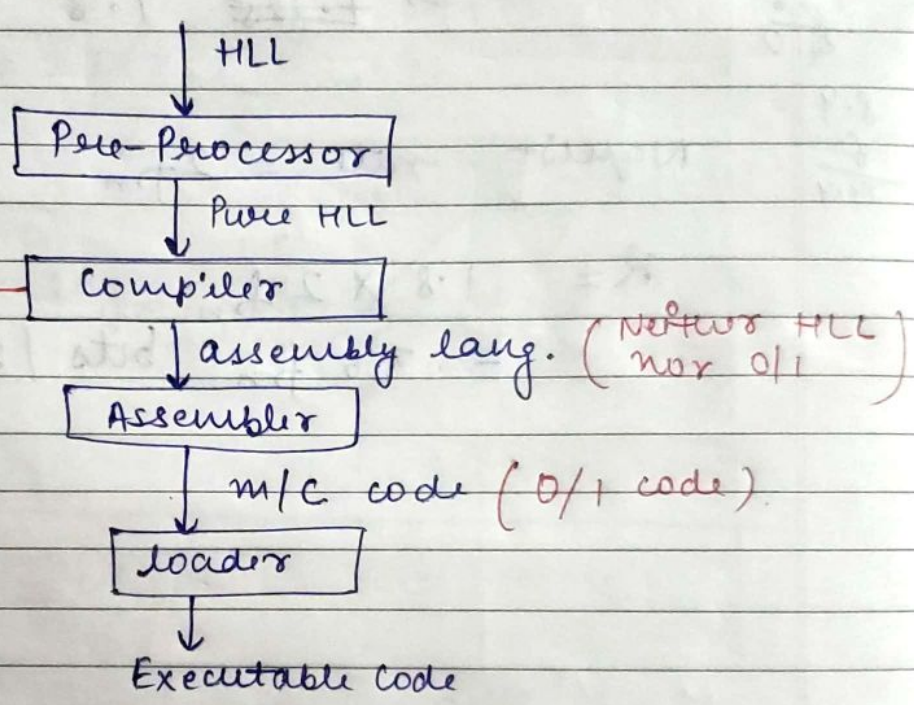
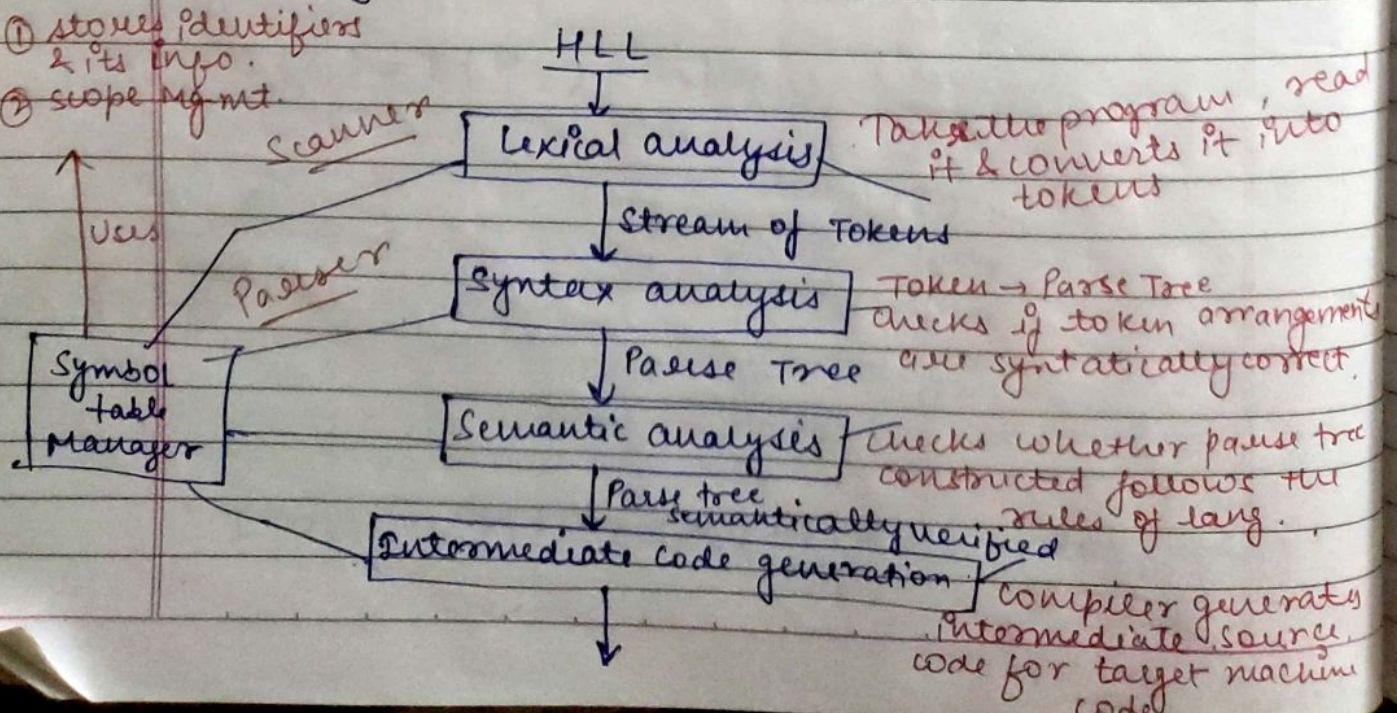
Compiler Design

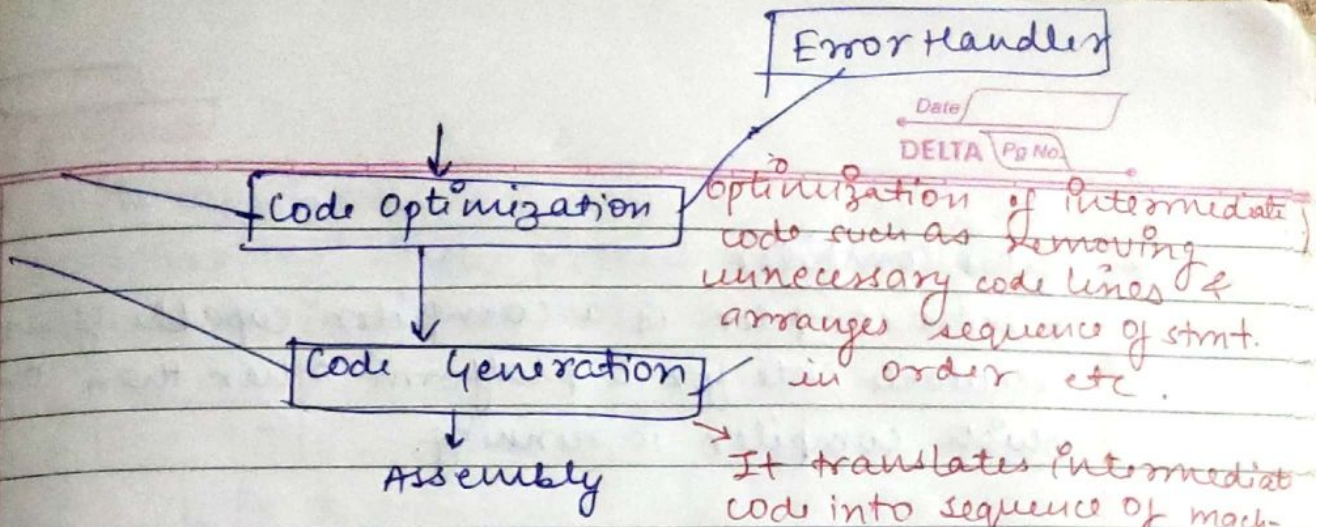
Unit - 1

* The compilation process is a sequence of various phases. Each phase takes input from previous stage, has its own representation of source program, and feeds its o/p to next phase of compiler.
It also reports errors present in source program as a part of its translation process.



* Phases of compiler :-





* Structure of compiler :

Two Modules

Front-end

a) Lexical analyzer
b) Semantic analyzer
c) Syntax analyzer
d) Intermediate code generator

Back-end

a) code Optimizer
b) code generator

} Synthesis

Analysis

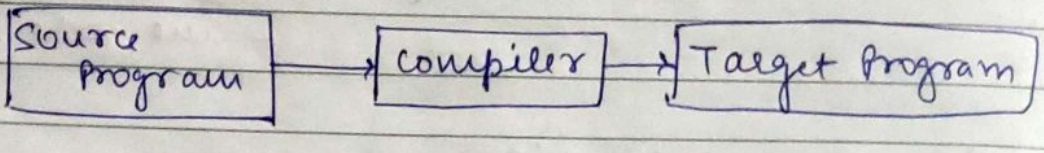
Linear – lexical analysis or scanning
Done by lexical analyzer or scanner

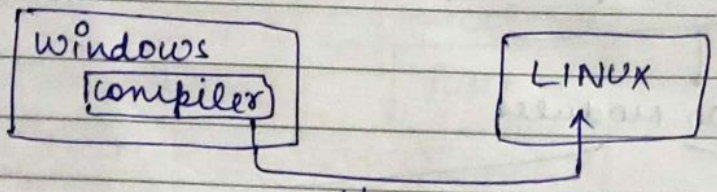
Hierarchical – Syntax analysis or Parsing
Done by syntax analyzer or Parser

» lexemes – Sequence of characters (alphanumeric)
» Tokens – Group of similar lexemes defined with a pattern.

* Cross compiler

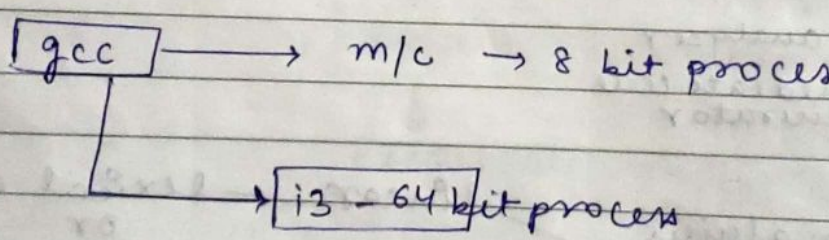
A cross compiler is a compiler capable of creating executable code for a platform other than one on which compiler is running.

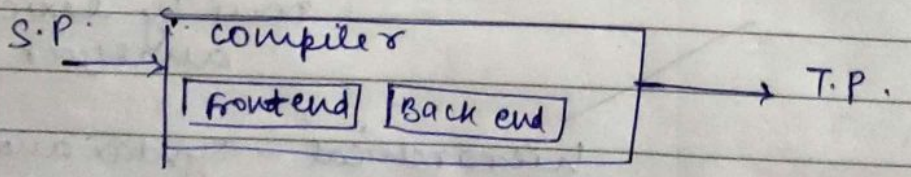




cross compiler / Retargeting

Eg - gcc



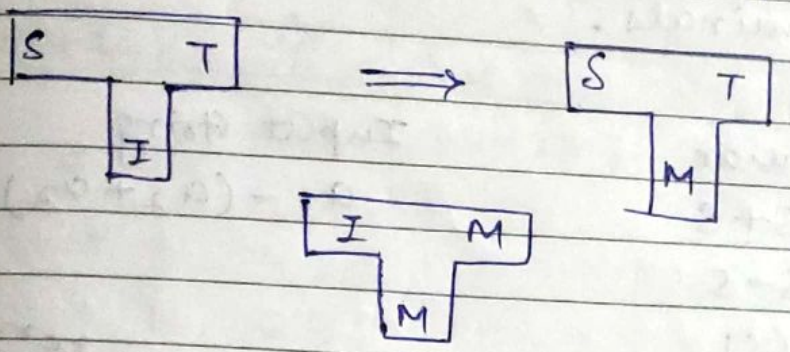


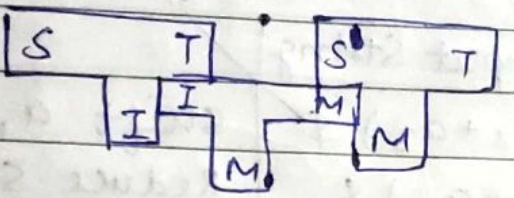
front end depends on source language; Since the language 'C' of gcc remains same but platform change, therefore, front end will remain same but backend will change.

Advantage:- We can still build code although hardware conditions of that machine are insufficient

* **Bootstrapping :-**

Bootstrapping is the process by which simple lang. is used to translate more complicated program.





$$S_M^T = S_I^T + I_M^M$$

Eg- gcc

* **Quick and dirty compiler :-**

A quick & dirty compiler are used to compile a code to check the functionality and correctness of a code. Once the written program is correct, so it is recompiled to make efficient. Its main function is to compile the codes as fast as possible.

* **Shift - Reduce Parsing :-** ($)

It is a process of reducing a string to start symbol of a grammar.

It uses stack to hold the grammar and an i/p to hold the string.

Steps: ① At shift action, the current symbol in i/p string is pushed to a stack.

② At each reduction, the symbol will be replaced by non-terminals.

Eg: Grammar

$S \to S+S$

$S \to S-S$

$S \to (S)$

$S \to a$

Input string

$a_1 - (a_2 + a_3)$

mandatory

Stack contents	Input String	Action
$	$a_1-(a_2+a_3)\$$	shift a_1
$\$ a_1$	$-(a_2+a_3)\$$	Reduce $S \to a$
$\$ S$	$-(a_2+a_3)\$$	shift −
$\$ S-$	$(a_2+a_3)\$$	shift (
$\$ S-($	$a_2+a_3)\$$	shift a_2
$\$ S-(a_2$	$+a_3)\$$	Reduce $S \to a$
$\$ S-(S$	$+a_3)\$$	shift +
$\$ S-(S+$	$a_3)\$$	shift a_3
$\$ S-(S+a_3$	$)\$$	Reduce $S \to a$
$\$ S-(S+S$	$)\$$	shift)
$\$ S-(S+S)$	$\$$	Reduce $S \to S+S$
$\$ S-(S)$	$\$$	Reduce $S \to (S)$
$\$ S-S$	$\$$	Reduce $S \to S-S$
$\$ S$	($)	Accept

Eg ②: Grammar

$E \to 2E2$

$E \to 3E3$

$E \to 4$

i/p string: 32423

Stack contents	i/p string	Action
$	~~1234~~ 32423 $	Shift 3
$3	2423 $	Shift 2
$32	423 $	Shift 4
$324	23 $	Reduce $E \to 4$
$32E	23 $	Shift 2
$32E2	3 $	Reduce $E \to 2E2$
$3E	3 $	Shift 3
$3E3	$	Reduce $E \to 3E3$
$E	$	Accept

* Operator Precedence Parsing :-

A grammar G is called an operator precedence grammar if it meets following cond^ns :-

1. There exist no production rule which contain ϵ (Epsilon) on its right hand side.
2. There exist no production rule which contains two non-terminals adjacent to each other on its RHS.

Eg :- $A \to A + A$; not adjacent

↓ non-T. ↓ Terminal → non-Terminal

Rules :

highest precedence – Terminals like a, b etc. and id

lowest precedence – $

$id \neq id$

$\$ \leftrightarrow \$ = A$

Q Consider the foll. grammar & construct an operator precedence parser

$E \rightarrow EAE \mid id$

$A \rightarrow + / *$

then parse the following string : id + id * id

Sol:

Step 1: check the grammar, if not operator precedence make it operator precendence grammar.

$\therefore E \rightarrow E+E / E*E / id$

Step 2: construct operator precedence table :

Terminal symbols are = { id , + , * , $ }

Table :

	id	+	*	$
id	–	>	>	>
+	<	>	<	>
*	<	>	>	>
$	<	<	<	A

Acc. to rules

- id ≠ id
- $ A $
- + > +
- + < *
- * > *

Step 3: Parsing the given string : id + id * id

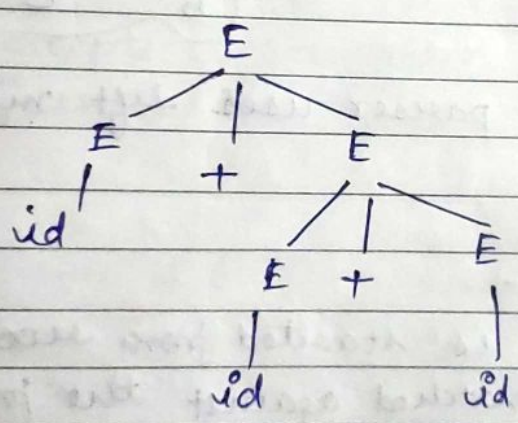
compare only terminals

Stack	Relation	i/p string	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
~~$ E + E~~	~~<~~	~~* id $~~	~~Reduce E → E+E~~

$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
$E	A	$	Accept

Note: Reduce only when stack side is greater (>) than i/p string.

step 4: Generating Parse Tree (Follow bottom up approach in table)



Types of Parser

- Top-down Parsing
  - Backtracking
  - Predictive Parsing
    - LL(1)
    - recursive descent
- Bottom up Parsing (shift-reduce parsing)
  - Operator precedence parsing
  - LR Parsing
    - SLR
    - LR
    - LALR

* Top- Down Parsing :-

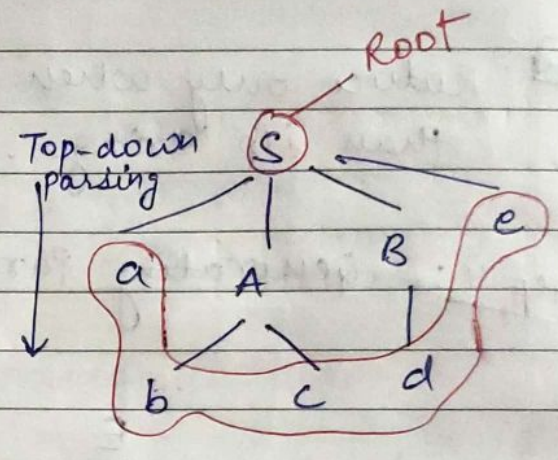
The process of construction of parse tree starting from root and proceed to children is called Top-down parsing i.e. starting from start symbol of grammar and reaching the i/p string.

Eg:

S → aABe
A → bc
B → d
w → abcde

start symbol

i/p string

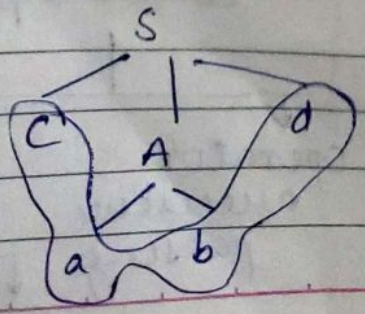


NOTE: Top down parser uses left most derivation.

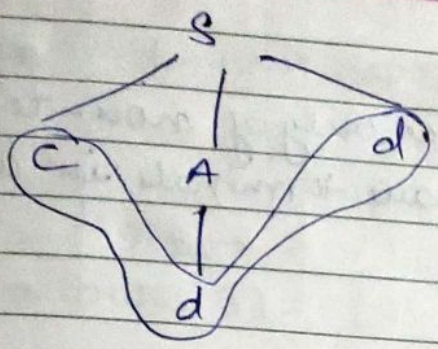
* Back-tracking :-

The parse tree is started from root node & its string is matched against the production rule for replacing them.

Eg:
S → CAd
A → ab | d
w → Cdd



String not matched, ∴ backtracked to A.



String matched.

>> Limitation of Backtracking : If the grammar has a large no. of alternatives (production rule) then cost of backtracking is high.

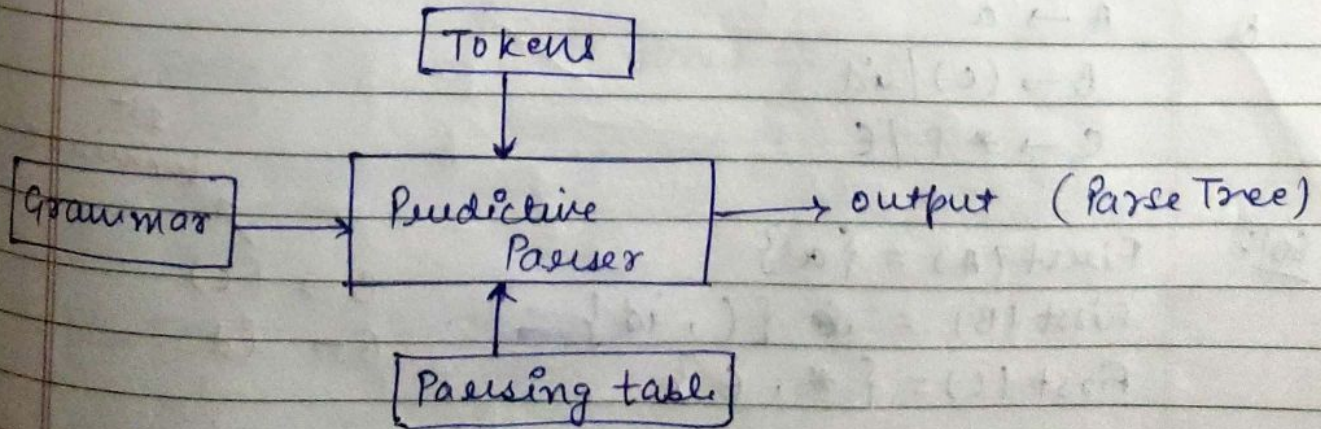
* Predictive Parsing :- or LL Parser

It accepts LL Grammar.

It is denoted as LL(k).

- k → no. of look aheads (tracking) (generally k = 1)
- L → i/p from left to right
- L → Left most derivation

A grammar G is LL(1) if there are two distinct productions.

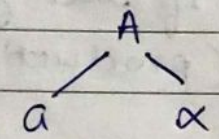


* **First and Follow :**

first and follow are only of non-terminals. and can only have terminals in answer.

>* **First** (left most)

case I: $A \rightarrow a\alpha$



First (A) = {a}

pehle hi terminal mil gya 'a' to aage kuch b ho wo ni dekhna.

case II: $A \rightarrow \epsilon$

First (A) = {ϵ}

case III: $A \rightarrow B$

$B \rightarrow C$

$C \rightarrow id$

First (A) = {id} (∵ Bcoz non-terminals cannot be first on any grammar.)

Q $A \rightarrow a$

$B \rightarrow (C) / id$

$C \rightarrow *T / \epsilon$

Sol: First (A) = {a}

First (B) = {(, id}

First (C) = {*, ϵ}

1st terminal

$B \rightarrow (C)$

$B \rightarrow id$

>> **Follow** (right most)

[Right most ka first = follow]

Case I: $A \to \alpha B \beta$

$\beta \to a$

Follow (B) = { a }

Eg: $A \to TE'$

$E' \to +F \mid id$

Follow (T) = { +, id }

Case II: $A \to \alpha B \beta$

$\beta \to \epsilon$

i.e. $A \to \alpha B$

Follow does not include ϵ (epsilon)

Follow (B) = { All Follow of A }

Eg: $E \to TE'$

given = Follow (E) = { $,) }

Follow (T) = { all follow of E }

= { $,) }

Case III: $A \to \alpha B \beta$

$\beta \to a \mid \epsilon$ → due to ϵ

∴ Follow (B) = { a, all follow of A }

Q

$E \to TE'$

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

$T \to FT'$

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

$F \to (E) / id$

	first	follow
E	(, id	), $
E'	+, ϵ	), $
T	(, id	+,), $
T'	*, ϵ	+,), $
F	(, id	*, +,), $

Q

$S \to iEtSS' / a$

$S' \to eS / \epsilon$

$E \to b$

	first	follow
S	i, a	e, $
S'	e, ϵ	e, $
E	b	t

Q

$E \to TQ$

$T \to FR$

$Q \to +TQ / -TQ / \epsilon$

$R \to *FR // FR / \epsilon$

$F \to (E) / id$

	first	follow
E	(, id	), $, +, -, (
T	(, id	+, -, (, id
Q	+, -, (, id	), $
R	*, /, (, id	+, -, (, id
F	(, id	*, /, (, id

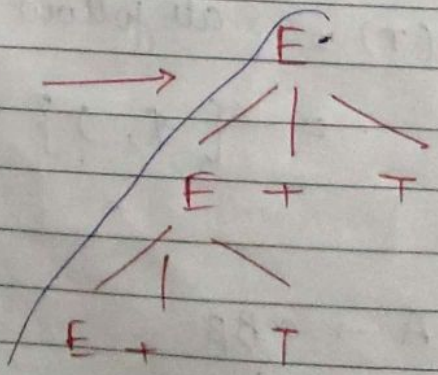
LL(1) Parser

Q

$E \to E+T / T$

$T \to T*F / F$

$F \to (E) / id$



left Recursion

>> Eliminating left recursion & left factoring:

Left recursion

Eg: $A \to A\alpha \mid \beta$ — general form

After eliminating left recursion

$A \to \beta A'$

$A' \to \alpha A' / \epsilon$

compiler can handle right recursion. but left recursion needs to be removed.

Q $E \to E+T / T$

α β

$\therefore E \to TE'$

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

(, id follow(Q), follow(R)

Q $T \to T*F / F$

Eliminating ....

$T \to FT'$

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

Left factoring

Eg: $S \to E+T / E*T / E \backslash T$

If compiler wants division, for that it had to open every production as it starts with same E, $\therefore$ we remove left factoring.

$S \to E+T / E*T / E / T$

$S \to ES'$

$S' \to +T / *T / /T$

Generalform: $A \rightarrow \alpha\beta_1 / \alpha\beta_2 / \alpha\beta_3$

then after eliminating left factoring

$A \rightarrow \alpha A'$

$A' \rightarrow \beta_1 / \beta_2 / \beta_3$

---

* LL(1) Parser:

$w = id + id * id$

Q
$E \rightarrow E+T / T$

$T \rightarrow T * F / F$

$F \rightarrow (E) / id$

Step 1: check for left recursion & left factoring.

Eliminating ~~rec~~ left recursion:

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) / id$

Step 2: find first and follow of new grammar.

	first	follow
E	(, id	), $
E'	+, ε	), $
T	(, id	+,), $
T'	*, ε	+,), $
F	(, id	*, +,), $

Step 3: Parsing Table generation

Non-Terminals	+	*	id	(	)	$
E			E→TE'	E→TE'		
E'	E'→+TE'				E'→ε	E'→ε
T			T→FT'	T→FT'		
T'	T'→ε	T'→*FT'			T'→ε	T'→ε
F			F→id	F→(E)		

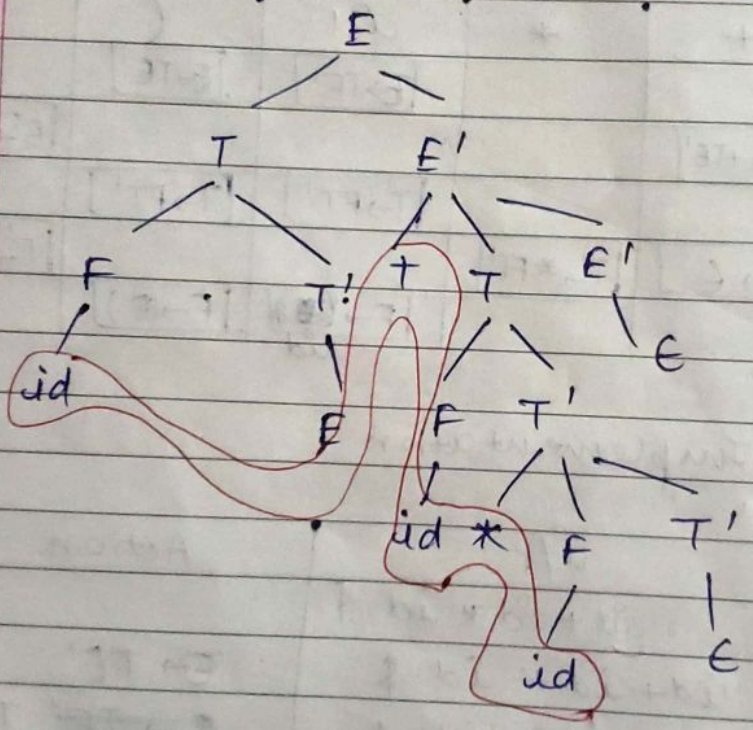
Step 4: Stack Implementation

not in case of Top-down

Stack	i/p	Action
E $	id + id * id $	
TE' $	id + id * id $	E→TE'
FT'E' $	id + id * id $	~~E→TE'~~ T→FT'
~~id~~ T'E' $	~~id~~ + id * id $	F→id
T'E' $	+ id * id $	~~Pop~~
E' $	+ id * id $	T'→ε
+TE' $	+ id * id $	E'→+TE'
TE' $	id * id $	
FT'E' $	id * id $	T→FT'
~~id~~ T'E' $	~~id~~ * id $	F→id
T'E' $	* id $	
*FT'E' $	* id $	T'→*FT'

Parse Tree

FT'E'$	id $	F → id
~~id~~ T'E'$	~~id~~ $	~~E' → ϵ~~
T'E'$	$	~~E' → ϵ~~ T' → ϵ
E'$	$	E' → ϵ
$	$	



Action Top-down

Q Solve the grammer using LL(1) parser.

S → A

A → aB / Ad

B → b

C → g

w = abd

Step ①: Eliminating left recursion:

$S \rightarrow A$

$A \rightarrow aBA'$

$A' \rightarrow dA' / \epsilon$

$B \rightarrow b$

$C \rightarrow g$

Step ②: first and follow :

	first	follow
S	a	$
A	a	$
A'	d, ϵ	$
B	b	d, $
C	g	NA

Step ③: Parse Table :

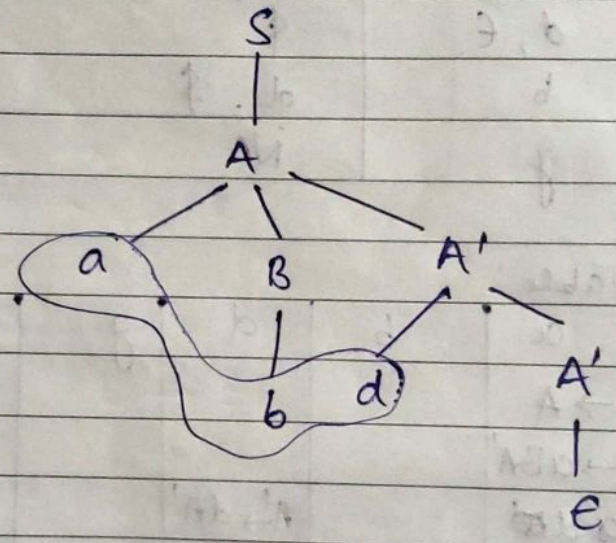
	a	b	d	g	$
S	$S \rightarrow A$				
A	$A \rightarrow aBA'$				
A'	~~A → aBA'~~		$A' \rightarrow dA'$		$A' \rightarrow \epsilon$
B		$B \rightarrow b$			
C				$C \rightarrow g$	

Step ④: Stack Implementation :

Stack	input string	Action
S $	abd $	
A $	abd $	~~Reduce~~ $S \rightarrow A$
aBA' $	~~a~~bd $	$A \rightarrow aBA'$
BA' $	bd $	
~~b~~A' $	~~b~~d $	$B \rightarrow b$
A' $	d $	
dA' $	~~d~~ $	$A' \rightarrow dA'$

A' $	$	
$	$	A' → ∈
$	$	Accept

step ⑤: Parse Tree



Q Check whether the given grammar is LL(1) or not.

S → iEtS / iEtSeS / a

E → b

step ①: Eliminating left ~~recursion~~ factoring.

S → iEtSS' / a

S' → eS / ∈ ∵ S → iEtS ↓ nothing left ∴ ∈

E → b

step ②:

	first	follow
S	i, a	e, $
S'	e, ϵ	e, $
E	b	t

step ③: Parsing Table

	a	b	e	i	t	$
S	S→a			S→iEtSS'		
S'			S'→eS S'→ϵ			S'→ϵ
E		E→b				

∴ The Grammer is not LL(1) because there are more than one production in S'.

---

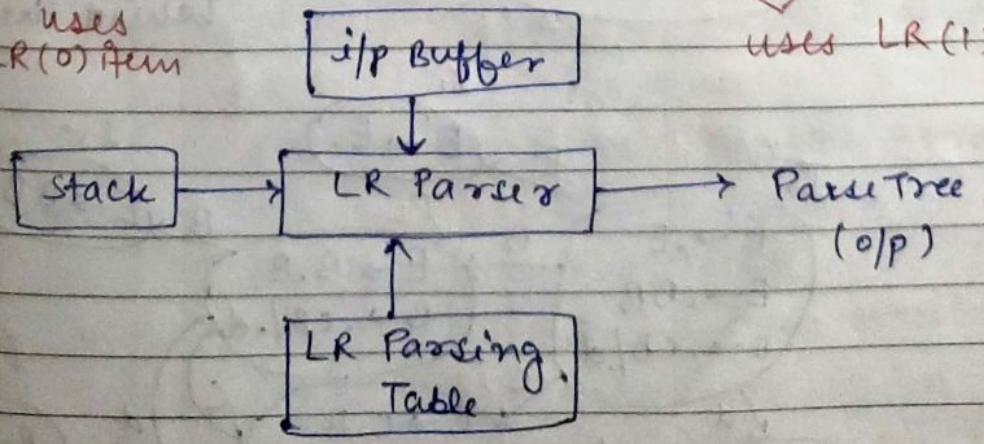
* Bottom-up Parser

LR → construction of right most derivation in reverse

Left to right scanning of i/p stream

LR Parser

- LR(0) ✓ — least powerful — uses LR(0) item
- SLR(1) ✓ (Simple LR)
- LALR(1) ✓ (Look ahead LR)
- CLR(1) ✗ (canonical LR) — most powerful — uses LR(1) item



* LR(0) :

>> Steps to solve LR(0) sums :

1. Augment the given grammar (self-created grammar)
2. Draw conical collection of LR(0) item
3. Number the production.
4. Create the Parsing Table.
5. Stack implementation.
6. Draw ParseTree.

Eg1: $E \rightarrow BB$
$B \rightarrow cB/d$ i/p = ccdd$

step ① : Augment grammar.

$E' \rightarrow E$
$E \rightarrow BB$
$B \rightarrow cB/d$

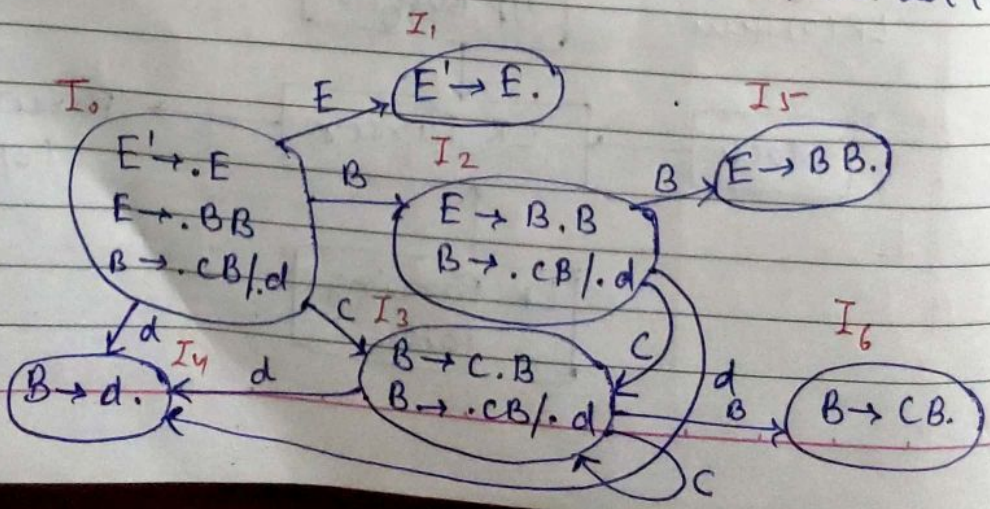
step ② : conical collection of LR(0) item

$E' \rightarrow .E$
$E \rightarrow .BB$
$B \rightarrow .cB/.d$

→ LR(0) Item Symbol

state 1

{ we have to open the non-terminal just behind the dot (.) }



Step ③: Number the Production

$E' \rightarrow E$

$E \rightarrow BB$ ①

$B \rightarrow cB$ ② / d ③

Step 4: Parsing Table

State	Action (Terminals)			Goto (Non-Terminals)	
	c	d	$	E	B
I_0	S_3	S_4		1	2
I_1			Accept (self-created)		
I_2	S_3	S_4			5
I_3	S_3	S_4			6
I_4	r_3	r_3	r_3		
I_5	r_1	r_1	r_1		
I_6	r_2	r_2	r_2		

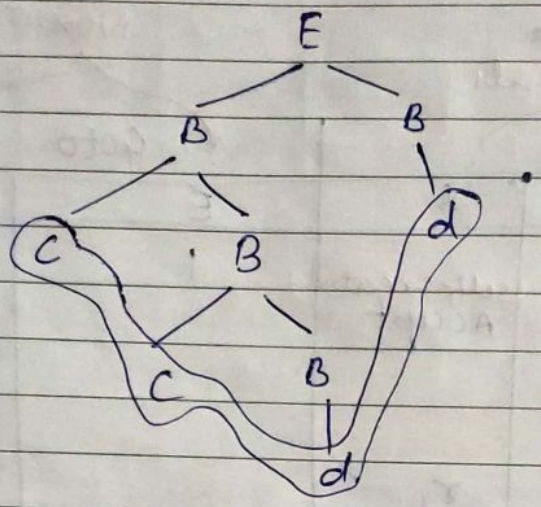
Step ⑤: stack implementation

stack (first state)	input	Action
$0	ccdd $	shift c in stack and goto state 3
$0c3	cdd $	shift c in stack and goto state 3
$0c3c3	dd $	shift d in " & " " S4.
$0c3c3d4	d $	Reduce r_3 i.e. $B \rightarrow d$
$0c3c3B6	d $	Reduce r_2 i.e $B \rightarrow cB$
$0c3B6	d $	Reduce r_2 i.e. $B \rightarrow cB$
$0B2	d $	shift d in stack & goto S4.
$0B2d4	$	Reduce r_3 i.e. $B \rightarrow d$
$0B2B5	$	Reduce r_1 i.e. $E \rightarrow BB$

$ 0 E 1	$	Accept

↑ Bottom up

Step ⑥: Parse Tree



Eg 2: Grammar:

$S \rightarrow AA$

$A \rightarrow aA \mid b$

Step ①: Augment grammar

$S' \rightarrow S$

$S \rightarrow AA$

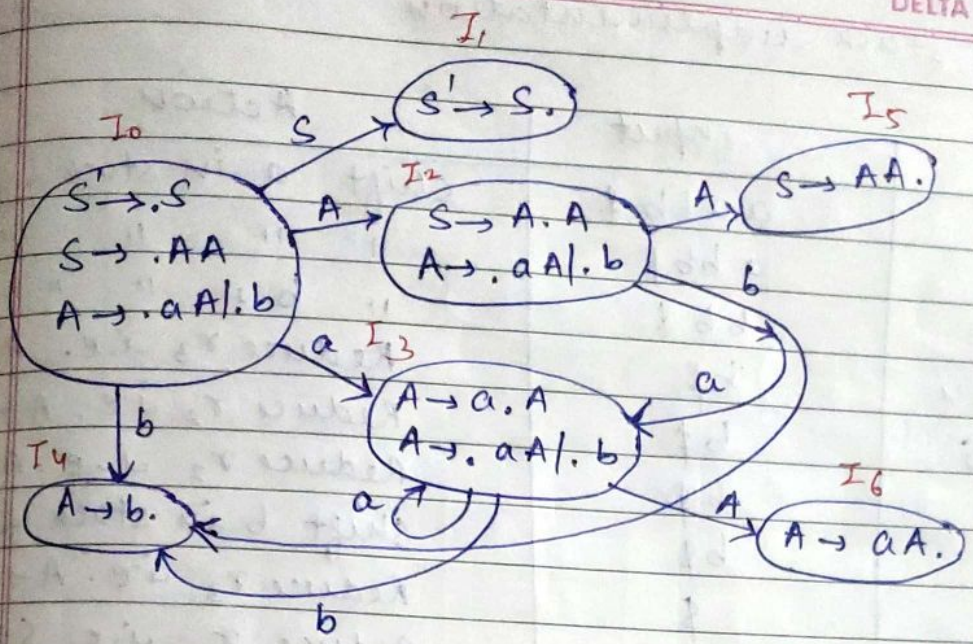
$A \rightarrow aA \mid b$

Step ②: Canonical collection of LR(0) items:

$S' \rightarrow .S$

$S \rightarrow .AA$

$A \rightarrow .aA \mid .b$



Step (3): Number the production.

$S' \rightarrow S$

$S \rightarrow AA$ (1)

$A \rightarrow aA$ (2) $| b$ (3)

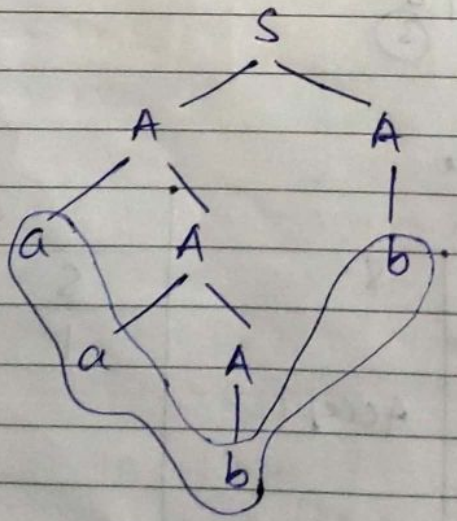
Step (4): Parsing Table

State	Action			Goto	
	a	b	$	S	A
I_0	S_3	S_4		1	2
I_1			Accept		
I_2	S_3	S_4			5
I_3	S_3	S_4			6
I_4	r_3	r_3	r_3		
I_5	r_1	r_1	r_1		
I_6	r_2	r_2	r_2		

Step ⑤ : stack implementation

Stack	input	Action
$ 0	a a b b $	shift a in stack & goto s3
$ 0 a 3	a b b $	" " " " "
$ 0 a 3 a 3	b b $	" b " " " " s4
$ 0 a 3 a 3 b 4	b $	Reduce r_3 i.e. A → b
$ 0 a 3 a 3 A 6	b $	Reduce r_2 i.e. A → aA
$ 0 a 3 A 6	b $	Reduce r_2 i.e. A → aA
$ 0 A 2	b $	shift b in stack & goto s4
$ 0 A 2 b 4	$	Reduce r_3 i.e. A → b
$ 0 A 2 A 5	$	Reduce r_1 i.e. S → AA
$ 0 S 1	$	Accept

Step ⑥ : Parse Tree



* SLR (1)

Same steps as LR(0)
only difference in parsing Table.

Eg 1 of LR(0)

$E' \to E$
$E \to BB$
$B \to CB/d$

State	Action			Goto	
	c	d	$	E	B
I_0	S_3	S_4		1	2
I_1			Accept		
I_2	S_3	S_4			5
I_3	S_3	S_4			6
I_4	r_3	r_3	r_3		
I_5			r_1		
I_6	r_2	r_2	r_2		

I_4 (B) $\to d$

follow(B) = c, d, $ i.e $

I_5 $E \to BB$

follow(E) = $

I_6 $B \to CB$

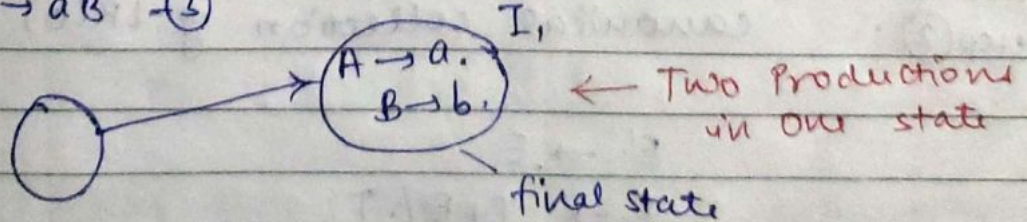
follow(B) = c, d, $

Q Why SLR(1) is used? → To remove or minimize these two conflicts to an extent.

conflict :-

① RR (reduce reduce)

$A \to a$ -①
$B \to b$ -②
$C \to aB$ -③

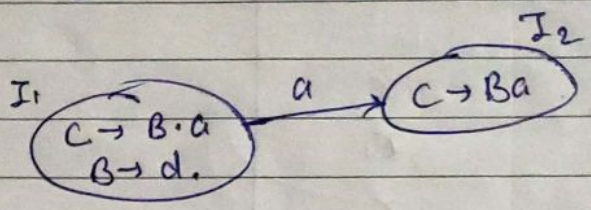


State	Action			Goto
	a	b	$	
I_1	r_1/r_2	r_1/r_2	r_1/r_2	

(2) SR (shift reduce)

$C \rightarrow Ba$ — (1)

$B \rightarrow d$ — (2)



State	Action			Goto
	a	d	$	
I_1	S_2/r_2	r_2	r_2	

$B \rightarrow d$ $B \rightarrow d$

Eg: Check whether the (i) grammar is LR(0) or not, (ii) grammar is SLR(1) or not.

$E \rightarrow T+E/T$

$T \rightarrow i$

sol:- step (1): Augment grammar

$E' \rightarrow E$

$E \rightarrow T+E/T$

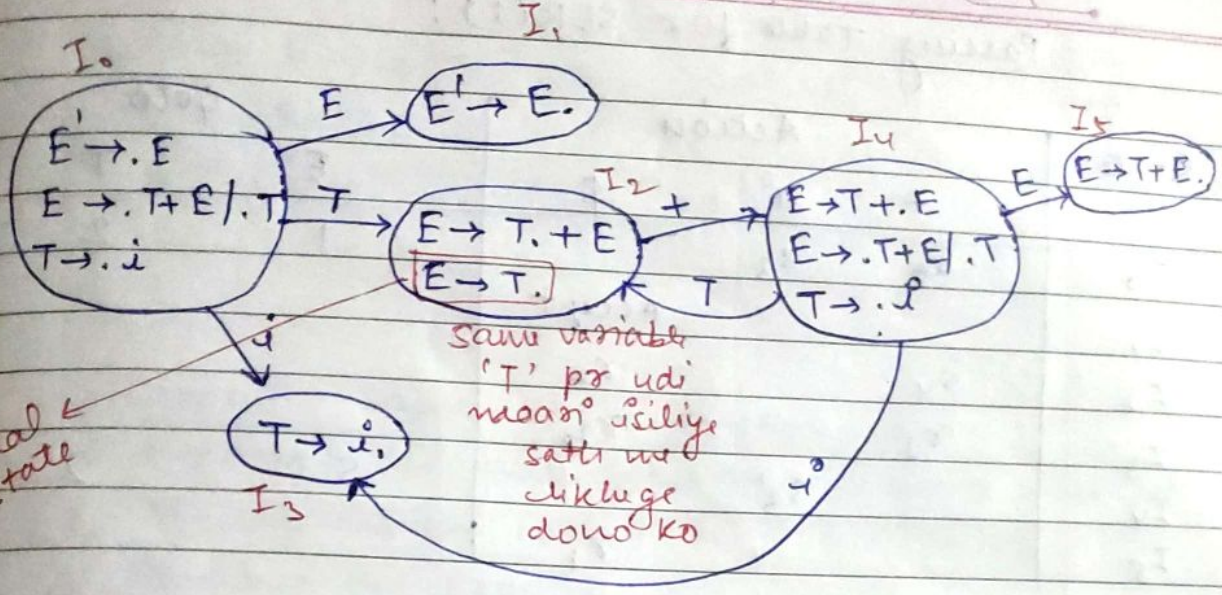
$T \rightarrow i$

step (2): canonical collection of LR(0) items:

$E' \rightarrow .E$

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

$T \rightarrow .i$



Step ③ : Number the prodⁿ.

$E' \rightarrow E$ ①

$E \rightarrow T+E$ / T ②

$T \rightarrow i$ ③

Step ④ : Passing Table

State	Action			Goto	
	i	+	$	E	T
I_0	S_3			1	2
I_1			Accept		
I_2	r_2	S_4/r_2	r_2		
I_3	r_3	r_3	r_3		
I_4	S_3			5	2
I_5	r_1	r_1	r_1		

State I_2 has two productions, i.e, grammar is not LR(0).

Parsing Table for SLR(1):

State	Action			Goto	
	+	id	$	E	T
I_0		s_3		1	2
I_1			Accept		
I_2	s_4		r_2		
I_3	r_3		r_3		
I_4		s_3		5	2
I_5			r_1		

∴ given Grammar is SLR(1) as it does not contain two productions for one state.

(*) LR(0) → Put reduce in full row
SLR(1) → Put reduce in follow of P
CLR(1) and LALR(1) → Put reduce only on Lookahead.

LR(0) item = (.) dot production
LR(1) item = LR(0) item + Look ahead

* How to find Lookahead :-

Eg: E → BB
B → cB | d

Step ①: Augment grammar and writing LR(1) item.

$E' \to .E$, $ → Lookahead of start symbol (augmented grammar)

Jis prod^n → $E \to .BB$, $

→ $B \to .CB/.d$, c/d

Jis prod^n ko hum open kr rhe hai, uske bad jo b hai uska first hi lookahead hoga.

* Steps to solve LR(1) items sums:

1. Augment the given grammar & write with look ahead
2. Draw canonical collection of LR(1) item
3. Number the prod^n.
4. Draw Parsing Table.
5. Stack Implementation.
6. Parse Tree

CLR(1)

Eg: $E \to BB$

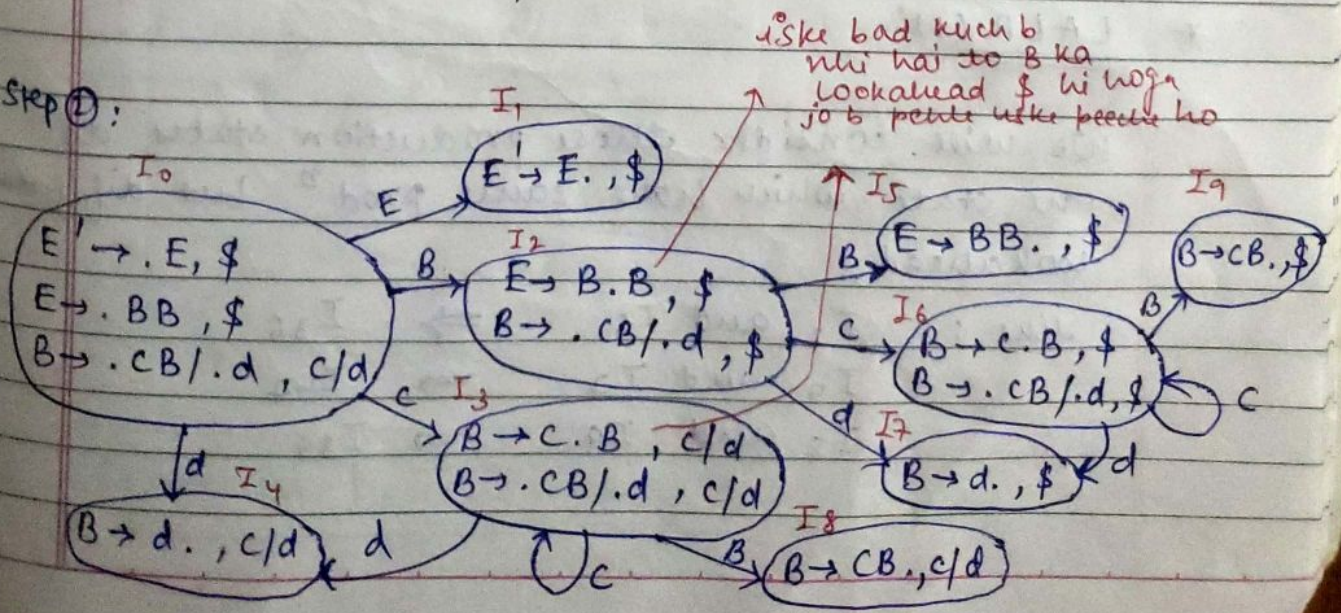
$B \to CB/d$

Step ① $E' \to .E$, $

$E \to .BB$, $

$B \to .CB/.d$, c/d

Step ②:



Step (3):

$E' \rightarrow E$

$E \rightarrow BB$ (1)

$B \rightarrow CB / d$
(2) (3)

Step (4):

State	Action			Goto	
	c	d	$	E	B
I_0	S_3	S_4		1	2
I_1			Accept		
I_2	S_6	S_7			5
I_3	S_3	S_4			8
I_4	r_3	r_3			
I_5			r_1		
I_6	S_6	S_7			9
I_7			r_3		
I_8	r_2	r_2			
I_9			r_2		

reduce comes only in lookahead in CLR(1).

* LALR(1)

We will consider those production states as one state which have same prod^n but different lookahead.

like: I_3 and I_6 $\Rightarrow$ I_{36}

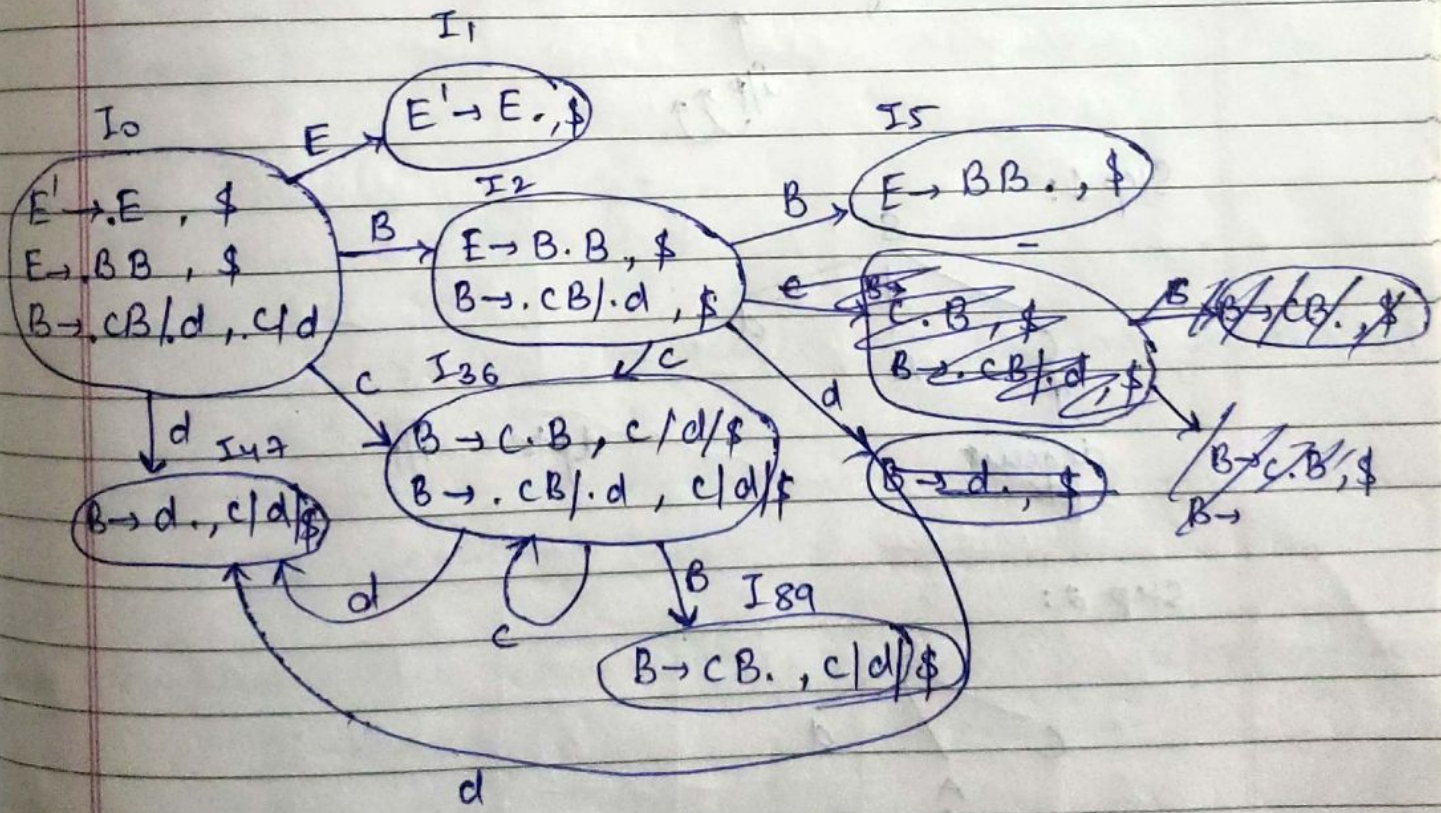
I_4 and I_7 $\Rightarrow$ I_{47}

I_8 and I_9 $\Rightarrow$ I_{89}

Parsing Table for LALR (1) :

state	Action			Goto	
	c	d	$	E	B
I_0	S_{36}	S_{47}		1	2
I_1			Accept		
I_2	S_{36}	S_{47}			5
I_{36}	S_{36}	S_{47}			89
I_{47}	r_3	r_3	r_3		
I_5			r_1		
I_{89}	~~r_2~~	~~r_2~~	r_2		

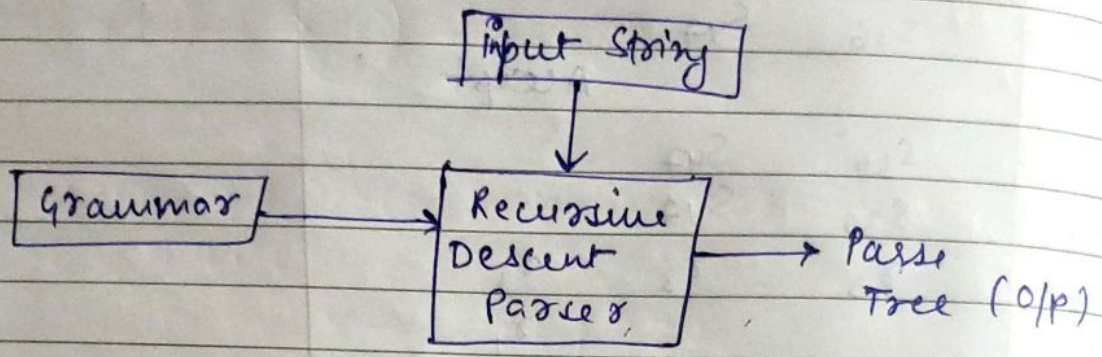
Reductions merge



Top-down Parser

* Recursive Descent Parsing :

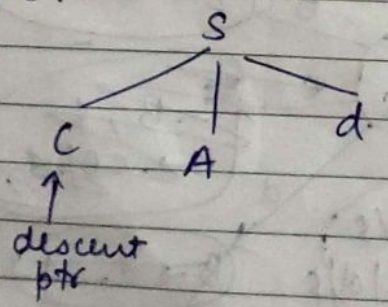
Recursive Descent Parsing is one of top-down parsing techniques that uses a set of recursive procedures to scan its input. May involve backtracking.



Eg: $S \rightarrow cAd$
$A \rightarrow ab / a$

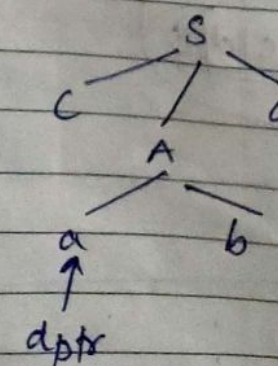
i/p string = cad
i/p

Step 1:



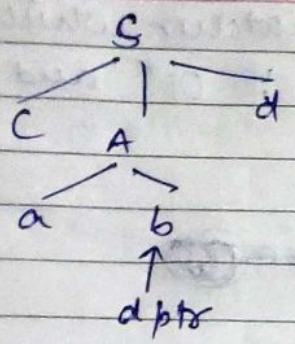
dptr = i/p

Step 2:



dptr = i/p

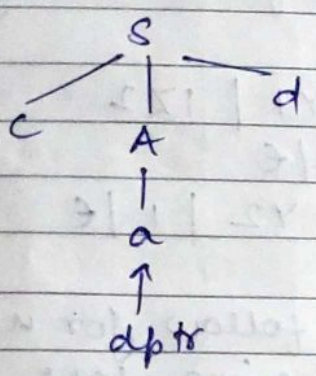
Step 3:



dptr ≠ i/p

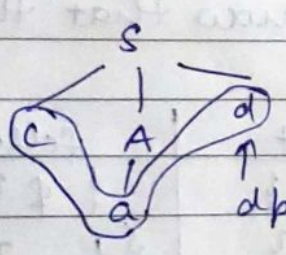
∴ Backtracks

Step 4:



dptr = i/p

Step 5:



∴ string matches, Parse Tree generated.

2018 Sessional Paper

Q Improve the grammar by removing left recursion.

$S \rightarrow AaB \mid b$

$A \rightarrow AaB \mid Aba \mid a$

$B \rightarrow a$

Ans:

$S \rightarrow AaB \mid b$

$A \rightarrow aA'$

$A' \rightarrow abA' \mid \epsilon \mid baA'$

$B \rightarrow a$

$A \rightarrow A\alpha \mid \beta$

$A \rightarrow Aba \mid a$

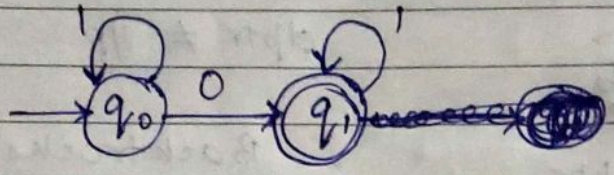
$A \rightarrow aA'$

$A' \rightarrow baA' \mid \epsilon$

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \epsilon$

Q Generate a pattern matcher which can recognise following two patterns : 01^* and 1^*0.



Q Consider the follow. grammar given below
ϵ = NULL.

$X \rightarrow 0Y1 \mid 1Z2$
$Y \rightarrow 2 \mid \epsilon$
$Z \rightarrow 1Y2 \mid 1 \mid \epsilon$

(a) calculate first & follow for non-terminals.
(b) create a predictive parsing table for above grammar. Also show that this grammar is not LL(1).

Sol: (a)

	first	follow
X	{0, 1}	{$}
Y	{2, ϵ}	{1, 2}
Z	{1, ϵ}	{2}

(b)

~~Predictive~~

	0	1	2	$
X	X→0Y1	X→1Z2		
Y			Y→ϵ / Y→2	Y→ϵ
Z		Z→1Y2	Z→ϵ	

∴ it is not LL(1) grammar.

Q consider the grammar

$S \rightarrow L = R \mid R$

$L \rightarrow * R \mid id$

$R \rightarrow L$

Design LALR Parser for this grammer

Sol:

Step 1: Augment grammar

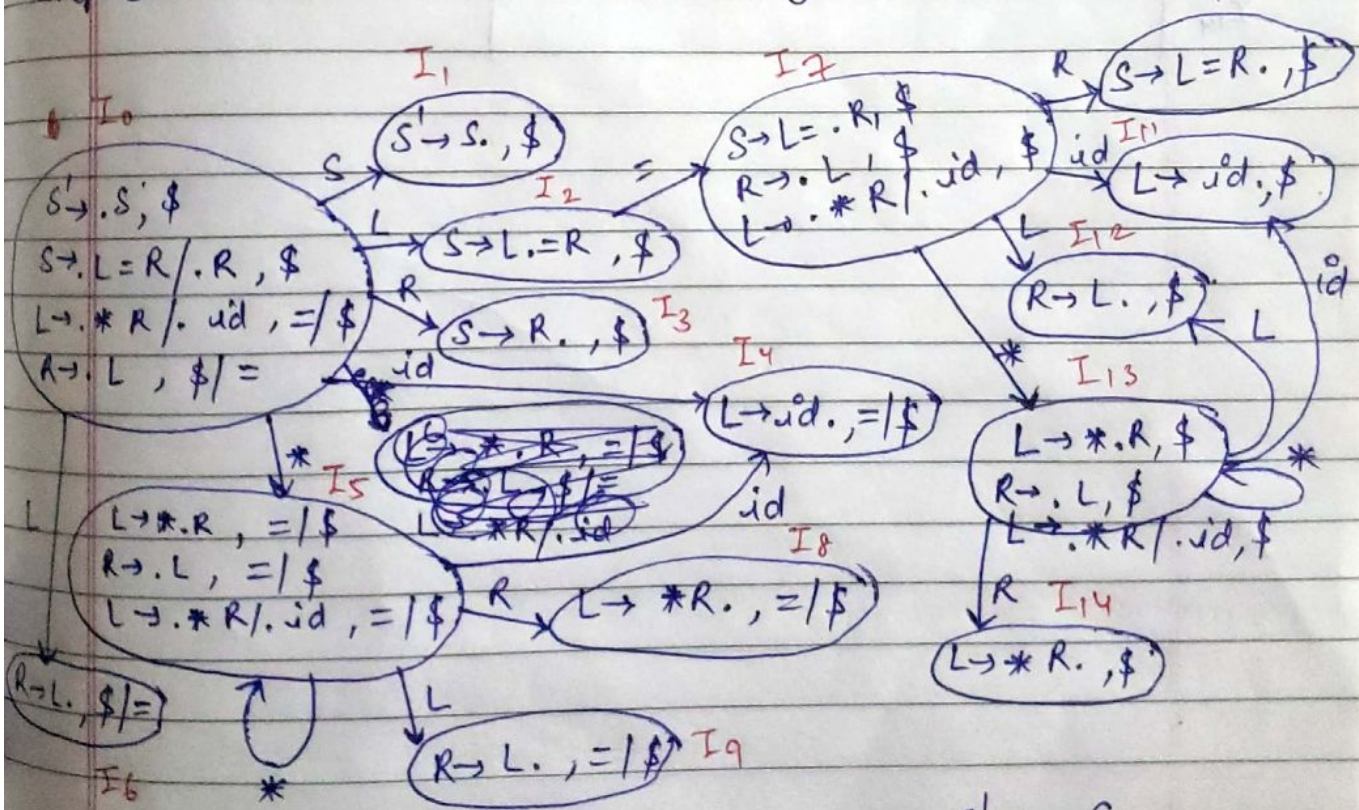
$S' \rightarrow .S \;,\; \$$

$S \rightarrow .L = R \;/\; .R \;,\; \$$

$L \rightarrow .* R \;/\; .id \;,\; = / \$$

$R \rightarrow .L \;,\; \$ / =$

Step (2): canonical collection of LR(1) item :



~~$I_2 + I_{10} = I_{2+10}$~~

Step (3):

$S' \rightarrow .S$

$S \rightarrow .L = R \;/\; .R$ ① ② ③

$S \rightarrow .*R \;/\; .id$ ④ ⑤

$R \rightarrow .L$ ⑤