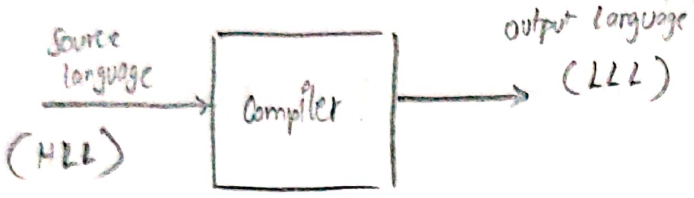
02/12/2024 + UNIT - I +

⇒ COMPILER :

→ Compiler is a Translater which converts high level language (Human Readable format) to low level language. (Assembly language).



⇒ Interpreater :

→ Interpreater is a Translater which convers high level language to low level language. But, it will check line-by-line if any mistake in one line it will skip that line.

⇒ Phases of Compiler :

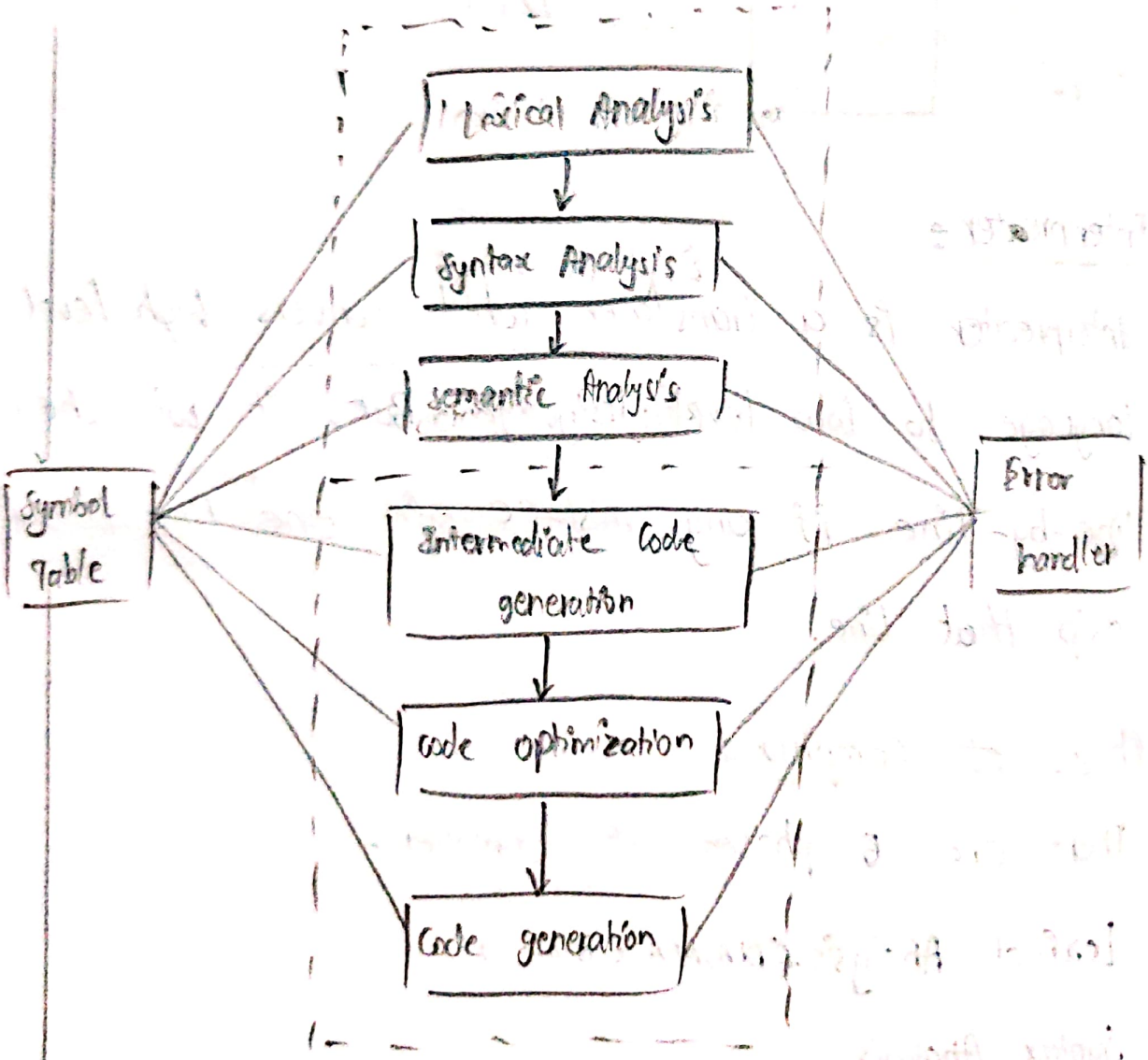
⇒ There are 6 phases of Compiler -

1) Lexical Analysis (DFA, NFA, ε-NFA, ...)
2) Syntax Analysis
3) Semantics Analysis
4) Intermediate Code generation
5) Code Optimization.
6) Code generation

02/12/2024

*NOTE:
```
int main() {
    int i=0;
    if (i==1)
      printf("Hi");
    return 0;
}
```
Dead code

* Output: (No output).



-> In lexical Analysis we divide the entire code into small tokens

02/12/2024

LEXICAL ANALYSIS -

→ In lexical analysis the input will be divided into tokens (or) lexemes.

→ It removes comment lines.

→ It removes white spaces

→ It ignores whatever we write in the double quotes.

→ It will divide the tokens into:

- Identifiers
- Numericals
- Key words
- seperaters {,, ;}
- Operators {=, >, <, >=}

SYNTAX ANALYSIS -

→ In syntax analysis we will verify the syntax of a given language

→ We can create syntax tree (or) parse tree for verifying the grammer.

→ In syntax analysis it will check balanced paranthesis (using PDA).

02/12/2024

Semantic Analysis:

→ In semantic Analysis we will check for "Type Mismatch errors" by using 'Annotated Parse Tree'.

Intermediate Code Generation: (ICG)

→ In ICG we generate a Temperory code for optimiz

→ There are three ways to generate code:

- Abstract syntax
- Polish Notation
- Tree address code
- ———————

→ We require ICG because any modification will be done on Temperory code.

Code Optimization:

→ In code optimization we can optimize the code by eliminating "Dead code", 'copy propagation' and so on.

Code Generation

→ It is the last stage of compiler design.

→ It generates the actual assembly code

05/12/2024

* Example: $c = i * j + 60;$

$id_1 = id_2 + id_3 + 60$

$z = id_1$

== assignment operator

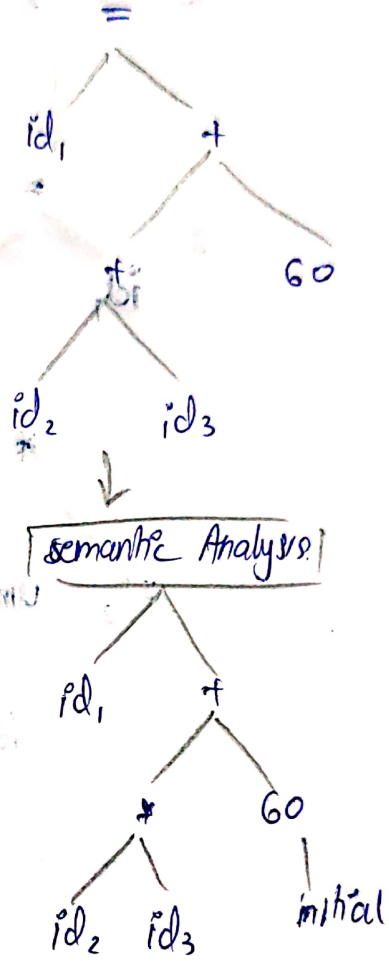
$i = id_2$

* : arithmatic operator

$j = id_3$

+ : arithmatic operator

60 - Constant



⇒ ICG (Intermediate Code generation)

$t_1 = id_2$

$t_2 = id_3$

$t_3 = t_1 * t_2$

$t_4 = t_3 + 60$

$c = t_4$

⇒ Code Optimization: $t_1 = id_2 * id_3$

$c = t_1 + 60$

* Example:

Q) Write different steps for each phase for the expression

$x = b^* - c + b^* - c$

Sol⇒ Given,

$x = (b^* - c) + (b^* - c)$

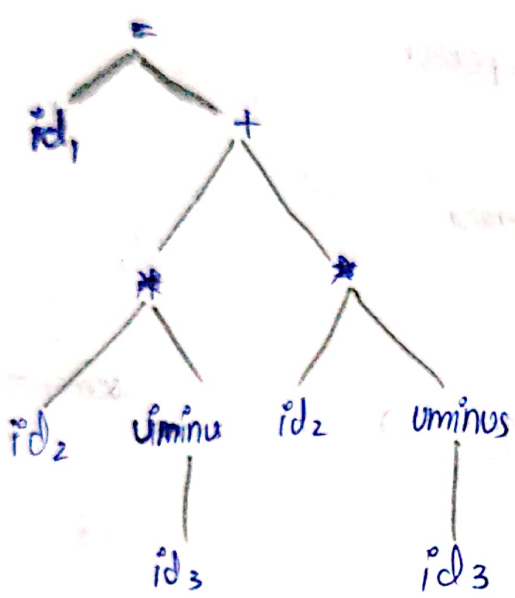
05/12/2024

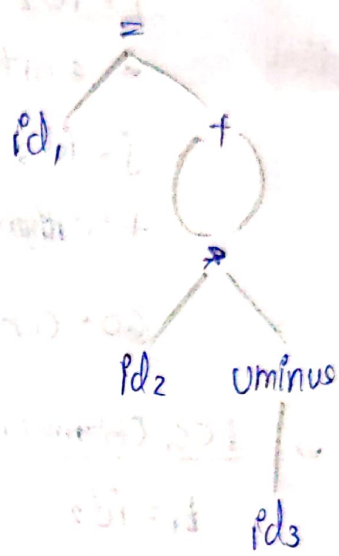
$x = (b^* - c) + (b^* - c)$

↓

$id_1 = (id_2^* - id_3) + (id_2^* - id_3)$

↓





⇓

t_1 = unary minus c

$t_2 = b * t_1$

t_3 = uminus c

$t_4 = b * t_3$

$t_5 = t_2 + t_4$

$c = t_5$

⇓

t_1 = uminus c

05/12/2024

$t_2 = b * t_1$

$c = t_2 + t_2$

MOVE : Unmul , R_1

MOVE : b , R_2

MOVE : R_1 , R_2

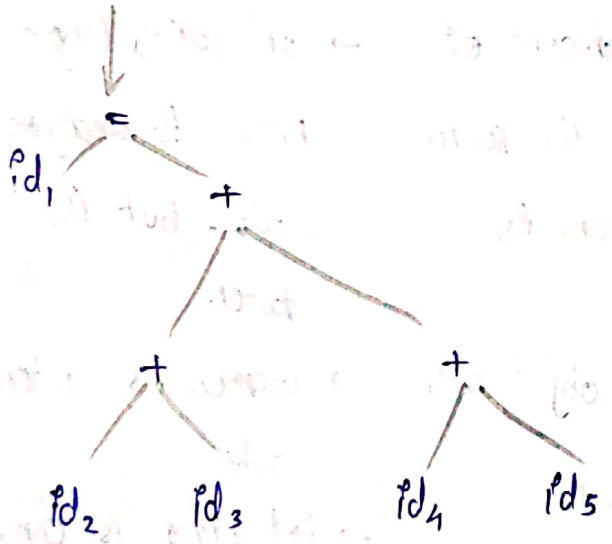
ADD : R_2 , R_2 , MOVE : R_2 , C

Q2) Write different steps for the expression :

$X = (a+b) + (c+d)$

Solln

Sol) $id_1 = (id_2 + id_3) + (id_4 + id_5)$



(ICG)

$t_1 = id_5$

$t_2 = id_4 + id_5$

$t_3 = id_3$

$t_4 = id_2 + id_3$

$t_5 = t_2 + t_4$

(Eqn)

$t_1 = id_2 + id_3$

$t_2 = id_1 + id_5$

$id_1 = t_1 + t_2$.

09/12/2024

↓

Code Generation (CG)

```
MOVF   id3 , R1
MOV    id4 , R3
ADD    R4 , R3  :=>  MOVF  R1
```

•> Differences between Interpreter and Compiler:

Interpreter	Compiler
→ Translates program one statement at a time	→ Scans the entire program and translates it as a whole into machine code.
→ It takes less amount of time to analyze the source code. but execution is slower.	→ It takes large amount of time to analyze the source code. but the execution is faster.
→ No intermediate object code	→ Generates intermediate object code
→ Debugging is easy	→ Debugging is comparatively hard.
→ Example: Python, Ruby	→ Example: C programming, C++.

•> Types of Compilers:

1) Incremental Compiler
2) Cross Compiler

09/12/2024

⇒ INCREMENTAL COMPILER -

→ It is a compiler which performs the run of the code only to the modified part of the code not the entire code.

→ It is a compiler which performs the recompilation of only the modified source rather than compiling the whole source program.

⇒ CROSS COMPILATION -

→ A compiler which runs on one machine and produces the target code for another machine such compiler is called as "Cross ~~Compilation~~ Compiler."

→ Compiler runs on platform X and target code runs on platform Y.

09/12/2024

BOOTSTRAPPING -

→ Bootstrapping widely used in compilation development.

→ It is a process in which simple language is used to translate more complicated program which in turn may handle far more complicated program.

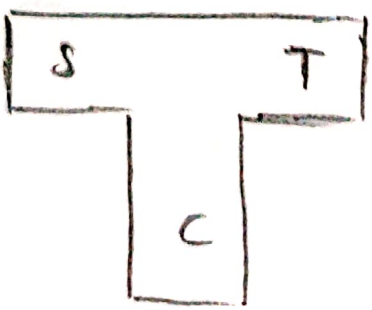
→ This complicated program can further handle even more complicated program and so on.

* A compiler can be characterised by three languages

1) Source Language
2) Target language
3) Implementation language

⇒ The T- Diagram shows a compiler S_C^T

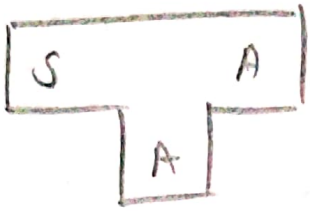
Source S, Target T, Implementation C.



09/12/2024

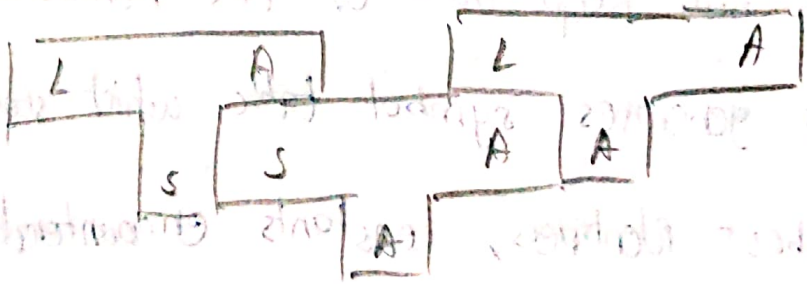
⇒ Follow steps to produce a new language L for

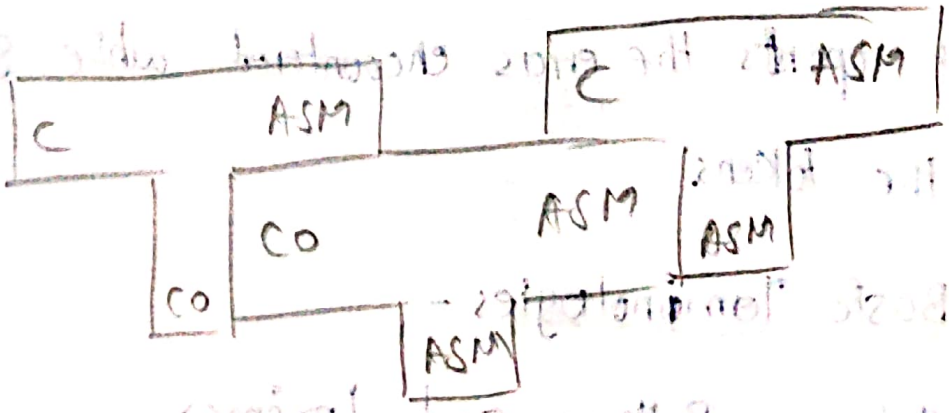
1) Create a compiler for subsets s, of the desired language L, is the layout 'A' and compiler runs on machine A.



2) Create a compiler L.A for lay L written in subset 'l'

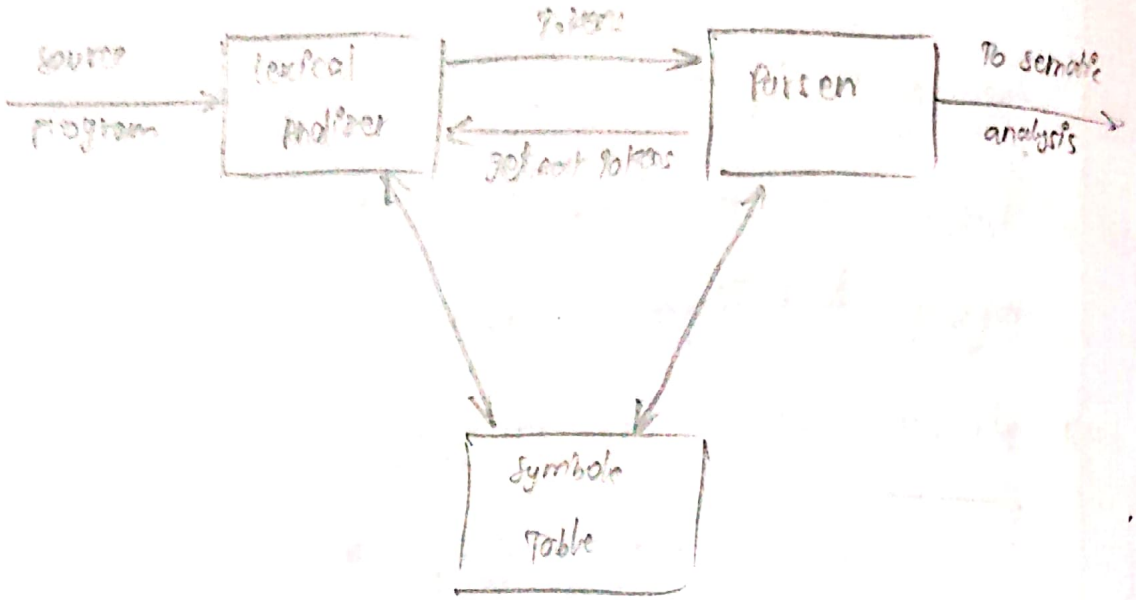
3) Compile $L_s A$ using compiler $S_A A$ to obtain L_A^A, L_A^A as a compiler for layer 'l' which runs on machine 'A' and produce machine A.





10/11/2024

* The role of Lexical Analyzer:



-> Functions of Lexical Analyzer:

- It produces stream of tokens
- It eliminates blanks and comment lines
- It will keep track of line numbers
- It generates symbol table which stores the information about identifiers, constants encountered in the input
- It reports the errors encountered while generating the tokens.

-> Basic Terminologies -

=> Tokens, Patterns and Lexemes.

10/12/2024

* Tokens - The smallest individual unit of a program is known as 'Token'.

* Pattern - A pattern rule is a description of the form that the lexeme of a token may taken.

* Lexemes - A lexeme is a sequence of characters in the source program that matches the pattern for a token

⇒ Example of lexical Analysis, Tokens, Non-Tokens -

```
#include<stdio.h>
int maximum (int x, int y) {
    // This will compare 2 number
    if (x > y) return x;
    else
    {
    return y;
    }
}
```

10/12/2024

Lexemes	Token
int	Keyword
maximum	identifier
(	operator
int	Keyword
x	Identifier
,	Operator
int	Keyword
y	Identifier
)	Operator
{	Operator
if	Keyword

=> Input Buffering :

- The lexical analyzer scans only one character in the input string at a time from left to right

10/12/2024

- It makes use of two pointers begin pointer (bptr) and forward pointer (fptr), for maintaining a track of the input scanned
- Initially both the pointers point to the first character of input string.

⇒ SPECIFICATION OF TOKENS:

- In theory of computations Regular Expressions are used to formalize the specification of tokens
- Regular expressions are means of specifying regular languages.
- Examples: letter_(letter_|digit)
- Each regular expression is a pattern specifying the form of strings.

⇒ Recognition of Tokens:

→ Each token such as identifier, keyword, constant and operators is represented with two values:

(i) Token value.

(ii) token type.

16/12/2024

Token	Code	value
if	1	-
else	2	-
while	3	-
for	4	-
identifier	5	Ptr to symbol table
Constant	6	Ptr to symbol table
<	7	1
<=	7	2
>	7	3
>=	7	4
!=	7	5
+	9	1
-	9	2
=	10	-

```
if (a<=0)
    i = i+2;
else
    j = j-2;
```

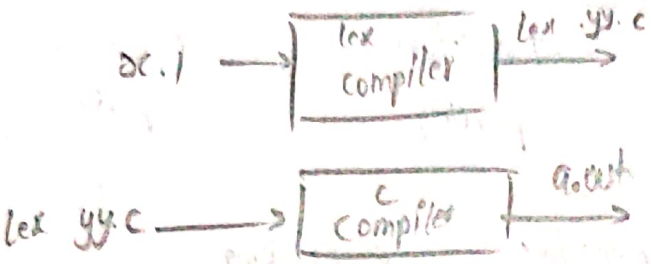
Token type	Token value

⇒ **Lex program-**

Unix provided lex tool to recognize the lex programs. The lex file will be saved with an extention of

16/12/2024

1. After compilation by lex compiler it generates c file which is lex yy.c.



lex ex.l
cc lex yy.c
./a.out

⇒ Recognition of token by using lex program -

lex program contains 3 sections

① Declaration section
② Rule section
③ Auxiliary C- section (procedure section)

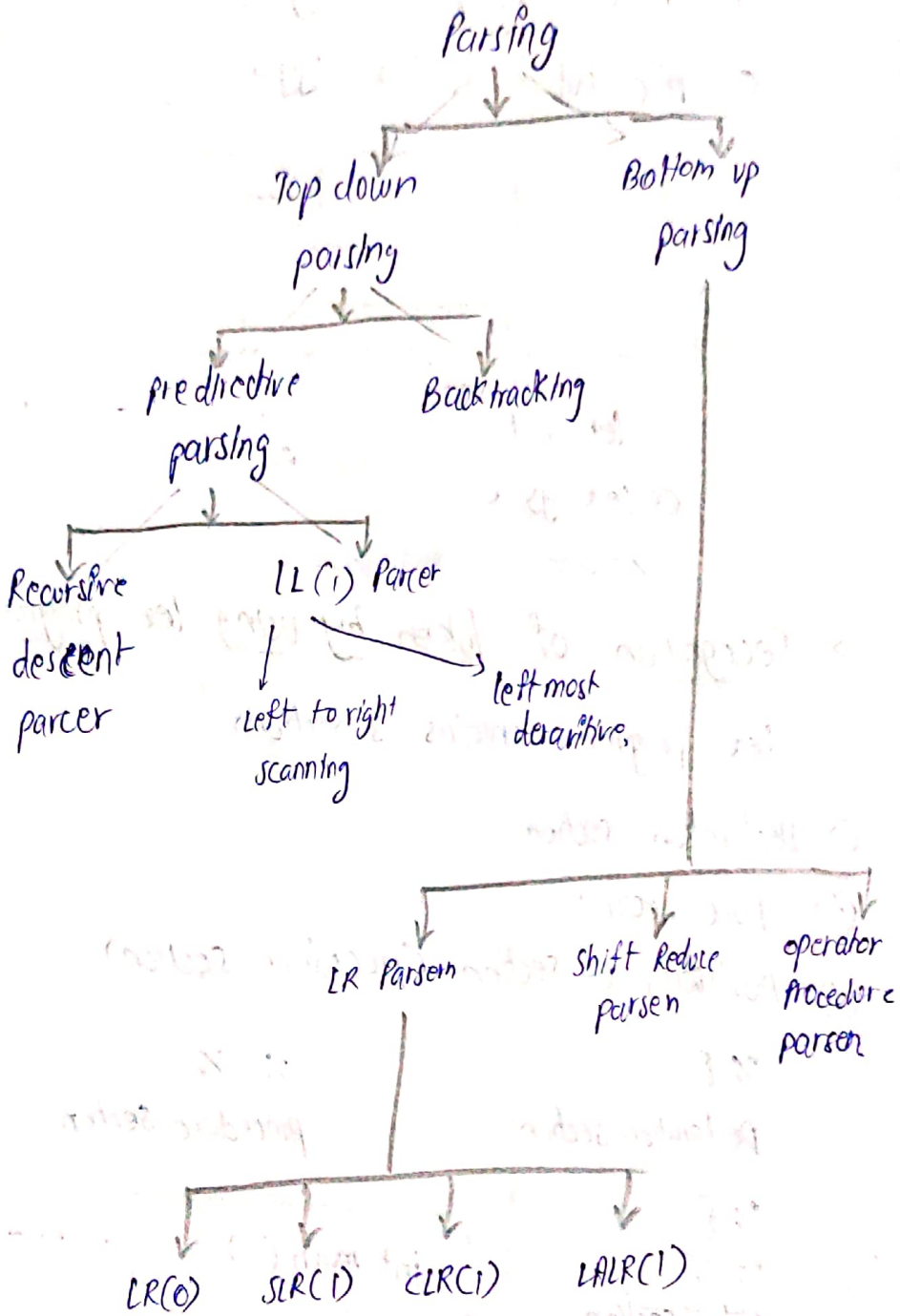
%{
Declaration section

%%
procedure section

%{
%%
Rule section
int vow cout = 0;
int const count = 0;
%?
%%

int main()
{ printf("Enter the string of vowels and consonents:");
yylex();
printf("Number of vowels are: %d\n, vow, cant);
printf("Number of constants are: %d\n, const, count);
return 0;
}

17/12/2024 UNIT - II



⇒ Problems in TOP DOWN PARSING:

→ There are three problems in top down parsing.

(i) Left Recursion (LR).

(ii) Left Factoring (LF)

(iii) Backtracking.

17/12/2024

⇒ Left Recursion-

→ If the production is in the form of $A \to A\alpha / \beta$ will be re-written as $A \to \beta A'$, $A' \to \alpha A' / \epsilon$

* Example:

$E \to E+T / T$

$A \quad A \; \alpha \; \beta$

∴ $A \to \beta A' \Rightarrow E \to TE'$

$A' \to \alpha A' / \epsilon \Rightarrow E' \to +TE' / \epsilon$

⇒ Left factoring-

→ If any production has more than one common prefixes on the right hand side as alternatives then left factoring problem arises.

→ Example: If problem arises as $A \to A\alpha_1 / A\alpha_2 / \beta$ then it will be rewritten as:

$A \to \beta A'$

$A' \to \alpha_1 / \alpha_2$

* Example: $S \to iEtS / iEtSeS / a$

$A \quad \alpha_1 \quad A \; \alpha_2 \quad \beta$

∴ $A \to \beta A' \Rightarrow S \to aiEt'$

∴ $A' \to \alpha_1 / \alpha_2 \Rightarrow$

17/12/2024

* Example- $S \rightarrow iEtS / iEtSeS / a$

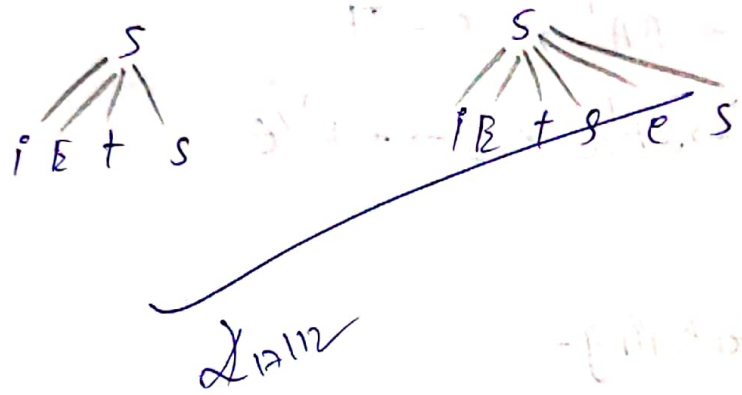
$E \rightarrow b$

let, $S \rightarrow iEtSeS$ [$\because S \rightarrow iEtSeS$]

$S \rightarrow iEtiEtSeS$ [$\because S \rightarrow iEtS$]

$S \rightarrow ibtibtSeS$ [$\because E \rightarrow b$]

$S \rightarrow ibtibtaea$ [$\because S \rightarrow a$]



* Example:

Do the left factoring for the grammer:

$S \rightarrow bSSaSS / bSSaSb / bSb / a$

Sol: Given,

$S \rightarrow \underset{A\ \alpha_1}{bSSaSS} / \underset{A\ \alpha_2}{bSSaSb} / \underset{A\ \alpha_3}{bSb} / \underset{\gamma}{a}$

$\therefore A = bS$

$\alpha_1 = SaSS$

$\alpha_2 = SaSb$

$\alpha_3 = b$

$\gamma = a$

19/12/2024

$\therefore S \rightarrow bSS'$

$S' \rightarrow \underset{A}{S}\underset{\alpha_1}{aS} / \underset{A}{S}\underset{\alpha_2}{aS}\underset{\gamma}{b}$ → still there is common prefix in this production

$S' \rightarrow SaS' / b$

$S'' \rightarrow aS / Sb$

$\therefore S \rightarrow bSS'$

$S' \rightarrow SaS'' / b$

$S'' \rightarrow aS / Sb$

* Example:

(2) $S \rightarrow aSSbS / aSaSb / abb / b$

Sol: Given,

$S \rightarrow \underset{A}{a}\underset{\alpha_1}{SSbS} / \underset{A}{a}\underset{\alpha_2}{SaSb} / \underset{A}{a}\underset{\alpha_3}{bb} / \underset{\gamma}{b}$

$\therefore A \rightleftharpoons a$

$\alpha_1 = SSbS$

$\alpha_2 = SaSb$

$\alpha_3 = abb$

$\gamma = b$

$S \rightarrow aS' / b$

$S' \rightarrow SSbS / SaSb / bb$

$S' \rightarrow SS' / bb$

$S'' \rightarrow SbS / aSb$

19/12/2024

③ Example: $A \rightarrow ABd / Aa / b$

$B \rightarrow Bc / b$

Sol: Here we have a problem called left Recursion problem.

If $A \rightarrow Aa / \beta$

Then $A \rightarrow AA'$ and $A' \rightarrow \alpha A' / \epsilon$

Given,

$A \rightarrow \underset{A}{ABd} / \underset{A}{Aa} / \underset{\gamma}{b}$

$\therefore B \rightarrow \underset{A}{Bc} / \underset{\beta}{b}$

$\therefore A \rightarrow aA'$

$A' \rightarrow BdA' / aA' / \epsilon$

$B \rightarrow bB'$

$B' \rightarrow cB' / \epsilon$

⊛ Problems:

① $E \rightarrow E+T / T$

$T \rightarrow T*F / F$

$F \rightarrow (E) / id$

Sol: $E \rightarrow TE'$

$E' \rightarrow +TE / \epsilon$

19/12/2024

$T \rightarrow F x'$

$T' \rightarrow *FT'/\epsilon$

$F \rightarrow (E)$

$E \rightarrow id$

② $S \rightarrow (L)/ax$

$L \rightarrow L,S/S$

Sol:

$S \rightarrow (L)/a$

$L \rightarrow SL'$

$L' \rightarrow ,SL'/\epsilon$

③ $A \rightarrow Ba/Aa/c$

$B \rightarrow Bb/Ab/d$

Sol:

$A \rightarrow BaA'/cA'$

$A' \rightarrow aA'/\epsilon$

$B \rightarrow AbB'$

$B' \rightarrow bB'/\epsilon$

Killl2

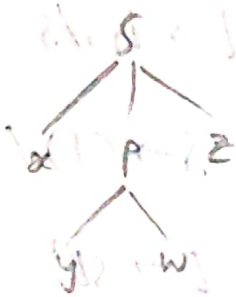
23/12/2021

Backtracking –

→ Backtracking is a technique in which for expansion of non-terminal we choose one alternative and if some mismatch occurs then we try another alternative

* Example – $S \rightarrow xPz$

$P \rightarrow yw/y$

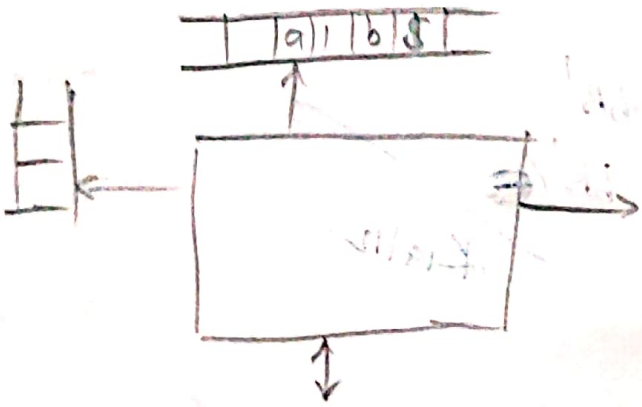


Predictive parsing (LL(1) parsen) –

→ It is a non-recursive type.

→ In LL(1) first 'L' means "left to right scanning" and second 'L' means "left most derivation" and '1' means "look ahead symbol"

→ It constructs LL(1) parsing table. structure of LL(1) parser :



23/12/2024

Steps for constructing LL(1) parsing Table:

- Step:1 - Calculate First and follow functions
- step:2 - By using first and follow construct parsing table
- step:3 - Parse the string by using parsing table.

- Note: First the given grammer should be free from
  1) left recursion
  2) left factoring and
  3) Backtracking problems.

First() and Follow():

- First function -
- First means, first appearing terminal on RHS.
- By using 3 rules we can calculate First function

- Rule: 1 -
- If a function $A \rightarrow a$

  Then First(A) = {a}

- Example - ① $S \rightarrow aABb$

  first(S) = {a}.

  ② $A \rightarrow aBD$

  ∴ First(A) = {a}.

23/12/2024

* **Rule: 2 -**

→ If the production is in the form of $A \rightarrow \epsilon$

$First(A) = \{\epsilon\}$

* **Rule: 3-**

→ If the production is in the form of $A \rightarrow X_1 X_2 X_3$.

then,

$First(A) = First(X_1) = First(X_2) = First(X_3)$

(excluding 'ϵ')

* **Example -**

① $S \rightarrow ABCD$

$C \rightarrow C/\epsilon$

$D \rightarrow d$

$A \rightarrow aA/\epsilon$

$B \rightarrow b/\epsilon$

$\therefore FIRST(S) = FIRST(A)$

$= \{a\}.$

$FIRST(S) = FIRST(B)$

$= \{b\}$

$FIRST(S) = FIRST(C)$

$= \{c\}$

$FIRST(S) = FIRST(D)$

$= \{d\}.$

$\therefore FIRST(S) = \{a, b, c, d\}.$

23/12/2024

* Problems:

① Calculate the FIRST() for the following grammer.

$S \rightarrow ABCDE$

$A \rightarrow a/\epsilon$

$B \rightarrow b$

$C \rightarrow c/\epsilon$

$D \rightarrow d/\epsilon$

$E \rightarrow e$

Sol: FIRST(S) = FIRST(A) = $\{a, \epsilon\}$ $\Rightarrow$ ∴

= FIRST(B) = $\{b\}$

= FIRST(C) = $\{c, \epsilon\}$

= FIRST(D) = $\{d, \epsilon\}$

= FIRST(E) = $\{e\}$.

FOLLOW() function:

→ follow() means set of terminal symbols that appear immediately to the right of A.

→ FOLLOW(A) = {a / $S \rightarrow \alpha A a \beta$, where α, β are alternating and terminals}

03/12/2024

→ follow() can be calculated using 3 rules:

* Rule:1-

→ For stationary symbols add $ in follow(S).

* Rule:2-

→ If the production is in the form of $A \to \alpha B$

then, follow(B) = follow(A).

* Rule:3-

→ If the production is in the form of $A \to \alpha B \beta$

then follow(B) = First(β), If First(β) contains 'ϵ' then instead of 'ϵ' substitute follow(A)

→ If the First(β) doesn't contain 'ϵ'

then, follow(B) = First(β).

* Example-

① $E \to E+T/T$

$T \to T*F/F$

$F \to (E)/id$

$\therefore \underset{A}{T} \to \underset{A}{T} \underset{\alpha}{*} \underset{\beta}{F}/\underset{\beta}{F}$

Sol→ $\therefore \underset{A}{E} \to \underset{A}{E} \underset{\alpha}{+} \underset{\beta}{T}/\underset{\beta}{T}$

$\therefore E \to TE'$

28/10/2024

$E' \rightarrow +TE' / \epsilon$ $\quad$ $E \rightarrow +TE'$

$T \rightarrow FT'$ $\quad$ $E' \rightarrow \epsilon$

$T' \rightarrow *FT' / \epsilon$

$F \rightarrow (E)$

$F \rightarrow id$

⇒ Table for First() and Follow():

	First	Follow
E	{(, id}	{), $}
E'	{+, ∈}	{), $}
T	{(, d}	{+,), $}
T'	{*, ∈}	{+,), $}
T	{(, id}	{*, +,), $}

∴ FIRST(E) = FIRST(T) = FIRST(F)

= {(, id}

FIRST(E') = {+, ∈}

FIRST(T') = {x, e}

FOLLOW(E) = {), $}

FOLLOW(E') = FOLLOW(E)

= {), $}

FOLLOW(T) = {+, ∈} • {+, FOLLOW(E)}

= {+,), $}

∴ $E \rightarrow TE'$ (A → B P)

FIRST(E') = {+, ∈}

28/12/2024

FOLLOW(T'): $T' \rightarrow *FT'$ $\quad T \rightarrow FT'$ (T: A; F: α; T': B)

FOLLOW(T), Rule 2.

FOLLOW(F): $T \rightarrow FT'$ ($A\ B\ \beta$) $\quad T' \rightarrow *FT'$

FIRST(T') = {*, +,), $}

= {*, ∈}

= {*, FOLLOW(T)}

= {*, +,), $}

Calculate:

$S \rightarrow aAB / bA / \epsilon$

$A \rightarrow aAb / \epsilon$

$B \rightarrow bB / \epsilon$

$S \rightarrow L, S$

$L \rightarrow L, id / id$.

→ Calculate FIRST and FOLLOW:

1) $S \rightarrow aBdh$
$B \rightarrow \epsilon$
$C \rightarrow bC / \epsilon$
$D \rightarrow Ef$
$E \rightarrow g / \epsilon$
$F \rightarrow f / \epsilon$

2) $S \rightarrow A$
$A \rightarrow aB / Ad$
$B \rightarrow b$
$C \rightarrow g$

3) $S \rightarrow ACB / CbB / Ba$
$A \rightarrow aa / BC$
$B \rightarrow g / \epsilon$
$C \rightarrow h / \epsilon$

4) $S \rightarrow AaAb / BbBa$
$A \rightarrow \epsilon$
$B \rightarrow \epsilon$

5) $S \rightarrow (L) / a$
$L \rightarrow L, S / S$ (L: A; L: A; S: α; S: β)
$L \rightarrow SL'$
$L' \rightarrow SL' / \epsilon$

Left Recursion

$L \rightarrow SL'$
$L' \rightarrow SL' / \epsilon$

28/12/2024

	First	follow
S	{c,a}	{,,), $}
L	{c,a}	{)}
L'	{,, ∈}	{)}

$L \rightarrow sL'$

$A \rightarrow \alpha\beta$

1) $S \rightarrow aBDh$

$B \rightarrow cC$

$C \rightarrow bC/\epsilon$

$D \rightarrow Ef$

$E \rightarrow g/\epsilon$

$F \rightarrow f/\epsilon$

First(D) = {g, f, ∈}

$B \rightarrow B\alpha\beta$

FOLLOW(S) = FOLLOW(D).

	First	follow
S	{a}	{$}
B	{c}	{g, f, $}
C	{b, ∈}	{g, f, $}
D	{g, f, ∈}	{h}
E	{g, ∈}	{f, h}
F	{f, ∈}	{h}

$S \rightarrow A$

$A \rightarrow aB/Ad$

$B \rightarrow bc$

$C \rightarrow y$

solving:

$A \rightarrow Ad/aB$

$A \rightarrow aBA'$

$A' \rightarrow dA'/\epsilon$

Construction of Predictive Parsing Table:

- **Step:1** - All the terminals in a grammer are written in columns and non-terminals are written in rows.
- **Step:2** - Fill the entries of the table with first of the non-terminals with the respective terminals.

26/12/2024

→ If first of non-terminal contains ϵ then fill in follow of that non-terminal

- Example: $E \to E+T/T$

$T \to T*F/F$

$F \to (E)$

$I \to id$

$\therefore E \to TE'$

$E' \to +TE'/\epsilon$

$T \to FT'$

$T' \to *FT'/\epsilon$

$F \to (E)$

$F \to id$

Note: Fill the stack with production rule in reverse order.

	First	Follow
E	{(, id}	{), $}
E'	{+, ε}	{), $}
T	{(, id}	{+,), $}
T'	{*, ε}	{+,), $}
F	{(, id}	{*, +,), $}
F'		

	+	*	(	)	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			$F \to (E)$		$F \to id$	

26/12/2024

stack	input	action
$E	id + id * id $	$E \to E'$
$E'T	id + id * id $	$T \to FT'$
$E'T'F	id + id * id $	$F \to id$
$E'T'id	id + id * id $	match rows pop id
$E'T'	+ id * id $	$T' \to \epsilon$
$E'	+ id * id $	$E' \to +TE'$
$E'T+	+ id * id $	match
$E'T	id * id $	$T \to FT'$
$E'T'F	id * id $	$F \to id$
$E'T'id	id * id $	pop
$E'T'	* id $	$T' \to *FT'$
$E'T'F*	* id $	pop
$E'T'F	id $	$F \to id$
$E'T'id	id $	$T' \to \epsilon$
$E'	$	$E' \to \epsilon$
$	$	accept.

26/12/2024

Q) Construct LL parser for the following grammer:

$E \rightarrow E+T / T$

$T \rightarrow T*F / F$

$F \rightarrow F* / a / b$

Q2) Construct LL, parser for the grammer:

$S \rightarrow SS / (S) / \epsilon$

Q3) First Reursion.

$S \rightarrow A / a$

$A \rightarrow a$

Sol:

	First	Follow
S	{a}	{$}
A	{a}	{$}

	a	$
S	$S \rightarrow A$ } $S \rightarrow a$ }	
A	$A \rightarrow a$	

→ There are two entries in a single column so, it is not a predictive parser.

26/12/2024

Sol:2 Given,

$S \rightarrow SS \mid (S) \mid \epsilon$

	First	Follow
E	{a, b}	{$}
E'	{+, ε}	{$}
T	{a, b}	{+, $}
T'	{*, ε}	
F	{a, b}	
F'	{*, ε}	

$E \rightarrow TE'$

$E' \rightarrow *TE' / \epsilon$

$T \rightarrow *T'$

$T' \rightarrow *FT'$

$F \rightarrow aF'$

$F \rightarrow bF'$

$F' \rightarrow *F' / \epsilon$

X27/12 predective

Q. Construct predrichic parser for the grammer

$S \rightarrow (L) / a$

$L \rightarrow L, S / S$

and parse the string $w = (a, (a, a))$

Sol: Given,

$S \rightarrow (L) / a$

$L \rightarrow L, S / S$

Here, $S \rightarrow (L) / a$

$S \rightarrow (L)$

$S \rightarrow Id$

30/12/2024

$\therefore \underset{A}{L} \rightarrow \underset{A}{L}, \underset{\alpha}{S} / \underset{\beta}{S}$

$\therefore A \rightarrow \beta A' \Rightarrow L \rightarrow SL'$

$A' \rightarrow \alpha A' / \epsilon \Rightarrow L' \rightarrow SL' / \epsilon$

* Finding first() and follow():

	first()	follow()
S	{c, a}	{,,), $}
L	{c, a}	{)}
L'	{,, ∈}	{)}

* Table:

	(	)	a	,	$
S	S→(L)		S→a		
L	L→SL'		L→SL'		
L'		L'→∈		L'→,SL'	

S

( )

L

30/12/2024

stack	input	action
$ S	(a,a) $	S → (L)
$)L(	(a,a) $	match pop '('
$)L	a,a) $	L → SL'
$)L'S	a,a) $	S → a
$ L'a	a,a) $	match pop 'a'
$ L'	,a) $	L' → SL'
$ L'S,	,a) $	match
$) L'a	a) $	match pop 'a'
$) L'	) $	match pop ')'
$	$	empty stack
		accept

20/12/2024

Q) Construct predictive parser for the grammar:

$S \rightarrow iEtS / iEtSeS / a$

$E \rightarrow b$

Sol) Given, $\underset{A}{S} \rightarrow \underset{A}{iEtS} / \underset{A\ \alpha_1}{iEtSeS} / a$

$\therefore S \rightarrow \underset{A}{iEtS}\underset{A'}{S'} / a$

$S' \rightarrow \underset{\alpha_1}{eS} / \underset{\alpha_2}{\epsilon}$

$E \rightarrow b$

* First() and Follow():

	First()	Follow()
S	{i, a}	{e, $}
S'	{e, ∈}	{e, $}
E	{b}	{t}

Consider: $\underset{A.}{S} \rightarrow iEtS\underset{B.}{S'} / a$

$\therefore A' \rightarrow \{e, \epsilon\} \Rightarrow \therefore \{e, \$\}$.

30/12/2024

	i	t	e	a	$	b
S	$S \to iEtSS'$			$S \to a$		
S'			$S' \to eS$ $S' \to \epsilon$		$S' \to \epsilon$	
E						$E \to b$

Note: If first of non-terminal intersection is not equals to $\emptyset$ then the grammer is said to be not a LL, grammer.

$\therefore First(NT) \cap Follow(NT) \neq \emptyset$

Important

→ In the above parsing we are having two multiple entries under single non-terminal. so the given grammer is not a LL, grammer.

Q3) Construct a predictive parser for the grammer;

$S \to AaAb / BbBa$

$A \to t$

$B \to t$

and $w = baa$

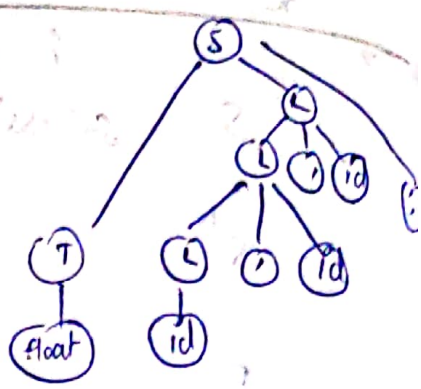
→ Bottom of parsing:

→ In bottom of parsing we will start from leaf node and reaches to root node.

→ In this type of parsing we use right-most derivation in reverse order.

02/01/2025

Q) Consider the following grammer:



S → TL ;

T → int / float / char

L → List, id / id

for the string w = float id, id, id

i) Shift Reduce parser:

→ In shift reduce parsers there are four actions

1) Shift
2) Reduce
3) Accept
4) Error

• Shift -

→ Move the symbols from the input buffer of the stack until desired hangle approaches

• Reduce -

→ If handle approaches reduce the handle with desired production rule.

• Accept -

→ If stack contains start symbol and input is

completed then that string is said to be parsed.

* Error-

→ If the parser unable to shift (or) reduce then parser is in error condition.

* Example-

Q) Construct a shift reduce parser for the following grammer:

$E \rightarrow E+E$
$E \rightarrow id$ for the string $w = id+id+id$

Sol →

Stack	Input buffer	Action
$	id+id+id$	shift 'id'
$id	+id+id $	Reduce id by E→id
$E	+id+id $	
$E+	id+id $	shift 'id'
$E+id	+id $	Reduce id by E→id
$E+E	+id $	
$E+E+	id $	Reduce E→E+E

02/01/2025

$E	+id$	shift +
$E+	id$	shift id
$E+id	$	Reduce id by E→ id
$E+E	$	Reduce E→ E+E
∴ $E	$	accept.

② Construct shift reduce parser for the grammer:

S→ (L)/a

L→ L,S/S

where, w⇒(a, (a,a))

Sol⇒ Given,

S→ (L)/a

L→ L,S/S

stack	input buffer	Action
$	(a, (a,a)) $	shift (
$(	a, (a,a)) $	shift a
$(a	, (a,a)) $	Reduce a by S→a
$(S	, (a,a)) $	Reduce S by L→S

02/01/2025

$(L	,(a,a))$	shift ,
$(L,	(a,a))$	Shift (
$(L,(	a,a))$	shift a
$(L,(a	,a))$	Reduce a by S→a
$(L,(S	,a))$	Reduce S by L→S
$(L,(L	,a))$	shift ,
$(L,(L,	a))$	shift a
$(L,(L,a	))$	Reduce a by S→a
$(L,(L,S	))$	Reduce L,S by L→L,S
$(L,(L	))$	Shift)
$(L,(L)	)$	Reduce (L) by S→(L)
$(L,S	)$	Reduce L,S by L→L,S
$(L	)$	shift)
$(L)	$	Reduce (L) by S→(L)
$S	$	accept.

06/01/2025

OPERATOR PRECEDENCE PARSER:

→ To construct Operator precedence parser the grammer should be operator grammer

OPERATOR GRAMMER-

→ A grammer G is said to be operator grammer if it follows:

1) No two non-terminals should be adjacent.
2) No production on the RHs contains (Epsilon) ϵ.

* Example:

$E \to EAE / (E) / -E / id$

$A \to + / - / * / / / \hat{}$

It is not a operator grammer since, it contains Adjacent Non-terminals and contains ϵ productions.

Convert to operator grammer.

$E \to E+E / E*E / E-E / (E) / E \hat{} E / E / E / id$

Q) Construct Operator precedence Grammer (OPG) for the grammer:

$E \to E+E$

$E \to E*E$

$E \to id$ and $w = id + id * id$

06/01/2025

Sol: Step:1 - First we have to construct operator precedence Relation Table.

→ In this table both rows and columns are terminals.

→ The precedence among between pair of terminals is defined as 3 ways - $p<q$, $p=q$, $p>q$!

* Procedure Relation Table -

	id	+	*	$
id		>	>	>
+	<	>	<	>
*	<	>	>	>
$	<	>	<	

Note: If incoming symbol is having highest priority precedence over stack push the element into the stack.

* Table:

Stack	Input Buffer	action
$	id + id * id $	shift id
$ id	+ id * id $	Reduce E→id
$ E	+ id * id $	shift '+'
$ E +	id * id $	shift 'id'
$ E + id	* id $	Reduce E→id

06/01/2024

stack	Input buffer	Action
$E+E	* id $	shift *
$E+E*	id $	shift id
$E+E* id	$	Reduce E→id
$E+E* E	$	Reduce E→E*E
$ E+E	$	Reduce E+E
$ E	$	Accept.

Ex: w = id * id + id

* Procedure:

→ Insert '$' symbol for the string 'starting' and 'Ending' and Insert the relation between adjacent two terminals. from the table.

∴ $ < id > * < id > + < id > $

→ Now scan the input from left to Right starting from (less than) '<' symbol to the (greater than) > symbol and replace the handle with 'E'.

$ E * E + E $.

06/01/2025

$ < * > + > $

→ Handle between '<' and '>' is id

Replace id as E → id.

→ Apply relations from the table, Remove non-terminals

$ < * > + > $

→ Again check Handle

$ * $ Reduce E → E * E

→ $ < + > $ Reduce E → E + E

$ $ Accept

* Advantage: It is easy to implement

* Disadvantage: The operator '-' (minus) has two different precedences, Hence. It is hard to handle tokens like minus

→ It is applied for small class of grammers. because as the no. of operators increases the table is also increases.

⇒ Algorithm:

1) Set 'i' pointer to the 1st symbol of string 'w.

06/01/2024.

2) If $ is on the top of the stack. and 'A' is the symbol pointed by 'i' then, return

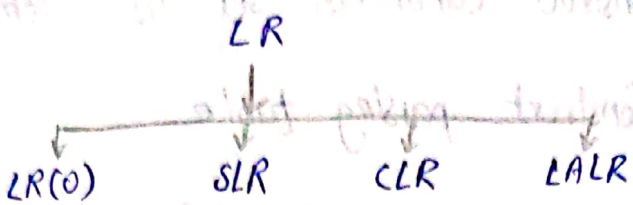
3) If 'A' is on the top of the stack and if 'B' is incoming symbol .. then:

(a) If A < B & C = B push element into the stack

(b) If A > B pop the element from stack with respective handle.

07/01/2025 ✦ UNIT - III ✦

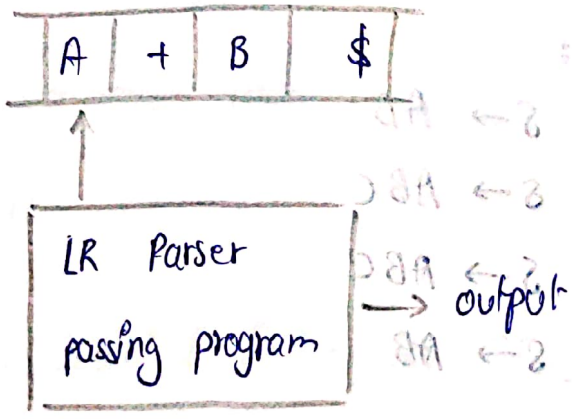
* LR Parsers:



⇒ LR - Stands for Left to Right scanning

1) LR(0)
2) Simple LR
3) Canonical LR
4) Look Ahead LR

* Structure:



07/01/2025

- Step:1- Construct canonical set of items.
- Step:2- Construct parsing table.
- Step:3- Pass the input string.

⇒ CANONICAL TRANSITION ITEMS:

→ The LR of 'o' item for grammer 'G' is the production in which . symbol inserted on the RHS of the production.

→ If the production is $S \rightarrow \epsilon$

then LR of o item is $S \rightarrow \cdot$

* Example:

$S \rightarrow AB$

$S \rightarrow ABC$

$S \rightarrow ABC$

$S \rightarrow AB$

→ By using closure and goto functions we have to construct the canonical items.

* Closure (I) - If the production rule is in form of $A \rightarrow \alpha . B\gamma$ and 'B' is having its own productions that is,

$B \rightarrow B_1 / B_2 / \cdots / B_n$

07/01/2025

we have to add all 'B' productions to the closure of 'I'. that is;

$B \to \beta_1$

$B \to \beta_2$

$\vdots$

$B \to \beta_n$

- Go to ()- on applying Go to() function (•) dot will be moved one position to the right

* Augumented Grammer-

→ For a given grammer 'G' we have to add a new symbol that is S' to the given grammer 'G'.

→ Construction of Canonical items is represented in the form of DFA.

Ex: Construct LR(0) Canonical items for the grammer

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

(or)

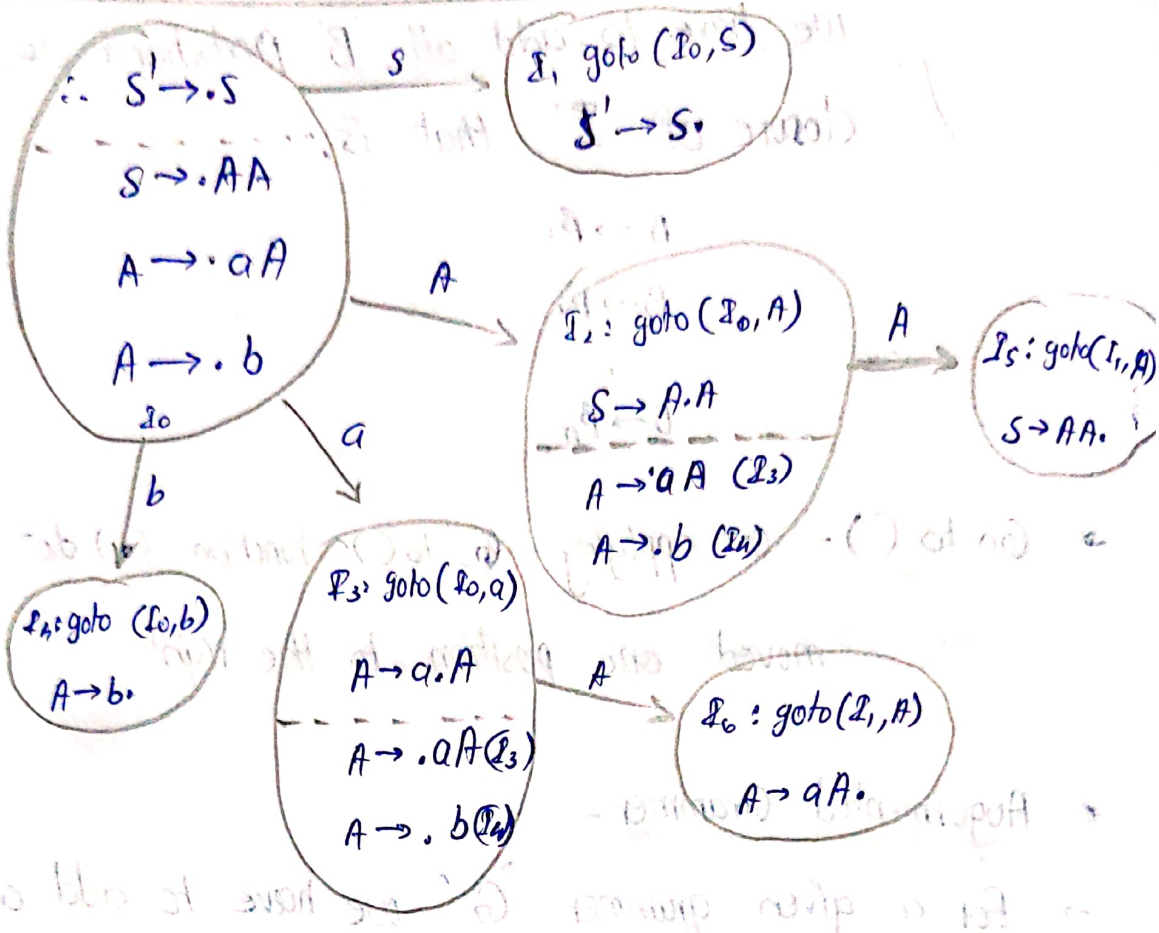
$S \to CC$
$C \to aC/d$

Sol: Given,

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

$\therefore$ $S \to AA$
$A \to aA$
$A \to b$

07/01/2025



⇒ Construction of Parsing Table-

S.No.	Action			goto	
	a	b	$	S	A
I_0	S3	S4		1	2
I_1			accept		
I_2	S3	S4			5
I_3	S3	S4			6
I_4	R3	R3	R3		
I_5	R1	R1	R1		
I_6	R2	R2	R2		

09/01/2025

Procedure-

If M[I_n, terminals] fill shift + destination number

If M[I_n, NT] fill destination state number in goto.

If any canonical item contains completed production (dot is moved to extreme right). Then fill with production not reduce in all the columns of Action part).

Example:

$S \rightarrow AA$

$A \rightarrow aA/b$

1) $S \rightarrow AA$
2) $A \rightarrow aA$
3) $A \rightarrow b$

passing the string: aabb$

↑ pointer

0	a	3

a on I_0 is S_2 From the "Construction of parsing Table. Increment the pointer (↑)

a a b b $

↑

09/01/2025

0	a	3	a	3

∴ a of I_3 is S_3 then, Increment the pointer.

∴ a a b b $
  ↑

0	a	3	a	3	b	4 ←

Stack

∵ b on I_3 = S_4, so 4 is pushed into ~~Queue~~ Then increment the pointer.

a a b b $
  ↑

0	a	3	a	3	b	4	A

push

b (I_3) = R_3 ⇒ A → b.

The no. of production is 1 so pop two elements from top. push the LHS into stack.

0	a	3	a	3	b	4	A	6	A

∴ 6(b) = R2 ⇒ A → aA

The no. of RHS = 2 so, pop 4 elements. and push LHS into stack

09/01/2025

⇒

0	a	3	a	3	b	4	A	6	A	6	A	2	4

$\therefore$ A (13) = 6

$I_6(b)$ · R_2 ⇒ $A \rightarrow \underline{aA}$.

↓

Pop - 4

$b(I_2) = 4$ → push 4 into stack

a a b b $

↑

⇒

0	a	3	a	3	b	4	A	6	A	6	A	2	4	2

$\$(I_4) = R_3$

$A \rightarrow \underline{b}.$

Pop - 2.

I_6 on b

Q) Construct LR(0) parser for the grammer:

$E \rightarrow E+T/T$

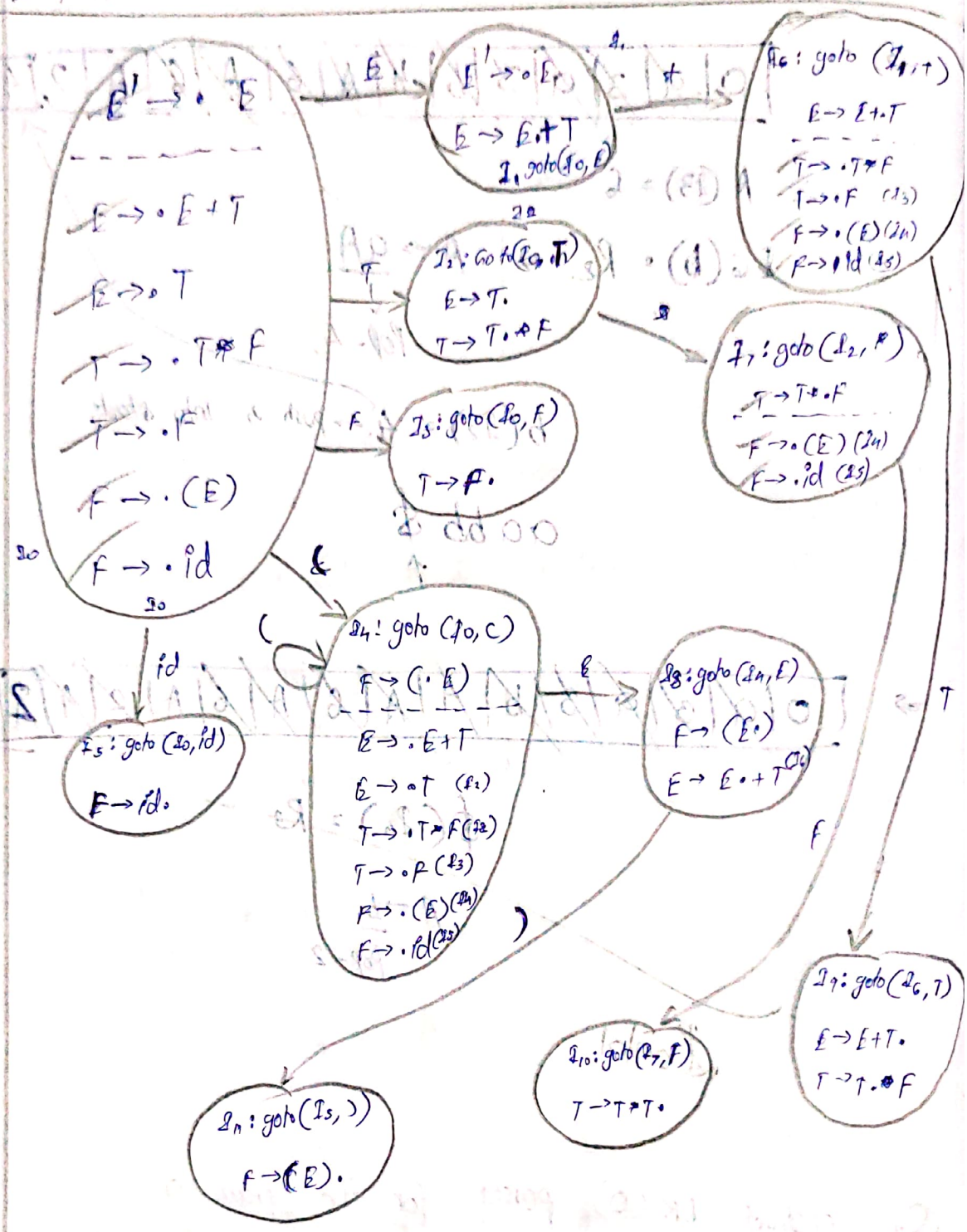
$T \rightarrow T * F/F$

$F \rightarrow (E)$

$F \rightarrow id$

Sol⇒ Construct the Argumented Grammer for the given grammer.

20/01/2025


I0: E' → . E
E → . E + T
E → . T
T → . T * F
T → . F
F → . (E)
F → . id
I1: goto(I0, E)
E' → E .
E → E . + T
I2: goto(I0, T)
E → T .
T → T . * F
I3: goto(I0, F)
T → F .
I4: goto(I0, ()
F → (. E)
E → . E + T
E → . T
T → . T * F
T → . F
F → . (E)
F → . id
I5: goto(I0, id)
F → id .
I6: goto(I1, +)
E → E + . T
T → . T * F
T → . F
F → . (E)
F → . id
I7: goto(I2, *)
T → T * . F
F → . (E)
F → . id
I8: goto(I4, E)
F → (E .)
E → E . + T
I9: goto(I6, T)
E → E + T .
T → T . * F
I10: goto(I7, F)
T → T * F .
I11: goto(I5,))
F → (E) .
(
id
E
T
F
+
*
)

20/01/2025

* Construction of Parsing Table:

SD's	Action						goto		
	id	+	*	(	)	$	E	T	F
I_0	S5			S4			1	2	3
I_1		S6				Accept			
I_2	r2	r2	S4/r2	r2	r2	r2			
I_3	r4	r4	r4	r4	r4	r4			
I_4	S5			S4			8	2	
I_5	r6	r6	r6	r6	r6	r6			
I_6	S5			S4				9	3
I_7	S5			S4					10
I_8		S6			S11				
I_9	r1	r1	S7/r1	r1	r1	r1			
I_{10}	r3	r3	r3	r3	r3	r3			
I_{11}	r5	r5	r5	r5	r5	r5			

SR Conflict

SR Conflict

* Table after solving SR Conflicts:

Follow: E : {$, +,)}

T : {*, +,), $}

F : {*, +,), $}.

20/01/2025

SD's	Actions						Goto		
	id	+	*	(	)	$	E	T	F
I_0	S5			S4			1	2	3
I_1		S6							
I_2		r_2	S7		r_2	r_2			
I_3		r_4	r_4		r_4	r_4			
I_4	S5			S4			8	2	
I_5		r_6	r_6		r_6	r_6			
I_6	S5			S4				9	3
I_7	S5			S4					10
I_8		S6			S11				
I_9		r_1	S7		r_1	r_1			
I_{10}		r_3	r_3		r_3	r_3			
I_{11}		r_6	r_5		r_5	r_5			

(i) The productions are as follows:

1) $E \rightarrow E+T$
2) $E \rightarrow T$
3) $T \rightarrow T * F$
4) $T \rightarrow F$
5) $F \rightarrow (E)$
6) $F \rightarrow id$

30/01/2025

The grammer is id + id * id.

∴ Solving it according to the stack:-

id + id * id

Initially the stack contains '0'. The stack pointer points at id, push id into stack.

id + id * id
↑

| 0 | id |

I_0, id = S5. → (5)
↓
push into stack increment the pointer.

id + id * id
↑

| 0 | id | 5 |

Now, $(I_5, +) = r6$

(6) ⇒ F → id
↓
we have one production on RHS pop 2 elements from stack. push (RHS) F. don't increment the pointer.

id + id * id
↑

| 0 | id | 5 | F |

$(I_0, F) = 3$
↓
push into stack.

20/01/2025

⇒ | 0 | id | 5 | F | 3 | id + id * id
↑

$(I_5, +) = r_4$

⇓

(T) → (F) → Pop 2 elements from stack

Push into stack. Don't increment the pointer.

⇒ | 0 | id | 5 | F | 3 | T | id + id * id
↑

* Shift Action:

→ Shift the symbol pointed by pointer to the stack.

→ Write destination state ~~no~~ on ~~TOS~~.

→ Increment the pointer.

* Reduce Action:

→ Find the production number and pop 2* (RHS_Symbols)

→ Write LHS on Top of stack

→ With previous symbol find the state numbe

→ Don't increment the pointer.

| 0 | id | 5 | F | 3 | T | id + id * id
↑

$(I_0, T) =$ (2) ← Push into stack.

⇒ | 0 | id | 5 | F | 3 | T | 2 | id + id * id
↑

20/01/2025

$(I_2, +) = r_2$

⇓

(E) → (T) → Pop two elements from stack

Push into the stack. Don't increment the pointer.

0	id	5	F	3	T	2	E

id + id * id

↑

$(I_0, E) =$ (T) → Push into stack

0	id	5	F	3	T	2	E	1

↳ for string id + id * id.

0	id	5	F	3	T	2	E	1	+	6	id	5	F	3	T	9	*	7	id	5	F

0	T	9	E	1	Accept

for string id * id + id

0	id	5	F	3	T	2	E	1	+	6	id	5	F	3	T	9	E	1	+	6	id	5

E	5	T	9	E	1	Accept

20/01/2025

- LR(1) : (CLR)

→ LR(1) = LR(0) + look Ahead symbol.

→ $A \to \alpha . B\beta, a$

and if B contains $B \to \gamma_1 / \gamma_2 / \dots / \gamma_n$

then

→ If $A \to \alpha : B\beta, a$ is the production form for.

$I \to T_1 / T_2 /$

and $B \to \gamma$, First(Bx) must be added to the closure.

→ Add a new production if 'S' is the starting symbol

$S' \to . S, \$$

(lookahead symbol)

- Example: Construct CLR parser for the grammar

$S \to AA$ (or) $S \to CC$

$A \to aA / b$ $C \to aC / d$

- Solution :-

Write the argumented grammer for the following:

$S \to AA$

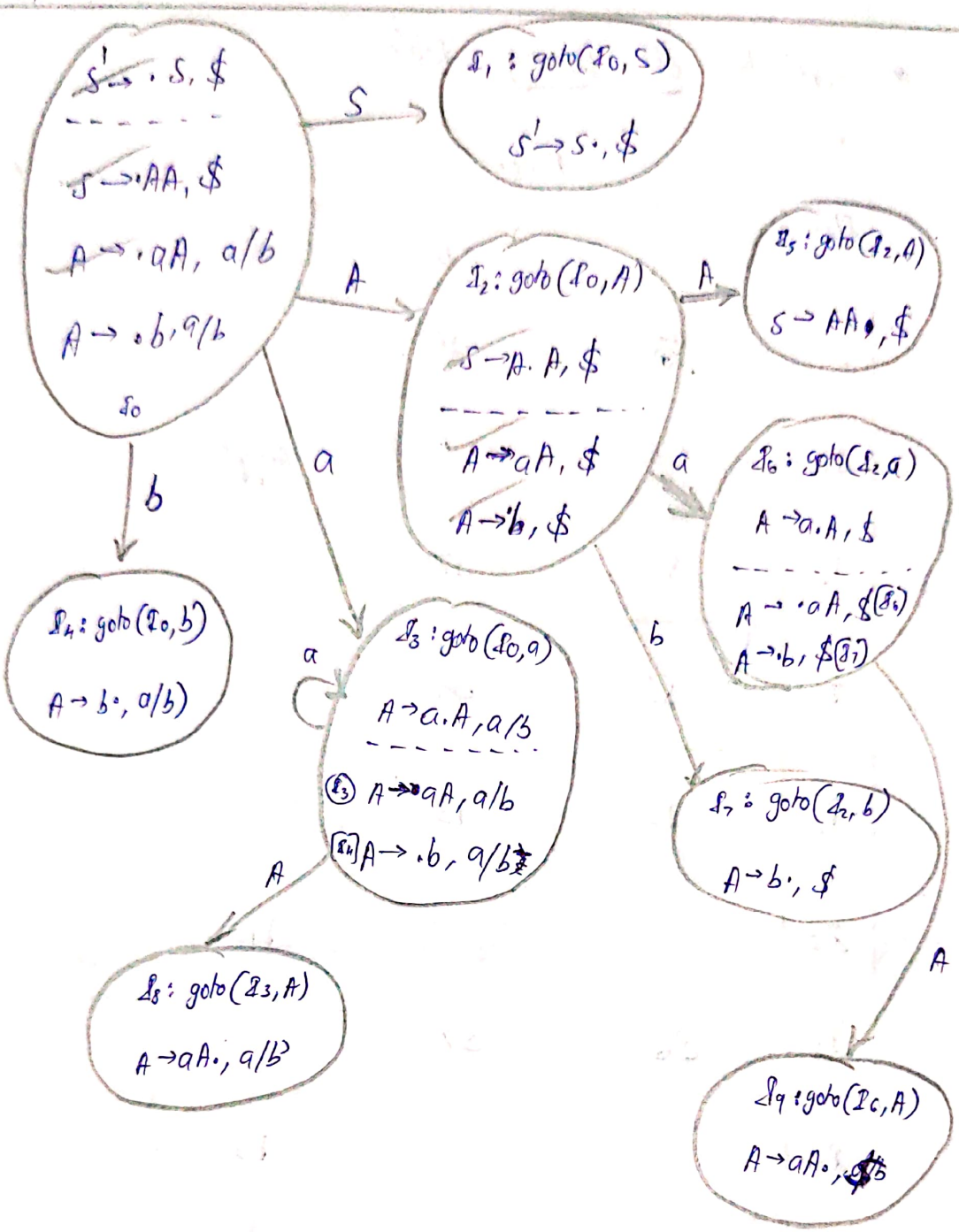
$A \to aA / b.$

→ First of 'A' : $A \to \alpha . B . \beta, a$

$\beta = \epsilon$ First($)

$a = \$$

20/01/2025



SD's	Action	goto
I0		
I1		
I2		
I3		
I4		
I5		
I6		
I7		
I8		
I9		

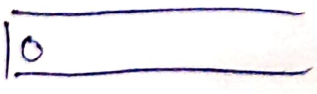
20/01/2025

* Construction of Parsing Table.

ID's	Action			go to	
	a	b	$	S	A
I_0	S3	S4		1	2
I_1			Accept		
I_2	S6	S7			5
I_3	S3	S4			8
I_4	r3	r3			
I_5			r_1		
I_6	S6	S7			9
I_7			r3		
I_8	r2	r2			
I_9			r2		

⇒ let the production be aaab.

⇒ Initial stack contains '0' pointer points on 'a'.



a a a b
↑

20/01/2025

→ Push 'a' into stack.

| 0 | a |

$(I_0, a) = S3 \Rightarrow (3)$ } push into stack. Increment pointer.

a a a b

↑

$(I_3, a) = (3)$ push into stack. increment pointer.

| 0 | a | 3 |

for string: aaab.

| 0 | a | 3 | a | 3 | a | 3 | b | 4 |

(iii) for string abab

| 0 | a | 3 | b | 4 | A | 5 | A | 2 | a | b | 7 | A | a | A | 5 | 5 | Accept |

→ LALR explanation:

$S' \rightarrow .S, \$$

$A \rightarrow \alpha . B \beta, a$

$\therefore$ First(βa) $\quad \beta = t$, $a = \$$

and First($\$$) = $\$$

$\therefore S \rightarrow . A A, \$$ (β, a)

First($A\$$) = a, b

21/01/2025

LALR:

→ In LALR we merge those canonical items which are having same LR(0) items and different look ahead symbols.

→ from the above problem I_4 & I_6 , I_2 & I_9 , I_8 & I_4 have same productions but different look ahead symbols.

→ And $I_{36} = I_3 + I_6$

I_{36}

$A \rightarrow a.A, a/b/\$$

$A \rightarrow .aA, a/b/\$$

$A \rightarrow .b, a/b/\$$

I_{47}

$A \rightarrow b., a/b/\$$

I_{89}

$A \rightarrow aA., a/b/\$$

21/01/2025

Construction of LALR Parsing Table:

SD's	Action			goto	
	a	b	$	S	A
I_0	S36	S47		1	2
I_1			Accept		
I_2	S36	S47			5
I_{36}	S36	S47			89
I_{47}	r3	r3	r3		
I_5			r1		
I_{89}	r2	r2	r2		

for the string a a b b $
↑

0	a	36	a	36	b	47	A	89	A	89	A	89	A	2	b	47

A	5	$	1	Accept

Relation between the Parsers:

CLR ≥ LALR ≥ SLR ≥ LR(0)

21/01/2025

Q2) Construct SLR parser for the grammar:

$E \to E+T / T$

$T \to TF / F$

$F \to F* / a / b$

and parse the string $w = a*b+a$

Sol ⇒ Follow ( ):

$E : \{+, \$\}$

$T : \{a, b, +, \$\}$

$F : \{a, b, *, +, \$\}$

- Argumented grammer:

$E \to .E+T$

$E \to .T$

$T \to .TF$

$T \to .F$

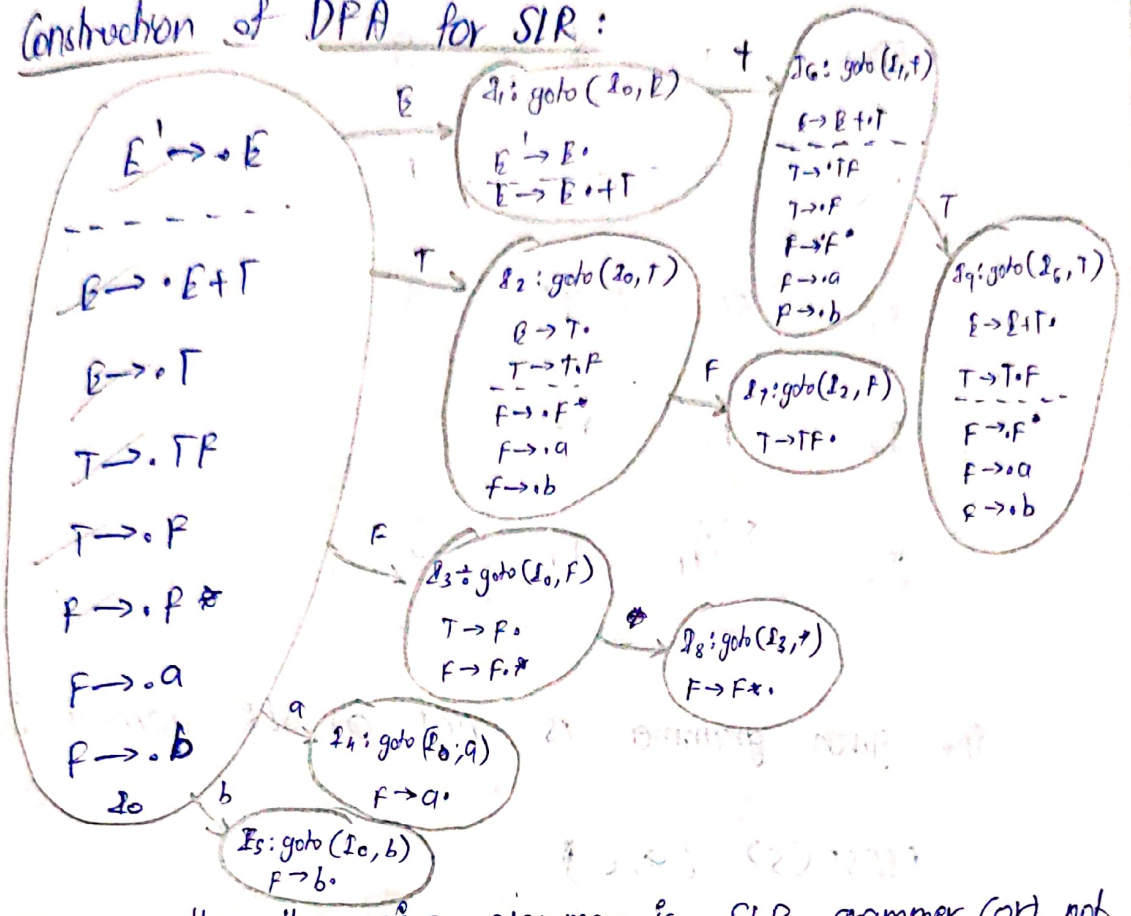
$F \to .F*$

$F \to .a$

$F \to .b$

21/01/2025

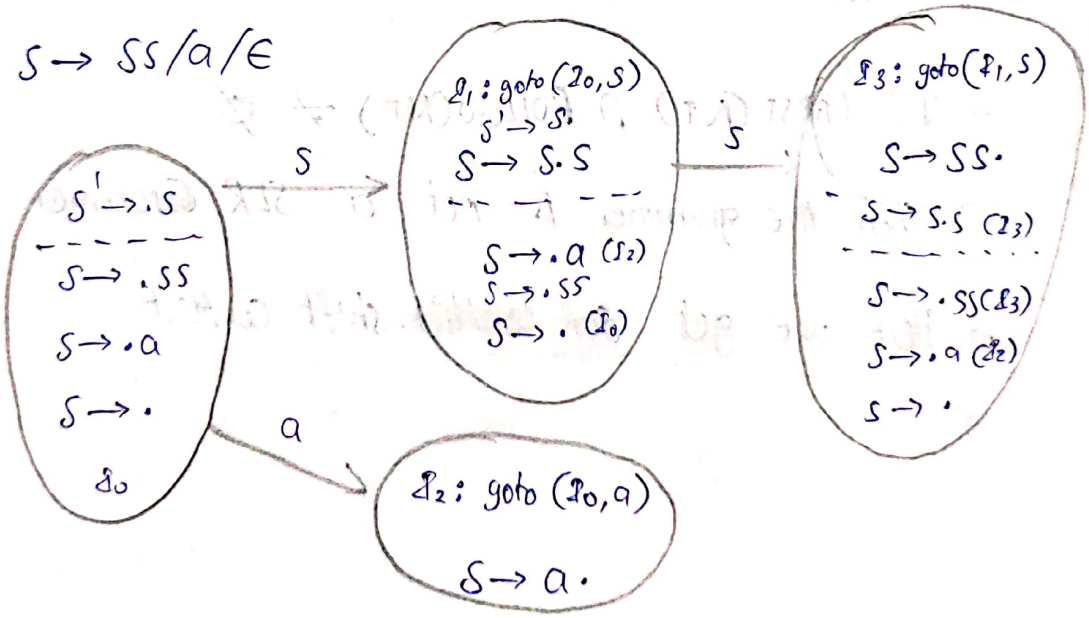
⇒ Construction of DFA for SLR:



Q3) Check weather the given grammar is SLR grammar (or) not

$S \to SS/a/\epsilon$

Sol⇒



23/01/2025

ID's	a	$	S
I_0	S2		1
I_1	S2	Accept	3
I_2	r_2	r_2	
I_3	S^2/r_1	r_1	3

→ The given grammer is not an SLR gramme. because

FIRST(S) = {a, ε}

FOLLOW(S) = {a, $}

→ If FIRST(NT) ∩ FOLLOW(NT) ≠ ∅

then the grammer is not a SLR Grammer

→ Here we get ~~SLR conflict~~. shift conflict

05/01/2025

Q20 ↓ Question in prev. page

Parsing Table

ID's	a	b	*	+	$	E	T	F
I_0	S4	S3				1	2	3
I_1				S6	Accept			
I_2	S4	S6		r2	r2			7
I_3	r4	r4	S8	r4	r4			
I_4	r6	r6	r6	r6	r6			
I_5	r7	r7	r7	r7	r7			
I_6	S4	S5						
I_7	r3	r3	S8	r3	r3			
I_8	r5	r5	r5	r5	r5			
I_9	S4	S5						

1) $E \rightarrow E+T$

2) $E \rightarrow T$

3) $T \rightarrow T*F$

4) $T \rightarrow F$

5) $F \rightarrow F*$

6) $F \rightarrow a$

7) $F \rightarrow b$

23/01/2025

=> Parsing the string a * b + a

0	a	4	F	3	4	8	F	3	T	2	2	b	5	+	T

2	E	1	6	a	4	F	3	T	9	E	T	accept

~~S → AaA~~

Q3. Construct a SLR parsing table for the grammer

S → AaAb / BbBa

A → ∈

B → ∈

Sol → Given,

S → AaAb / BbBa

A → ∈

B → ∈

Agument grammer;

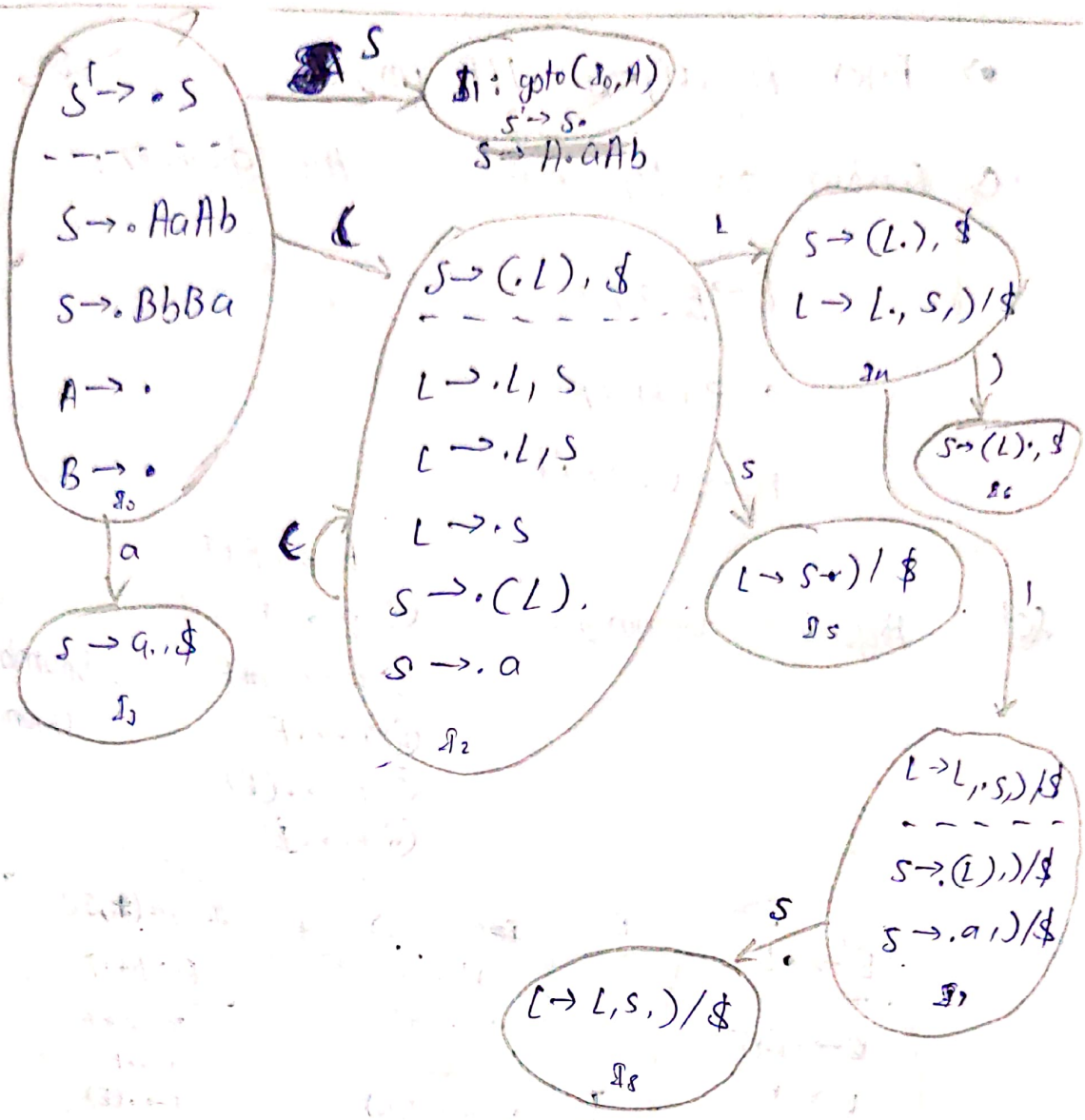
S → .AaAb

S → .BbBa

A → .

B → .

23/01/2025



* Parsing Table:

ID's	a	/	(	)	$	S	L
I0	S3		S2			1	
I1					Accept		
I2						5	4
I3	r2						
I4		S7					
I5							
I6							
I7							
I8							

28/01/2025

Error Recovery in LLR parser:

Q) Construct an LLR parser for the grammer:

$E \rightarrow E+T \mid T$

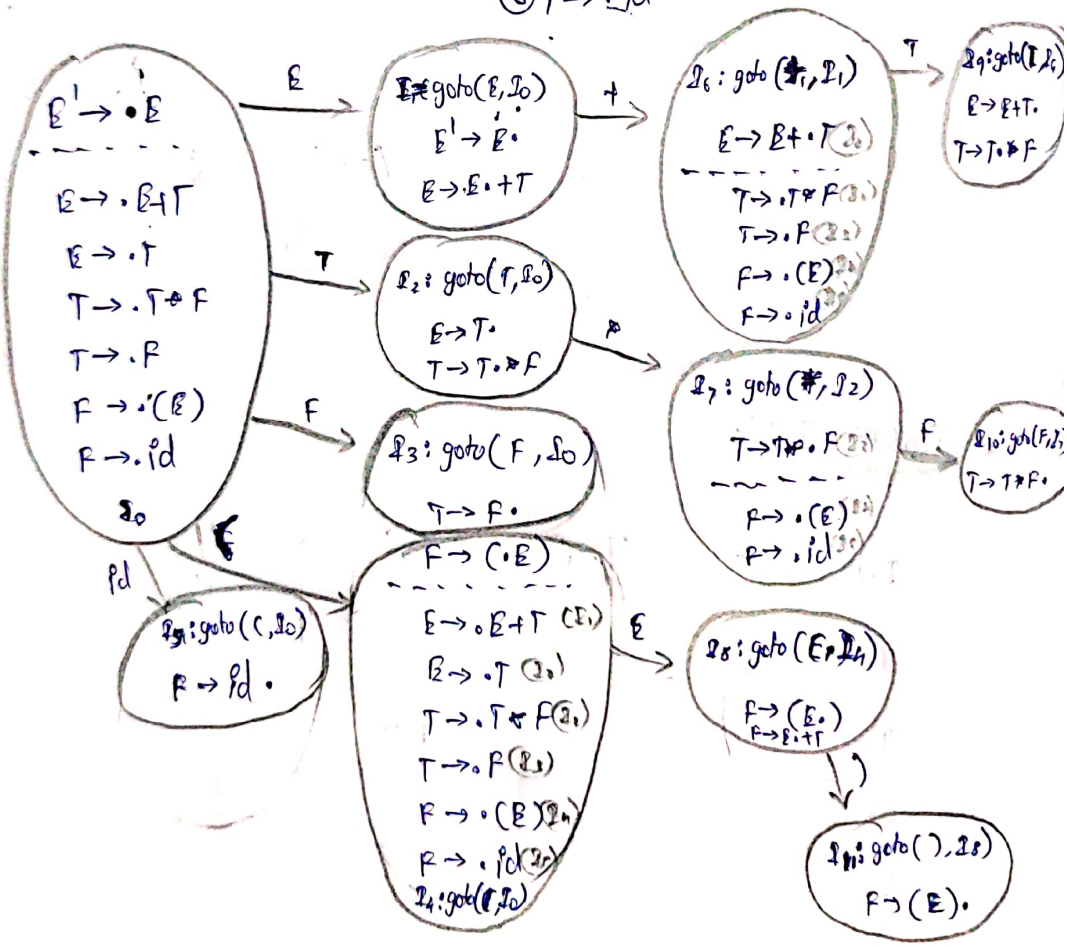
$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

Sol) Argumented Grammer:

① $E' \rightarrow .E$
② $E \rightarrow .T$
③ $T \rightarrow .T*F$
④ $T \rightarrow .F$
⑤ $F \rightarrow .(E)$
⑥ $F \rightarrow .id$

→ Unambigious Grammer



28/01/2025

Parsing Table:

STATE'S	id	+	*	(	)	$	E	T	F
I_0	S5			S4			1	2	3
I_1		S6				accept			
I_2		r2	S7		r2	r2			
I_3		r4	r4		r4	r4			
I_4	S5			S4			8	2	3
I_5		r6	r6		r6	r6			
I_6									
I_7									
I_8									
I_9									
I_{10}									
I_{11}									

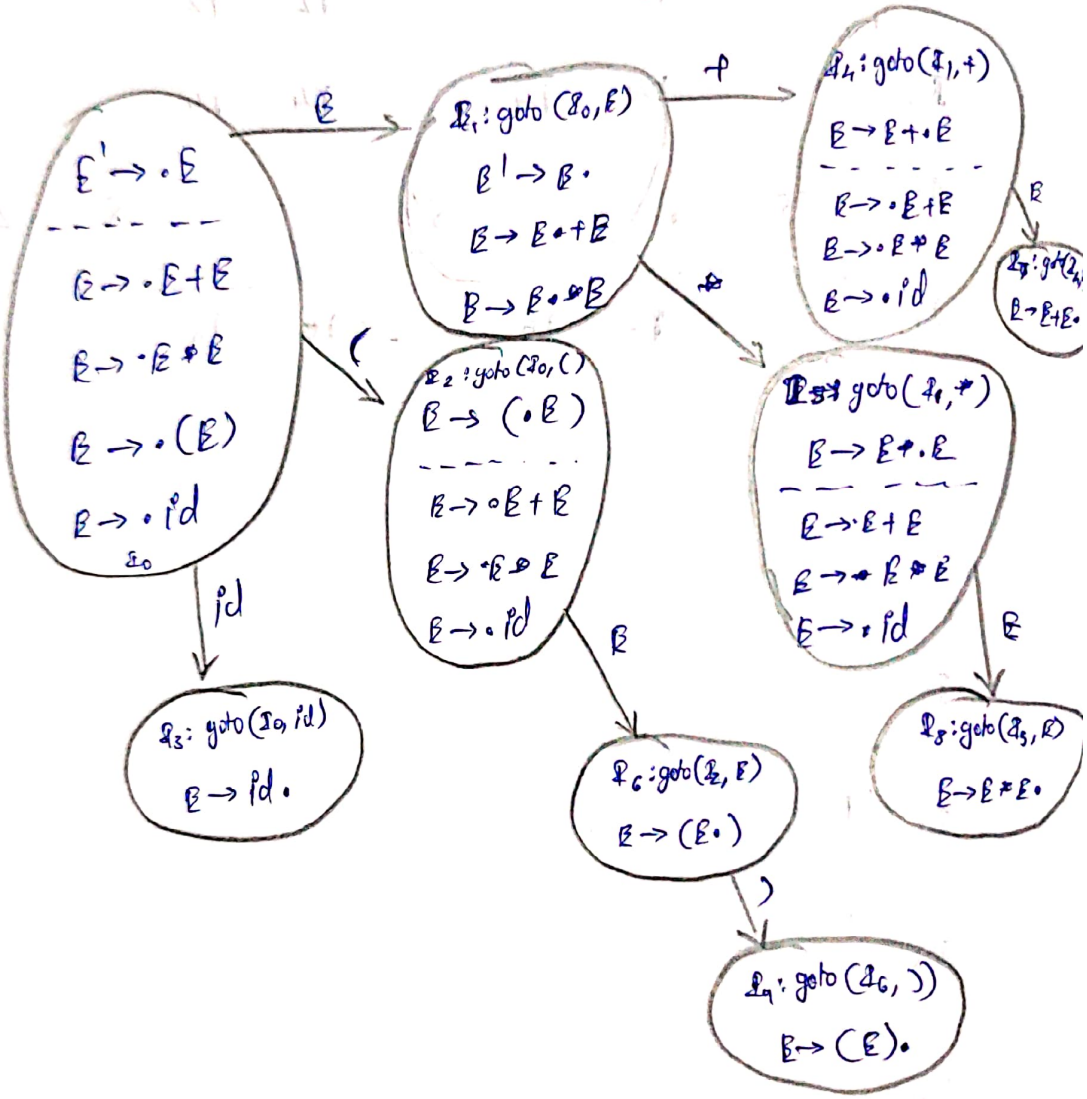
30/01/2025

Q) Construct SLR Parser for the grammer.

$E \to E + E$
$E \to E * E$
$E \to (E)$
$E \to id$

} Ambigious Grammer

Sol:



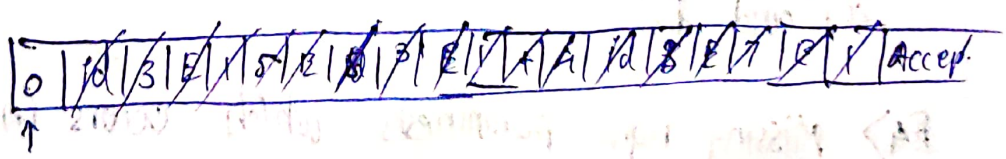
follow(E) = {+, *,), $}

30/01/2025

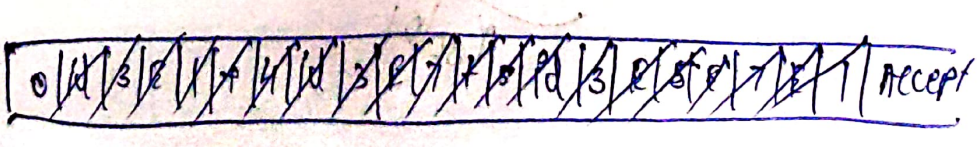
SD's	id	+	*	(	)	$	E
I_0	S_3	E_1	E_1	S_2	E_2		1
I_1		S_4	S_6			Accept	
I_2	S_3	E_1	E_1	S_2			6
I_3		r_4	r_4	r_4	r_4	r_4	
I_4	S_3	E_1	E_1	S_2			7
I_5	S_3	E_1	E_1	S_2			8
I_6		S_4	S_5				
I_7		S_4/r_1	S_5/r_1 $\downarrow S_5$		r_1	r_1	
I_8		S_4/r_2	S_5/r_2		r_2	r_2	
I_9		r_3	r_3		r_3	r_3	

i) String: id * id + id

Sol:

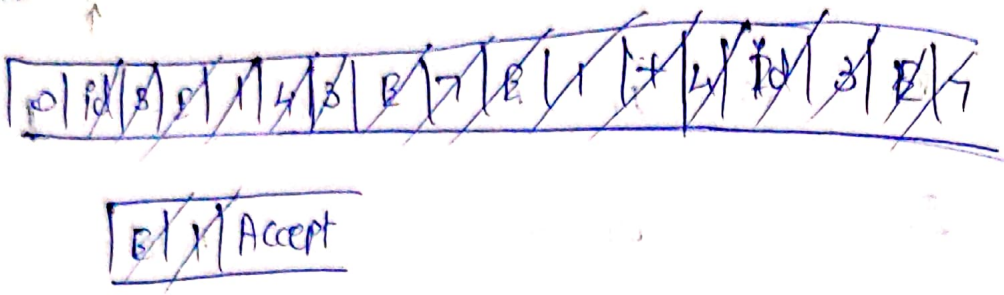


ii) id + id * id



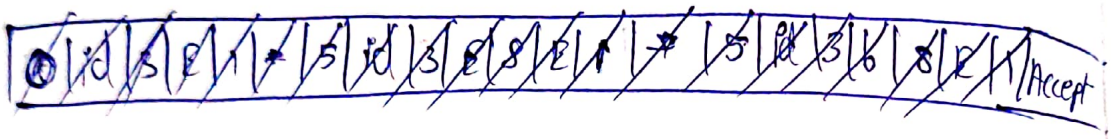
30/01/2025

(iii) id + id + id



E	$	Accept

(iv) id * id * id



→ There are four kinds of errors.

E1> This error occurs in I_0, I_2, I_4 and I_5 this error is known as missing operand

E2> This error is known as unbalanced right parathesis. It occurs in the I_0, I_1, I_2, I_4 and I_5.

E3> This is known as missing operator. it occurs in I_1 and I_6.

E4> Missing right parenthesis. which occurs in I_6 under $ column

30/01