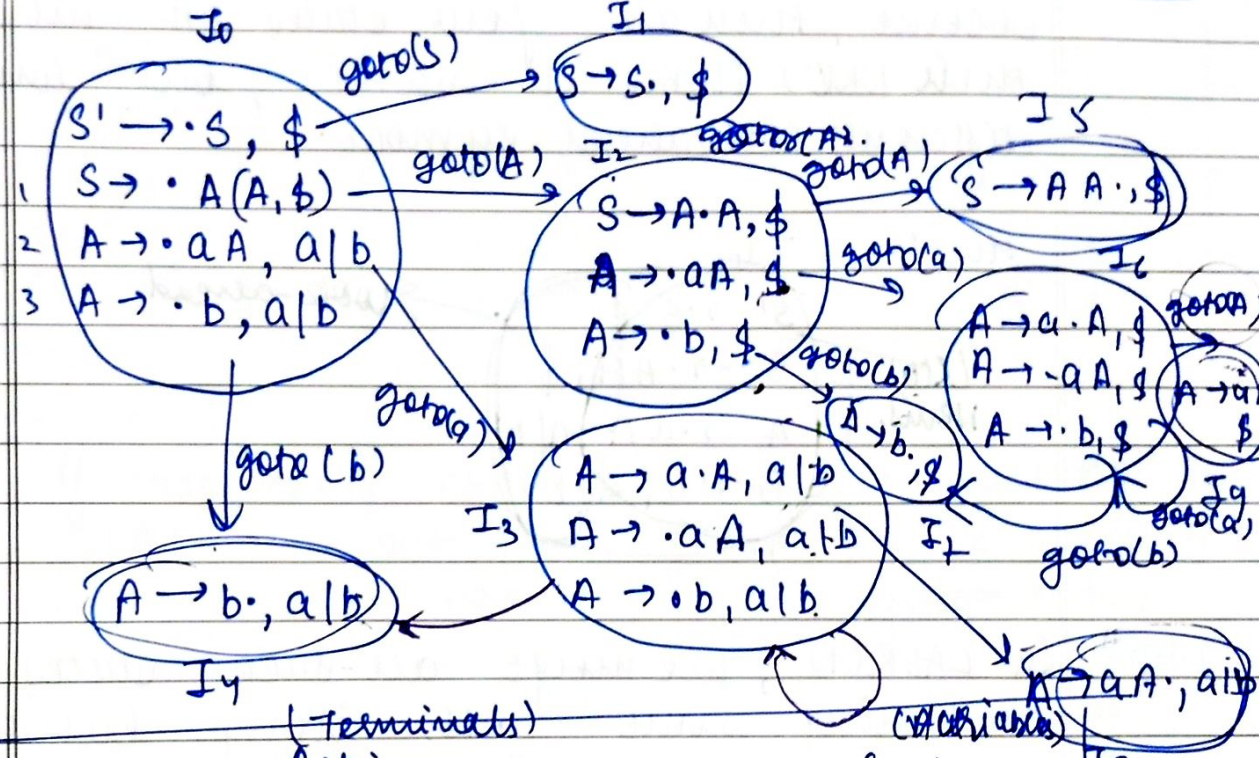
a.

.a.b

Topic _____ Date __/__/____ Page No. _____

find(A$)





	(Terminals) Action			(Variables) Goto	
States	a	b	$	A	S
0	S3	S4		2	1
1			"accept"		
2	S6	S7		5	
3	S3	S4		8	
4	r3	r3			
5			r1		
6	S6	S7		9	
7			r3		
8	r2	r2	r2		
9					

MR

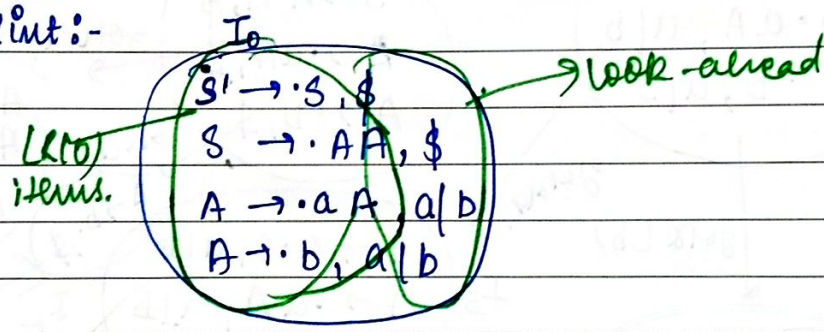
Topic _____ Date 10/Nov/2021 Page No. :_____

→ LALR(1) PARSER:-

After the discussion of CLR(1), now we partially modify CLR(1) concept to achieve LALR(1) parser.

In the previous example of CLR(1) we can observe, there are few states in which their LR(0) items are same, but having different look-ahead symbol.

Hint:-



" In LALR(1), we merge all such states which are having same LR(0) items but different look-ahead symbols. " And in the parsing table of CLR(1), we simply delete such rows which follow this property & keep only one row along with their shift move merge.

→ In the previous example, following are the pair of states which are having same LR(0) items but different look-ahead symbol.

MR

Topic _____ Date ___/___/_____ Page No. :_____

I_3 & I_6 → I_{36}

I_4 & I_2 → I_{47}

I_8 & I_9 → I_{89}

(corresponding LALR(1) parsing table is :)

	ACTION			GOTO	
States	a	b	$	A	S
0	S36	S47		2	
1			"accept"		
2	S36	S47		5	
36	S36	S47		8	
47	R3	R3	R3		
5			R1		
89	R2	R2	R2		

NOTE :-

If grammar is not CLR(1), then it must not be LALR(1), but if the grammar is CLR(1), then we manually need to check whether the grammar is LALR(1) or not, bcoz it might be a chance, that during the merge operation in LALR(1), we must face S-R or R-R conflicts.

Topic _____ Date 11/NOV/2021 Page No. _____

UNIT-3

SYNTAX DIRECTED TRANSLATION

(GRAMMAR + SEMANTIC RULE)

* SEMANTIC RULES :-

→ Informal notation. Semantic rules are the meaningful rules that are helpful for a compiler, as well as intermediate code generator of expression evaluation.

→ Eg:-

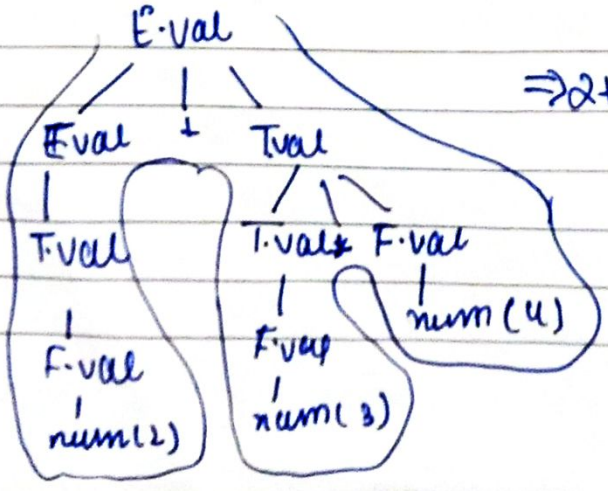
SDT ⇒

GRAMMAR	SEMANTIC RULES
E → E + T / T	{ E.val = E.val + T.val }
	{ E.val = T.val }
T → T * F / F	{ T.val = T.val * F.val }
	{ T.val = F.val }
F → num	{ F.val = num }

"val = value"
"num = number"

Now, evaluate the expression "2+3*4" using the given SDT (Syntax Directed Translation).

⇒



⇒ 2+(3*4)

→ Top to Down
→ Left to Right

M/R

Topic _____ Date __/__/____ Page No. :_____

There are the few points regarding the above soln. :-

(1.) num = token & 2, 3, 4 represents the actual values or lexical values.

(2.) Traverse the parse tree :- (i.) Top to down
(ii.) Left to right

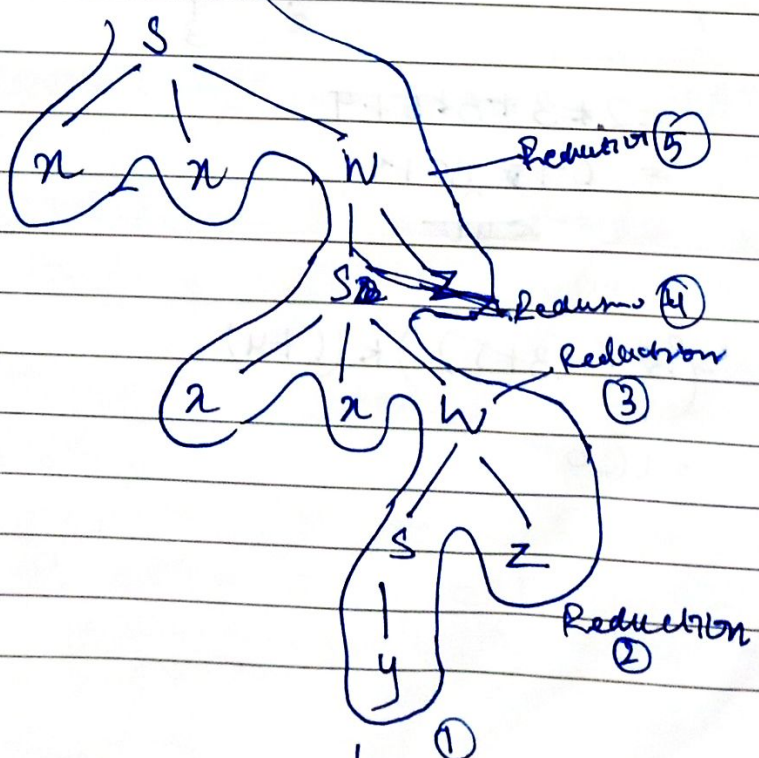
(3.) Whenever you find any Reduction, the corresponding semantic rule will be activated.

[Hint: F-val
num(2) ↑ Reduction] The corresponding semantic rule i.e. F-val = num gets activated

NOTE :- SDT can be implemented using both Top-down Parser & Bottom-Up Parser.

Eg-2 : S → xxW | { print("1"); }
y { print("2"); }
W → Sz { print("3"); }

What is the output of this SDT while generating the string: "xxxyzz"



⇒ 23131

Topic _____ Date __/__/__ Page No. :

Eg-3

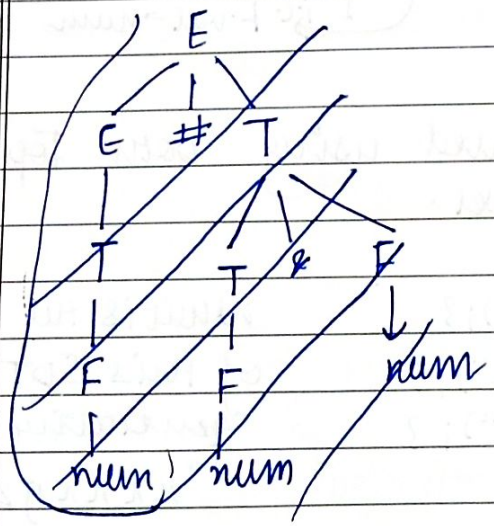
$E \to E \# T \mid T$	{ E.val = E.val * T.val } { E.val = T.val }
$T \to T \& F \mid F$	{ T.val = T.val + F.val } { T.val = F.val }
$F \to num$	{ F.val = num }

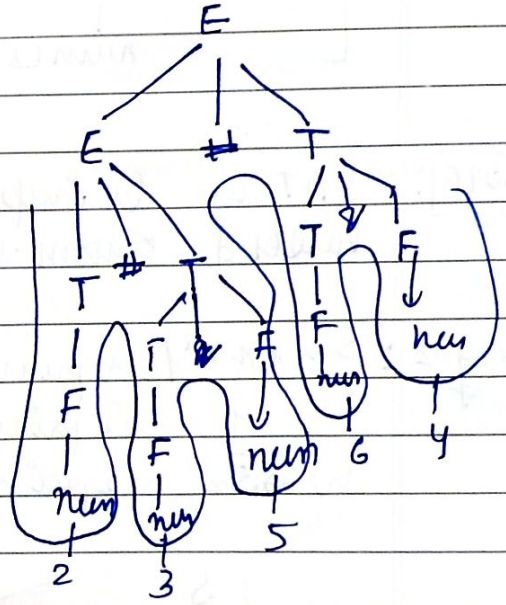
(16)

Evaluate the following string acc. to the given SDT.

2 # 3 & 5 # 6 & 4

⇒





~~= 2 * 3 + 5 * 6 + 4~~

~~= 6 + 30 + 4~~

~~= 40~~

= (2 * (3 + 5)) * (6 + 4)

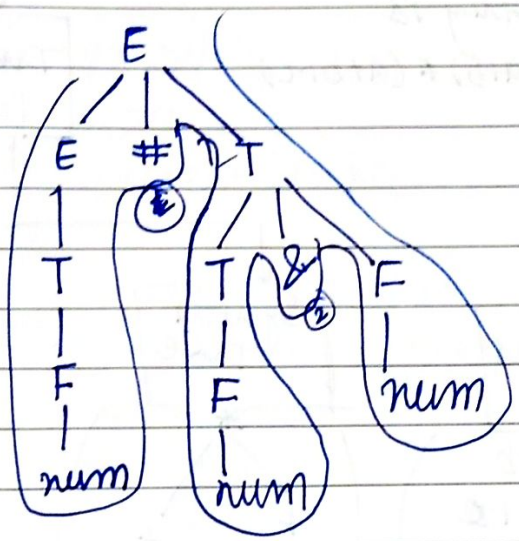
= 160

* Special Example:-

semantic rule can also arise in b/w production.

(1) E → E # & print("1") } T | T
T → T & & print("2") } F | F
F → num { F.val = num }

(3)



(2) 2# (3 & 4)

="12" (output)

But if the ~~too~~ semantic rule was after the grammar, then we must ~~write~~ get the output as "21".

---

* INTERMEDIATE CODE GENERATION:-

It works as a middle-ware in between source lang. & machine lang. / assembly lang.

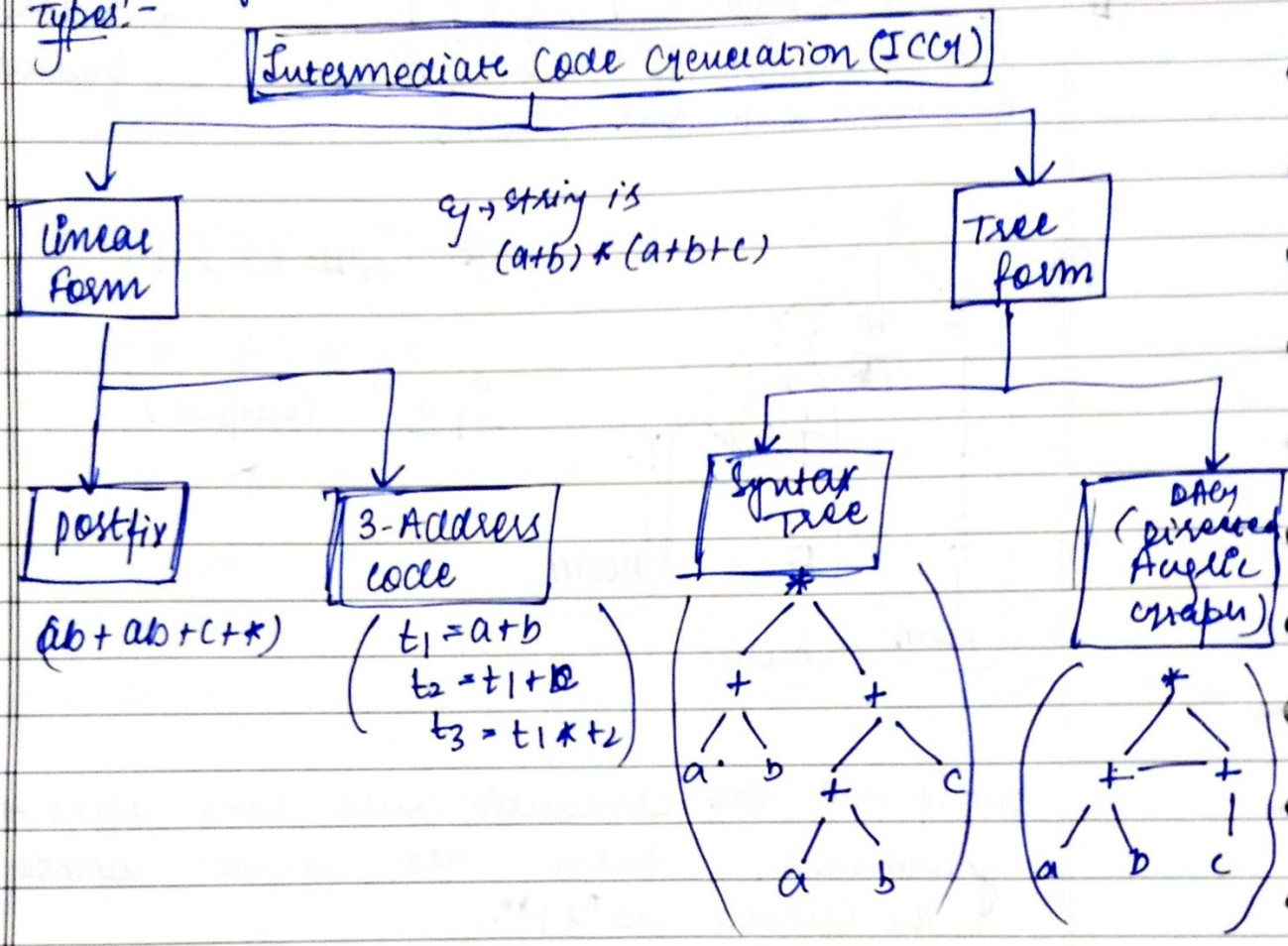
We cannot directly convert the source lang. directly into machine lang. because for this reason we require native compiler for each & individual machine with different configuration. (like different O.S)

M R

Topic _____ Date __/__/____ Page No. : _____

ICG helps to generate a intermediate code which is native for each & every individual machine.

* Types:-



* TYPES OF ICG :-

I. POSTFIX NOTATION :-

1. Eg :- (a+b)*(a+b+c) → (ab+ab+c+*)
   a+b → ab+
   a+b*c → abc*+

Topic _____ Date ___/___/_____ Page No. : _____

II 3-ADDRESS CODE:-

This is the most frequent & famous 3address code representation. There are 3 basic types of 3-Address code:-

1. Quadriples
2. Triple
3. Indirect Triple.

15/Nov/2021

eg:- ~~t=a+b~~ -(a+b)*(c+d)+(a+b+c)

The general representation of ICG through a 3-Address code is like :-

1. $t_1 = a+b$
2. $t_2 = -t_1$
3. $t_3 = c+d$
4. $t_4 = t_2 * t_3$
5. $t_5 = a+b$
6. $t_6 = t_5+c$
7. $t_7 = t_4+t_6$

After writing the general representation we will proceed for quadriples, triples, indirect triple representation of the expression.

1. Quadriple representation:-

Topic _____ Date ___/___/_____ Page No. :_____

	operator	operand (1)	Operand (2)	Result
1.	+	a	b	
2.	−	t_1		
3.	+	c	d	
4.	*	t_2	t_3	
5.	+	a	b	
6.	+	t_5	c	
7.	+	t_4	t_6	

→ Advantage:- Statements can be move around.

→ Disadvantage:- Too much space is wasted.

(2) Triple Representation:-

	operator	Operand (1)	Operand (2)
1	+	a	b
2	−	(i)	
3	+	c	d
4	*	(ii)	(iii)
5	+	a	b
6	+	(v)	c
7	+	(iv)	(vi)

→ Advantage:- Space is not wasted.

→ Disadvantage:- Statements cannot move around.

M/R

Topic _____ Date __/__/__ Page No. : _____

(3) Indirect Triple representation:-

1. (i.)
2. (ii.)
3. (iii.)
4. (iv.)
5. (v.)
6. (vi.)
7. (vii.)

In indirect triple, each instruction is not need to put into main memory only keep your instruction no.(pointer) in main memory & let the instructions to be in secondary memory

⇒ disadvantage – atleast 2 memory access required

⇒ advantage – Statements can be moved around.

→ BACK & PATCHING :-
(Conversion to 3-Address code)

Eg:- If (a < b) then t = 1
else t = 0

→ here we have 4-Address (a, b, t=0, t=1) so we need to convert it into 3-Add. code.

(i.) if (a < b) goto (i+3)
(i+1) t = 0
(i+2) goto (i+1)
(i+3) t = 1
(i+4)

MR

Topic _____ Date __/__/____ Page No. _____

NOTE:- During the conversion of statements into 3 Address code, leaving the ~~blanks~~ ~~spaces~~ of levels as empty and filling them later is called Backpatching.

eg: **a<b and c<d or e<f**

100) if (a<b) goto ___
101) $t_1 = 0$
102) goto ___
103) $t_1 = 1$
104) if (c<d) goto ___
105) $t_2 = 0$
106) goto ___
107) $t_2 = 1$
108) if (e<f) goto ___
109) $t_3 = 0$
110) goto ___
111) $t_3 = 1$
112) $t_4 = t_1$ and t_3
113) $t_5 = t_4$ / t_3

Note: [t_1 a<b | t_2 = c<d / t_3 = e<f]

MR