System Programming.

PAGE :
DATE : / /

System
- H/w → µp → opcode, microcode
- S/w → OS.

↳ compiler
↳ Assembler
↳ Interpreter

1.) high level language (source code)
2.) Assembly language
3.) Machine code.

* source code → compiler → intermediate code
↳ assembly code

Java	C++
• java	• cpp
↓	↓
• class	• exe
Intermediate code (Bytecode)	assembly code

source code
↑ compiler
assembly language
↑ assembler
machine language
↓
machine

Compiler : Architecture :

Lexical analysis (HLL) — Lex.
Syntatic analyser — Yacc.
Semantic (meaningful statement)
↓
Intermediate code generation
↓
Code optimization
↓
~~Assembly~~ Target code generation
↓
Output (Assembly language)

Symbol Table — interchangeable

Error handling

Eg. – $x = a + b * 6.5$;

⇓

Lexical analyzer — Identify the tokens, variables, const etc.

(id, 1) = (id, 2) + (id, 3) * 6.5 .

address.

$id_1 = id_2 + id_3 * 6.5$

Symbol Table
1. int x
2. int a
3. int b

Suppose the rules are defined as :–

$S \rightarrow id = E$
$E \rightarrow E + T / T$
$T \rightarrow T * F / F$
$F \rightarrow id / num$

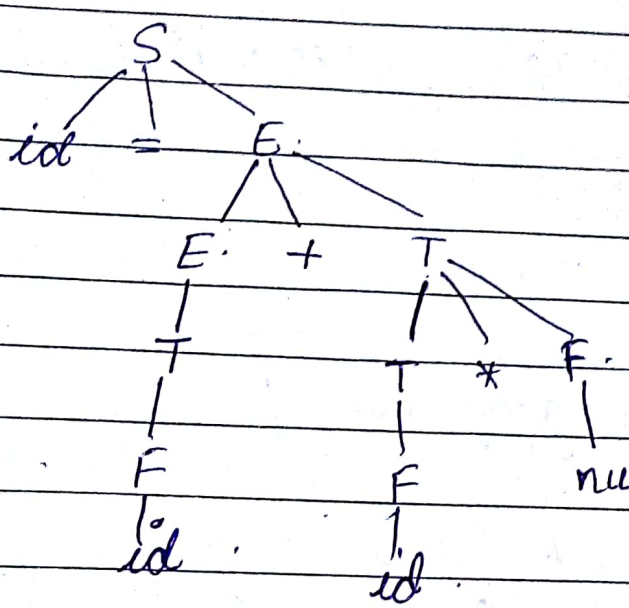
Production rule.

DATE : / /

Symbols :-

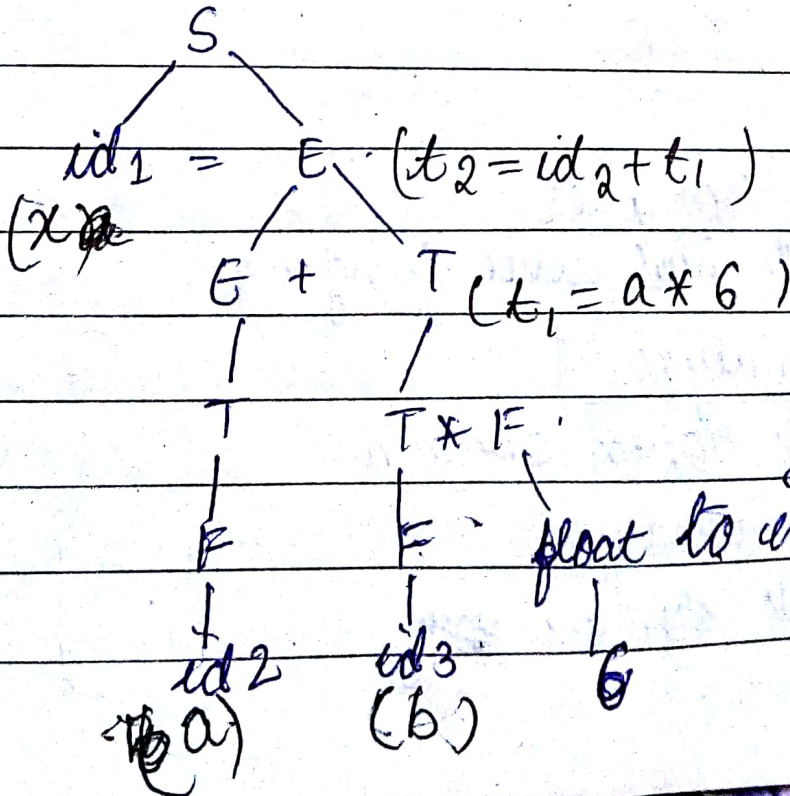
* Terminal symbol (small letter)
Non-Terminal symbol (capital letter)
('S') starting symbol.

Syntatic Analyzer



Parse Tree
i) ambiguous
ii) unambiguous

Semantic

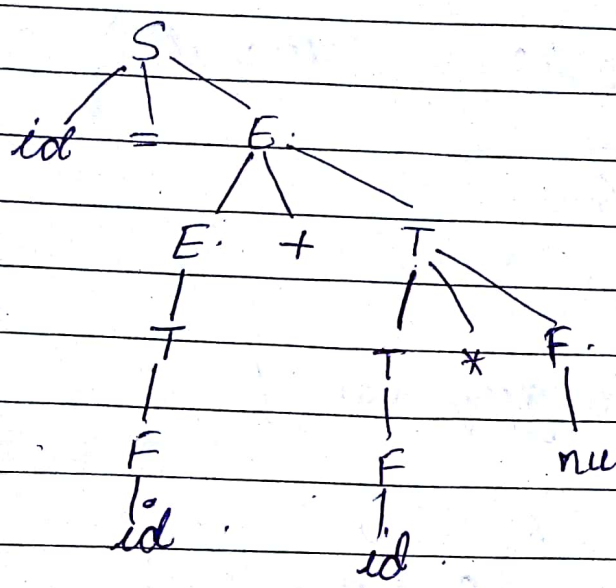


PAGE :
DATE : / /

Symbols :-

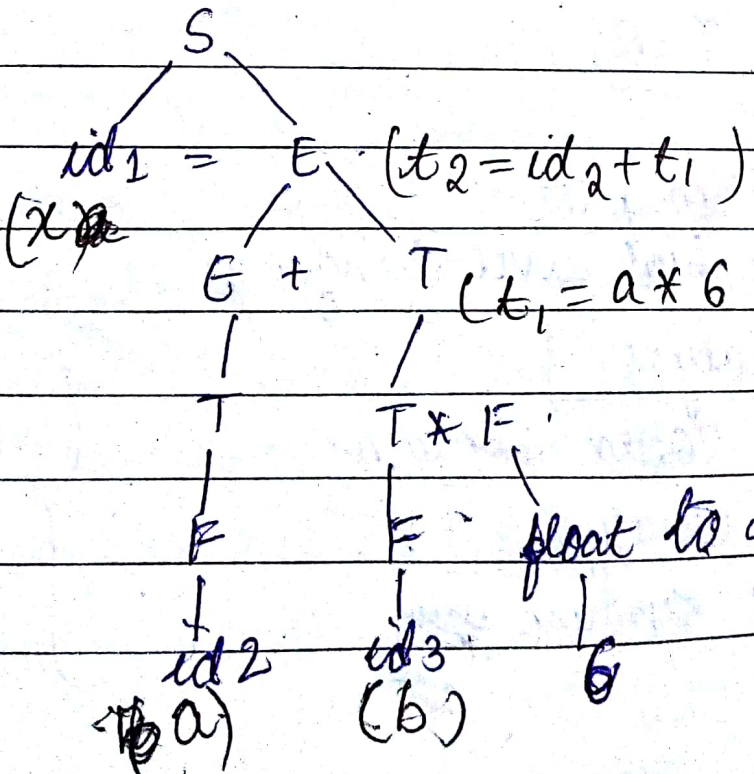
* Terminal symbol (small letter)
Non-Terminal symbol (capital letter)
('S') Starting Symbol.

Syntatic Analyzer



Parse Tree
(i) ambiguous
(ii) unambiguous

Semantic



Intermediate Code Generator

$t_1 = id3 * 6$ or $t_1 = a * 6$
$t_2 = id2 + t_1$ or $t_2 = b + t_1$
$x = t_2$ $x = t_2$

Code Optimization (optional)

$t_1 = b * 6$
$x = a + t_1$

Assembly Language Code

```
mov R1, b   (R1 ← b)
mul R, 6    (A ← R1 * 6)
mov R2, A   (R2 ← A)
ADD R1, R2
mov x, R1
```

* Phase of Compiler :
  ⇓ high level language
  Lexical analyzer
  ⇓ Token stream
  Syntax analyzer
  ⇓ Syntax tree

analysis part, front end.

analysis part, front end
Symbol table
Synthetic part, Back end

⇓
Semantic Analyzer
⇓ Syntax tree
Intermediate code Generator
⇓ Intermediate Representation
Machine independent code optimizer
⇓ Intermediate Representation
Code Generation
⇓ target code
M/C Dependent Code Optimizer (machine)
⇓ target

* Code Optimization
  - M/C Dependent (up): A - ADD, SUB; B - MUL, ADD.
  - M/C Independent (logic)

* Compiler mainly consists of two parts :-
  (i) analysis part
  (ii) Synthetic part.

* Compiler

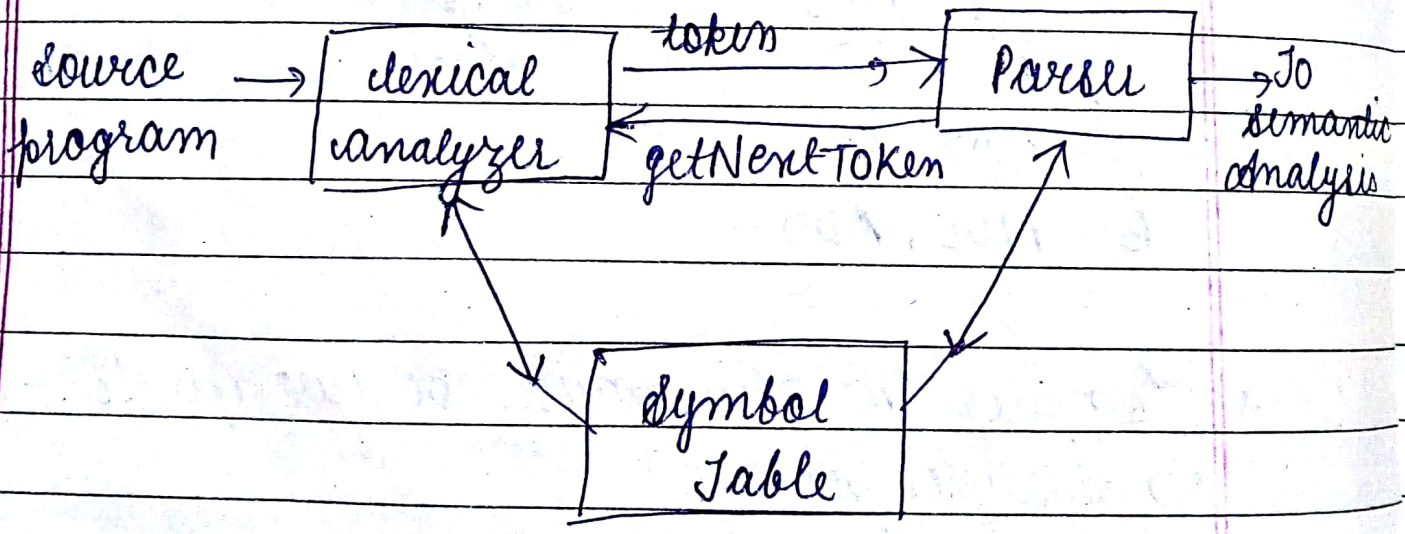
Single pass
(all the modules are loaded at the same time in memory)
(ie, all modules placed in main memory)
eg.- C, C++

Multi-pass.
First load front end then load back end.
eg.- Java.

Disadvantage: more memory
Adv: less time.

Adv: less memory.
Disad: more time

* Lexical analyzer:



- Read the input character of source program, group them into lexemes and process as output a sequence of token for each lexemes in the source program.

int a, b ;

lexemes. and inke corresponding jo load hota h us token.

* Finite automata : mathematic structure to check if data can be converted as lexemes.

* Lexical analyzer :
- It remove comment line and whitespace character (i.e new line, Tab etc.)
- It will help to provide error message by providing line no.

Eg :- int a, b ;

5 tokens.

printf ("Hello") ;

5 tokens.

int max (i, j)
{ /* max of i & j */
return i > j ? i : j ;
}

18 tokens.

printf ("i = %d , &i = %d", i, &i);

10 tokens.

* Lexical Analyzer
  - ↳ Scanning
  - ↳ Lexical Analysis.

int - key
a - var
23 - Count

int a=23;

* printf ("Total = %d", score);
  id literals id

No. of tokens = 7.

TOKEN :	FORMAL DESCRIPTION	EXAMPLE.
IF	character i,f	if
ELSE	character e,l,s,e	else
Comparison	< or > or <= or =>	>, =>, <=
id	letter followed by letter or digit	variable name or keywords
number	constant	23
literal	anything in " "	"total = %d"

(b) using Regular expression:
ex: letter (letter / digit)*

PAGE :
DATE : / /

(ii)
* Input Buffering :

* Two types of pointer :
(i) Lexeme Begin
(ii) Forward

*. Terms related to string :
Prefix : $w_1 = aw$, a is called prefix.
Postfix Suffix $w_1 = wb$, b is a suffix.
Substring : $w_1 = awb$, w is a substring.
Proper prefix
Proper suffix
Proper substring if $w \neq \epsilon$, $w \neq w_1$
Subsequence.

Eg:- w_1 = IP College, Prefix : I P, IP Col, IP College
Not in proper prefix { X, X
Suffix : College, : ege
$\wedge$ IP college = IP college

* (Proper subst : A ⊂ S but A ≠ S)

* College
'ol' is substring
'oec' is subsequence.
but 'OC' not a subsequence. X

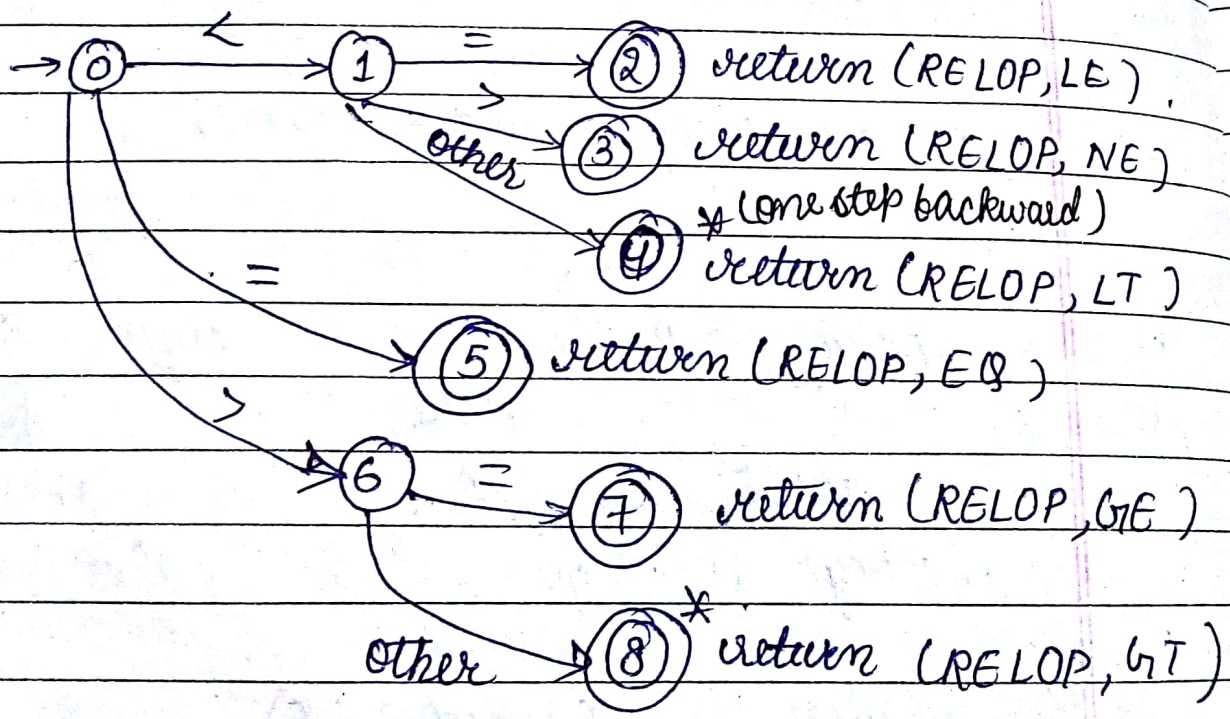
PAGE :
DATE : / /

* Transition Diagram :

* Relational operation :

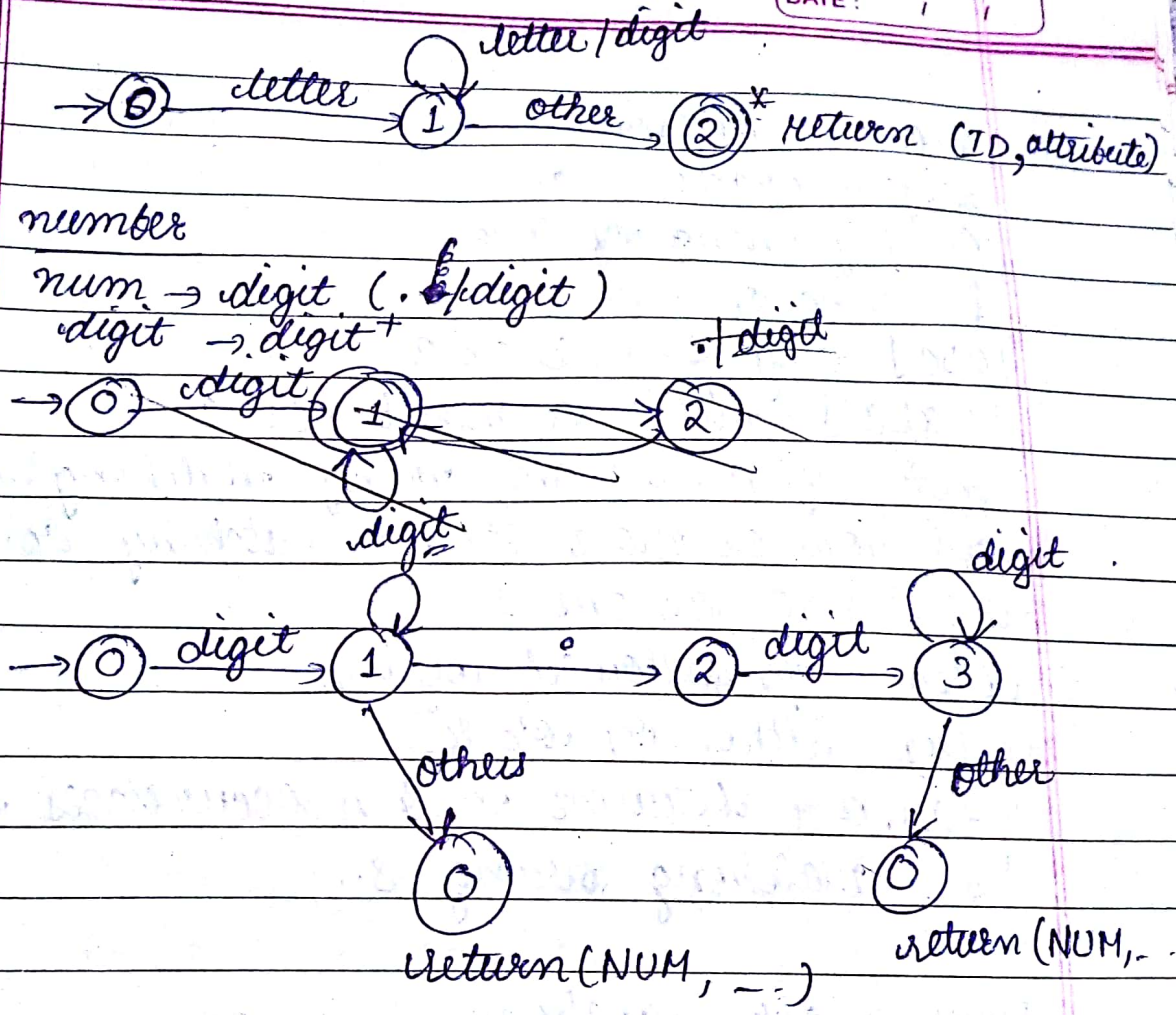
Lexemes	Token Name	Attributes
>	RELOP	GT
<	RELOP	LT
>=	RELOP	GE
<=	RELOP	LE
<>	RELOP	NE
=	RELOP	EQ

* FA / Transition Diagram :



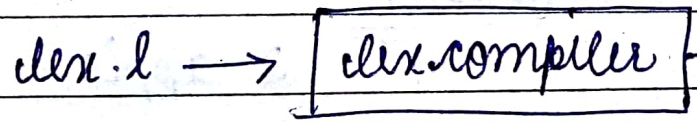
* Transition diagram for Reserve word and identifier

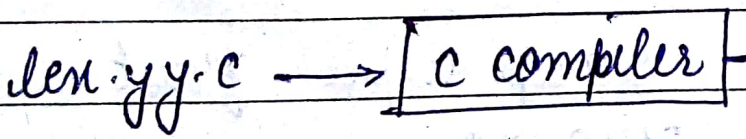
id → (letter) (letter/digit)*

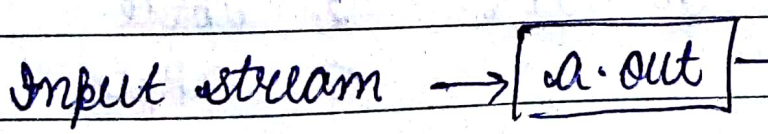


6. Lexical Analysis (Lex or Flex):

- Lex is a compiler.







* Regular Expression:

any character .
^ beginning of line
$ end of line
[abc] any of a, b or c
[^abc] nether a nor b nor c
$r*$ zero or more string matching r
$r+$ one or more string matching r.
$r?$ zero or one r
$r_1 r_2$ r_1 followed by r_2
$r_1 | r_2$ either r_1 or r_2
$r\{m, n\}$ between m & n occurences of r
"s" matching string s.

* $grep → for searching.

fruit.

1	Apple
2	apple
3	APPLE
4	mango
5	aPple
6	abple
7	an Apple
8	Apple Red

eg - $grep "apple" fruit

OUTPUT:

2 apple.

$grep a.ple fruit.

OUTPUT:

2. apple
5. aPple
6. abple.

[a-z A-Z]
[0-9]

$grep [aA]pple fruit.

OUTPUT: 1 Apple
2. apple

$grep apple/Apple fruit.

$grep Apple$ fruit. → OUTPUT: 1. Apple
$grep ^Apple fruit. 7. An Apple.

OUTPUT: 1. Apple
8. Apple Red.

* id → (letter) (digit / letter)*.
letter [a-z A-Z]
digit [0-9]

id {letter} ({letter} | {digit})*

* Lexical:

Parsing
↙ Top-Down Parser
↘ Bottom-up Parser.

Grammar: (small letter)
- A grammar G = (V, T, S, P)
V → set of non-terminal symbol (capital)
T → set of terminal symbol
S → starting symbol
P → set of Production Rule.

PAGE :
DATE : / /

$V = \{S, A\}$
$T = \{a, b, c, d\}$
$S = \{S\}$
$P = \{ S \rightarrow cAd$
$A \rightarrow ab \mid d$

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

E is starting symbol
+, *, id, ε, c,) is T.S.
N.T.S = { E, E', T, T', F }.
Does the given grammar satisfy the below sequence: syntax
id + id * id ?

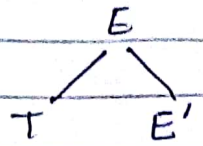
→ Top-Down
Grammar → Syntax.
root → leaf
leftmost derivatives first.

Bottom-Up
Syntax → Grammar.
leaf → root

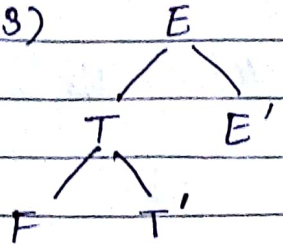
Syntax tree / Parse tree.

1) E

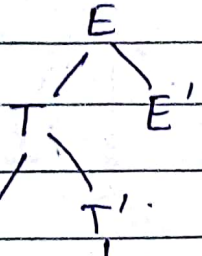
2) E



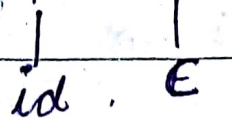
3) E



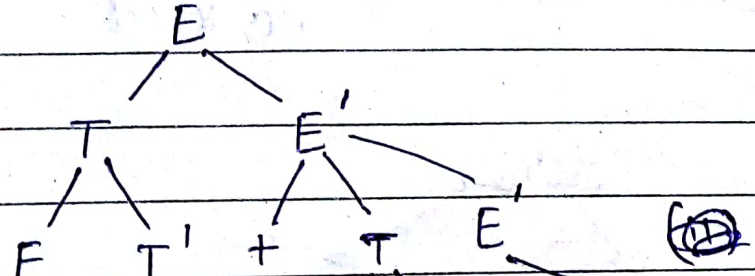
4) E



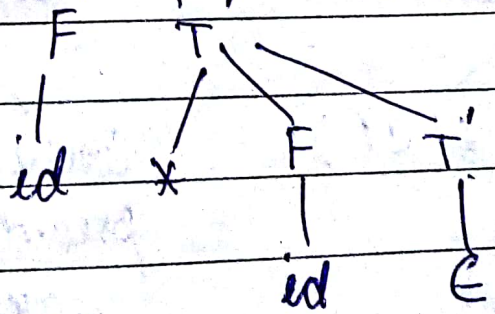
5)



6)



7)



8)

9)

10)

id ϵ id * id ϵ ϵ

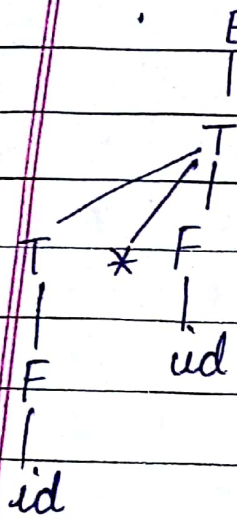
id + id * id.

PAGE :
DATE : / /

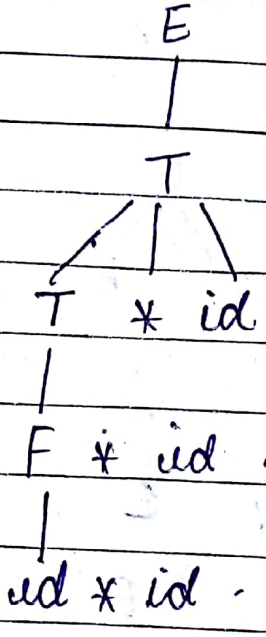
* Grammar : $E \rightarrow E+T \mid T$
$T \rightarrow T*F \mid F$
$F \rightarrow (E) \mid id$.

Check : id * id .

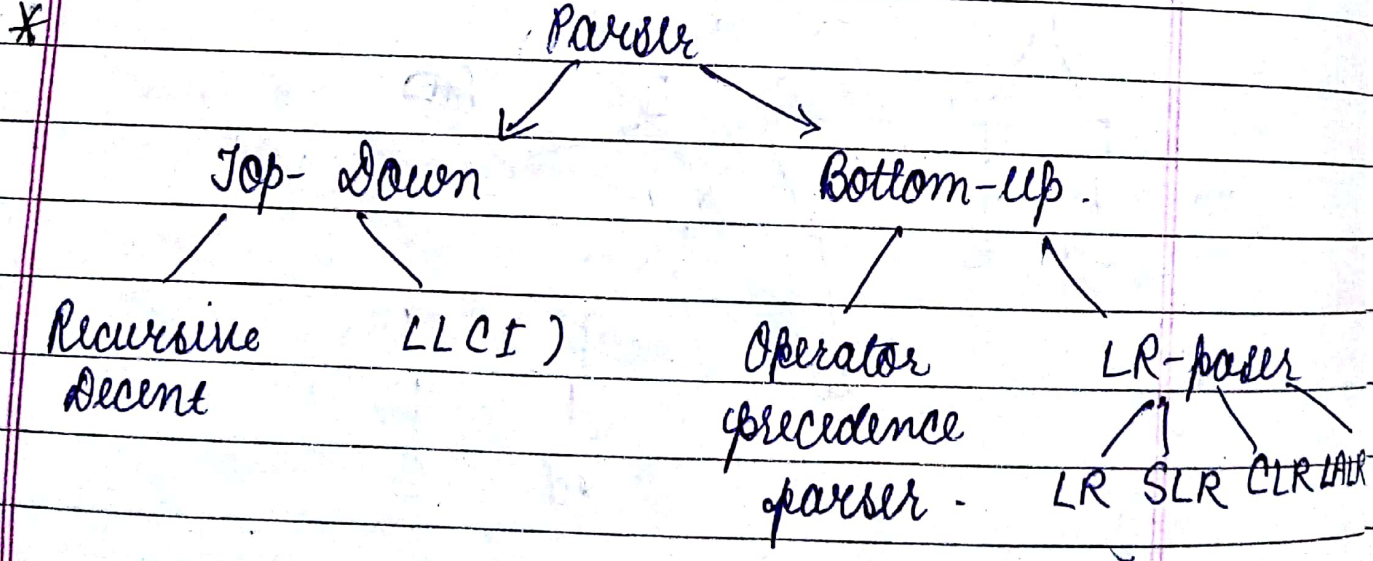
→ Top-down



Bottom-up



*



Operators:
First
Follow

First (α) :

(i) First (α) = α if α is terminal symbol

(ii) First (ϵ) = ϵ

(iii) 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) ∪ First ($x_2 x_3$)

Grammar:

$S \rightarrow aABe$

$A \rightarrow a$

$B \rightarrow c \mid d$

* We calculate first ( ) of non-terminal symbol.

i.e, we calculate :

First (S) = a.

First (A) = First (a) = a

First (B) = c, d.

First (S) = First (aABe)
= First (a)
= a.

PAGE :
DATE : / /

* $S \rightarrow (L) \mid a$
$L \rightarrow SL'$
$L' = \epsilon \mid , SL'$

First (S) → (, a
First (L) → First (SL') = First (S) = (, a
First (L') → ., ϵ, ,

* $A \rightarrow aA \mid bB \mid c \mid d$

Or. $A \rightarrow aA$ First (aA) = First (a) = a
$A \rightarrow bB$ First (bB) = First (b) = b
$A \rightarrow c$ First (A) = First (c) = c
$A \rightarrow d$ First (A) = First (d) = d

∴ First (A) = {a, b, c, d}.

* S → aABe
B → c | d
A → a.

First (A) = First (a) = {a}
First (B) = {c, d}
First (S) = {a}

PAGE :
DATE : / /

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

First (E) = First (TE') = First (T) = { id, (}.

First (T) = First (F) = { id, (}.

First (F) = { id, (}.
First (E') = { ϵ, + }.
First (T') = { ϵ, * }.

* $S \to aBDh$
$B \to cC$
$C \to bC / \epsilon$
$E \to g / \epsilon$
$F \to f / \epsilon$
$D \to EF$.

First (B) = { c }.
First (C) = { b, ϵ }.
First (E) = { g, ϵ }.
First (F) = { f, ϵ }.
First (D) = First (E) = { g, ϵ }
First (S) = { a }.

First (F) = { f, ϵ }
∴ First (D) = { g, ϵ } − ϵ ∪ { f, ϵ }
= { g, f, ϵ }

PAGE :
DATE : / /

* $S \rightarrow (L) / a$
$L \rightarrow SL'$
$L' \rightarrow \epsilon / , SL'$
First (S) = { (, a }
First (L) = First (S) = { (, a }.
First (L') = { ϵ, , }.

* Follow (A)
(i) If A is starting symbol
then Follow (A) = $
(ii) If $x \rightarrow \alpha AB$
then Follow (A) = First (B)
(iii) If $x \rightarrow \alpha A$ or $x \rightarrow \alpha AB$
$B \rightarrow \epsilon$
then Follow (A) = Follow (x)

* Follow (S) = Follow (L') = $, , ,)

Follow (L) = First (')') =)
Follow (L') = Follow (L) =)

Follow (S) = Follow (SL') = First (L') = ϵ, ,
= Follow (, SL') = Follow (L') =)
∴ Follow (S) = $, , ,)

PAGE :
DATE : / /

* $S \rightarrow aABe$
$A \rightarrow a$
$B \rightarrow c \mid d$.
First (S) = a
First (A) = a
First (B) = c, d.

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

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

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

Follow (F) = $, id,)
Follow (E) = Follow (F) = $, ~~id~~)

Follow(E') = Follow(E) = $,)
Follow(T) = $, First(E') = $, ϵ, +
But Follow ≠ ϵ. Hence,
Follow(T*) = $, Follow(E') = $, +,)

Follow(T') = Follow(T) = $, +,)

Follow(F) = $ ∪ First(T')
= $, ϵ, *
But ϵ ∉ Follow.
∴ Follow(F) = $, * ∪ Follow(T')
= $, *, +,)

* Left Recursion:
If $A \to A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \ldots \mid \beta_1 \mid \beta_2 \mid \beta_3 \ldots$

$$A \to \beta_1 A' \mid \beta_2 A' \mid \ldots$$
$$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' \ldots \mid \epsilon$$

* $E \to E+T \mid T$
It is in left recursion.
$A = E$
$\alpha_1 = +T$
$\beta_1 = T$

$\Rightarrow E \to TA'$ or TE'
$E' \to +TE' \mid \epsilon$

PAGE
DATE

* $F \rightarrow F * T \mid T$
$F \rightarrow TF'$
$F' \rightarrow *TF' \mid \epsilon$

$A = F$
$\alpha_1 = *T$
$\beta_1 = T$

* $A \rightarrow A\alpha \mid \beta \Rightarrow \{ A \rightarrow \beta A' \;\; A' \rightarrow \alpha A' \mid \epsilon \}$.

* LL(1) Parser (Top-Down)
  - Left to right string parsing
  - use left most derivative

Procedure:
(i) Make grammar free from left recursio
(ii) Find first & follow.
(iii) Create table as:-

Non-terminal \ Terminal	a	b	*	+	$
S					
A					
B					

(i) Add $A \to \alpha$ under $M[A, b]$ where $b \in First(A)$

If First(A) contain ϵ, then add
$A \to \alpha$ under $M[A, C]$
$A \to \epsilon$ where $C \in Follow(A)$

Q.

	First	Follow
$S \to (L) / a$	(, a	$, , ,)
$L \to SL'$	(, a	)
$L' \to , SL' / \epsilon$	, , ϵ	)

	a	(	)	,	$
S	$S \to a$	$S \to (L)$			
L	$L \to SL'$	$L \to SL'$			
L'			$L' \to \epsilon$	$L' \to , SL'$	

No conflict, hence LL(1) parser

Q. $E \to E + T / T$
$T \to T * F / F$
$F \to id / (E)$.

~~First~~ Follow

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

$S \to S\alpha_1 / S\alpha_2 / \beta_1 / \beta_2$

$\Downarrow$

$S \to \beta_1 S' / \beta_2 S'$
$S' \to \alpha_1 S' / \alpha_2 S' / \epsilon$

	First	Follow
$E \to TE'$	id, C.	$,)
$E' \to +TE' / \epsilon$	+, ϵ	$,)
$T \to FT'$	id, C	+, $,)
$T' \to *FT' / \epsilon$	*, ϵ.	+, $,)
$F \to id / (E)$.	id, C	*, +, $,)

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

∴ There is no conflict, hence LL(1) parser no cell contains more than 1 production.

		First	Follow
Q.	$S \to iEtSS' / a$	i, a.	$, e.
	$S' \to eS / \epsilon$	e, ϵ	$, e.
	$E \to b$.	b	t.

✗ $S \to S\alpha_1 / S\alpha_2 / S\alpha_3 \ldots / \beta_1 / \beta_2$.

$S = S$

$\alpha_1 =$

$S \to aS'$

$S' \to iEtS' / \epsilon$

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

conflict

∴ Not LL(1) parser.

NOTE:

(i) If $A \to \alpha_1 | \alpha_2 | \alpha_3$

then it said to be LL(1) iff

$First(\alpha_1) \cap First(\alpha_2) = \phi$

$First(\alpha_2) \cap First(\alpha_3) = \phi$

$First(\alpha_3) \cap First(\alpha_1) = \phi$

(ii) If $A \to \alpha_1 | \alpha_2 | \epsilon$

then it is said to be LL(1) iff

$First(\alpha_1) \cap Follow(A) = \phi$

$Follow(A) \cap First(\alpha_2) = \phi$

$First(\alpha_1) \cap First(\alpha_2) = \phi$

Q. | | First | Follow |
|---|---|---|
| $S \to E/a$ | a | |
| $E \to a$ | a | |

$First(E) \cap First(a)$

$\{a\} \cap \{a\} \neq \phi$

Hence, not LL(1) parser.

Q. $S \to aSA \mid \epsilon$ — First: a, ϵ — Follow: $\$, a, c$

$A \to c \mid \epsilon$ — First: c, ϵ — Follow: $\$, a, c$

First(aSA) ∩ Follow(S)

$\{a\} \cap \{\$, c\} = \phi$

First(c) ∩ Follow(A)

$\{c\} \cap \{\$, c\} \neq \phi$

Hence, not LL-1 parser.

Q. =

	First	Follow
$S \to aABb$	a	
$A \to a \mid \epsilon$	a, ϵ	d, b
$B \to d \mid \epsilon$	d, ϵ	b

First(a) ∩ Follow(A)

$= \{a\} \cap \{d, b\} = \phi$.

First(d) ∩ Follow(B)

$= \{d\} \cap \{b\} = \phi$.

∴ LL(1) parser.

PAGE :
DATE : / /

* $LR^{(K)}$ Parser :
  - Bottom-up parser.
  - 'L' stands for left-to-right scanning.
  - 'R' stands for constructing rightmost derivative in Reverse order.
  - 'K' → no. of lookahead symbol.

* SLR (simple LR)
  LALR (Lookahead LR)
  CLR (Canonical LR)

Let G1 be a grammar with start symbol S. then augmented grammar 'G1' with a new start symbol S' such that S' → S.

- augmented grammar is used to announce the acceptance of input.

* G1 : S → AA
       A → aA | b

G1' : S' → S
      S → AA.
      A → aA | b.

} augmented Grammar.

* LR item : (contains the production rules)

$S' \to S$

$S \to AA$

$A \to aA \mid b$

* Closure of any item :

Closure (I) = 1. add item I to closure (I)

2. If $\alpha \to \beta . A \alpha$ is item (I)

and $A \to BC$

then, add $A \to .BC$ to closure (I)

• Repeat 1 & 2 for newly added item.

* $S \to AA$

$A \to aA \mid b$.

Closure ($S' \to .S$)

$= S' \to .S$

$S \to .AA$

$A \to .aA \mid .b$

Q: $E' \to E$

$E \to E + T / T$

$T \to T * F / F$

$F \to (E) / id$

Closure ($E' \to \cdot E$)

$E' \to \cdot E$

$E \to \cdot E+T \mid \cdot T$

$T \to \cdot T * F \mid \cdot F$

$F \to \cdot (E) \mid \cdot id$

* LR Algo for G :

(i) Create G'

(ii) Find iclosure of ($S' \to \cdot S$) and say it item I_0.

Q.n: $S \to AA$

$A \to aA \mid b$.

→ $G' =$ $S' \to S$

$S \to AA$

$A \to aA \mid b$.

Closure ($S' \to \cdot S$) = $S' \to \cdot S$

$S \to \cdot AA$

$A \to \cdot aA \mid \cdot b$. } I_0

$S' \to \cdot S \xrightarrow{S} S' \to S\cdot$ I_1

$S \to \cdot AA \xrightarrow{A} S \to A\cdot A$, $A \to \cdot aA \mid \cdot b$ I_2

$A \to \cdot aA \xrightarrow{a} A \to a\cdot A$, $A \to \cdot aA \mid \cdot b$ I_3

$A \to \cdot b \xrightarrow{b} A \to b\cdot$ I_4

$I_2 \xrightarrow{A}$ $S \to AA.$ I_5

$I_2 \xrightarrow{a}$ $A \to a.A$, $A \to .aA \mid .b$ (I_3)

$I_2 \xrightarrow{b}$ $A \to b.$ (I_4)

$I_3 \xrightarrow{A}$ $A \to aA.$ I_6

$I_3 \xrightarrow{a}$ $A \to a.A$, $A \to .aA \mid .b$ (I_3)

$I_3 \xrightarrow{b}$ I_4.

* Shift & reduce :

Reduce : item at last.

I_1 , I_5 , I_6.

Q. $E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid id$.

→ $G' =$ ~~$S' \to .S$~~ $E' \to .E$

$E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid id$.

Closure $(E' \to .E) = E' \to .E$.

$E \to .E+T \mid .T$

$T \to .T*F \mid .F$

$F \to .(E) \mid .id$

PAGE :
DATE : / /

$E' \to \cdot E \xrightarrow{E} E' \to E\cdot$; $E \to E\cdot + T$ $\Big\}$ I_1.

$E \to \cdot E + T \xrightarrow{E} E \to E\cdot + T$

$E \to \cdot E + T \xrightarrow{T} E \to T\cdot$; $T \to T\cdot * F$ $\Big\}$ I_2.

$E \to \cdot T \xrightarrow{T} E \to T\cdot$

$T \to \cdot T * F \xrightarrow{T} T \to T\cdot * F$.

$T \to \cdot F \xrightarrow{F} T \to F\cdot$ I_3.

$F \to \cdot (E) \xrightarrow{(} F \to (\cdot E)$
$E \to \cdot E + T \mid \cdot T$
$T \to \cdot T * F \mid \cdot F$, $F \to \cdot (E) \mid \cdot id$

$F \to \cdot id \xrightarrow{id} F \to id\cdot$ I_4.

* Difference between LL and LR Parser :

LL	LR
(i) Top-Down parser	(i) Bottom-up parser.
(ii) leftmost derivation	(ii) Rightmost derivation in reverse order.

$S \rightarrow AB$

derivative of S in AB.

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

$T \rightarrow T * F / F$.

$F \rightarrow id / (E)$

Check if id * id can be generated by the given grammer.

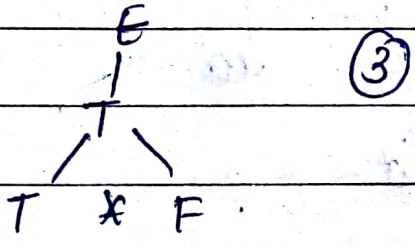
Using LL: (left most derivative)

E — (1)

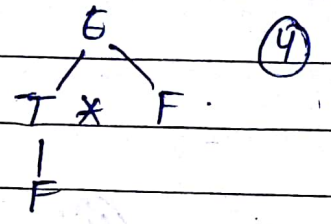
E

E — (2)

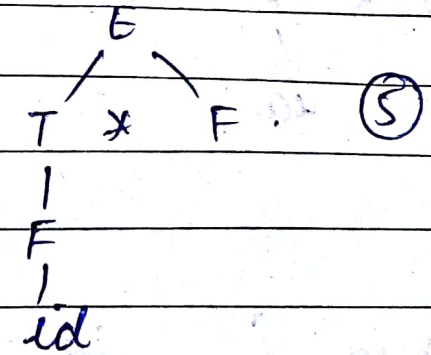
T



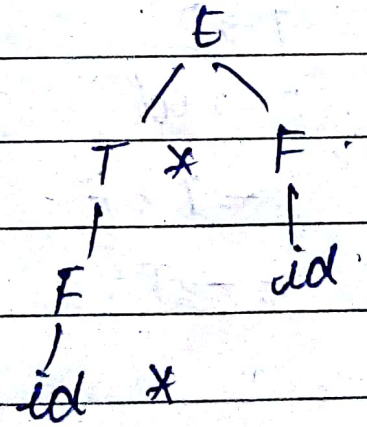
(3)



(4)



(5)



(6)

PAGE :
DATE : / /

Using LR:

Bottom-up

id * id (1)

F (2)
id * id

T * id (3)
F
id

T * F (4)
F id
id

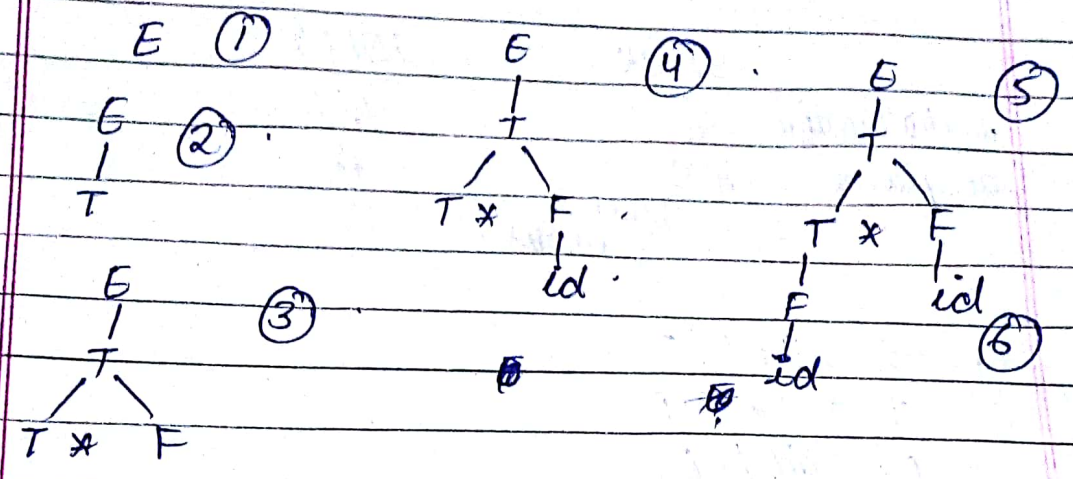
T (5)
T * F
F id
id

E (6)
T
T * F
F id
id

*	id * id	id	F → id
	F * id	F	T → F
	T * id	id	F → id
	T * F	T * F	T → T * F
	T	E	E → T
	E		

Table for Bottom-up.

* Topdown using rightmost derivation.



* Shift-Reduce Parser:

Four possible action :-

(i) Shift → shift the next i/p symbol on top of stack.

(ii) Reduce → The right end of the string to be reduced must be at the top of the stack.

(iii) Locate left end of the string within the stack and decide with what non-terminal to replace the string.

(iii) Accept → Successful completion of parsing.

(iv) Error → Discovery of a syntax error and call error recovery routine.

	STACK	INPUT
initialization	$	w$
acceptance	$S	$

S → starting symbol.

Eg - $E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow id \mid (E)$

$w = id * id$

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

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

System Programming

* Shift-Reduce Parser:

ACTION:
- Shift
- Reduce
- Accept
- Error

Two types of Item:
(i) Kernel item: The initial item $S' \to .S$ and all item whose '.' are not left end.
Eg.- $S' \to .S$, $S \to A.$
(ii) Non-Kernel item: Except $S' \to .S$, where '.' on left side.

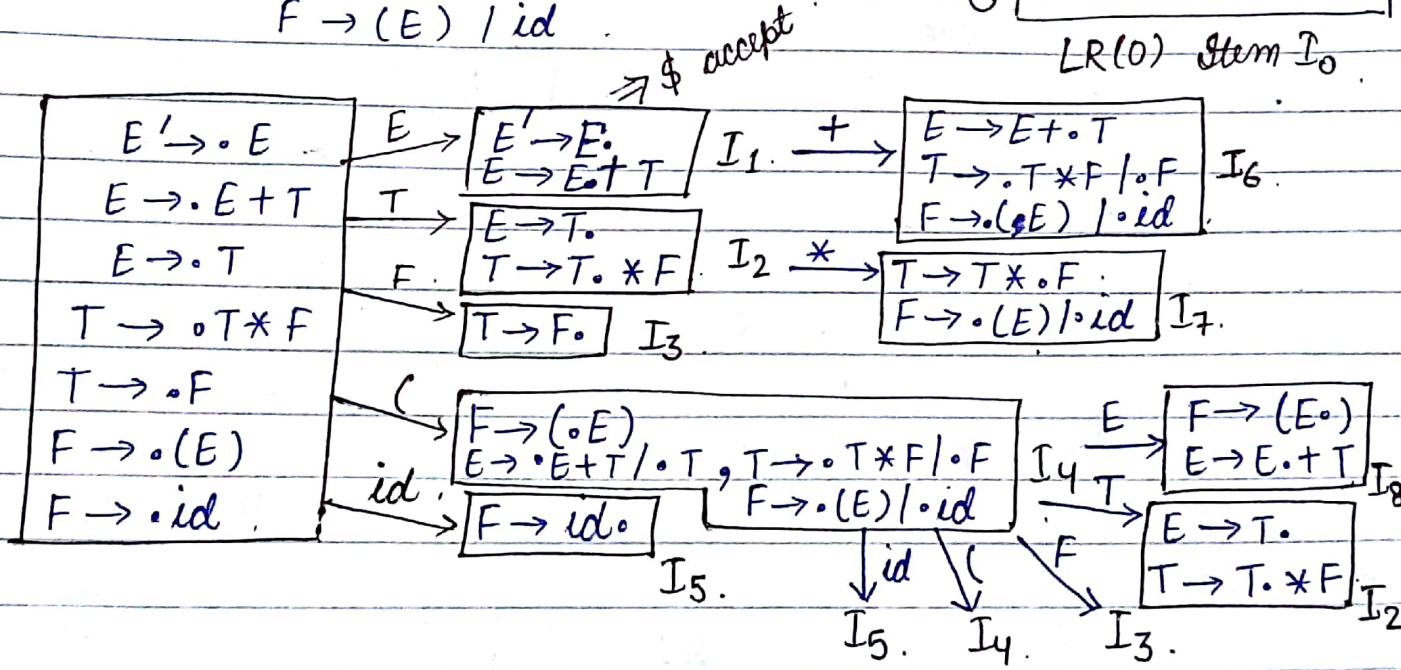
Q. $E \to E+T \mid T$
$T \to T*F \mid F$
$F \to (E) \mid id$

→ Augmented grammer:
$E' \to E$
$E \to E+T$
$E \to T$
$T \to T*F \mid F$
$F \to (E) \mid id$

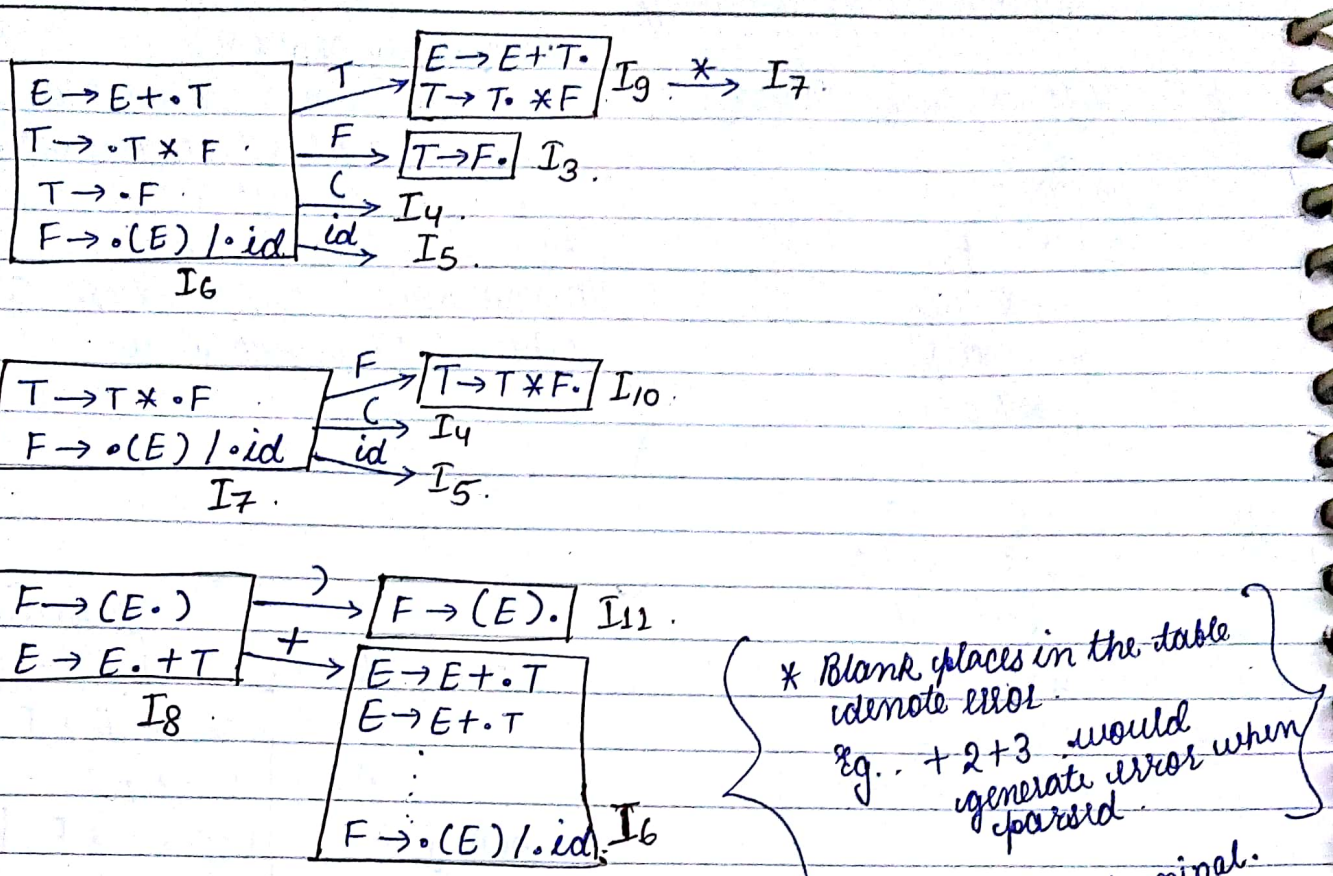
Closure $(E' \to .E) =$ Kernel item $E' \to .E$
Non-Kernel item: $E \to .E+T$, $E \to .T$, $T \to .T*F \mid .F$, $F \to .(E) \mid .id$

LR(0) Item I_0

Shift:



* If there is non-terminal symbol, then we find its closure and mark the item.



* Blank places in the table denote error.
eg. +2+3 would generate error when parsed.

List of items	action (shift) — terminal						goto — Non-terminal		
	id	+	*	(	)	$	E	T	F
0	S_5			S_4			1	2	3
1		S_6				accept			
2	r_2	r_2	S_7/r_2	r_2	r_2	r_2			
3	r_4	r_4	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	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	r_1	S_7/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			

** ◯ Not LR(0) parser.

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 each container contain only single action then it is LR(0) parser. (cont.)
else, not LR(0) parser.

ACTION ~~GOTO~~

* Simple LR:
We fill reduce entry only in the follow of production.

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

Follow(E) = +, $,)
Follow(T) = *, $,), +.
Follow(F) = Follow(T) = *, $,), +

	id	+	*	(	)	$	E	T	F.
0	s5			s4.			1	2	3.
1		s6				accept			
2		r_2	s7		r_2	r_2			
3		r4	r4		r4	r4.			
4	s5.			s4.			8	2	3.
5		r6	r6		r6	r6.			
6	s5			s4				9	3
7	s5			s4.					10.
8		s6			s11.				
9		r_1	S7		r_1	r_1			
10		r3	r3		r3	r3.			
11.		r5	r5		r5	r5.			

∵ No conflict, hence SLR parser.

shift → just push into stack
reduce → first pop then push

id * id $ (Input symbol)

STACK	Symbol	Input	Action
0	$	id * id $	shift
05 popped	$id popped	* id $	~~shift~~ Reduce F→id
~~06~~	~~$id *~~	~~id $~~	~~reduce~~
~~065~~	~~$id * id~~	~~$~~	~~shift~~
03 popped	$F popped	* id $	Reduce T→F
02	$T	* id $	
027	$T*	id $	shift
0275	$T*id	$	shift
027(10)	$T*F	$	Reduce F→id
027~~10~~	$T*F	$	Reduce T→T*F
02~~7~~	$T	$	
02	$T	$	
01	$E	$	Reduce E→T

Check if the given grammar is SLR or not.

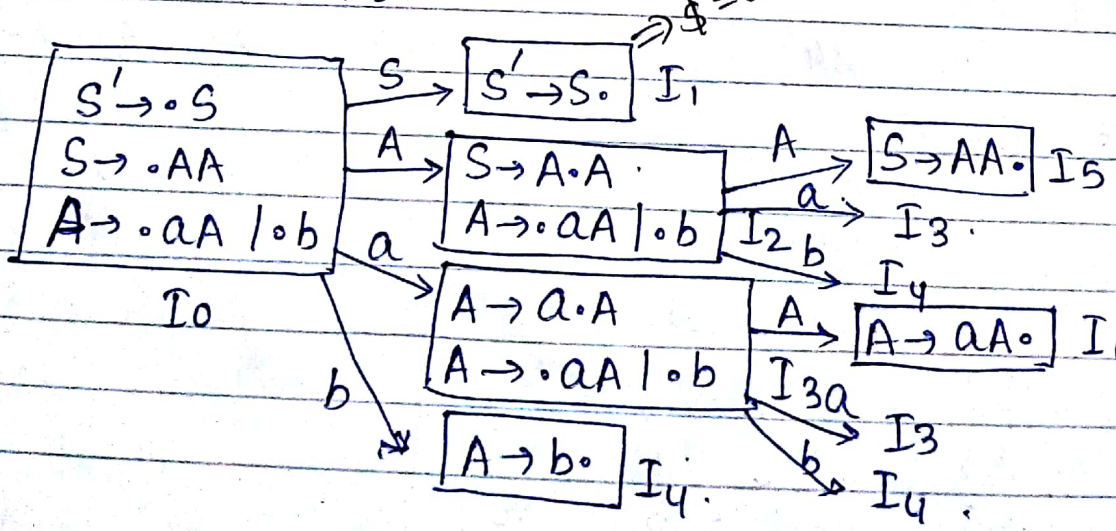
* (1) $E \to E+T \mid T$
$T \to TF \mid F$
$F \to F* \mid a \mid b$

(2) $S \to AA$
$A \to aA \mid b$

Check if LR or not.

→ $S' \to S$
$S \to AA$
$A \to aA \mid b$

Reduce
1.) $S \to AA$
2.) $A \to aA$
3.) $A \to b$



	a	b	$	S	A
0	s3	s4		1	2
1			accept		
2	s3	s4		5	
3	s3	s4		6	
4	r_3	r_3	r_3		
5	r_1	r_1	r_1		
6	r_2	r_2	r_2		

Since no conflict, hence LR parser.
also SLR parser.

* LR Parser Algo:

Let 'a' be the first symbol of w$
while(1)
{ Let S be the state on top of stack
if (ACTION [s, a] = shift t)
{ push t onto the stack;
Let 'a' be the next input symbol;
}
else if (ACTION [s, a] = reduce A→β
{ pop |β| symbol of the stack; pop
Let state 't' now on top of stack; [t=0 since 5 is popped]
push GOTO [t, A] onto the stack;
output the production A→B
}
else if (ACTION [s, a] = accept)
break;
else
call error-recovery routine;
} }

Let input = id * id.
initially S=0.
a = id.

* Input : id * id * id

State	symbol	INPUT	ACTION
0	$	id * id * id $	shift
05	$id	* id * id $	Reduce F→id
03	$F	* id * id $	Reduce T→F.
02	$T	* id * id $	shift
027	$T*	id * id $	shift
0275	$T*id	* id $	Reduce F→id
0210	$T*F	* id $	Reduce T→T*F.
027	$T	* id $	
02	$T	* id $	shift
027	$T*	id $	shift

→ Repeated again.

* LR(1) item :

LR(0) item + look ahead symbol

S' → .S , $ → look ahead symbol.

LR(0)
- LR.
- SLR.

LR(1)
- CLR
- LALR.

Closure (I) = 1. add I to closure (I)

2. If I is α → B . ABC , $

&&

A → EFG

then add A → .EFG , First (BC, $)

3. Repeat it for newly add member.

Eg.- G1 = S → AA

A → aA | b

Closure (S' → .S , $) = ?

Closure $(S' \to \cdot S, \$)$ =

$S' \to \cdot S, \$$

$S \to \cdot AA, \$$

$A \to \cdot aA, a/b$

$A \to \cdot b, a/b$

* $S' \to S$

$S \to CC$

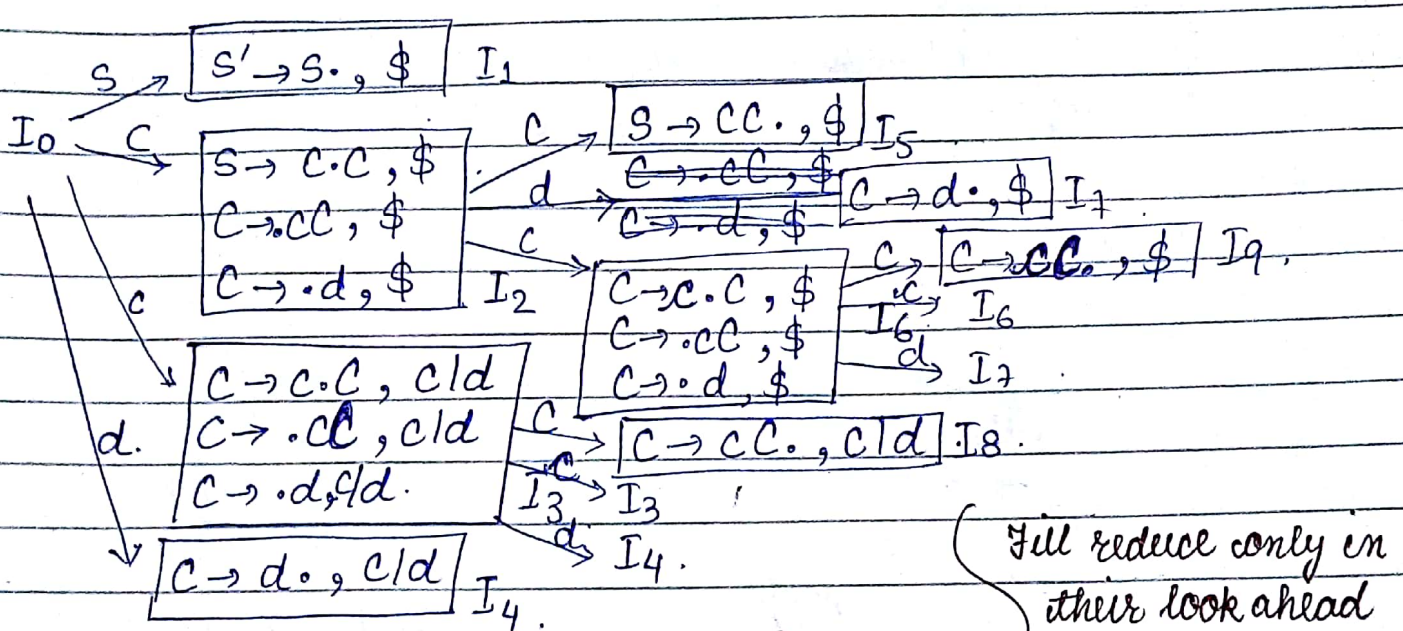
$C \to cC / d$

Closure $(S' \to \cdot S, \$)$ =

I_0:
- $S' \to \cdot S, \$$
- $S \to \cdot CC, \$$
- $C \to \cdot cC, c/d$
- $C \to \cdot d, c/d$

Reduce:

1. ~~$S' \to S$~~
2. $S \to CC.$
3. $C \to cC.$
4. $C \to d.$



Fill reduce only in their look ahead symbol.

State	ACTION c	ACTION d	ACTION $	GOTO S	GOTO C
0	s3	s4		1	2
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		

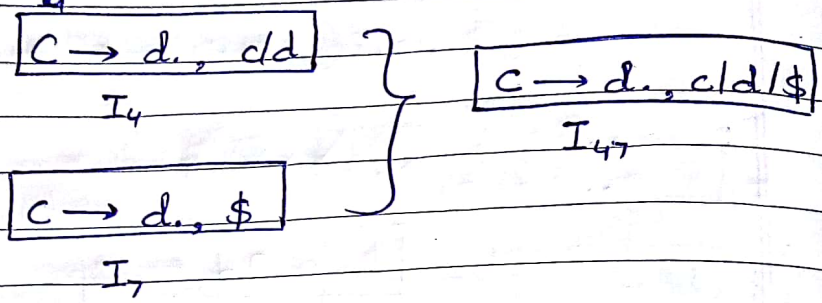
16/9/17

Date :
Page No.

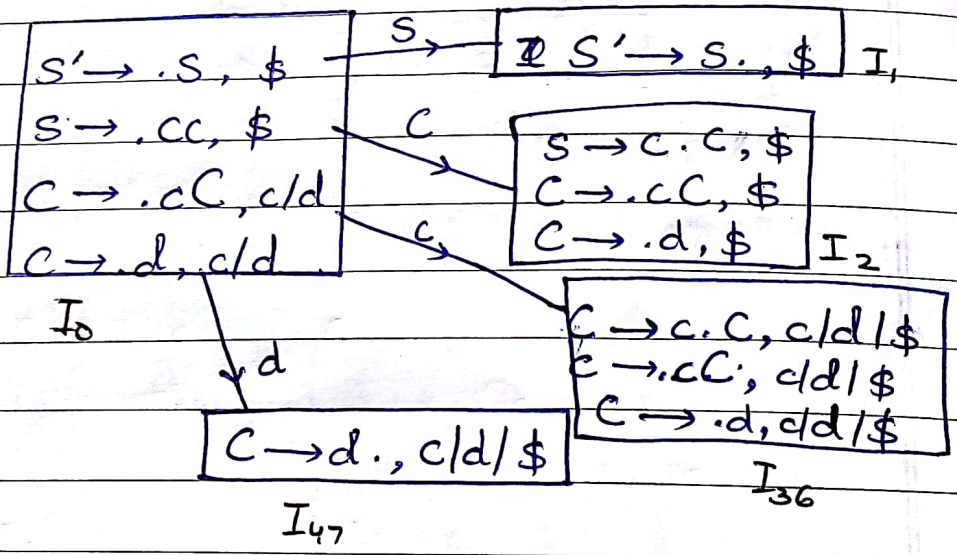
LALR

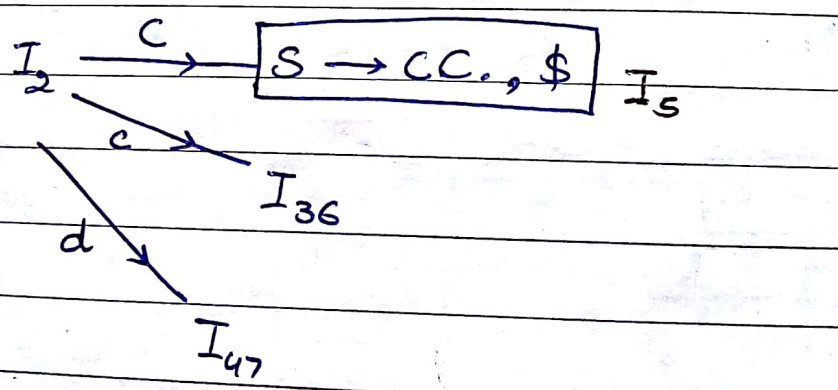
Ref Ques1- In LALR, merge ~~state~~ stage / item which have only diff. look ahead symbol.

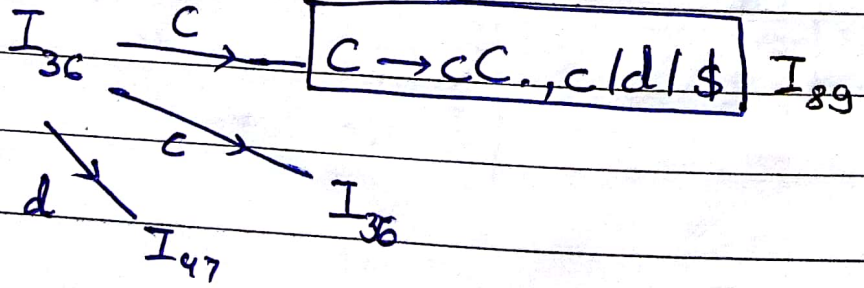
eg.



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







STATE
0
1
2
36
47
5
89

Ques3-

Date :
Page No.

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- $S' \rightarrow S$ r_1

$S \rightarrow Aa$ r_2

$S \rightarrow bAc$ r_3

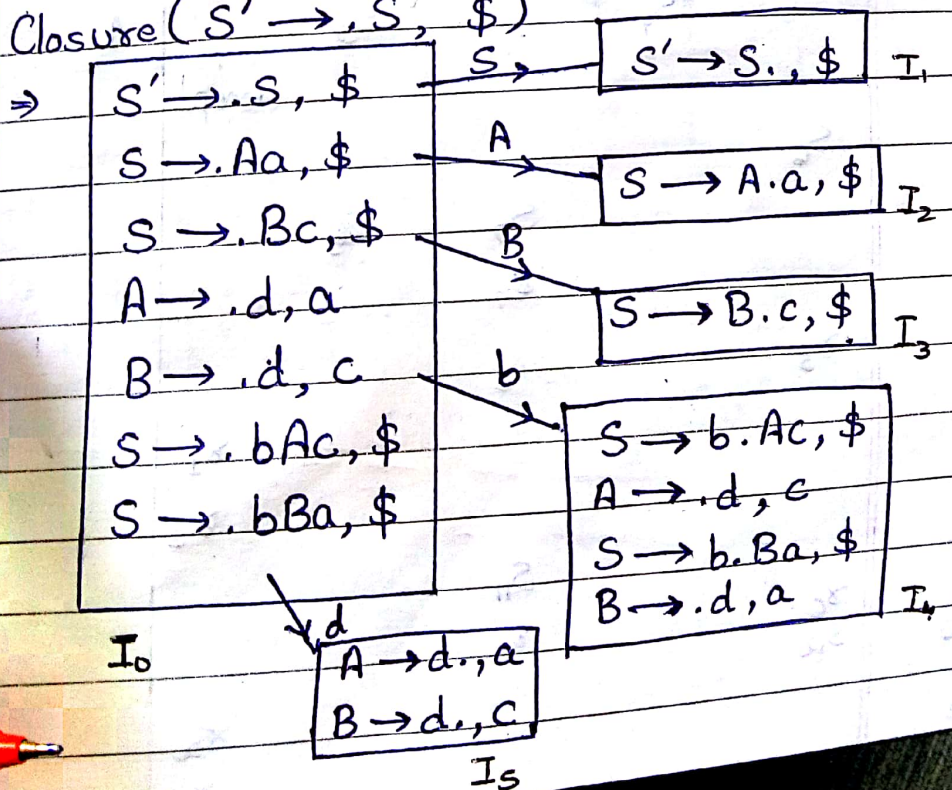
$S \rightarrow Bc$ r_4

$S \rightarrow bBa$ r_5

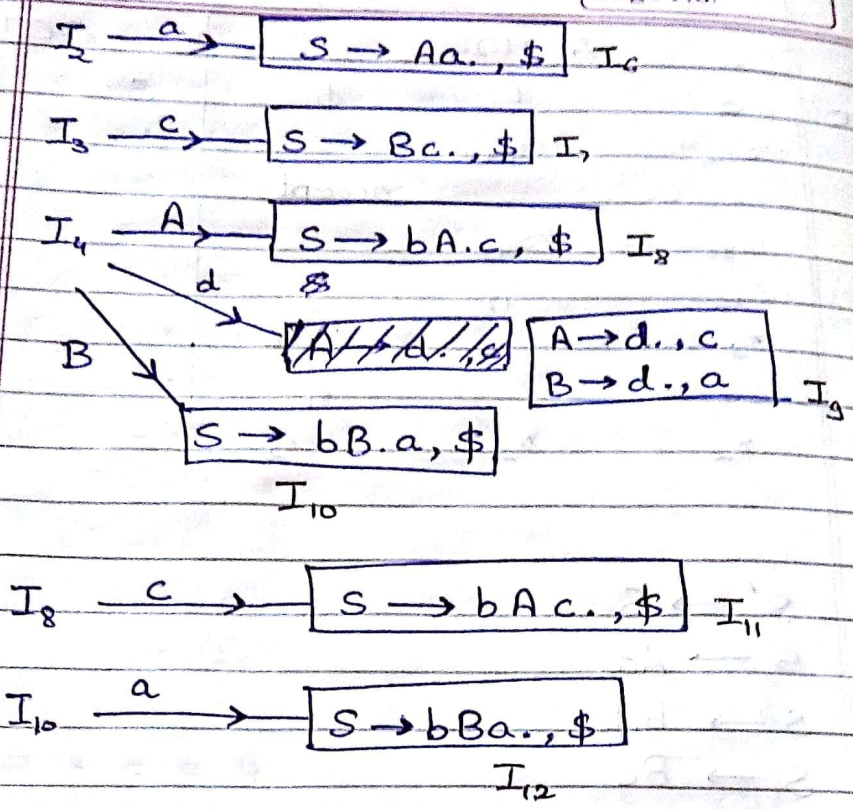
$A \rightarrow d$ r_6

$B \rightarrow d$

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



Date :
Page No.



STATE	ACTION a	b	c	d	$	GOTO S	A	B
0		S_4		S_5		1	2	3
1					accept			
2	S_6							
3			S_7					
4				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 → d., c/a B → d., c/a

I_{59}.

New Table.

STATE	ACTION a	b	c	d	$	GOTO 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.

* $S' \rightarrow S$.
$S \rightarrow aAd \mid bBd \mid aBe \mid bAe$
$A \rightarrow c$
$B \rightarrow c$

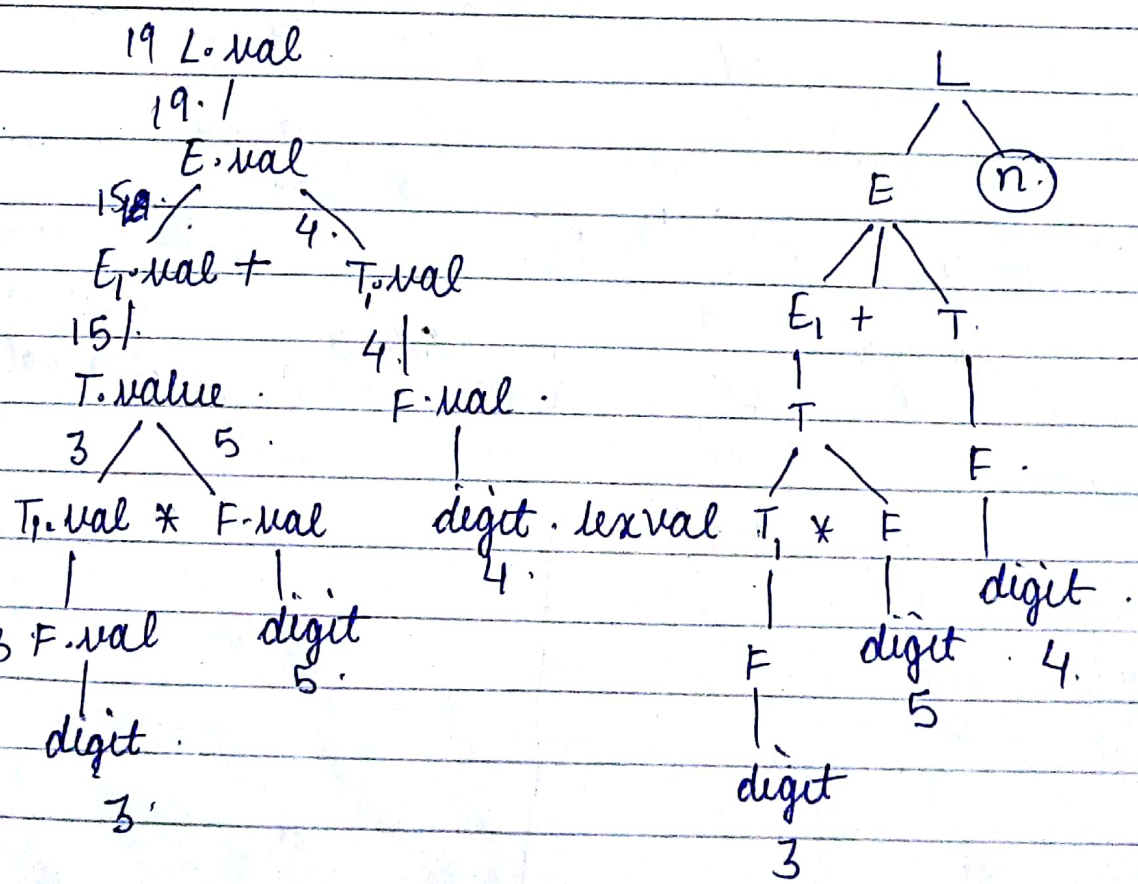
* Syntax-Directed Translated:

Q. $S \rightarrow S+T$

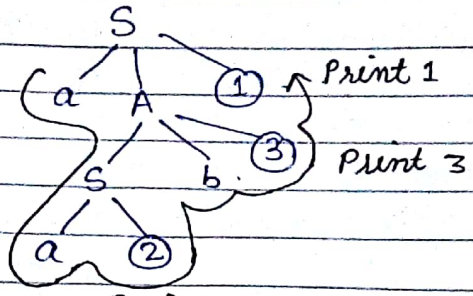
Calculator of '+' & '*'

$L \Rightarrow E\,n$ { L.value = E.val }
$E \rightarrow E_1 + T$ { E.val = E_1.val + T_1.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 (SDT)

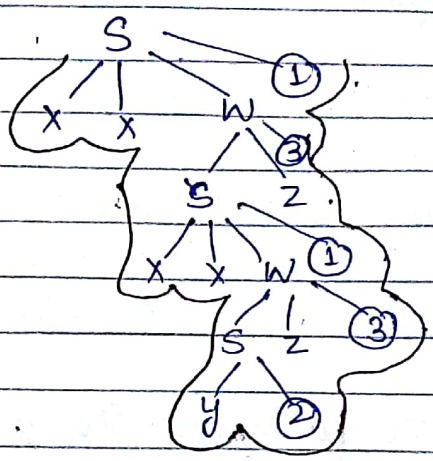


Q. $S \to aA$ {print 1}
$S \to a$ {print 2}
$A \to Sb$ {print 3}
Parse aab
Output = ?

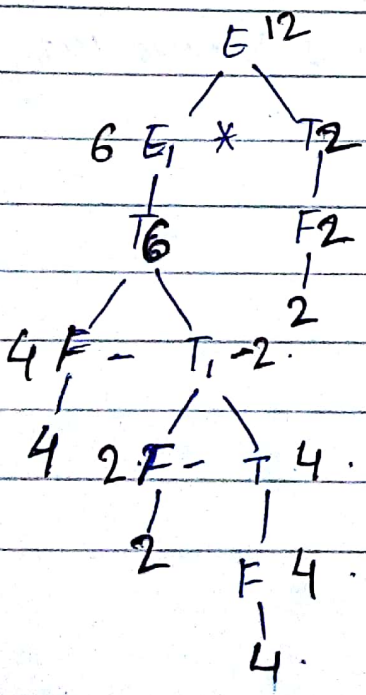


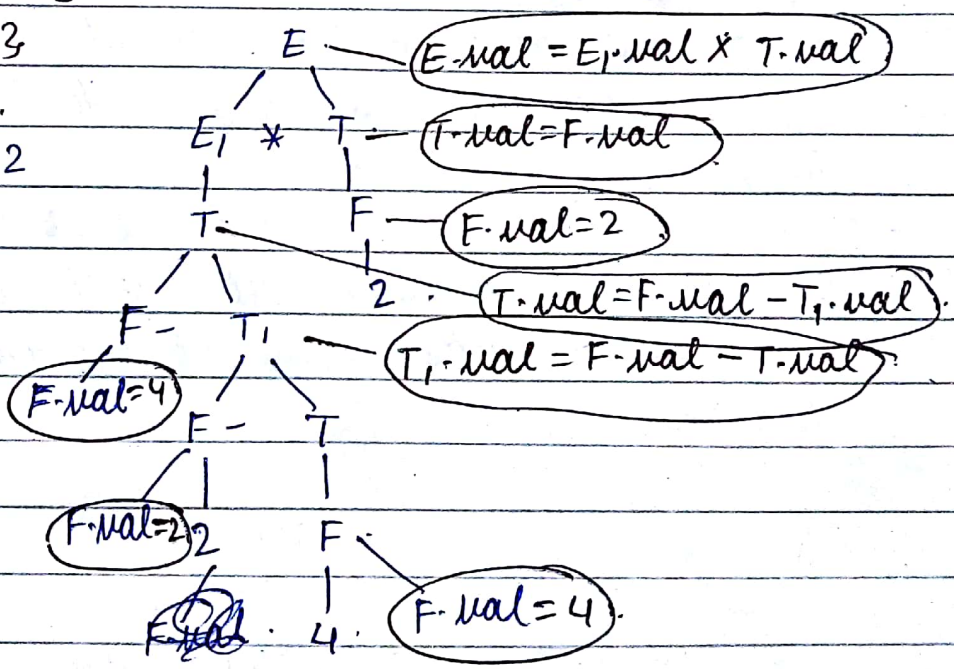
Print 2.3.1.

Q. $S \to XXW$ {print "1"}
$S \to y$ {print "2"}
$W \to Sz$ {print "3"}
i/p → xxxxyzz
Output = ?
→ Output : 23131.



Q. $E \to E_1 * T$ {E.val = E_1.val * T.val}
$E \to T$ {E.val = T.val}
$T \to F - T_1$ {T.val = T.val − T_1.val}
$T \to F$ {T.val = F.val}
$F \to 2$ {F.val = 2}
$F \to 4$ {F.val = 4}.
input → 4-2-4*2
Output = ?





Syntax-Directed Definitions (SDT):

- It is a context-free grammer together with attributes and rules.

* attributes are associated with grammer symbols and rules are associated with productions.

$X \to \alpha \cdot \{ X.a = F.a \}$ → Semantic Rule.

$X.a$: X → symbol, a → attributes; α → Production

* attributes are of many kinds : number, string, types, table reference.

* Types of attributes :

(i) Synthesized attributes
(ii) Inherited attributes

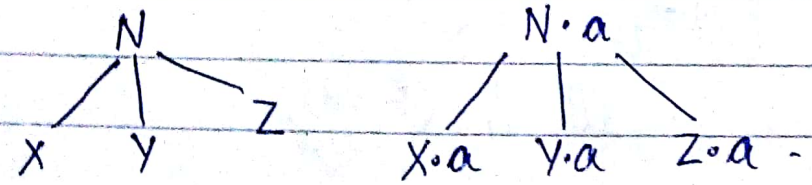
Parent class → child class.

* Synthesized attributes 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 as its head.

A synthesized attributes at node N is defined only in term of attributes value at the children of N and at N itself.



$N.a = f(X.a \mid Y.a \mid Z.a)$

* Inherited attributes for a non-terminal B :
at a parse tree node N is defined by semantic rule associated with the production at the parent of N.
- An inherited attribute at node N is defined only in terms of attribute value at N's parent, N itself and N's siblings

i.e, B.a

N.a N_1.a N_2.a .

* application of SDT :
(i) Converting infix to postfix
(ii) Converting infix to prefix .
(iii) Binary to decimal conversion.
(iv) DAG (Directed Acyclic Graph)
(v) Intermediate code generation

Let the given grammer is :

$L \rightarrow E\,n$.
$E \rightarrow \{print(+)\}\ E_1 + T$.
$E \rightarrow T$
$T \rightarrow \{print(*)\}\ T_1 * F$.
$T \rightarrow F$
$F \rightarrow (E)$
$F \rightarrow digit\ \{print(digit.lexval)\}$

i/p → 3 * 5 + 4 .
O/p → + * 3 5 4 .

→