26/7/2017

Date :
Page No

System Programming

System - Consists of hardware components and s/w components or programs.

System Programming
Invalves designing & writing computer programs that allow computer hardware to interface with programmer & user, leading to effective execution of application s/w on the computer system.

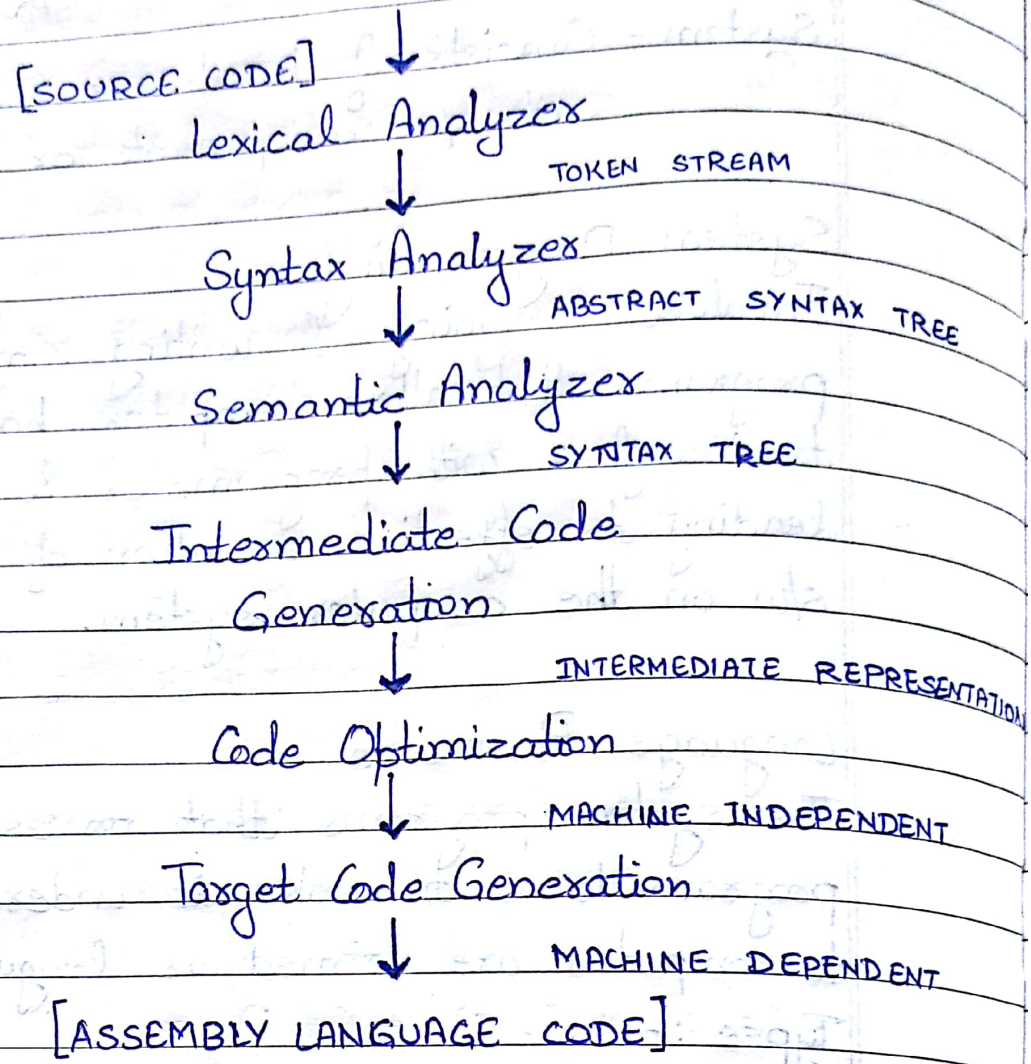
Language Processors
The system programs that process HLL program to as to make it understandable to computer are termed as language processor

Types :-

1. Assembler - Converts program written in assembly language into machine language.
2. Interpreter - Converts HLL program into machine language, by converting & executing it line by line.
3. Compiler - Also converts HLL program into machine language but converts entire program in one go.

28/7/17

Compiler Architecture



1. Lexical Analyzer

- Works as text scanner. It scans the source code as a stream of characters & converts into tokens.

2. Syntax Analysis

- Takes token produced by lexical analysis as input & generates a parse tree.
- In this phase, token arrangements are checked against the source code grammar (i.e checks if expression made by tokens is

syntactically correct)

3. Semantic Analysis
- Checks whether the parse tree constructed follows the rules of language. eg- datatypes are compatible or not.
- Also keeps track of identifiers, their types & expressions.

4. Intermediate Code Generation
- In this phase, the compiler generates a intermediate code of source code for the target machine.
- It is in b/w HLL & machine language

5. Code Optimisation
- Some unnecessary code lines are removed from the intermediate code.

6. Target Code Generation
- In this phase, the code generator takes the optimised code & maps it to target machine language.
- The code generator translates the intermediate code into sequence of instructions or machine code.

Symbol Table

- A data structure maintained throughout all the phases of compiler.
- All the identifier's names & their types are stored here.

Error Handling

- Each phase can encounter errors. After detecting an error, a phase must handle the error so that compilation can proceed

Illustrative Example

Source code :- `x += a + b * 6.5;`

1.) Lexical analyzer

$(id_1) = (id_2) + (id_3) * (6.5);$

Symbol Table

symbol name	type	address
x	int	...
a	int	...
b	int	...

2.) Syntax Analysis

⇒PRODUCTION RULE

$S \rightarrow id = E$

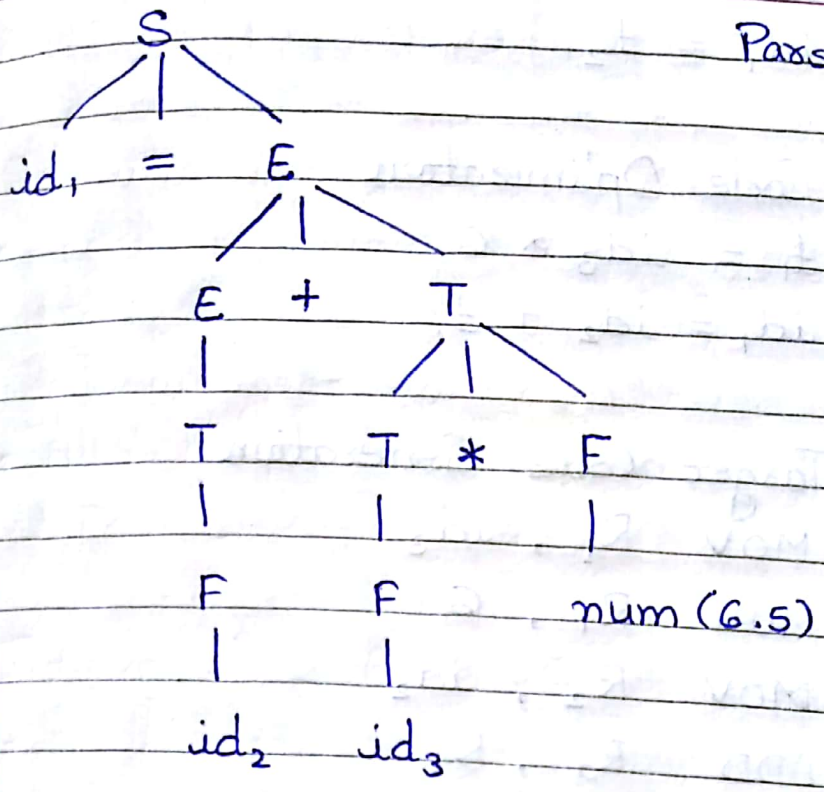
$E \rightarrow E + T / T$

$T \rightarrow T * F / F$

$F \rightarrow id / num$

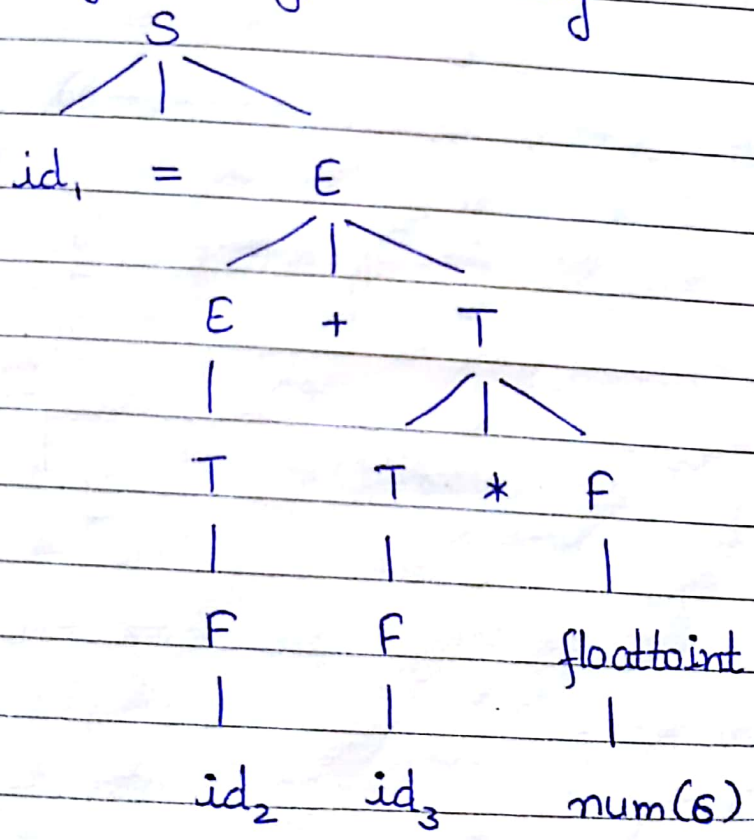
{ Starting symbol (S)
Terminating symbol (small letters)
Non-terminating }

Parse Tree



3.) Semantic Analysis

⇒ Performs type checking



4.) Intermediate Code Generation

$t_1 = id_3 * \text{num } 6$

$t_2 = id_2 + t_1$

$id_1 = t_2$

5.) Code Optimization

$t_1 = id_3 * 6$

$id_1 = id_2 + t_1$

6.) Target Code Generation

MOV R_1 , id_3

MUL R_1 , 6

MOV R_2 , id_2

ADD R_2 , R_1

MOV id_1 , R_2

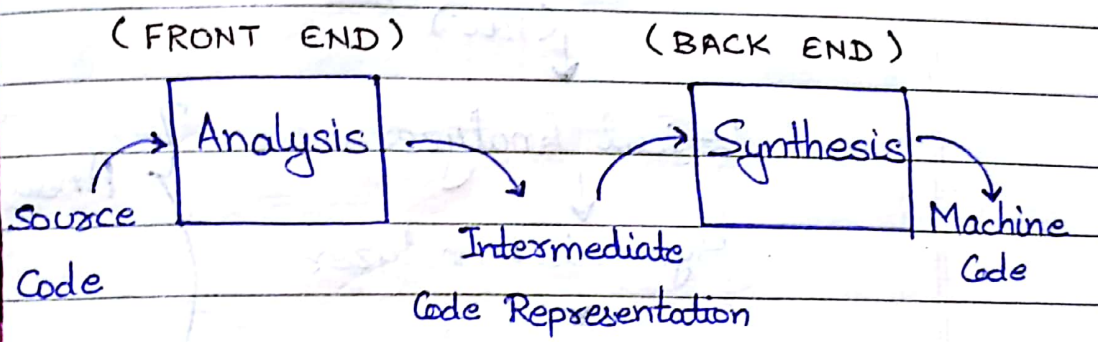
29/7/2017

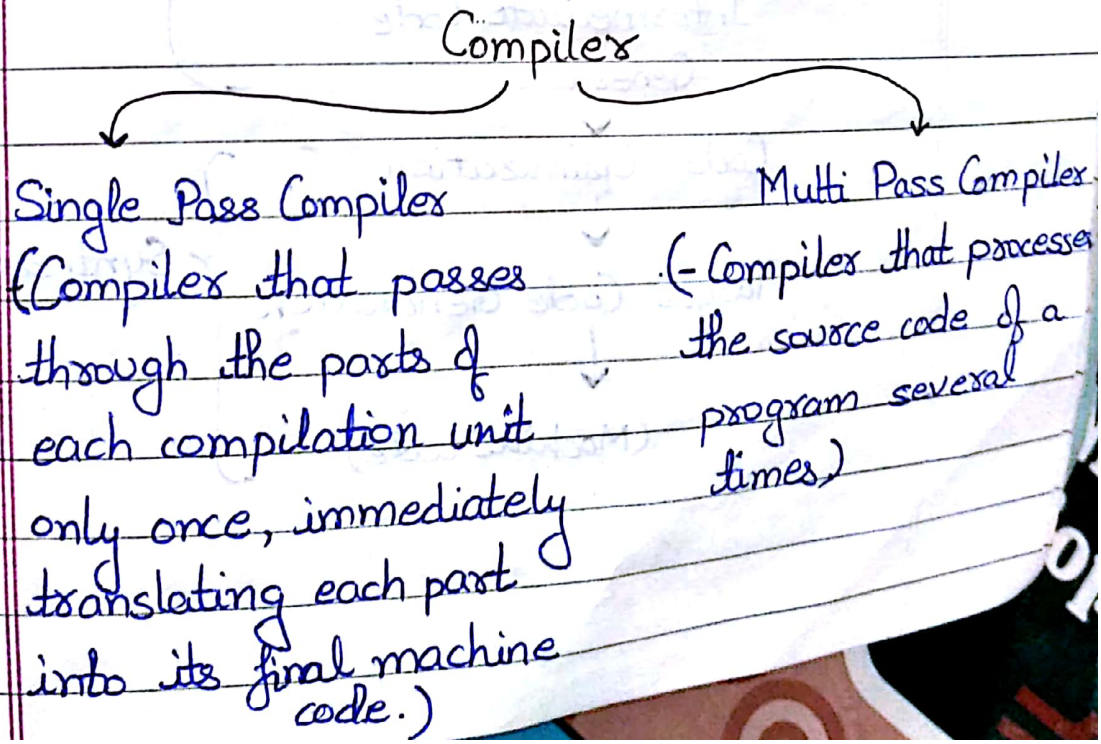
Page No.

Phases of Compilation

1. Analysis Phase
   - Known as front end of compiler
   - Reads the source code, checks for lexical grammar & syntax errors.
   - Then generates an intermediate representation of source code & a symbol table.

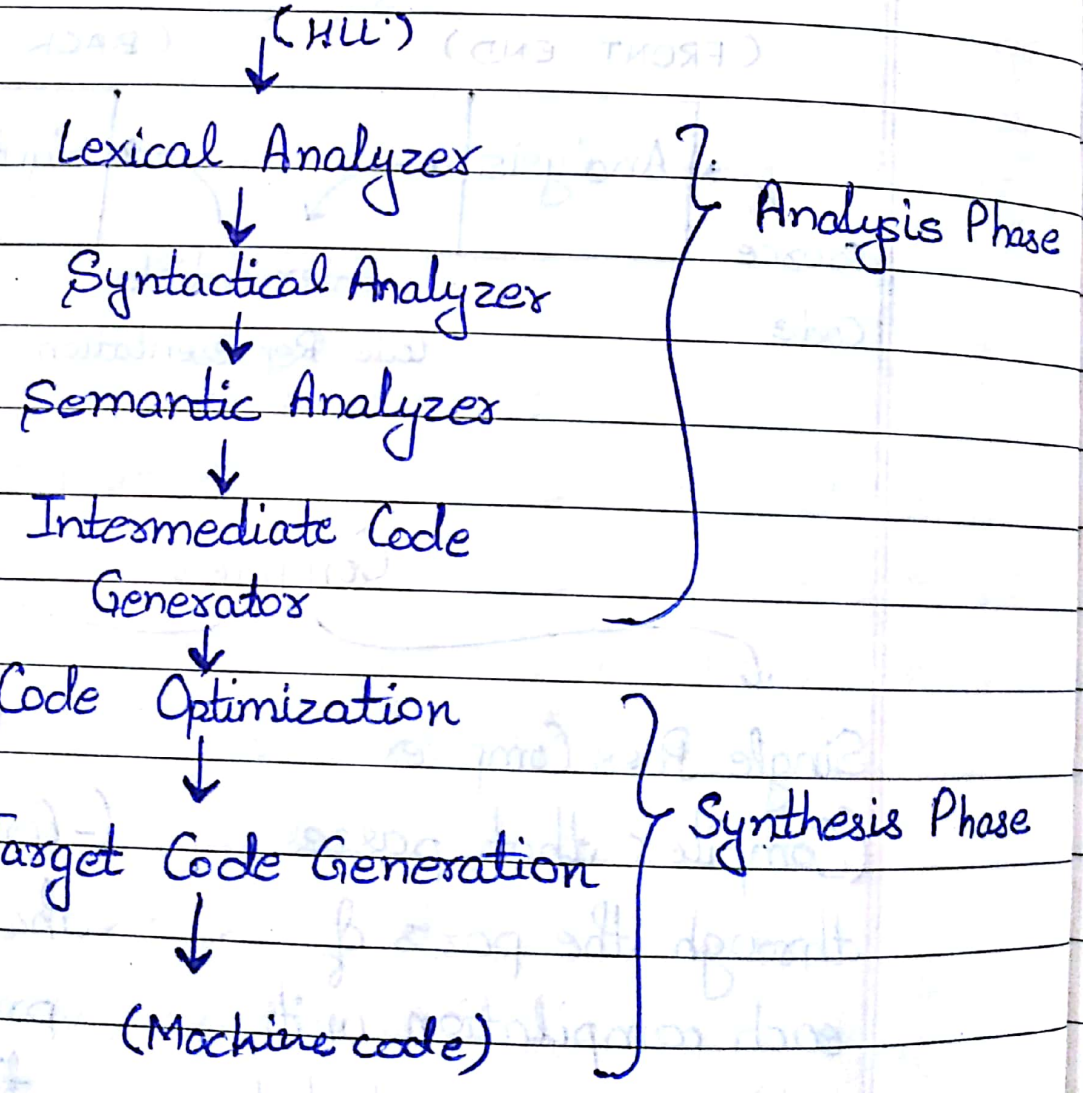
2. Synthesis Phase
   - Known as back end of compiler.
   - Generates target program with the help of intermediate source code representation.





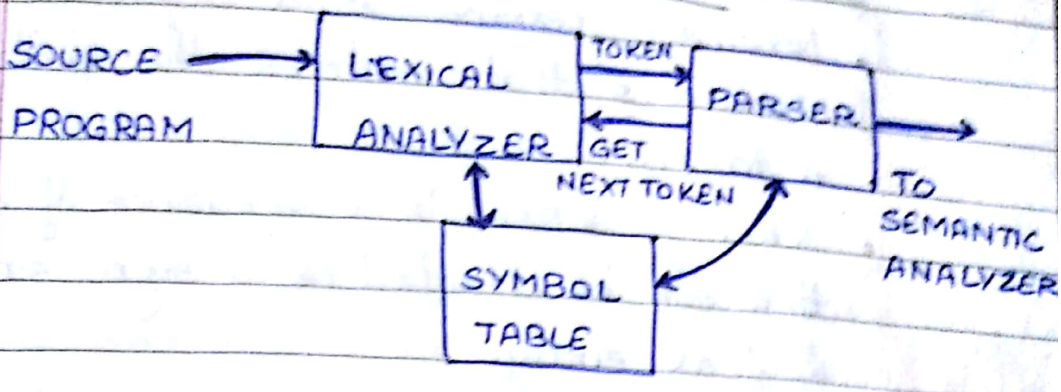
Date :
Page No.

Single Pass Compiler	Multipass Compiler
- Takes less time for compilation	- Takes more time for compilation.
- Limited scope of passes	- Wide scope of passes
- Produces less efficient programs.	- Helpful in optimizing code.
- Machine Dependent	- Machine Independent
- Requires more memory space.	- Requires less memory space.
- Used by languages like C++,	- Java compilers are multipass compilers.



Date :
Page No.

LEXICAL ANALYSER



Role of Lexical Analyzer

- It reads the input character of source code & group them into lexemes.
- Then breaks these lexemes into a series of tokens, by removing any whitespace or comments in the source code.
- If it finds a token invalid, it generates an error message, with its occurrence by specifying its line no.

Lexical analysis consists of 2 stages of processing -

- Scanning - Reading input characters & removing whitespaces & comments.
- Tokenization - Produce tokens as output.

Need of lexical Analyzer

- Simplicity of design of compiler
- Compiler efficiency is improved
- Compiler portability is enhanced.

Lexeme and Tokens

- Lexeme - Lexeme is a sequence of characters that matches the pattern for a token.
- Token - Token is a sequence of characters that can be treated as a ~~sign~~ single logical entity.

Common tokens in a programming language are :-

- Keywords
- constants
- Identifiers
- Numbers
- Puntuation symbols

Specification & Recognisation of Tokens

There are some predefined rules for a lexeme to be a token.

- Valid tokens - keywords, punctuation marks, semicolon, numbers etc.
- Non tokens - Comments, blanks, tabs, macros etc.

eg- int a, b ;
has 4 tokens.

printf("hello");
has 5 tokens

Date :
Page No

Practice Question

1. Count no. of tokens

a)
```
int max(i,j)
{  // max of i & j
    return i>j ? i : j;
}
```
18 tokens

b)
```
int main()
{  int a = 10, b = 20;
    printf("sum is : %d", a+b);
    return 0;
}
```
27 Tokens

c)
```
printf("i = %d, &i = %x", i, &i);
```
10 Tokens

4/8/17

Date :
Page No.

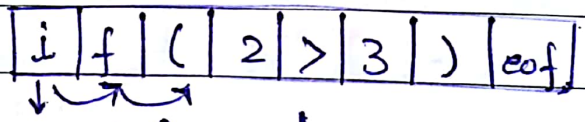
1) Lexical Analyzer
   - → Scanner
   - → Lexcial Analysis

TOKEN	INFORMAL DESCRIPTION	EXAMPLE
IF	character i, f	if
ELSE	character e, l, s, e	else
comparison	< or > or <= or =>	>, =>, =<
id	letter followed by letter / digit	variable name
number	constant	23
literal	inside " "	"total = %d"

Input Buffering

eg-

if (2>3)



lexeme begin — forward Pointer

Terms related to strings

- Prefix
- Suffix
- Substring
- Proper prefix
- Proper suffix
- Proper substring
- Subsequence

Date :
Page No.

1) Prefix - string obtained by removing zero or more trailing symbols from a string.
eg- pre is prefix in prepaid

2) Suffix - string obtained by removing zero or more leading symbols from a string.

3) Substring - string obtained by removing prefix & suffix

4) Proper prefix or proper suffix - string obtained by removing leading or trailing symbols that are not null.

5) Subsequence - string obtained by deleting zero or more arbitrary symbols.
eg- stn is a subsequence of string.

5/8/2017

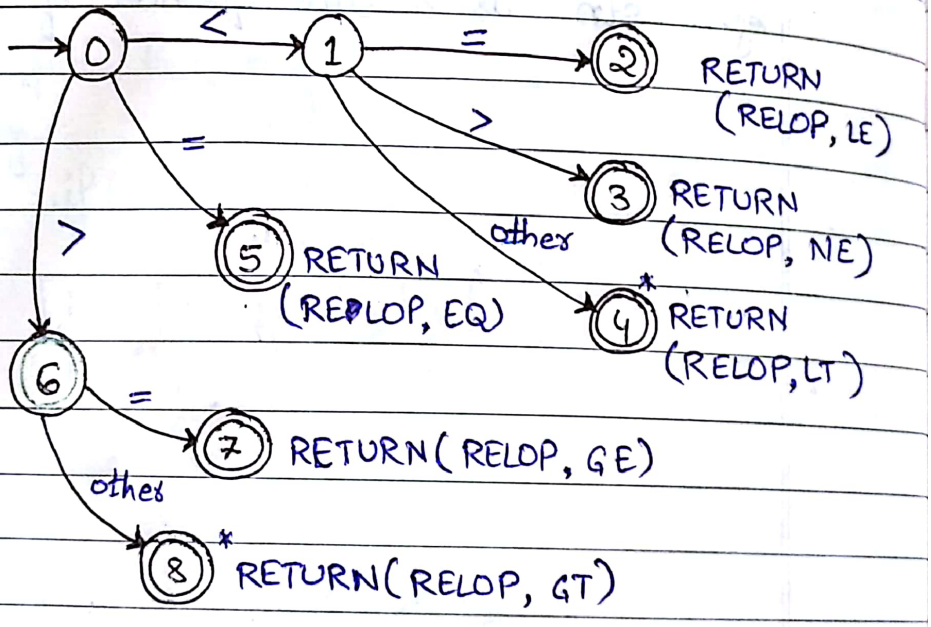
Date :
Page No.

Transition Diagram.

Relational Operation

Lexeme	Attribute	Token Name
>	(GE) greater than	RELOP
<	(LT) less than	"
>=	(GE) greater than / equal to	"
<=	(LE) less than / equal to	"
<>	(NE) not equal to	"
=	(EQ) equal to	"

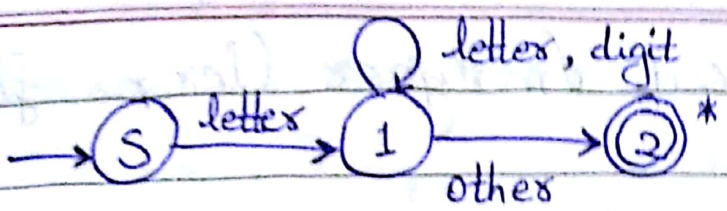
FA Transition Diagram.



* → Represents we need to go a step back

Transition Diagram for Reserve word & Identifier

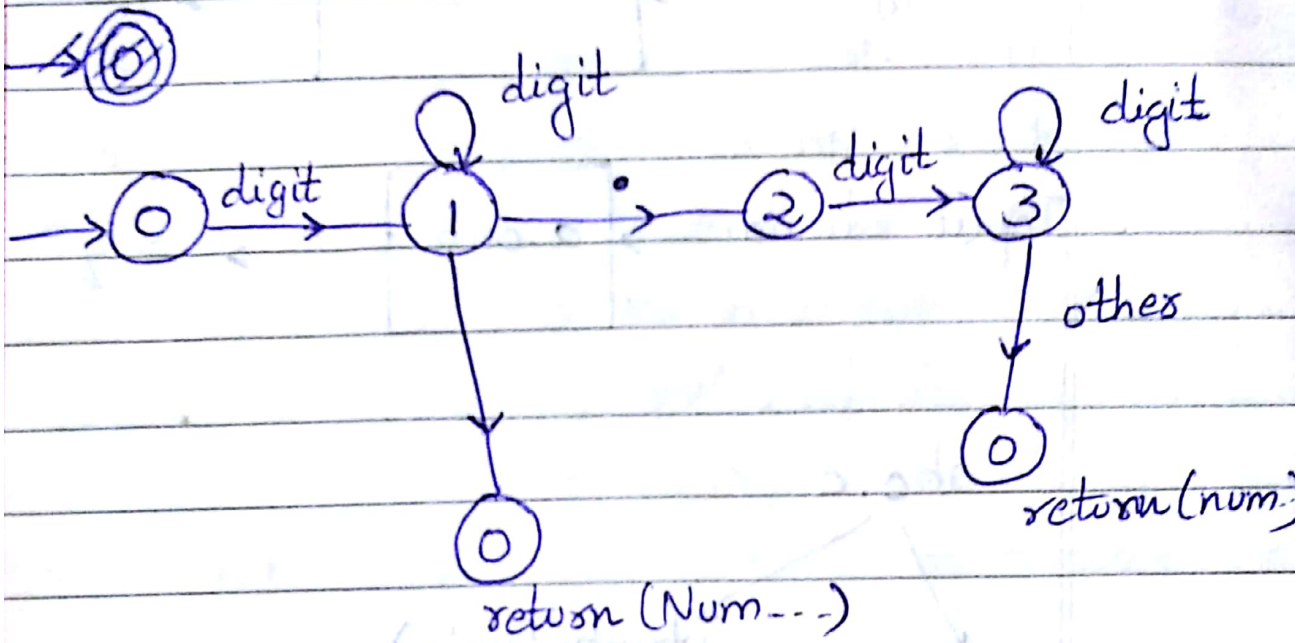
id → (letter) (letter / digit)*



Number

num → digit(. / digit.)

digit → digit +

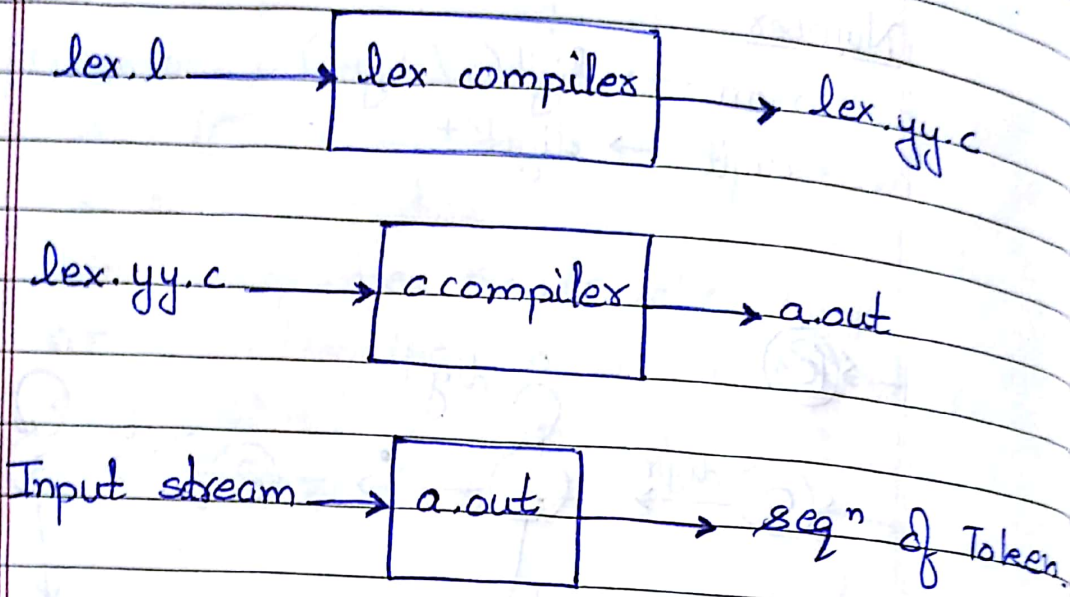


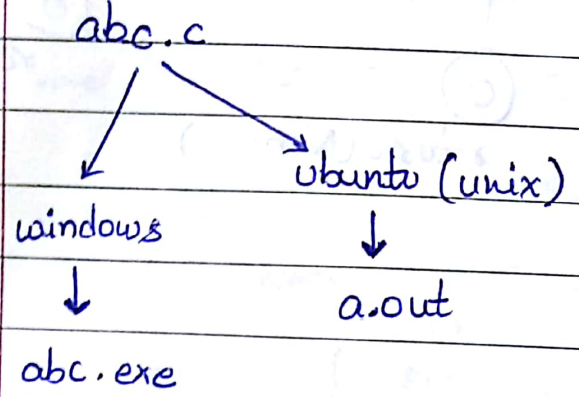
3/8/17

Date :
Page No.

Lexical Analyser (lex or flex)

- Lex is a compiler





What is lex ?

- Lex is a computer program that generates lexical analyzers.
- It's main job is to break up an input stream into more usable elements.

Date :
Page No

Regular Expressions (REGEX)

SYMBOL	DESCRIPTION
.	any character
^	begining of line
$	end of line
[abc]	any of a, b or c
[^abc]	neither a, b nor c
r*	zero or more string matching r
r+	one or more string matching r
r?	zero or more r
$r_1 r_2$	r_1 followed by r_2
$r_1 \mid r_2$	r_1 or r_2
$r\{m,n\}$	b/w m & n occurrences of r
" s "	matching string s

* In unix, "grep" command is used for REGEX searching.

ex - $ grep a.pple fruit

a.pple → for matching
fruit → filename

⇒ a.pple will look for such matchings where starting letter is 'a' & last 4 are 'pple' & second letter can be any characters.

Date :
Page No.

Example- A filename "fruits", has contents as :-

Apple
apple
aPPLE
mango
aPple
abple
Apple Red
An Apple

Write the correct answers for the following -

a) a.ple - apple, aPple, abple
b) [aA]pple - Apple, apple
c) apple/Apple - Apple, apple
d) ^Apple - Apple, Apple Red
e) Apple $ - Apple, An Apple

11/8/17

Date :
Page No.

2) Parsing

Parser

Top Down — Bottom Up

Grammar - A grammar can be defined as
$G = (V, T, S, P)$ → starting symbol set
→ Production rule
↓ set of terminal symbols
↓ set of non terminal symbol

S → starting symbol
Capital letter → non-terminal symbol (V)
Small letter → ~~set of~~ Terminal symbol (T)

eg - $S = \{S\}$
$T = \{a, b, c, d \dots\}$
$V = \{S, A, B \dots\}$
$P = \{ S \rightarrow cAd$
$A \rightarrow ab/d\}$

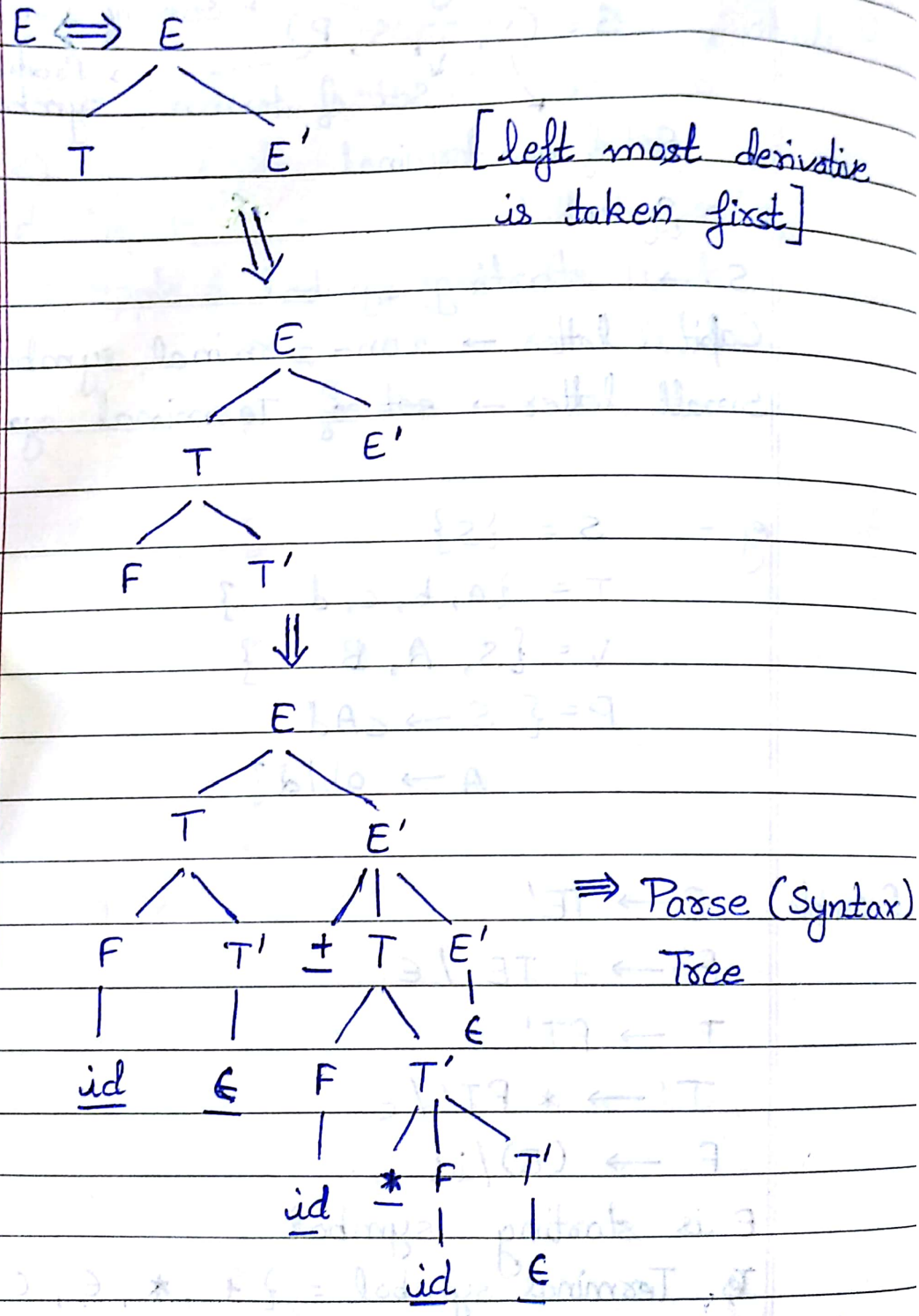
Example - $E \rightarrow TE'$
$E' \rightarrow +TE' / \epsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' / \epsilon$
$F \rightarrow (E) / id$.
E is starting symbol
Terminal symbol = $\{+, *, \epsilon, c,)\}$

Non-terminal symbols = { E, T, E', T', F }
for given grammar check id + id * id is valid or not.

a) Top-Down Approach

Grammar → Syntax

⇒ Root → Leaf



⇒ Parse (Syntax) Tree

Date :
Page No.

Thus, given syntax satisfied by the grammar

id ∈ + id * id ∈
= id + id + id

b) Bottom - Up Approach

Syntax ⟶ Grammar
Leaf ⟶ Root

id * id
⇓

Example- Grammar

$E \rightarrow E + T / T$

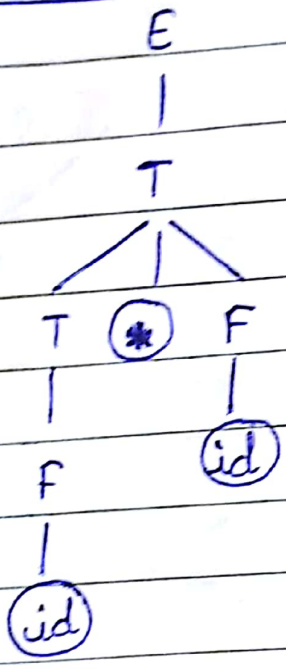
$T \rightarrow T * F / F$

$F \rightarrow (E) / id$

Syntax

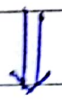
id * id

a) Top-Down Parsing



b) Bottom-Up Parsing

id * id

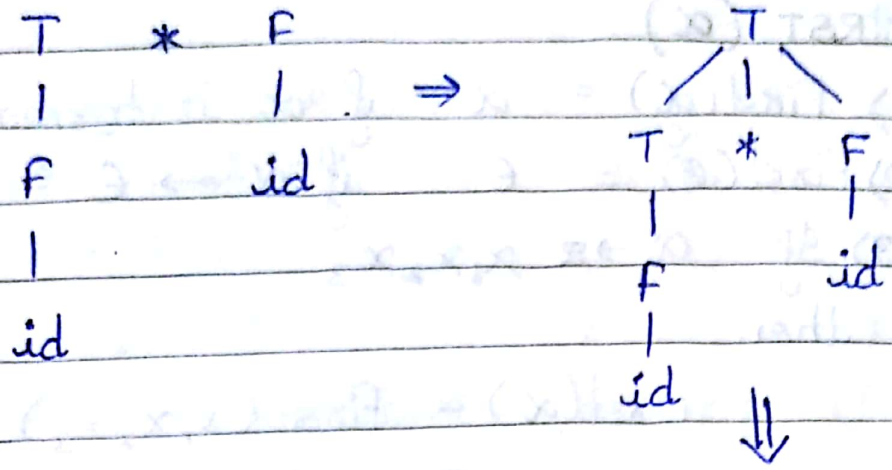


F id * id ⇒ T * id

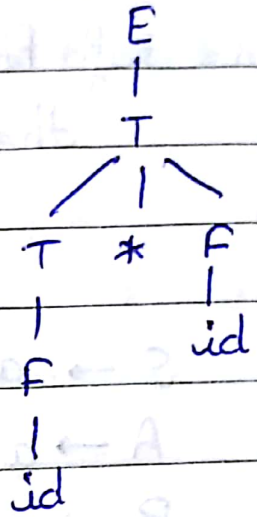
F

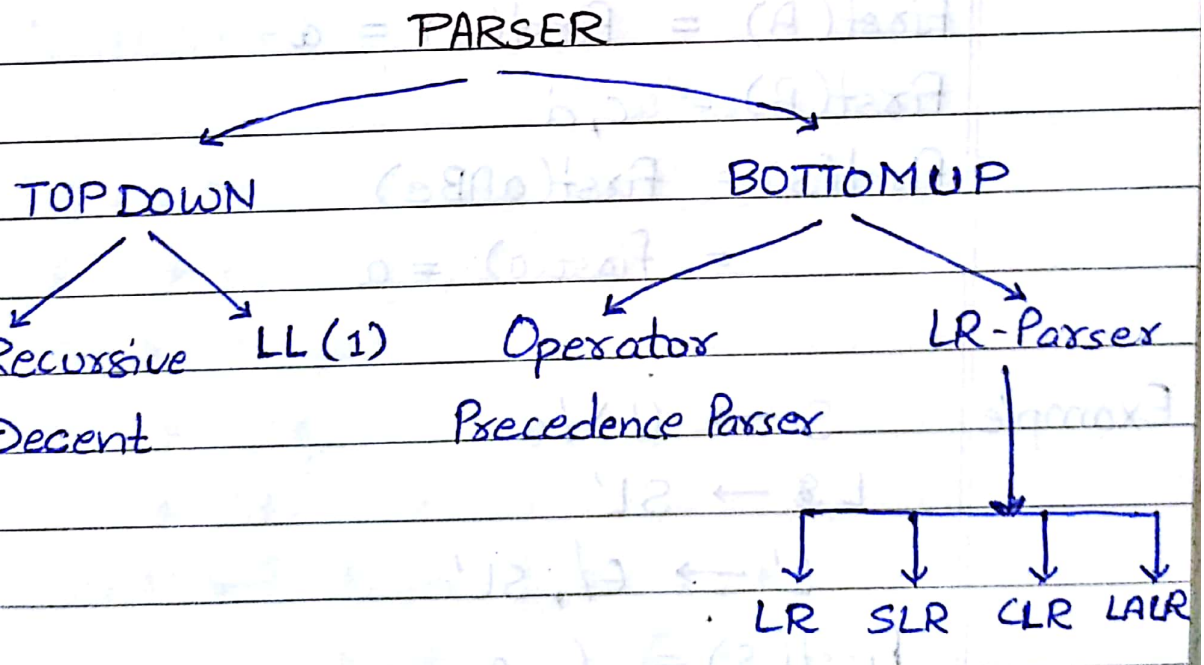
id

id



We reached the root (starting symbol).





First* First & follow

Date :
Page No.

FIRST (α)

1) First(α) = α if α is terminal symbol
2) First(α) = ϵ if $\alpha \rightarrow \epsilon$
3) If $\alpha \rightarrow x_1 x_2 x_3$
then
First(α) = first($x_1 x_2 x_3$)
= first(x_1)
if First(x_1) contain ϵ
then First(α) = (First(x_1) - ϵ) $\cup$ First($x_2 x_3$)

Example- $S \rightarrow aABe$
$A \rightarrow a$
$B \rightarrow c/d$

First(A) = First(a) = a
First(B) = c, d
First(S) = First(aABe)
= first(a) = a

Example $S \rightarrow (L)/a$
$L \rightarrow SL'$
$L' \rightarrow \epsilon/,SL'$
First(S) = (, a
First(L) = First(SL')
= First(S)
= (, a
First(L') =), ϵ, ,

18/8/17

Date :
Page No

Example: $A \rightarrow aA \mid bB \mid c \mid d$

Find First(A)

First(A) = c

First(A) = d

First(A) = First(aA) = First(a)

= a

First(bBA) = First(bB) = First(b) = b

$\Rightarrow$ First(A) = {a, b, c, d}

Example: $S \rightarrow aABe$

$B \rightarrow c \mid d$

$A \rightarrow a$

First(S) = {a}

First(B) = {c, d}

First(A) = {a}

Example: $E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow id / (E)$

First(E') = ϵ, +

First(T') = ϵ, *

First(F) = id, (

First(T) = First(F) = id, (

First(E) = First(T) = id, (

Date :
Page No.

Example- $S \rightarrow aBDh$

$B \rightarrow cC$

$C \rightarrow bC / \epsilon$

$E \rightarrow g / \epsilon$

$F \rightarrow f / \epsilon$

$D \rightarrow EF$

$First(S) = \{a\}$

$First(B) = \{c\}$

$First(C) = \{b\}$

$First(E) = \{g, \epsilon\}$

$First(F) = \{f, \epsilon\}$

$First(D) = First(EF)$

$= First(E) = \{g, \epsilon\}$ {∵ First(E) contains ϵ}

$= (\{g, \epsilon\} - \epsilon) \cup First(F)$

$= \{g, f, \epsilon\}$

Example $S \rightarrow (L) / a$

$L \rightarrow SL'$

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

$First(S) = (, a$

$First(L) = (, a$

$First(L') = \epsilon, ,$

Date :
Page No.

Follow (A)

1) If A is starting symbol, then
   Follow(A) = $
2) If $x \rightarrow \alpha AB$, then
   Follow(A) = First(B)
3) If ($x \rightarrow \alpha A$) or ($x \rightarrow \alpha AB$ and $B \rightarrow \epsilon$)
   then Follow(A) = Follow(α)

Example- $S \rightarrow (L)/a$
$L \rightarrow SL'$
$L' \rightarrow \epsilon/,SL$

Follow(S) = {$, , ,)}
Follow(L) = First()) {$\because$) $\in T$}
=)
Follow(L') = Follow(L) (Rule 3)
=)

Follow(S) = Follow First(L')
= , , ϵ
= , , Follow(L)
= , ,)
follow(S) = $ ($\because$ S is starting pt.)

Example $S \rightarrow aABe$
$A \rightarrow a$
$B \rightarrow c/d$
First(S) = a First(B) = c, d
First(A) = a

Date :
Page No.

Follow (S) = $
Follow (A) = First(B) = c, d
Follow (B) = First(e) = e

19/8/17

Example: Find Follow(α)

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

First(E) = id, (
First(T) = id, (
First(E') = ∈, +
First(T') = ∈, *
First(F) = id, (

Follow (E) = $,)
Follow (T) = +, $,)
Follow (E') = Follow(E) = $,)
Follow (T') = First(E') = ∈ Follow + , $,)
Follow (F) = First(T')
= *, ∈
= *, Follow(T)
= *, +, $,)

Left Recursion

If $A \rightarrow A\alpha_1 | A\alpha_2 | A\alpha_3 | \cdots \beta_1 | \beta_2 | \beta_3 \cdots$ is in production rule, then problems are created. So this has to be removed.

eg- $A \rightarrow Aa$

$\downarrow$

Aa

$\downarrow$

$Aa \cdots$

Thus, new production rule is defined as

$A \rightarrow \beta_1 A' | \beta_2 A' | \cdots$

$A' \rightarrow \alpha_1 A' | \alpha_2 A' | \cdots | \epsilon$

Example - $E \rightarrow E+T / T$

Here $A \Rightarrow E$

$\alpha_1 \Rightarrow +T$

$\beta_1 \Rightarrow T$

Thus,

$E \rightarrow E+T / T$

$\Downarrow$

$E \rightarrow TE'$

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

Example - $F \rightarrow F * T / T$

Here $A \Rightarrow F$ $\quad \alpha_1 = *T$ $\quad \beta_1 = T$

$\Rightarrow$ $F \rightarrow TF'$

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

Remark

For direct conversion,

$A \to A\alpha / \beta \Rightarrow \begin{cases} A \to \beta A' \\ A' \to \alpha A' / \epsilon \end{cases}$

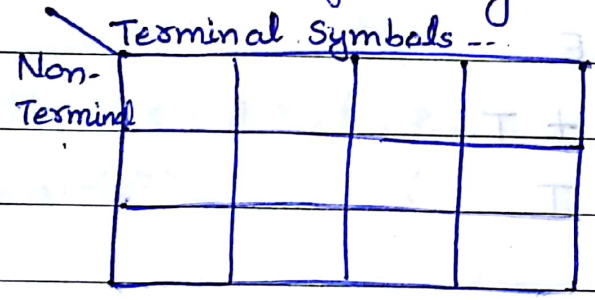
LL(1) Parsers (Top down)

Left to Right Scanning Parsing

Use Left most derivative.

Procedure :

1. Make grammar free from left recursion
2. Find First & Follow
3. Make the following table as -



Table

1. Add $A \to \alpha$

   under $M[A, b]$ where $b \in$ First(A)

2. If First(A) contain ~~Non-terminal / Termi~~ ϵ, then add

   $A \to \alpha$ under $M[A, c]$

   $A \to \epsilon$ where $c \in$ Follow(A)

Example-Consider the following set of rules -

$S \rightarrow (L) / a$

$L \rightarrow S / L'$

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

	First	Follow
S	(, a	$, , ,)
L	(, a	)
L'	, , ε	)

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

Since, one cell contains only one entry then it has LL(1) Parsing.

23/8/17

Date :
Page No.

LL(1) Parser

Example- $E \rightarrow E+T/T$

$T \rightarrow T*F/F$

$F \rightarrow id/(E)$ Is LL(1) parser?

Remove left recursion $S \rightarrow S\alpha_1 | S\alpha_2 | \beta_1 | \beta_2 \ldots$

$\Downarrow$

$S \rightarrow \beta_1 S' | \beta_2 S'$

$S' \rightarrow \alpha_1 S' | \alpha_2 S' | \epsilon$

STEP 1- Using above formula to remove left recursion

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow id / (E)$

STEP 2-

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

STEP 3-

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

Example- $S \rightarrow iEtSS' / a$
$S' \rightarrow eS / \epsilon$
$E \rightarrow b$

Is LL(1) Parser?

	First	Follow
S	i, a	$, e, t
S'	e, ϵ	$, e
E	b	t

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

* This is not a LL(1) parser

NOTE:

(1) If $A \rightarrow \alpha_1 / \alpha_2 / \alpha_3$ then it is said to be LL(1) iff -
$First(\alpha_1) \cap First(\alpha_2) = \phi$
$First(\alpha_2) \cap First(\alpha_3) = \phi$
$First(\alpha_3) \cap First(\alpha_1) = \phi$

(2) If $A \rightarrow \alpha_1 / \alpha_2 / \epsilon$ then it is said to be in LL(1) iff
$First(\alpha_1) \cap Follow(A) = \phi$
$Follow(A) \cap First(\alpha_2) = \phi$
$first(\alpha_1) \cap Follow(A) = \phi$

Date :
Page No

Example- $S \rightarrow E/a$

$E \rightarrow a$

Check LL(1) parser or not ?

First(E) $\neq \cap$ First(a) $= \{a\} \cap \{a\}$

$\neq \phi$

$\Rightarrow$ Thus, given grammar isn't in LL(1) parser.

Example- $S \rightarrow aSA/\epsilon$

$A \rightarrow c/\epsilon$

First(aSA) $\cap$ Follow(S) $= \{a\} \cap \{\$, c\}$

$= \neq \phi$

First(c) $\cap$ Follow(A) $= \{\$, c\} \cap \{c\} = \{c\}$

$\neq \phi$

$\Rightarrow$ Thus, this is not in LL(1) parser

Example- $S \rightarrow aABb$

$A \rightarrow a/\epsilon$

$B \rightarrow d/\epsilon$

	First	Follow
A	a, ϵ	d, b
B	d, ϵ	b

First(a) $\cap$ Follow(A) $= \{a\} \cap \{d, b\} = \phi$

First(d) $\cap$ Follow(B) $= \{d\} \cap \{b\} = \phi$

$\Rightarrow$ Thus, the given grammar is in LL(1) parser.

Date :
Page No.

30/8/17

LR(K) Parser

- Bottom-up parser
- 'L' → left to right scanning
  'R' → constructing rightmost derivative in reverse order
  'k' → #lookahead symbol

- Let G be a grammar with start symbol S. then augmented grammar G' with a new start symbol S' s.t. S' → S.
- An augmented Grammar is used to announce the acceptance of input.

Example: G is given by

S → A

A → aA / b

Find G' (augmented grammar)

Sol- G' is given as

S' → S

S → A

A → aA / b

★ **LR item**

Contains production rule.

Closure of any item

Closure (I) = is defined /obtained by -

1. Add item I to closure (I)
2. If $\alpha \rightarrow \beta . A \alpha$ is Item I

and $A \rightarrow BC$

then add

$A \rightarrow BC$ to closure (I)

3. Repeat 1 & 2 for newly added item.

Example

$S \rightarrow AA$

$A \rightarrow aA / b$

~~Closure~~ ($S \rightarrow S$ Find closure ($S' \rightarrow .S$)

1. Make augmented grammar

$S' \rightarrow S$

$S \rightarrow AA$

$A \rightarrow aA / b$

2. Closure ($S' \rightarrow .S$) $\Rightarrow$

$S' \rightarrow .S$

$S \rightarrow .AA$

$A \rightarrow .aA / .b$

Item I_0

Example -

$E' \rightarrow E$

$E \rightarrow E+T / T$

$T \rightarrow T*F / F$

$F \rightarrow (E) / id$

~~Close~~ Closure ($E' \rightarrow .E$)

$= E' \rightarrow .E$

$E \rightarrow .E+T / .T$

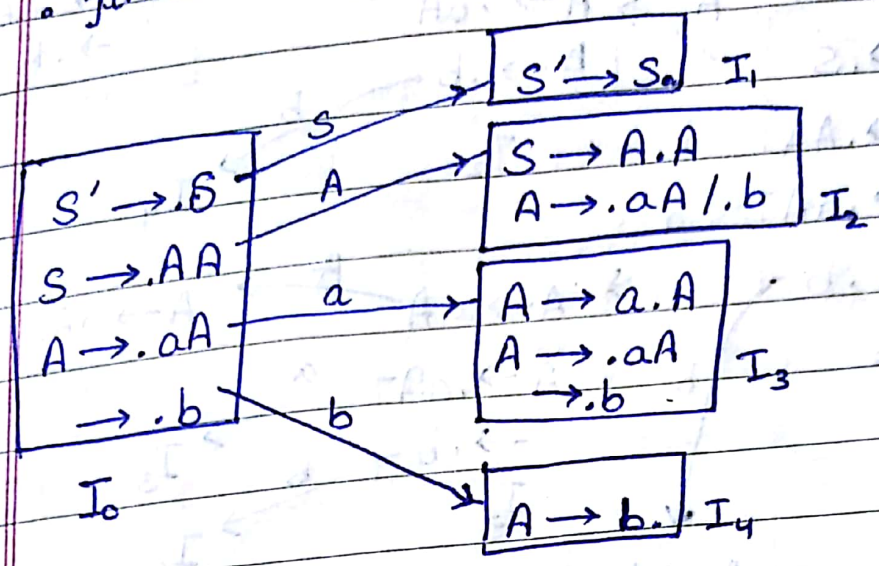
$T \rightarrow .T*F / .F$

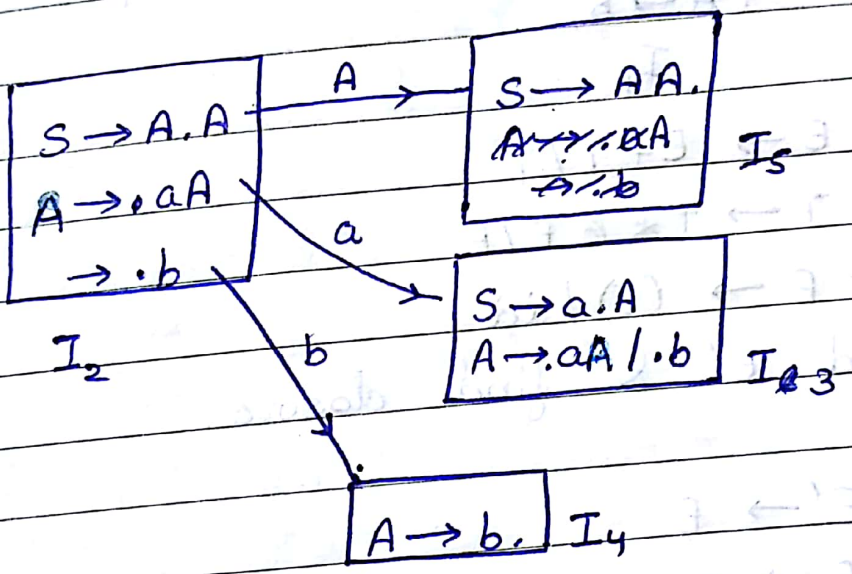
$F \rightarrow .(E) / .id$

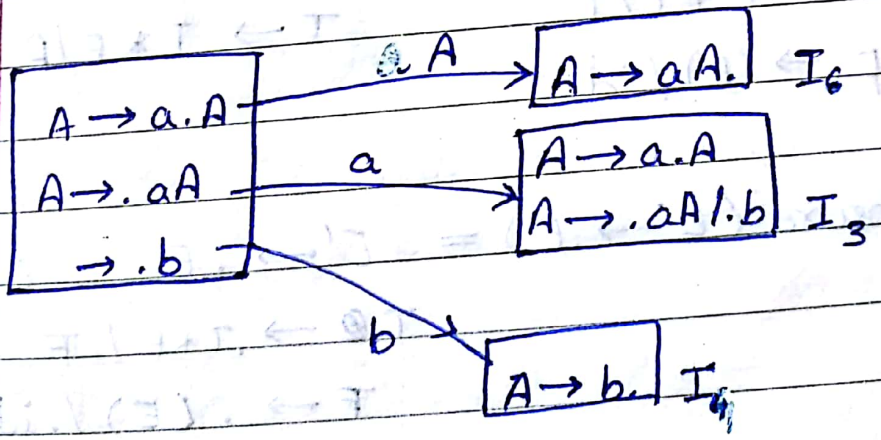
LR Algo for G

- Create G'
- find closure of (S' → .S) & say it item I_0

ex -



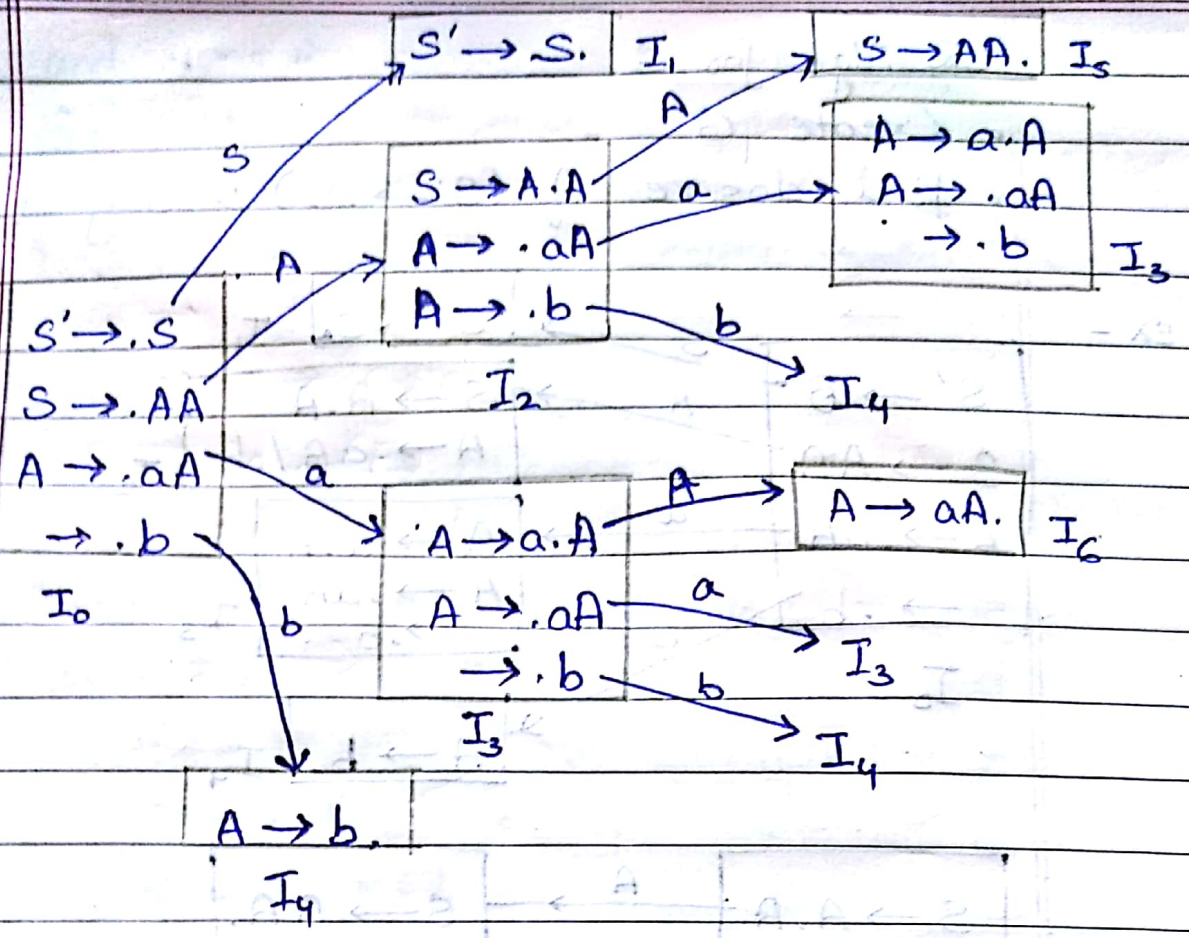




Reduce = I_1, I_5, I_4, I_6

⇓

(Cannot go further)



Example- $E \rightarrow E+T/T$

$T \rightarrow T*F/F$

$F \rightarrow (E)/id$

* Add E' & find closure.

$E' \rightarrow E$

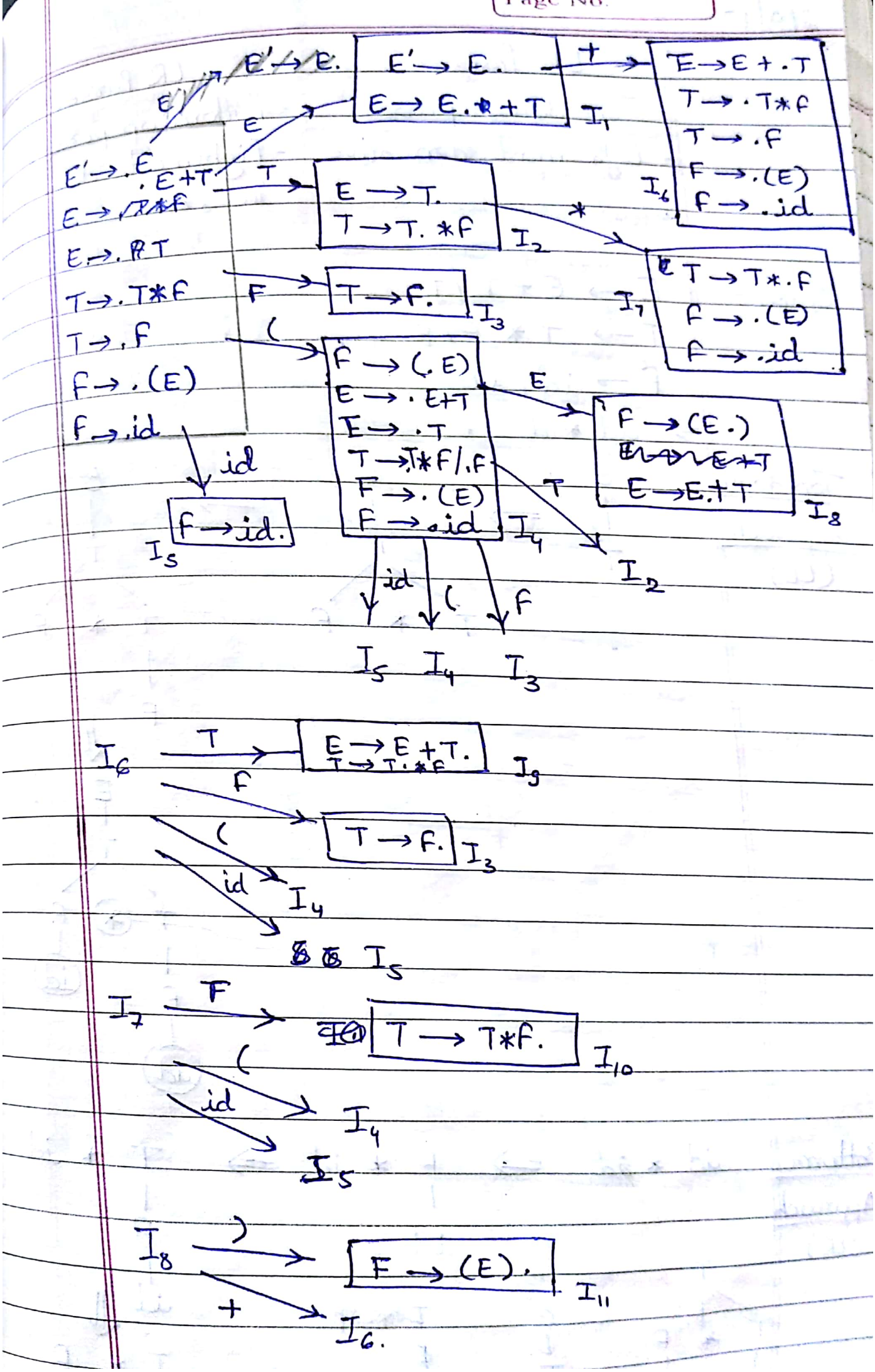
$E \rightarrow E+T/T$ $\quad T \rightarrow T*F/F$

$F \rightarrow (E)/id$

Closure $(E' \rightarrow .E) = E' \rightarrow .E \quad E \rightarrow .E+T/.T$

$T \rightarrow .T*F/.F$

$F \rightarrow .(E)/.id$



6/9/17

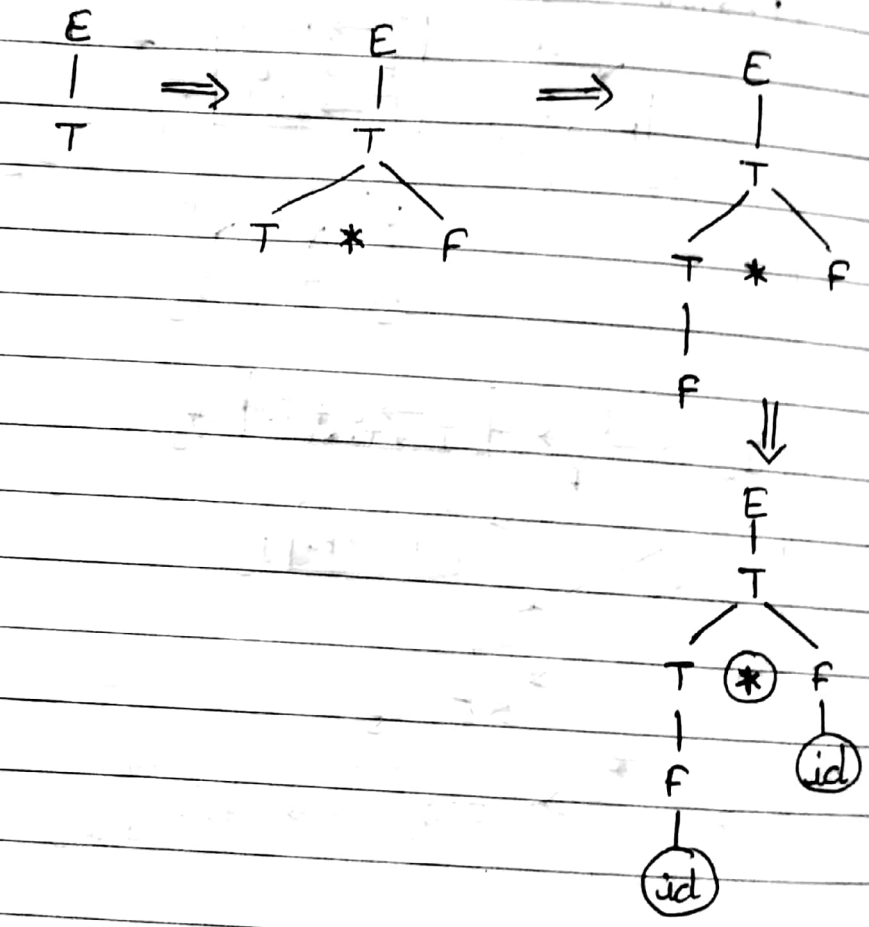
Date :
Page No

LL Parser
- Top down parser
- Left most derivative

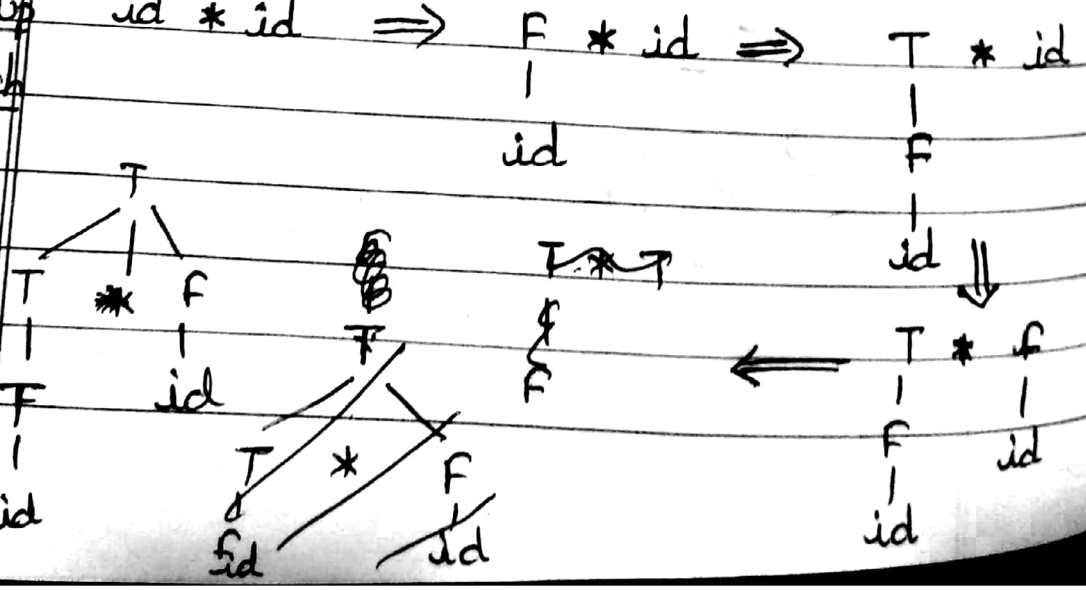
LR Parser
- Bottom up parser
- Rightmost derivative in reverse order.

Ques - $E \rightarrow E+T \,/\, T$
$T \rightarrow T * F \,/\, F$
$F \rightarrow id \,/\, (E)$
$\Rightarrow id * id$

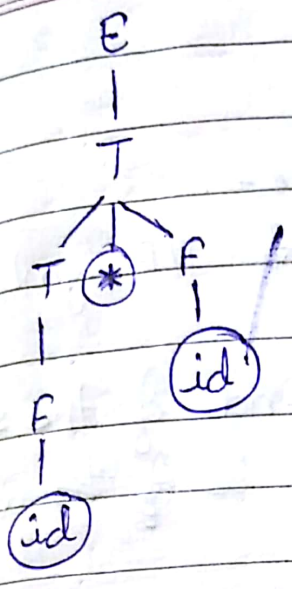
Top down Approach (LL)



Bottom up Approach (LR)



Date :
Page No.



id * id	id	f → id
f * id	f	T → f
T * id	T id	f → id
T * f	T * f	T → T * f
T	E	E → T

Top-down using rightmost derivation -

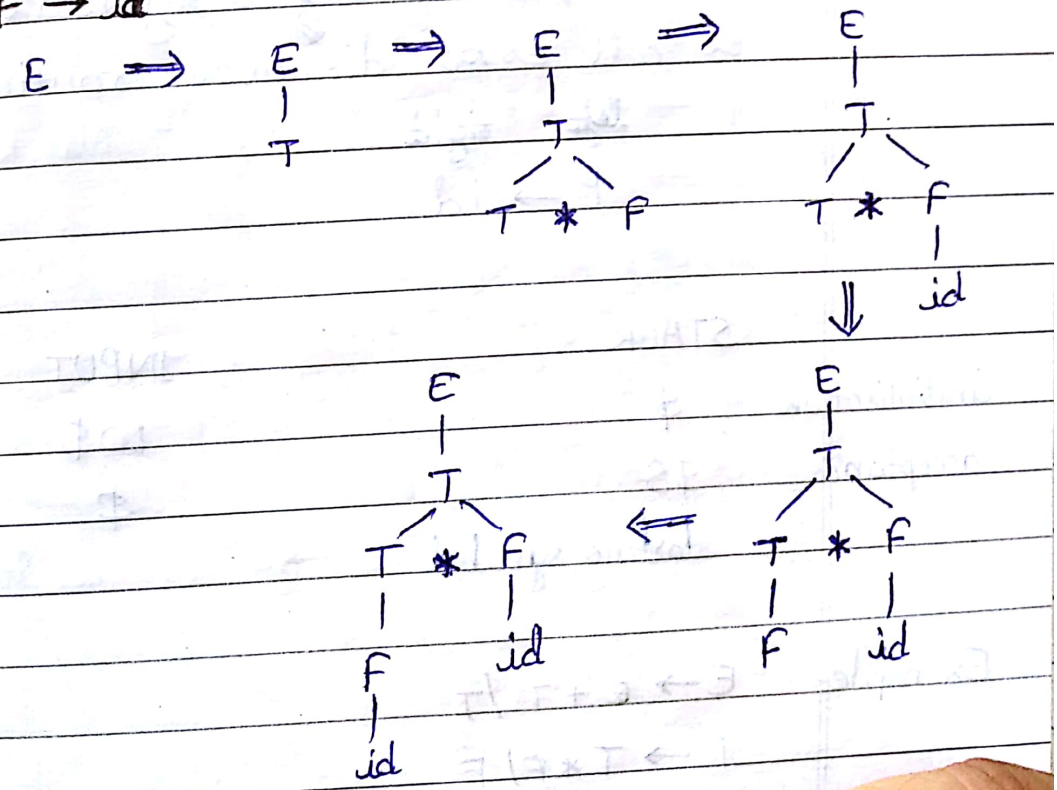
E → T

T → T * F

f → id

T → f

F → id



Shift - reduce parser

Four possible actions are -

1. Shift - Shift the next I/P symbol on top of stack.
2. Reduce - The right end of the string to be reduced must be at the top of the stack.
   Locate left end of the string within the stack & decide with that non-terminal to replace the string.
3. Accept - Successful completion of parsing.
4. Error - Discovery a syntax error & call error discover routine.

left right

$F \rightarrow id$

	STACK	INPUT
initialisation	$	w$
acceptance	$S	$

↓ starting symbol

Example $E \rightarrow E + T / T$

$T \rightarrow T * F / F$

$F \rightarrow id / (E)$

$w = id * id$

STACK	INPUT	ACTION
$	id * id $	} shift
$ id	* id $	} Reduce by F → id
$ F	* id $	} F → T
$ T	* id $	Shift
$ T *	id $	Shift
$ T * id	$	Reduce F → id
$ T * F	$	T → T * F
$ T	$	E → T
$ E	$	Accept

Hence w = id * id is acceptable by the parser.

Date :
Page No.

8/9/2017

Example- $E \rightarrow E+T/T$

$T \rightarrow T*F/F$

$F \rightarrow (E)/id$

Augmented grammar.

$E' \rightarrow E$

$E \rightarrow E+T$

$E \rightarrow T$

$T \rightarrow T*F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id$

Create LR zero item (I_0)

Kernel Item

$E' \rightarrow .E$

$E \rightarrow .E+T$

$E \rightarrow .T$

$T \rightarrow .T*F$

$T \rightarrow .F$

$F \rightarrow .(E)$

$F \rightarrow .id$

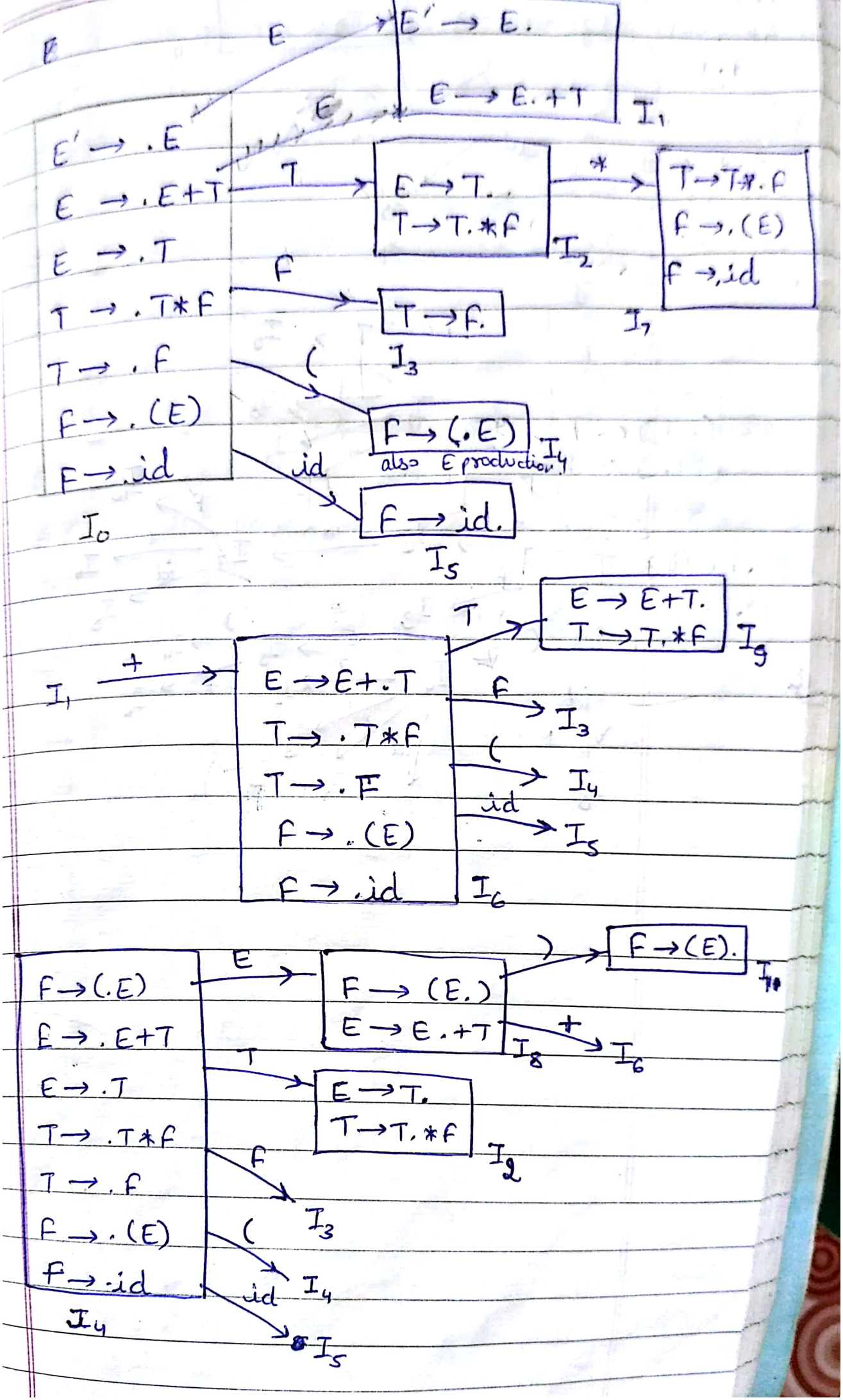
Non kernel item

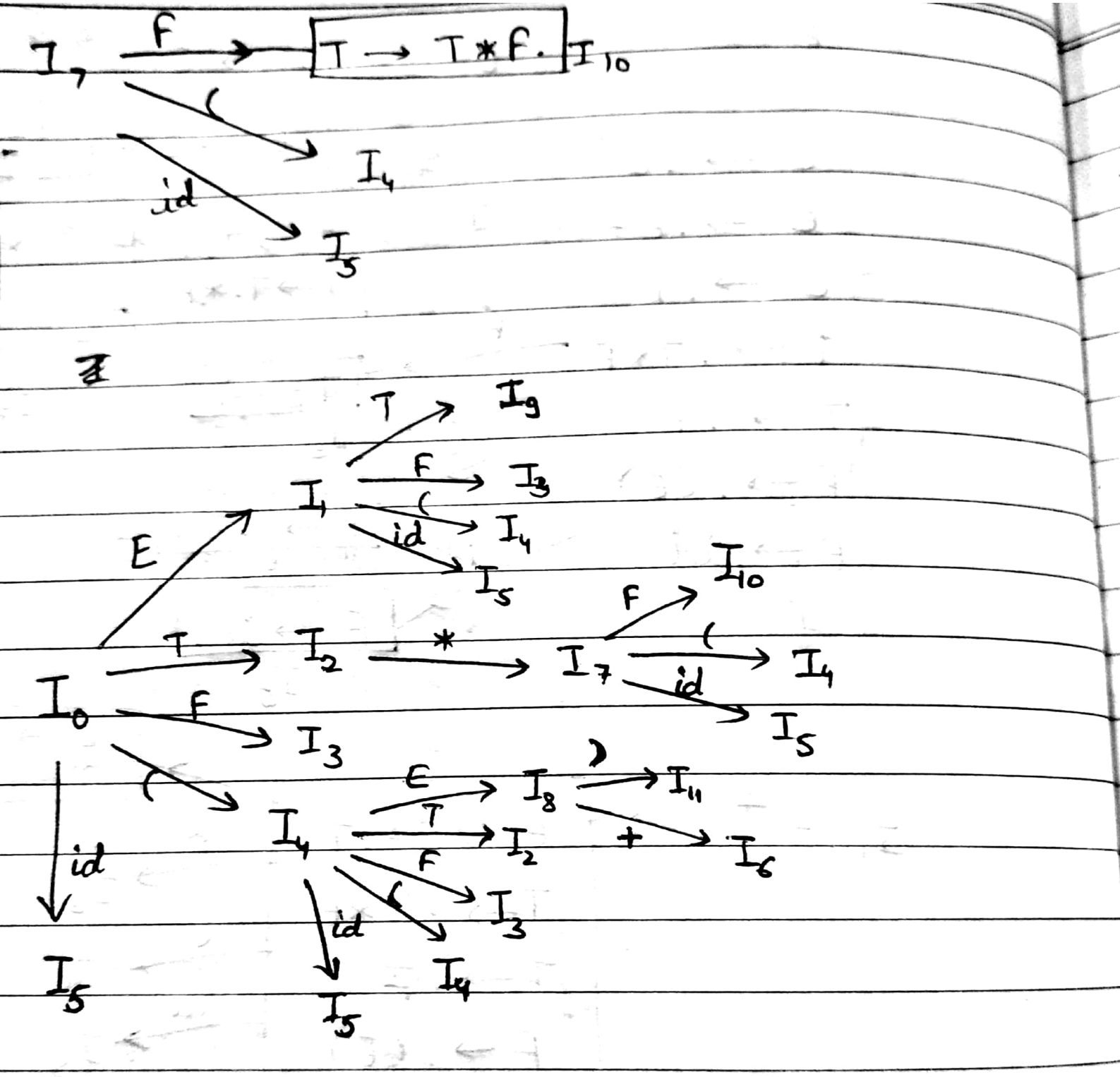
Kernel Item

- The initial item $S' \rightarrow .S$ & all item whose '.' are not left end.

Non-kernel Item

- Except $S' \rightarrow .S$ where '.' on left side





* 1 acceptance only for $E' \to E$.

Date :
Page No.

Create corresponding table.

Item	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				($E' \to E.$) accept			
2	r_2	r_2	S7 / r_2	r_2	r_2	r_2			
3	r_4	r_4	r_4	r_4	r_4	r_4			
4	S5			S4			8	2	3
5	r_6	r_6	r_6	r_6	r_6	r_6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9	r_1	r_1	S7 / r_1	r_1	r_1	r_1			
10	r_3	r_3	r_3	r_3	r_3	r_3			
11	r_5	r_5	r_5	r_5	r_5	r_5			

Reduce.

1 $E \to E + T.$

2 $E \to T.$

3 $T \to T * F.$

4 $T \to F.$

5 $F \to (E).$

6 $F \to id.$

* If any cell in the table has multiple values then the string is not accepted by LR(0) parser.

else

it is in LR(0)

Date
Page No

9/9/17

SLR (Simple LR)

- Write reduce entry only in follow of that production.

State	id	+	*	(	)	$	E	T	F
0	S_5			S_4			1	2	3
1		S_6				accept			
2		r_2	S_7		r_2	r_2			
3		r_4	r_4		r_4	r_4			
4	S_5			S_4			8	2	3
5		r_6	r_6		r_6	r_6			
6	S_5			S_4				9	3
7	S_5			S_4					10
8		S_6			S_{11}				
9		r_1	S_7		r_1	r_1			
10		r_3	r_3		r_3	r_3			
11		r_5	r_5		r_5	r_5			

Thus, the grammer is accepted by SLR parser.

eg- check for id * id

STACK	Symbol	Input	ACTION
0	$	id * id$	shift
05	$id	* id$	Reduce F→id
03	$F	* id$	R: T→F
02	$T	* id$	shift
027	$T*	id$	shift 5
0275	$T*id	$	R: F→id
02710	$T*F	$	R: T→T*F
02	$T	$	R: E→T
01	$E	$	accept

Date :
Page No

Ques- Check whether the given grammar are acceptable by LR or SLR parser.

1. $S \rightarrow AA$

$A \rightarrow aA / b$

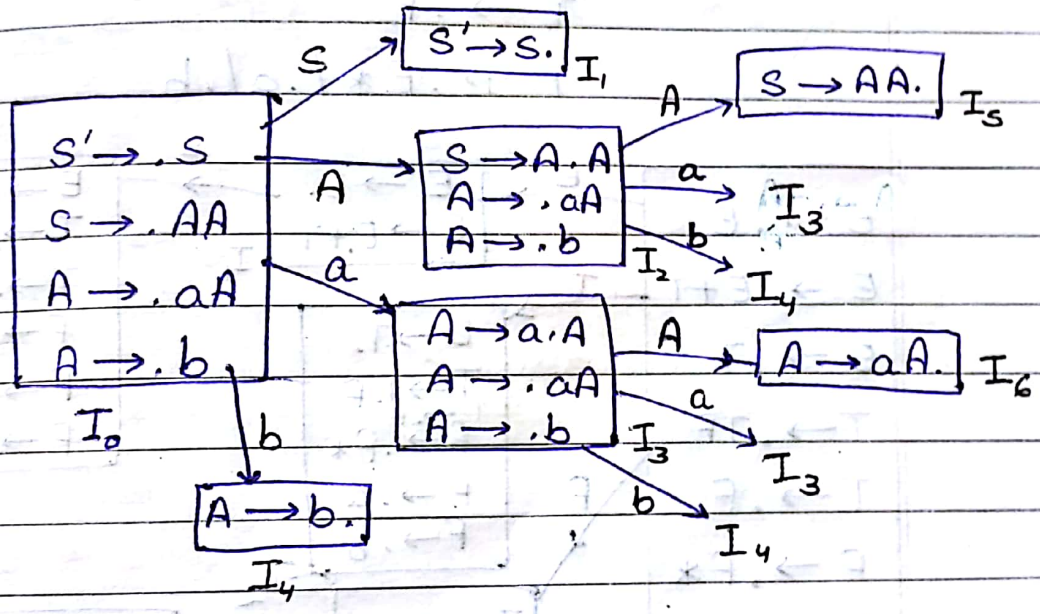
Augmented grammar

$S' \rightarrow .S$

$S \rightarrow .AA$

$A \rightarrow .aA / .b$

Closure of $S' \rightarrow .S$, $S \rightarrow .AA$, $A \rightarrow .aA / .b$ } I_0



State	a	b	$	S	A
0	S_3	S_4		1	2
1			accept		
2	S_3	S_4			5
3	S_3	S_4			6
4	r_3	r_3	r_3		
5	r_1	r_1	r_1		
6	r_2	r_2	r_2		

No, Yes the given grammar is LR. parser acceptable.

Ques- $E \rightarrow E+T / T$

$T \rightarrow TF / F$

$F \rightarrow F* / a / b$

. Make augmented grammar

. Closeure of $E' \rightarrow .E$

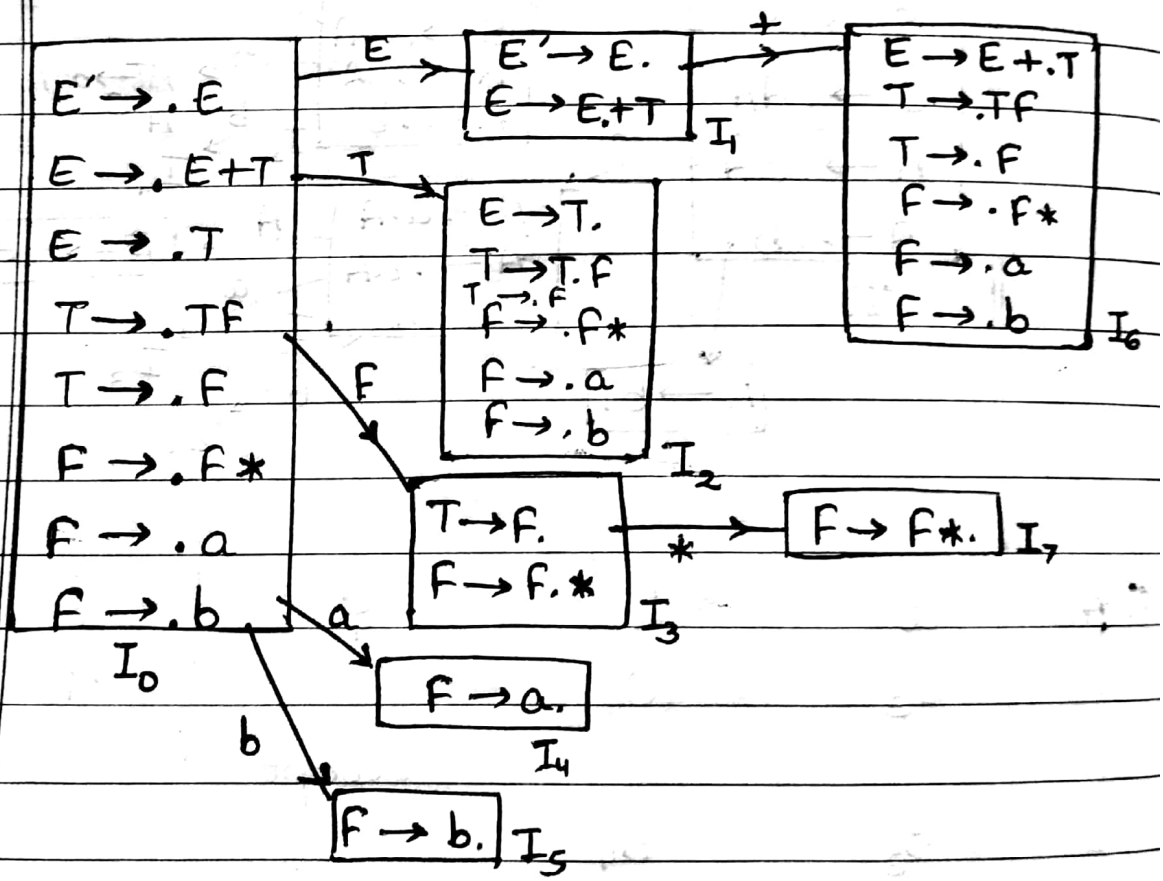
$E \rightarrow .E+T$

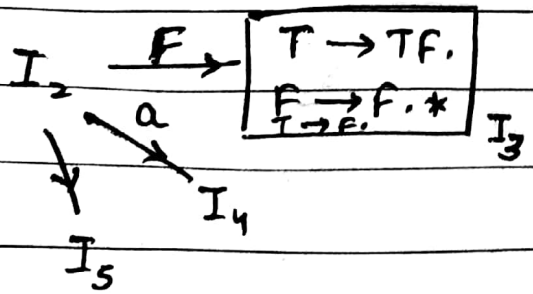
$E \rightarrow .T$

$T \rightarrow .TF$

$T \rightarrow .F$

$F \rightarrow .F* / .a / .b$





Date :
Page No

I_6 —T→

$E \rightarrow E + T.$
$T \rightarrow T . F$
$F \rightarrow . F *$
$F \rightarrow . a$
$F \rightarrow . b$

Date :
Page No.

13/9/17

LR Parser Algo

Let a be the first symbol of w$
Let S be the state on top of stack.

```
if ( Action [$, a] = Shift){
      Push s onto the stack;
      let a be the next input symbol
}
else if (Action [S, a] = reduce A→B)
{
      if Pop [β] symbol of the stack;
      let state t now on top of stack
      Push Goto [t, A] onto stack;
      output the production A→B
}
else if (Action[S,a] = accept)
      break;
else call error-recovery routine
}
```

15/9/17

LR(1) Item

↓

LR(0) item + look Ahead Symbol

Algorithm

LR(1) closure (I) ⇒ 1. Add I to closure (I)

2. If I is $\alpha \rightarrow B.ABC, \$$

&&

$A \rightarrow EFG$

then add ~~$A \rightarrow .EFG$ /first(B~~

$A \rightarrow .EFG$, first(BC, $)

3. Repeat it for newly added number.

LR(0) closure (I) ⇒ 1. Add I to closure (I)

2. If I is $\alpha \rightarrow B.ABC$, &

$A \rightarrow EFG$

then add,

$A \rightarrow .EFG$

3. Repeat for new number.

Example- $S \rightarrow AA$

$A \rightarrow aA/b$

Find LR(1) Closure

Closure $(S' \rightarrow .S, \$)$ *

$\Rightarrow S' \rightarrow .S, \boxed{\$}$

$S \rightarrow .AA, \$$ $\{S' \rightarrow .^{S}A, first(\$)\}$

$A \rightarrow .aA, a/b$

$A \rightarrow .b, a/b$

Date :
Page No.

Ques - 1

$S' \rightarrow S$

$S \rightarrow CC$

$C \rightarrow cC/d$ Check for CLR

Closure $(S' \rightarrow .S, \$)$

$\Rightarrow$

I_0:
- $S' \rightarrow .S, \$$
- $S \rightarrow .CC, \$$
- $C \rightarrow .cC, \$/c/d$
- $C \rightarrow .d, c/d$

I_1: $S' \rightarrow S., \$$ (on S)

I_2: $S \rightarrow C.C, \$$; $C \rightarrow .CC, \$$; $C \rightarrow .d, \$$ (on C)

I_3: $C \rightarrow c.C, c/d$; $C \rightarrow .cC, c/d$; $C \rightarrow .d, c/d$ (on c)

I_4: $C \rightarrow d., c/d$ (on d)

I_5: $S \rightarrow CC., \$$ (on C from I_2)

I_6: $C \rightarrow c.C, \$$; $C \rightarrow .cC, \$$; $C \rightarrow .d, \$$ (on c from I_2)

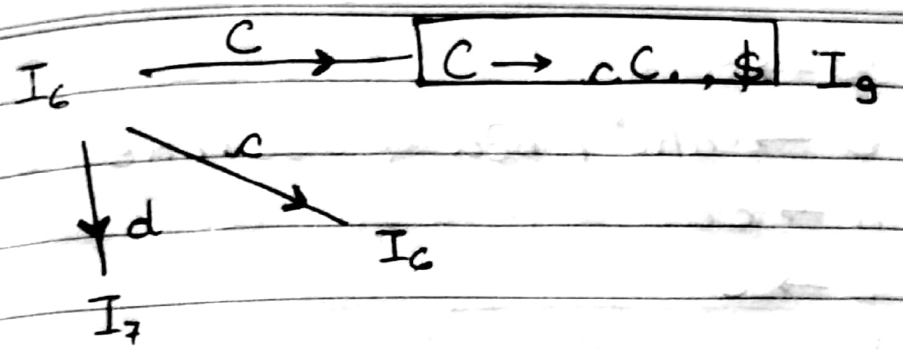
I_7: $C \rightarrow d., \$$ (on d from I_2)

I_8: $C \rightarrow cC., c/d$ (on C from I_3)

I_3: $C \rightarrow c.C, c/d$; $C \rightarrow .cC, c/d$; $C \rightarrow .d, c/d$ (on c from I_3)

I_4: $C \rightarrow d., c/d$ (on d from I_3)

I_3 (on c from I_3)



1. $S \rightarrow CC.$
2. $C \rightarrow cC.$
3. $C \rightarrow d.$

} Reduce

STATE	ACTION			GOTO	
	c	d	$	S	C
0	S_3	S_4		1	2
1			accept		
2	S_6	S_7			5
3	S_3	S_4			8
4	r_3	r_3			
5			r_1		
6	S_6	S_7			9
7			r_3		
8	r_2	r_2			
9			r_2		

★ Fill r_n ~~where~~ under the look ahead symbol column.

For CLR, this has no conflict. Thus, the grammar is in **CLR**. (Canonical LR)

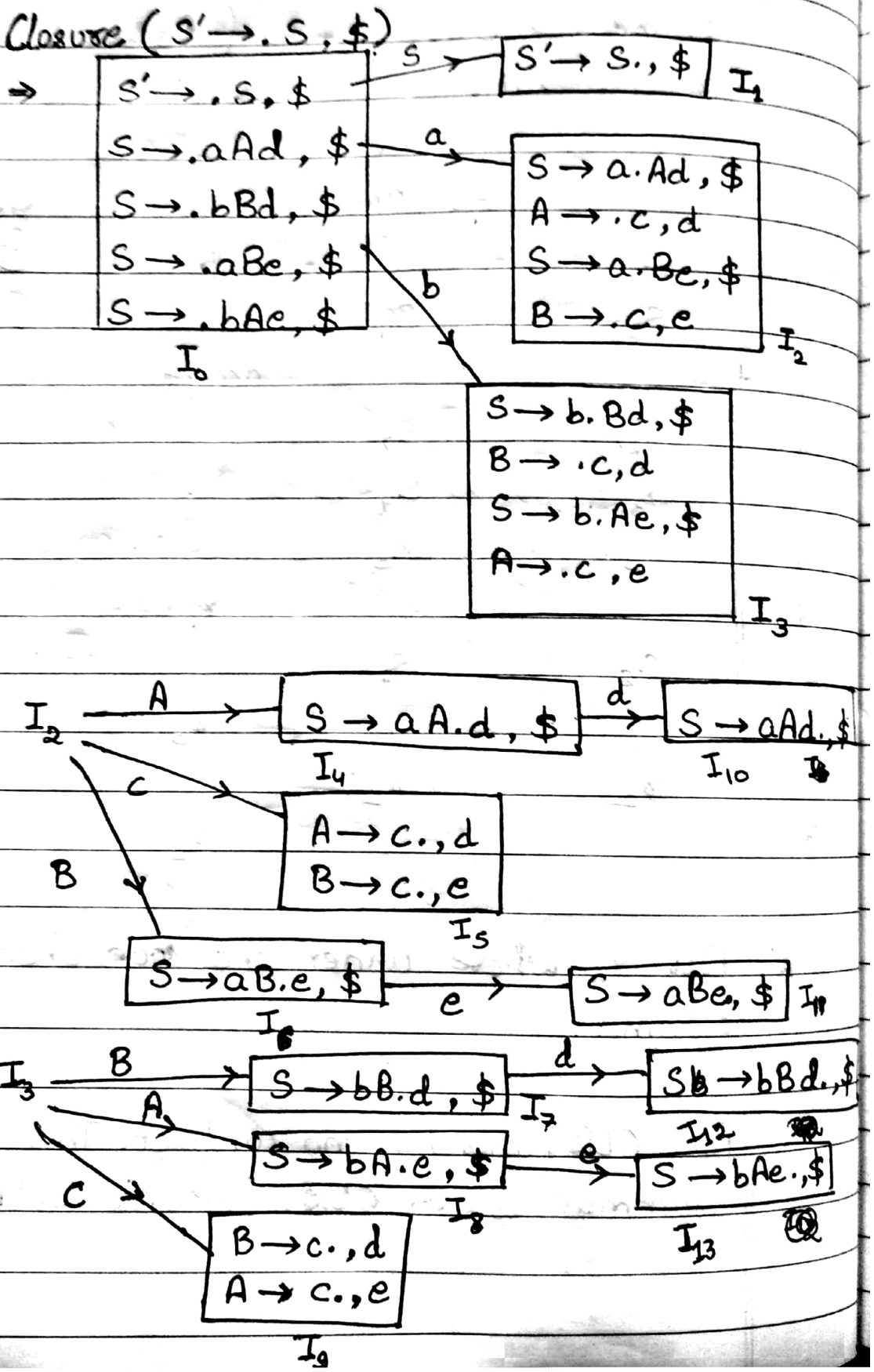
Ques-2 $S' \rightarrow S$

$S \rightarrow aAd \mid bBd \mid aBe \mid bAe$

$A \rightarrow c$

$B \rightarrow c$

Check for CLR Parser.

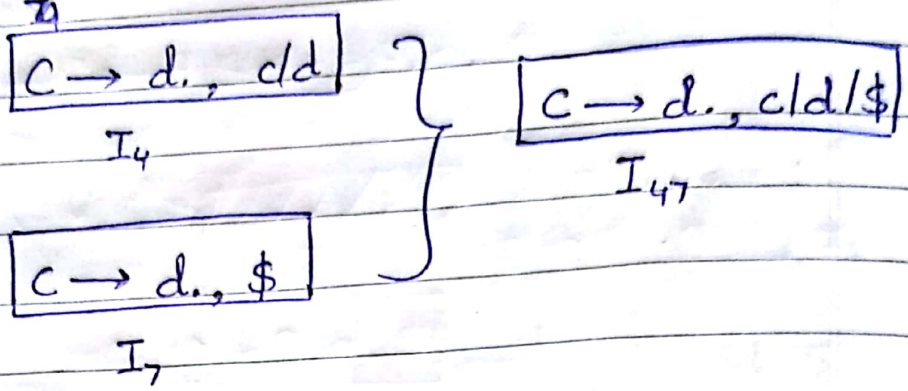


16/9/17

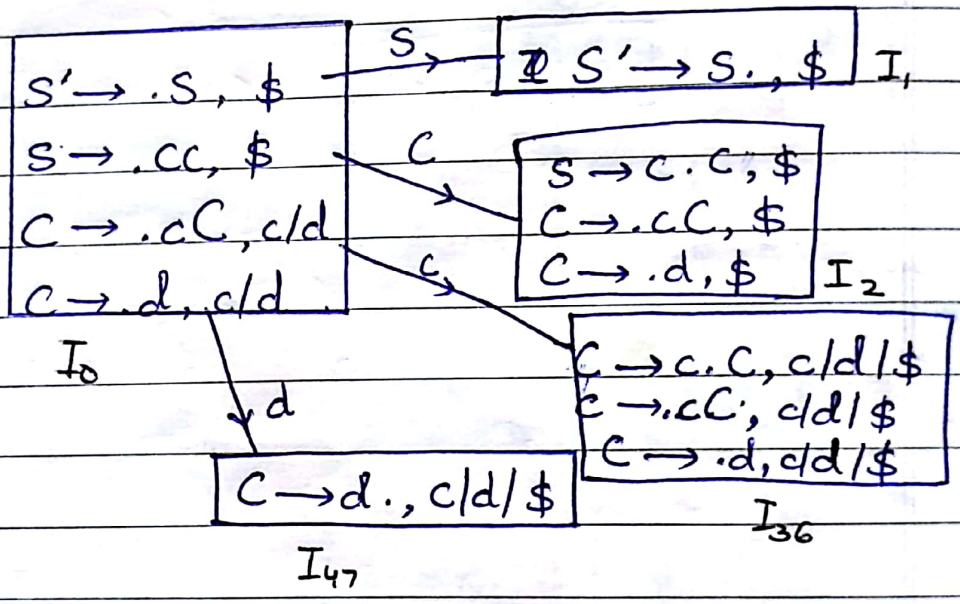
LALR

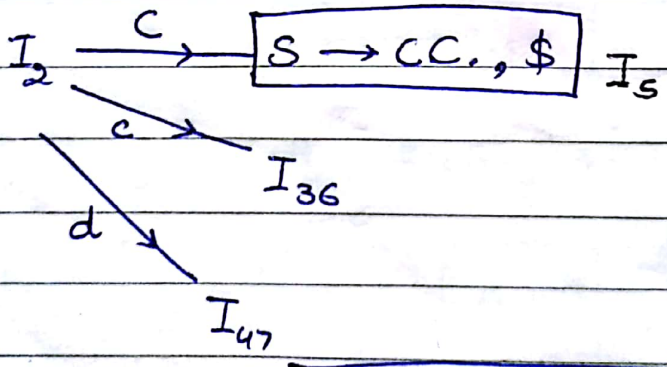
Ref Ques1 - In LALR, merge ~~state~~ stage / item which have only diff. look ahead symbol.

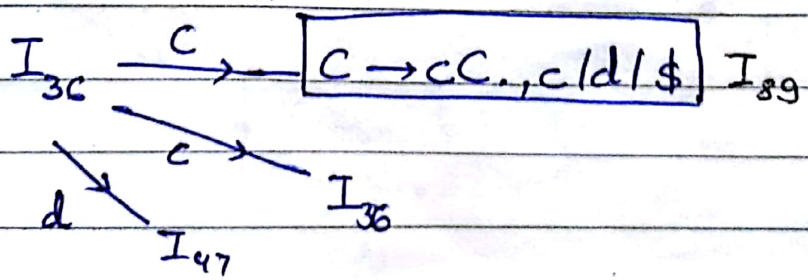
eg-



- Replace I_4 & I_7 with I_{47}
- Do d this for all such items.







STATE	ACTION c	ACTION d	ACTION $	GOTO S	GOTO C
0	S_{36}	S_{47}		1	2
1			accept		
2	S_{36}	S_{47}			5
36	S_{36}	S_{47}			89
47	r_3	r_3	r_3		
5			r_1		
89	r_2	r_2	r_2		

Ques3- $S' \longrightarrow S$

$S \longrightarrow Aa$ r_1

$S \longrightarrow bAc$ r_2

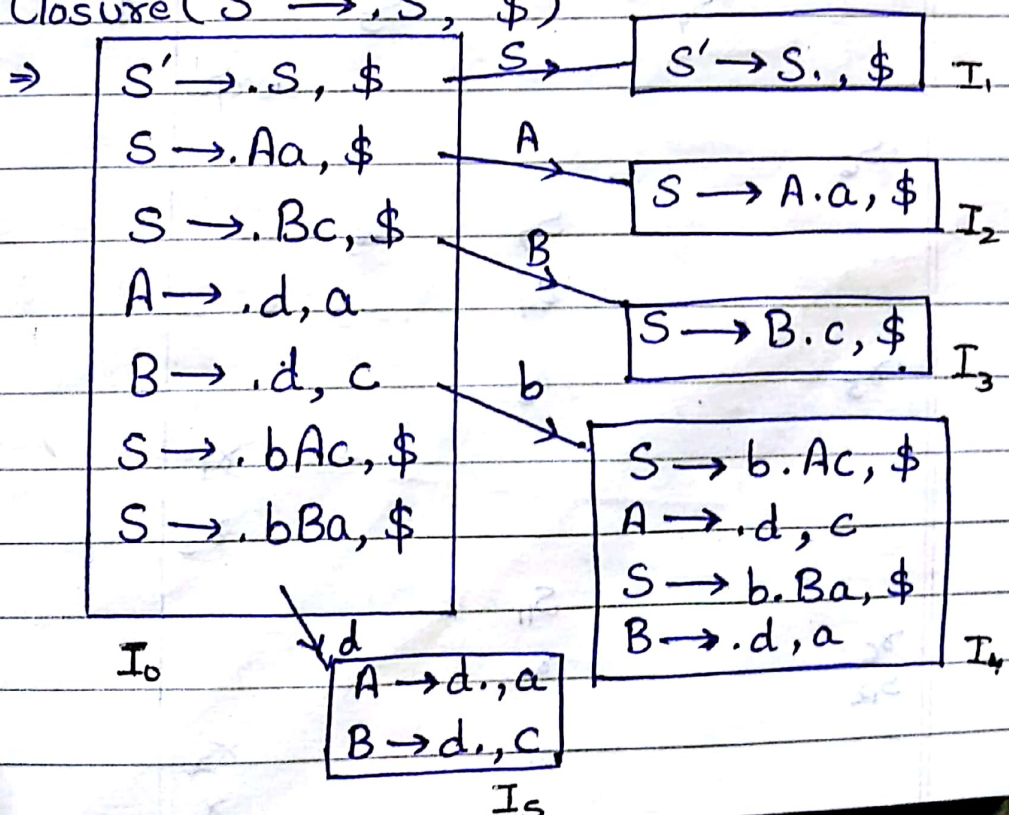
$S \longrightarrow Bc$ r_3

$S \longrightarrow bBa$ r_4

$A \longrightarrow d$ r_5

$B \longrightarrow d$ r_6

Closure($S' \longrightarrow .S, \$$)



$I_2 \xrightarrow{a}$ [S → Aa. , $] I_6

$I_3 \xrightarrow{c}$ [S → Bc. , $] I_7

$I_4 \xrightarrow{A}$ [S → bA.c , $] I_8

$I_4 \xrightarrow{d}$ [A → d. , c / B → d. , a] I_9

$I_4 \xrightarrow{B}$ [S → bB.a , $] I_{10}

$I_8 \xrightarrow{c}$ [S → bAc. , $] I_{11}

$I_{10} \xrightarrow{a}$ [S → bBa. , $] I_{12}

	ACTION					GOTO		
STATE	a	b	c	d	$	S	A	B
0		S_4		S_5		1	2	3
1					accept			
2	S_6							
3			S_7	S_9			8	10
4			S_7	S_9			8	10
5	γ_5		γ_6					
6					γ_1			
7					γ_3			
8			S_{11}					
9	γ_6		γ_5					
10	S_{12}							
11					γ_2			
12					γ_4			

Date :
Page No.

⇒ Solving for LALR Parser

- I_5 & I_9 are identical irrespective of lookahead symbols.

$A \rightarrow d.,\ c/a$
$B \rightarrow d.,\ c/a$

I_{59}.

New Table.

STATE	ACTION					GOTO		
	a	b	c	d	$	S	A	B
0		S_4		S_{59}		1	2	3
1					accept			
2	S_6							
3			S_7					
4				S_{59}			8	10
6					γ_1			
7					γ_3			
8			S_{11}					
10	S_{12}							
11					γ_2			
12					γ_4			
59	γ_5		γ_6					

Yes, the grammer is LALR parser.

20/9/17

Syntax Directed Translation (SDT)

$S \rightarrow S + T$

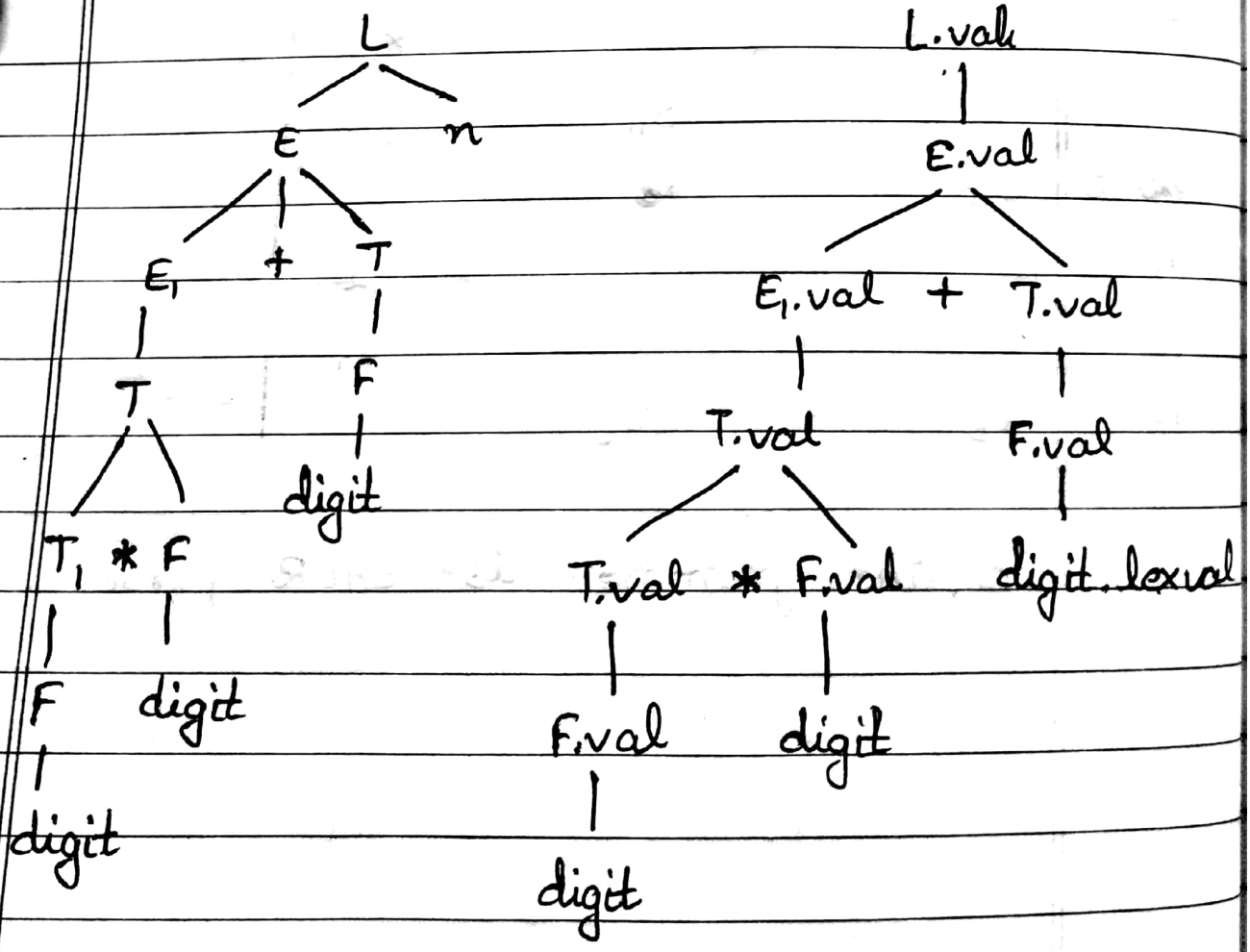
$S \rightarrow S + T$ {T.value = F.value}

printf("f") → semantic attribute

Calculation of + & *

$L \rightarrow En$	{ L.val = E.val }
$E \rightarrow E_1 + T$	{ E.val = E_1.val + T.val }
$E \rightarrow T$	{ E.val = T.val }
$T \rightarrow T_1 * F$	{ T.val = T_1.val X F.val }
$T \rightarrow F$	{ T.val = f.val }
$F \rightarrow (E)$	{ F.val = E.val }
$F \rightarrow$ digit	{ F.val = digit.lexval }

3 * 5 + 4n

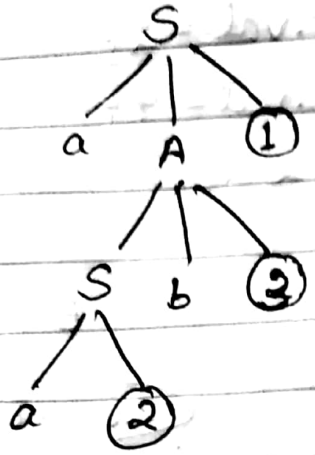


Ex- $S \longrightarrow aA$ {print 1}

$S \longrightarrow a$ {print 2}

$A \longrightarrow Sb$ {print 3}

Parse "aab"



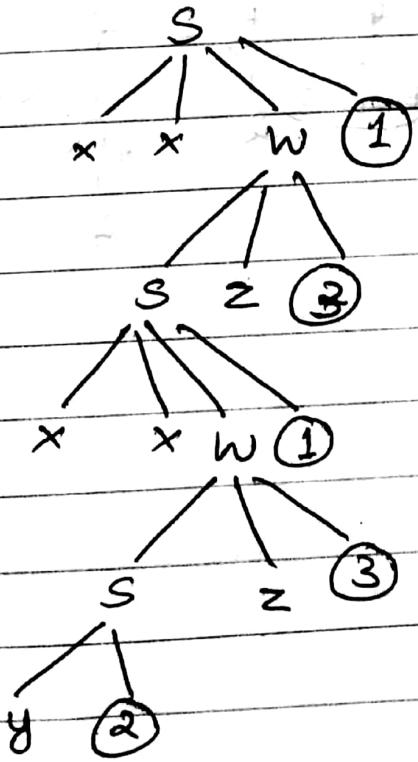
→ 2 3 1

Ex- $S \rightarrow xxW$ {print 1}

$S \rightarrow y$ {print 2}

$W \rightarrow Sz$ {print 3}

Parse "xxxxyzz"

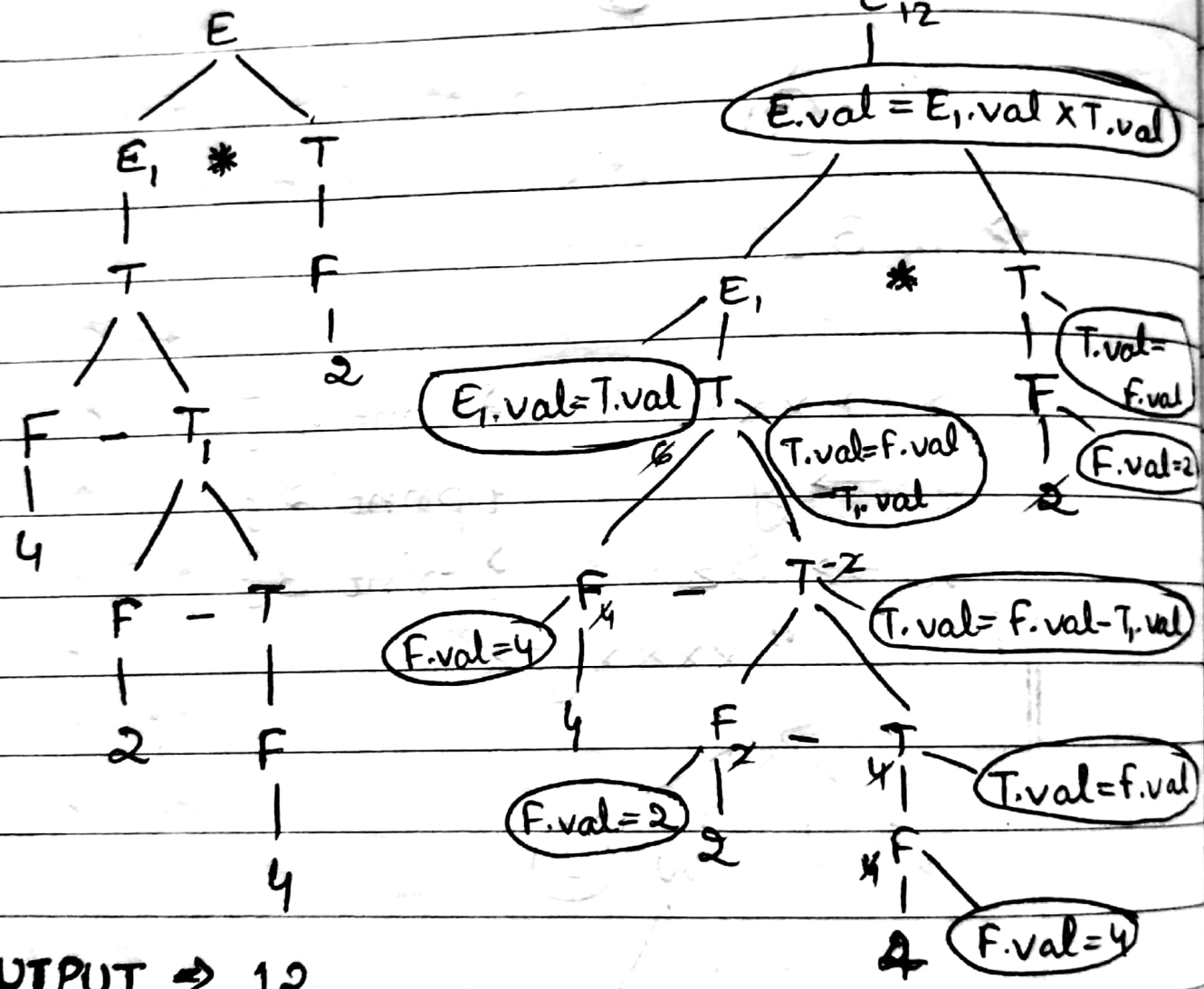


→ 23131

Ex-

$E \rightarrow E_1 * T$	$\{E.val = E_1.val \times T.val\}$
$E \rightarrow T$	$\{E.val = T.val\}$
$T \rightarrow F - T_1$	$\{T.val = F.val - T_1.val\}$
$T \rightarrow F$	$\{T.val = F.val\}$
$F \rightarrow 2$	$\{F.val = 2\}$
$F \rightarrow 4$	$\{F.val = 4\}$

INPUT ⇒ 4 - 2 - 4 * 2



OUTPUT ⇒ 12

22/9/17

Page No.

SDT

Syntax Directed Definition
- It is a CFG together with attributes & rules. Attributes are associated to grammar symbols & rules are associated with productions

eg- X.a (X → Symbol, a → Attribute.) $X \rightarrow a$ Production rule

Attributes of many types - number, String, type, table refrence

Type of Attributes-
1. Synthesised Attribute -
2. Inherited Attribute -

Synthesized Attribute for a non-terminal A At a parse tree node N is defined by semantic rule associated with the production at N.
- The production must have A at its head A synthesised attribute at node N is defined only in term of attribute value at its children of N & at N itself.

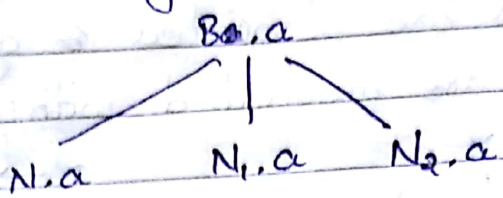
eg- E.value = T.val + F.val

Inherited Attribute for non-terminal B At a parse tree node N is defined by semantic rule associated with the production at the

parent of N.

- A inherited attribute at Node N is defined only in term of attribute values at N's parent, siblings & itself.

eg-



Application of SDT

1. Converting infix to postfix
2. Converting infix to prefix.
3. Binary to decimal conversion

Example- L → En

E → E1 + T {print(+)}

E → T

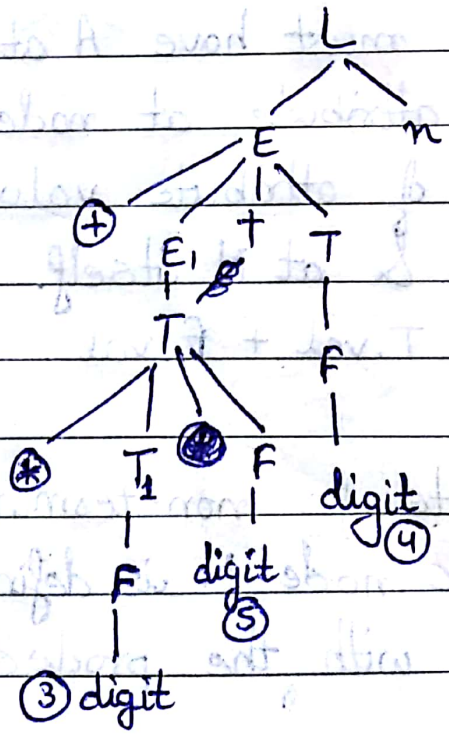
T → T * F {print(*)}

T → F

F → (E)

F → digit {print(digit.lexval)}

INPUT ⇒ 3 * 5 + 4



OUTPUT ⇒ 3 5 * 4 +

More Applications of S.D.T

- Draw DAG (Directed Acyclic Graph)
- Generate Intermediate Code