Chapter 04

TOP DOWN PARSING:

* To resolve the ambiguity :-
  1). disambiguity rules (explicitly)
  2). " " (implicitly)

(ambiguous grammar)

* $E \rightarrow E+E \mid E-E \mid E*E \mid E/E \mid (E) \mid -E \mid id.$

- $E \rightarrow E+T \mid E-T \mid T$ → (lower priority & same priority) (Left rec.) (L-R)
- $T \rightarrow T*F \mid T/F \mid F$ → Right rec. (R-L)
- $F \rightarrow -E \mid (E) \mid id$

$a+b-c$ → Associativity is Left to right.

$E \rightarrow E+T$ (left recursion) (left to right)
$E \rightarrow T+E$ (right recursion) (right to left)

* LEFT RECURSION REMOVAL :

left rec. is commonly used to make operations left associated as in the simple expression grammar:

1) 2) 3)
$E \rightarrow E+T \mid E-T \mid T$
4) 5) 6)
$T \rightarrow T*F \mid T/F \mid F$
7) 8) 9)
$F \rightarrow -E \mid (E) \mid id$

where production, 1, 2, 4, 5 has left recursion.

In general, the left recursion is represented as $A \rightarrow A\alpha|\beta$ where, α and β are strings of terminals & non-terminals & β doesn't begin with A.

* Removal of left Recursion:
Rewrite the grammar as follows:

$A \rightarrow A\alpha|\beta$

$\Downarrow$

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' | \epsilon$

In general, if we have the production of the form $A \rightarrow A\alpha_1|A\alpha_2|\text{-----}|A\alpha_n|\beta_1|\beta_2\text{---}|\beta_n$

$\Downarrow$

$A \rightarrow \beta_1 A' | \beta_2 A' | \text{-------} | \beta_m A'.$

$A' \rightarrow \alpha_1 A' | \alpha_2 A' | \text{----} | \alpha_n A' | \epsilon.$

Eg:- $\overset{A}{E} \rightarrow \overset{A}{E}+T|T$

$\alpha \quad \beta$

$\Downarrow$

$E \rightarrow TE'$

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

For $E \rightarrow E+T | E-T | T$

$E \rightarrow TE'$

$E' \rightarrow +TE' | -TE' | \epsilon.$

$T \rightarrow T*F \mid T/F \mid F$

$T \rightarrow FT'$

~~$T' \rightarrow *FT' \mid /F \mid$~~ $T' \rightarrow *FT' \mid /FT' \mid \epsilon$.

All these grammars are G' (which doesn't have left recursion).

Ques Write the parse tree for 3-4-5 applying G'.

G' $E \rightarrow TE'$

$E' \rightarrow +TE' \mid -TE' \mid \epsilon$.

$T \rightarrow FT'$

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

$F \rightarrow E \mid (E) \mid$ ~~id~~ num.

TE'

$T-TE'$

$FT'-TE'$

$F-TE'$

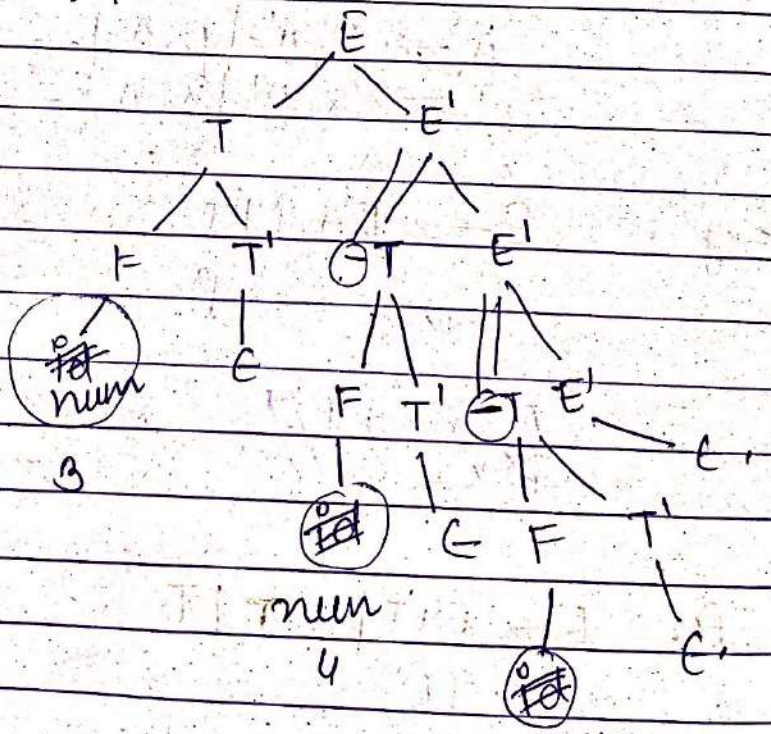
$id-T-TE'$

$id-T-T$

$id-FT'-FT'$

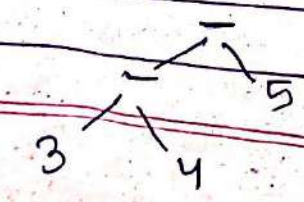
$id-F-F$

$id-id-id$



Abstract Syntax / Parse Tree.

AST



Ques: Stmt-sequence → Stmt-sequence ; Stmt | Stmt
Stmt → S

G = ({Stmt-sequence, Stmt}, {S, ;}, {Stmt-sequence}, P)

L(G) = { S, S;S, S;S;S }

Ques: Write the L(G). & Remove left recursion.

1). Stmt-seq → Stmt-seq ; Stmt | Stmt
Stmt → S

2). declarn → type var-list
var-list → var-list , id | id
type → int | float A α β

3). lexp → atom | list
atom → number | id
list → (lexp-seq)
lex-seq → lex-seq lexp | lexp

Soln: 1). Stmt-seq → Stmt-seq ; Stmt | Stmt
A → Aα | β
A α β
∴ α = ; Stmt β = Stmt.
∴ A → βA' & A' → αA' | ∈.

Ans →
Stmt-seq → Stmt {Stmt-seq}'
(Stmt-seq)' → ; Stmt {Stmt-seq}' | ∈
Stmt → S

L(G) = {S, S;S, S;S;S, ...}
∴ L(G) = Any set of Stmt separated by ;.

Only change the production having left recursion. Rest all production will remain same.

(2). declarn → type var-list

var-list → var-list , id | id

A → A α | β

L(G1) = { int a , int a, b ; float c, d ...}

declarn → type var-list
var-list → id var-list'
var-list' → , id var-list' | ∈ .
type → int | float

Put num1, num2 } Ans

(3).

→ brackets.

lexp → atom | list
atom → number | id.
list → (lexp-seq)

lexp-seq → lexp-seq lexp | lexp

A → A α | β

lexp-seq → lexp lexp-seq'
lexp-seq' → lexp lexp-seq' | ∈ .

L(G1) = { ~~atom~~ , ~~list~~ , number , id , (~~atom~~ num) ,

23 a (num id)

L(G1) = { 23 , a , (23 50) , (50 60 a) ,

((50 60) 70 a) }

★ LEFT FACTORING: It can be applied whenever we have common prefix.

- some of the parsers do not allow common prefix.
- So, for the removal of ~~tet~~ common prefix we use left factoring.

$$A \longrightarrow \alpha\beta \mid \alpha\gamma$$

$$\Downarrow$$

$$A \longrightarrow \alpha A'$$
$$A' \longrightarrow \beta \mid \gamma$$

α $\beta = \epsilon$ α γ

E.g.: if-stmt → if(exp)stmt | if(exp) stmt else stmt
stmt → s
exp → 0 | 1
(non-ter.) T (ter.)

$G_1 = (\{$ if-stmt, stmt, exp $\}, \{$ if, else, s, 0, 1, (,) $\}$
$\{$ if-stmt $\}$, P)
S (start var)

$L(G_1) = \{$ if (0) s, if (1) s, if (1) s else s,
if (0) else s $\}$ → only 4 lexemes

if-stmt → if(exp) stmt if-stmt'
if-stmt' → ϵ | else stmt

$\}$

Ques: stmt-seq → stmt ; stmt-seq | stmt (α = stmt, β = ; stmt-seq, r = ϵ)
stmt → S

G1 = ({stmt-seq, stmt}, {S, ϵ}, {stmt-seq}, P)

stmt-seq → stmt stmt-seq'
stmt-seq' → ; stmt-seq | ϵ
stmt → S

∴ L(G1) = { S, S;S, S;S;S, ... }

Ques: lexp → atom | list
atom → number | id
list → (lexp-seq)
lexp-seq → lexp , lexp-seq | lexp (α = lexp, β = , lexp-seq, α = lexp, r = ϵ)

lexp-seq → lexp lexp-seq'
lexp-seq' → , lexp-seq | ϵ

L(G1) = { 23, a, (a, 50), ... }

★ PARSING:

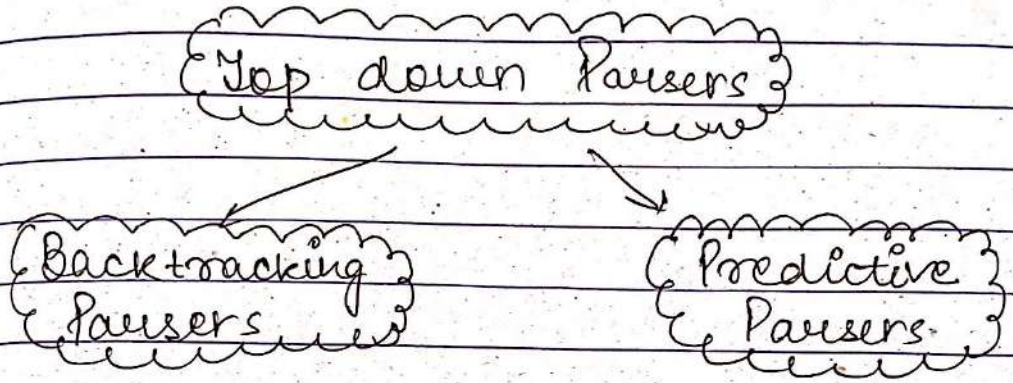
Parsers

- Top down Parser
  - Recursive descent
  - LL(1) → no. of look aheads symbol in this.
    - scan i/p: Left to right
    - leftmost derivation
- Bottom Up Parser
  - SLR(1), LR(1), LALR(1). (YACC tool)
    - (more efficient)

↳ 1 look ahead symbol

- SLR(1)
  - ↳ reverse of right recursion
  - ↳ scan ip from left to right

- LALR : Look Ahead LR



- more powerful
- less efficient
  efficiency: exponential

- LL(1)

* LL(1) parser:

* Constraints:
- The grammar should not have the production which have left recursion or common prefix.

1). Check whether grammar is suitable for LL(1).

2). If not suitable, i.e. it has left recursion or common prefix
3). make it suitable by removing left recursion / common prefix (left factoring).

3). Compute first & follow sets.

4). construct the parsing table ∴ (P.T)

5). verify the PT taking valid string by showing the contents of parsing stack.

Eg. $S \rightarrow (S)\,S \mid \epsilon$

$L(G_1) = \{ (), (())(), \ldots \}$ // balanced Parenthesis.

- Embed dollar ($) sign at the end of input.

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

$S \Rightarrow (S)S$
$\Rightarrow (\epsilon)S$
$\Rightarrow ()\epsilon$
$\Rightarrow ()$

input	parsing stack	action
()$ ↑	$S	$S \rightarrow (S)\,S$
()$ ↑	$S)S((enter staring from right) top	matched adv i/p ptr & pop symbol
)$ ↑	$S)S	$S \rightarrow \epsilon$
)$ ↑	$S)	match
$ ↑	$S	$S \rightarrow \epsilon$
$ ↑	$ ↑	accept.

→ If first i/p symbol is same as top of stack then it is a match.

So, we adv. the top symbol & pop.

when i/p ptr is dollar $ top of stack is also dollar then we say action is accepted.

Ques 1. exp → exp addop term | term
3. addop → + | -
5. term → term mulop factor | factor
7. mulop → *
8. factor → (exp) | number

1 & 5 are having left recursion
∴ remove it.

G'

exp → term exp'
exp' → addop term exp' | ∈
addop → + | -
term → factor term'
term' → mulop factor term' | ∈
mulop → *
factor → (exp) | number

Computation of first set:

1). first(X) = X if X is a terminal.

2). X → Y1 Y2 ... Yk i.e. if X is a non-terminal.

first(X) = first(Y1)

3). first(∈) = ∈. if X → ∈

	pass 1	pass 2	pass (3)
exp	first(term)		{ (, number }
exp'	{first(addop), ∈}	{ +, -, ∈ }	
addop	{ +, - }		
term	first(factor)	{ (, number }	
term'	{first(mulop), ∈}	{ *, ∈ }	
mulop	{ * }		
factor	{ (, number } ↳ opening bracket		

∴ First Set

	First
exp	{ (, number }
exp'	{ +, -, ∈ }
addop	{ +, - }
term	{ (, number }
term'	{ *, ∈ }
mulop	{ * }
factor	{ (, number }

* LL(1) Parser:

exp → term exp'
exp' → addop term exp' | ∈
term → factor term'
term' → mulop factor term' | ∈
addop → + | -
mulop → *
factor → (exp) | number

} same as prev

* FOLLOW SET:

Def: Given a non terminal A the set follow(A) consisting of terminals & $ (dollar's) is defined as follows:

Rule 1: If A is a start symbol then $ is in follow of A (follow(A)).
{Add $ to follow(A)}
{Rule 1 will be applied only once because we have only 1 start symbol}

Rule 2: If there is a production A → αBβ where B is a non-terminal, then first(β) - {∈} is in follow(B).
↓
except (∈)

• α, β are strings containing terminals & non-terminals. (N) (T).
i.e. $\alpha, B = (T \cup N)^*$.

Rule 3: If there is a production $A \to \alpha B$ or $A \to \alpha B \beta$ where first(β) contains 'ϵ', then whatever is in follow(A) will become follow(B) i.e. **follow(B) = follow(A)**

NOW, apply for same prev Grammar G' to compute follow sets.

	Pass 1
exp → term exp'	follow(exp) = {$} (by Rule 1) follow(exp') = {$} (by Rule 3) = follow(exp)
exp' → addop term exp' | ϵ	follow(term) = first(exp') − ϵ (by Rule 2) = {+, −} follow(term) = {follow(exp'), +, −} (by Rule 3) = {+, −, $}
term → factor term'	follow(term') = follow(term) (by Rule 3) = {+, −, $}
term' → mulop factor term' | ϵ	follow(factor) = first(term') − {ϵ} (by Rule 2) = {*} follow(factor) = follow(term') (by Rule 3) = {+, −, $, *}
addop → + | − mulop → *	} both are terminals, ∴ cannot get any follow sets.
factor → (exp) | number α B β	follow(exp) = first()) − {ϵ} by Rule 2 = {$,)} ∵ follow(exp) changed.

∴ follow(exp') will also change, follow(term) will change, etc. ∴ Go to for next page.

- α, β are strings containing terminals (T), & non-terminals. (N)
  i.e. $\alpha, \beta = (T \cup N)^*$.

Rule 3: If there is a production $A \rightarrow \alpha B$ or $A \rightarrow \alpha B \beta$ where first(β) contains 'ϵ', then whatever is in follow(A) will become follow(B) i.e. **follow(B) = follow(A)**

NOW, apply for same prev Grammar G' to compute follow sets.

	Pass 1
exp → term exp'	follow(exp) = {$} (by Rule 1) follow(exp') = follow(exp) = {$} (by Rule 3)
exp' → addop term exp' / ϵ	follow(term) = first(exp') − ϵ (by Rule 2) = {+, −} follow(term) = {follow(exp'), +, −} (by Rule 3) = {+, −, $}
term → factor term'	follow(term') = follow(term) (by Rule 3) = {+, −, $}
term' → mulop factor term' | ϵ	follow(factor) = first(term') − {ϵ} (by Rule 2) = {*} follow(factor) = follow(term') (by Rule 3) = {+, −, $, *}
addop → + | −	} both are terminals, ∴ cannot get any follow sets.
mulop → *	
factor → (exp) | number	follow(exp) = first( ) − {ϵ} by Rule 2 = {$,)} ∵ follow(exp) changed.

∴ follow(exp') will also change, follow(term) will change etc. ∴ Go for next page.

Pass 2

exp → term exp'	follow(exp') = { $,) }
exp' → addop term exp'	follow(term) = { +, −, $,) }
term → factor term'	follow(term') = { +, −, $,) }
term' → mulop factor term'	follow(factor) = { *, +, −, $,) }
factor → exp.	NO change.

∴ Final follow set & first set :-

	first	follow
exp	{ (, number }	{ $,) }
exp'	{ +, −, ∈ }	{ $,) }
term	{ (, number }	{ +, −, $) }
term'	{ *, ∈ }	{ +, −, $,) }
addop	{ +, − }	{ (, number }
mulop	{ * }	{ (, number }
factor	{ (, number }	{ +, −, *, $,) }

exp' → addop term exp'
follow(addop) = first(term) ⇒ first of whatever is next
= { (, number }

# term' → mulop factor term'
follow(mulop) = first(factor)
= { (, number }

* Construction of Parsing Table: (LL(1)):

Repeat the following steps for each non-terminal A & for the production $A \to \alpha$.

1). for each token a in first(α), add $A \to \alpha$ to the entry M[A,a]

2). If ϵ is in first(α) for each element of follow(A). {a is a token (terminal) or $} then add the production $A \to \alpha$ to the entry M[A,a]

M[N,T]	number	+	−	*	(	)	$
exp	Exp → term exp'				exp → term exp'		
exp'		exp' → addop term exp'	exp' → addop term exp'			exp' → ∈	exp' → ∈
addop		addop → +	addop → −				
term	term → factor term'				term → factor term'		
term'		term' → ∈	term' → ∈	term' → mulop factor term'		term' → ∈	term' → ∈
mulop				mulop → *			
factor	factor → number				factor → (exp)		

Based on all the productions taking one-by-one.

All blank entries represent errors.

P.T.O

* Verifying Parsing Table: let I/p = 23+15

parsing stack (start symbol ↑)	input	action
$exp	23+15$	exp → term exp'
$exp' term	23+15$	term → factor term'
$exp' term' factor	23+15$	factor → number
$exp' term' number	23+15$	match, pop & advance I/p pointer.
$exp' term'	+15$	term' → ∈
$exp'	+15$	exp' → addop term exp'
$exp' term addop	+15$	addop → +
$exp' term +	+15$	match, pop & advance I/p pointer.
$exp' term	15$	term → factor term'
$exp' term' factor	15$	factor → number
$exp' term' number	15$	match, pop & advance I/p pointer.
$exp' term'	$	term' → ∈
$exp'	$	exp' → ∈
$	$	accept

Also check for 2+

Parsing Stack	Input	action
$ exp	2+3*5$ ↑	exp → term exp'
$ exp' term	2+3*5$ ↑	term → factor term'
$exp' term' factor	2+3*5$ ↑	factor → number
$exp' term' number	2+3*5$ ↑	match, pop $ adv i/p ptr.
$exp' term'	+3*5$ ↑	term' → ∈.
$ exp'	+3*5$ ↑	exp' → addop term exp'
$exp' term addop	+3*5$ ↑	addop → +
$ exp' term +	+3*5$ ↑	match, pop $ adv i/p ptr.
$ exp' term	3*5$ ↑	term → factor term'
$exp' term' factor	3*5$ ↑	factor → number
$ exp' term' number	3*5$ ↑	match, pop $ adv. i/p ptr.
$exp' term'	*5$ ↑	term' → mulop factor term'
$exp' term' factor mulop	*5$ ↑	mulop → *
$exp' term' factor *	*5$ ↑	match, pop $ adv i/p ptr.
$exp' term' factor	5$ ↑	factor → number
$exp' term' number	5$ ↑	match, pop $ adv i/p ptr.
$ exp' term'	$ ↑	term' → ∈
$ exp'	$ ↑	exp' → ∈
$	$ ↑	accept

Ques ①. Construct LL(1) PT for the given CFG

stmt_seq → stmt ; stmt_seq | stmt.

stmt → S.

② if_stmt → if (exp) stmt |
if (exp) stmt else stmt.

stmt → S

exp → 0 | 1

Sol^n: ① This grammar contains common prefix. α β α r=ϵ

∴ stmt_seq → stmt ; stmt_seq | stmt ;

$$\begin{cases} A \rightarrow \alpha\beta \mid \alpha r \\ \Downarrow \\ A \rightarrow \alpha A' \\ A' \rightarrow \beta \mid r \end{cases}$$

$$\begin{cases} \text{stmt\_seq} \rightarrow \text{stmt stmt\_seq}' \\ \text{stmt\_seq}' \rightarrow ; \text{stmt\_seq} \mid \epsilon \\ \text{stmt} \rightarrow S \end{cases}$$

* FIRST SET

	Pass 1	Pass 2	Pass 3
stmt_seq	first(stmt)	S	
stmt_seq'	first(;)=;		
stmt	S		

∴ First set

	First
stmt_seq	S
stmt_seq'	{;, ϵ}
stmt	S

Follow Set :-

stmt_seq → stmt stmt_seq'	follow(stmt_seq) = {$} Rule 1 follow(stmt_seq') = follow(stmt_seq) = {$} Rule 2
stmt_seq' → ; stmt_seq | ∈ .	

Production 1:

stmt_seq → stmt stmt_seq'

By Rule 1 : follow (stmt_seq) = {$}.

By Rule 2 : stmt_seq → stmt stmt_seq'
(α = ∈, B, β)

∴ follow(stmt) = first(stmt_seq')
= {;, ∈} − ∈
= {;}.

By Rule 3 : stmt_seq → stmt stmt_seq'
(α = ∈, B, β)

follow(stmt) = follow(stmt_seq)
= {;, $}.

follow(stmt_seq') = follow(stmt_seq)
= {$}

Production 2: stmt_seq' → ; stmt_seq.
(α, B, β = ∈)

By Rule 2 :- follow (stmt_seq) = {$} = first(β) − ∈

By Rule 3 :- follow (stmt_seq) = follow(stmt_seq')
= {$}.

Production 3 : stmt → s.

Follow set

stmt-seq	$
stmt-seq'	$
stmt	;, $

M-2 Prod 1

stmt-seq → stmt stmt-seq'

follow(stmt-seq) = {$} by Rule 1.

For Rule 2: stmt-seq → stmt stmt-seq'
($\alpha = \epsilon$, B, β)

∴ follow(stmt) = first(stmt-seq') − ϵ.
= {;}

For Rule 3: stmt-seq → stmt stmt-seq'
(1st alternative) A → αB (α, B)

follow(stmt-seq') = follow(stmt-seq)
by Rule 3.1 = {$}

(2nd alternative) A → αBβ: stmt-seq → stmt stmt-seq'
(α, B, β)

∴ by Rule 3.2 follow(stmt) = follow(stmt-seq)
= {;, $}

Prod 2

stmt-seq' → ; stmt-seq
(α, B, $\beta = \epsilon$)

follow(stmt-seq) = first(β) − ϵ.
↓ It has nothing except ϵ
∴ we cannot apply Rule 2.

For Rule 3.1 stmt-seq' → ; stmt-seq
(A, α, B)

follow(stmt-seq) = follow(stmt-seq')
= {$}.

Not getting new set i.e. no changes.

Q4)

	First	Follow
stmt-seq	{S}	{$}
stmt-seq'	{;, ∈}	{$}
stmt	{S}.	{;, $}

★ Parsing Table :-

M[N,T]	;	S	$
stmt-seq		stmt-seq → stmt stmt-seq'.	
stmt-seq'	stmt-seq' → ; stmt-seq		stmt-seq' → ∈.
Stmt		stmt → S	

★ Parsing Stack & Verification :

Valid language : S, S;S, S;S;S.

Parsing Stack	Input	Action
$ stmt-seq	S;S $ ↑	stmt-seq → stmt stmt-seq'
$ stmt-seq' stmt	S;S $ ↑	stmt → S
$ stmt-seq' S	S;S $ ↑	match, pop $ adv. ip ptr.
$ stmt-seq'	;S $ ↑	stmt-seq' → ; stmt-seq.
$ stmt-seq ;	;S $ ↑	match, pop $ adv. ip ptr.
$ stmt-seq	S $ ↑	stmt-seq → stmt stmt-seq'
$ stmt-seq' stmt	S $ ↑	stmt → S
$ stmt-seq' S	S $ ↑	match, pop $ adv ip ptr.
$ stmt-seq'	$ ↑	stmt → seq' → ∈.
$	$ ↑	Accept

(2) stmt → if_stmt | other

if_stmt → if (exp) stmt | if (exp) stmt else stmt

exp → 0 | 1.

A → α β | α γ

if_stmt → if (exp) stmt | if (exp) stmt else stmt

∴ A → αA'

A' → β | γ

∴ stmt → if_stmt | other

if_stmt → if (exp) stmt if_stmt'

if_stmt' → else stmt | ∈.

exp → 0 | 1.

* FIRST SET

	Pass 1	Pass 2
stmt	first(if_stmt) first(other) = other	{if, other}
if_stmt	first(if) = if	{if}
if_stmt'	first(else) = else, first(∈) = ∈.	{else, ∈}
exp	{0, 1}	{0, 1}.

* FOLLOW SET

Prod 1 stmt → if_stmt $

follow(stmt) = $ by Rule 1

stmt → α if_stmt β

Rule 3.2 follow(if_stmt) = follow(stmt) = $.

Prob 2 α β β

~~if-stmt → if (exp) stmt | if (exp) stmt else stmt~~

if-stmt → if (exp) stmt if-stmt'

if-stmt' → if (exp) stmt if-stmt'

* LL(1) PT with error recovery entries.

M[N,T]	(	number	)	+	−	*	$
exp	exp → term exp'	exp → term exp'	POP	scan	Scan	scan	POP
exp'	Scan	Scan	exp' → ∈	exp' → addop term exp'	exp' → addop term exp'	Scan	exp' → ∈
addop	POP	POP	scan	addop → +	addop → -	Scan	POP
term	term → factor term'	term → factor term'	POP	POP	POP	Scan	POP
term'	Scan	Scan	term' → ∈	term' → ∈	term' → ∈	term' → mulop factor term'	term' → ∈
mulop	POP	POP	Scan	scan	Scan	mulop → *	POP
factor	factor → (exp)	factor → number	POP	POP	POP	POP	POP

Invalid Strings:

2 + * : missing operand

(2+3) 5 : missing operator

Write all left-out box in $ as POP. P.T.O

* To fill LL(1) PT with Error Recovery Entries:

Given a non-terminal A at the top of the stack Φ. an input token that is not in first(A) or [follow(A)], If ϵ is in first(A), there are three possible alternatives:

1). Pop the non terminal 'A' from the stack.
2). Successively pop tokens from the I/p until a token is seen for which we can restart the parse. (Panic Mode Error Rec.)
3). Push a new non-terminal onto the stack.

- We choose alternative 1, if the current I/p token is $ or is in follow(A).
- We choose alternative 2, If the current I/p token is not $. Φ is in not in first(A) $\cup$ follow(A).
- Option 3 is occassionally useful in special cases that is rarely appropriate.
- We indicate the 1st action in the parsing table by the notation POP & the 2nd by the notation SCAN.

NOTE: The POP action is equivalent to the reduction by an ϵ-production

Scan → adv i/p ptr.
POP → POP from stack.

* Parse for (2+*)

Parsing Stack	Input	Action
$ exp	(2+*)$	exp→
$ exp' term	(2+*)$	term exp'
$ exp' term' factor	(2+*)$	term→ factor term'
$exp'term') exp (	(2+*)$	factor→ (exp)
$exp'term') exp	2+*)$	match, pop & adv i/p
$exp'term') exp'term	2+*)$	→term exp'
$exp'term') exp'term factor	2+*)$	term→ factor term'
$exp'term') exp'term no.	2+*)$	factor→ number
$exp'term') exp'term	+*)$	match, pop & adv
$exp'term') exp'	+*)$	term' → ϵ
$exp'term') exp'term addop	+*)$	→addop term exp'
$exp'term') exp'term +	+*)$	addop→ +
$exp'term') exp'term	*)$	match, pop & adv i/p
$exp'term') exp'term	)$	Scan (adv. i/p ptr) (error)
$exp'term') exp'	)$	POP ;
$ exp'term')	)$	pop exp'→ ϵ.
$ exp'term'	$	match, adv i/p ptr
$ exp'	$	term'→ ϵ.
$	$	exp'→ ϵ.
	$	Accept
		Parsed

* Parse for (2+3)5

$exp	(2+3)5$	exp→term exp'
$exp'term	(2+3)5$	term→factor term'
$exp'term' factor	(2+3)5$	factor→ (exp)
$exp'term') exp(	(2+3)5$	match.
$exp'term') exp	2+3)5$	exp→term exp'
$exp'term') exp'term	2+3)5$	term→factor term'
$exp'term') exp'term factor	2+3)5$	factor→number
$exp'term') exp'term no.	2+3)5$	match.
$exp'term') exp'term'	+3)5$	term'→ ϵ.
$exp'term') exp'	+3)5$	exp'→ addop term exp'

$exp'term')exp'term addop	+ 3) 5 $	addop → +
$exp'term')exp'term+	+ 3) 5 $	match.
$exp'term')exp'term	3) 5 $	term → factor term'
$exp'term')exp'term'factor	3) 5 $	factor → no.
$exp'term')exp'term'no.	3) 5 $	match.
$exp'term')exp'term'	) 5 $	term' → ∈
$exp'term')exp'	) 5 $	exp' → ∈
$exp'term')	) 5 $	match.
$exp'term'	5 $	Scan (Error)
$exp'term'	$	term' → ∈
$exp'	$	exp' → ∈
$	$	Parsed

Limitation –

- Top down parsers are less efficient than bottom up parsing.
- Because it cannot parse all the languages.
- Even if it is a context free grammar, there is a limitation that the grammar should not contain left recursion or common prefixes.

22/10/19

BOTTOM UP PARSING:

(Shift / Reduce Parsers)

- The action could be either shift or reduce parsers.

Parsing stack	I/p string	Action
$	I/p $	shift / reduce
⋮	⋮	⋮
$ start var	$	accept.

Shift / Reduce Parsers
(LR Parsers)

- SLR(1)
  - Scan I/p from left to right
  - Reverse Rightmost
  - no. of look ahead symbols
- LR(1) — canonical LR parser
- LALR(1) — look ahead LR parser — YACC TOOL

- YACC TOOL uses LALR technique.
- LALR is more efficient out of all these.

* Steps Involved to build the SLR parser:

1). Augment the grammar with a new start var. by adding the production $S' \rightarrow S$.

2). Compute the collection of LR(0) items.

3). Construct or build the DFA of LR(0) items.

4). Construct the SLR(1) parsing table by using the SLR(1) algorithm.

5). Verify the parsing table by taking a valid i/p string & parse it using the parsing stack.

LR(0) Items :

- LR(0) item of a context free grammar is the production choice with the distinguished position in its right hand side.
- We will indicate this distinguished position by a period '.'.

E.g : $A \longrightarrow \alpha \underset{\downarrow}{X} \beta$ (T)

α : $(VUT)^*$; β : $(VUT)^*$

$A \longrightarrow .\alpha X \beta$ (LR(0) item) ; initial item

Shift $A \longrightarrow \alpha . X \beta$ { 'α' is parsed & expected next i/p is X.

Shift $A \longrightarrow \alpha X . \beta$ { 'αX' is parsed & expected next i/p is β.

Reduce $A \longrightarrow \alpha X \beta .$ { Complete item } { replace it by LHS }.

Ques: P. $E \to E+n \mid n$

CFG = $G_1 = (V, T, S, P)$

$= (\{E\}, \{n, +\}, \{E\}, P)$

$L(G_1) = \{2, 2+3, 23+50, 1+5+2\}$

step 1). G_1: $E' \to E$ (step 1, $S' \to S$).

$E \to E+n$

$E \to n$

step 2). I_0 set.

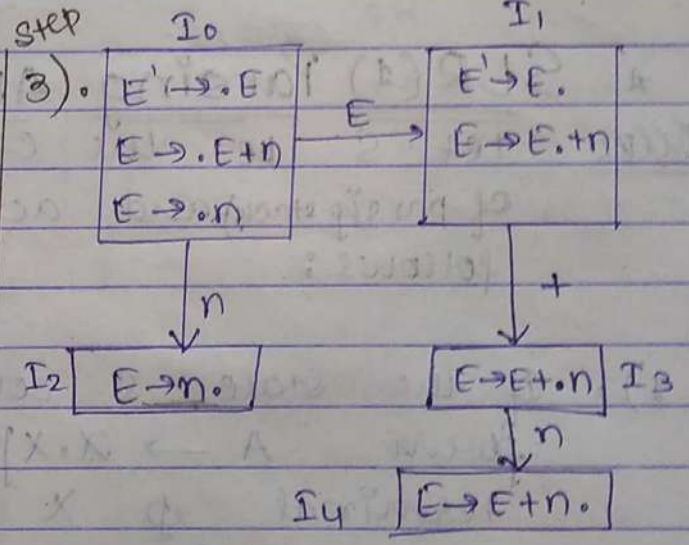
$E' \to .E$

$E \to .E+n$

$E \to .n$

If a dot is followed by a var, then take those alternatives of that var & make them as initial item.

step 3).



* Goto $(I_0, E) = I_1$

I_1 $E' \to E.$

$E \to E.+n$

~~$E \to n.$~~

* Goto $(I_0, n) = I_2$

I_2 $E \to n.$

* Goto $(I_1, +) = I_3$

I_3 $E \to E+.n$

* Goto $(I_3, n) = I_4$

I_4 $E \to E+n.$

- 5 states.
- DFA of LR(0) ~~parsers~~ items

follow(E) = {+, $}

1. $E' \rightarrow .E$
2. $E \rightarrow .E+n$
3. $E \rightarrow .n$

★ SLR(1) PT:

States	Input (terminals)			Goto (non-terminals)
	+	n	$	E → (non terminal other than E)
I_0 (0)		S2 ↳ to $I_2 = 2$		1 Goes to I_1
I_1 (1)	S3		accept → ∵ Item is $S' \rightarrow S.$	
I_2 (2)	reduce (R3)		reduce (R3) ∵ we have one complete item (reduce)	
I_3 (3)		S4		
I_4 (4)	reduce R2 ↳ prod no.		reduce R2 ↳ prod no.	

★ SLR(1) Parsing Algorithm:

Steps Let S be the current state (at the top of parsing stack) and actions are defined as follows:

1). If the state S contains any item of the form $A \rightarrow \alpha.X\beta$ where X is the terminal & X is the next token in the i/p string then the action is to **shift** s the current i/p token onto the stack & the new state ~~will~~ to be pushed on the stack is the state containing the Item $A \rightarrow \alpha X.\beta$

2). If state S contains a complete Item i.e. $A \rightarrow \alpha X\beta.$ $(A \rightarrow r.)$ and the next token in i/p string is in follow(A) then the action is to **reduce** r by rule $A \rightarrow r.$

contd. Point (2).

The reduction by the rule $S' \rightarrow S.$, where 'S' is start variable i.e. equivalent to acceptance & this will happen if the next i/p token is in $.

3). If the next i/p token is such that neither of the above cases then it is declared as error.

* Parsing Stack: ① i/p = 25+10 i.e. n+n

Parsing Stack	Input String	Action
$0 (start state (I0))	25+10$ (n+n)	S2 (adv. i/p ptr & push 2)
$0n2 (pop)	+10$	r3 (E→n) (Pop 2*length of RHS.) ∴ states are also pushed & replace by RHS
$0E1 (→ 0 on E.)	+10$	S3
$0E1+3	10$	S4
$0E1+3n4	$	r2 (E→E+n) ∴ pop 6 symb.
$0E1	$	accept

② i/p = 1+10+20 i.e. n+n+n

Parsing Stack	Input String	Action
$0	n+n+n$	S2
$0n2	+n+n$	r3 (E→n)
$0E1	+n+n$	S3
$0E1+3	n+n$	S4
$0E1+3n4	+n$	r2 (E→E+n) pop 6, push E
$0E1	+n$	S3
$0E1+3	n$	S4
$0E1+3n4	$	r2 (E→E+n)
$0E1	$	accept

Ques : $E \to E+T \mid E-T \mid T$

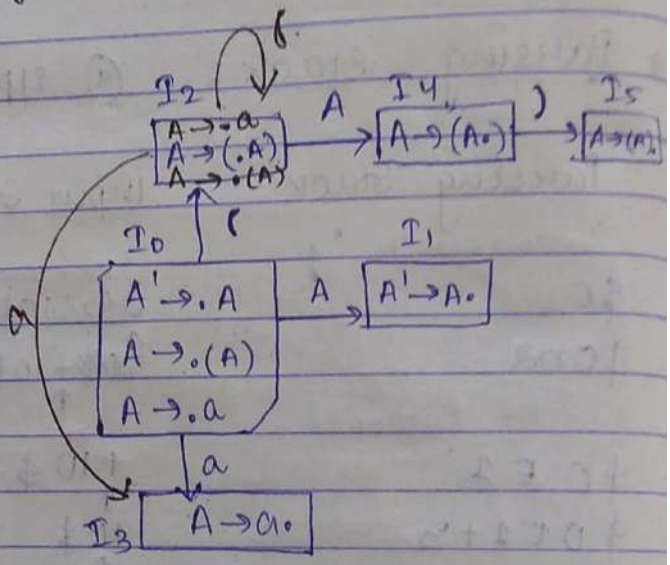
$T \to T*F \mid F$

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

* Tutorial 24/10/19

Construct the SLR(1) PT & verify by taking valid i/p string :-

1). $A \to (A) \mid a$



G' 1. $A' \to A$

2. $A \to (A)$

3. $A \to a$

I_0 $A' \to .A$

$A \to .(A)$

$A \to .a$

* Goto $(I_0, A) = I_1$

I_1 $A' \to A.$

* Goto $(I_0, () = I_2$

I_2 $A \to (.A)$

$A \to .(A)$

$A \to .a$

* Goto $(I_0, a) = I_3$

I_3 $A \to a.$

I_2

$A \to (.A)$

$A \to .(A)$

$A \to .a$

* Goto $(I_2, A) = I_4$

I_4 $A \to (A.)$

* Goto $(I_2, () = I_2$

* Goto $(I_2, a) = I_3$

$A \to a. = I_3$

* Goto $(I_4,)) = I_5$

I_5 $A \to (A).$

follow(A) = {$,)}

*SLR(1) PT :-

States	Terminals				Goto
	(	)	a	$	Non-terminals A
0	S2		S3		1
1				accept	
2	S2		S3		4
3		r3		r3	
4		S5			
5		r2		r2	

* Parsing Stack :

stack	i/p string	Action
$0	(a) $	S2
$0(2	a) $	S3
$0(2a3	) $	r3 A→a
$0(2A4	) $	S5
$0(2A4)5	$	r2 A→(A)
$0A1	$	accept

2). S → (S)S | ∈

G' S' → S
S → (S)S
S → .

I0 S' → .S
S → .(S)S
S → .

* Goto (I0, S) = I1
I1 S' → S.

* Goto (I0, ( ) = I2
I2 S → (.S)S
S → .(S)S
S → .

* Goto (I2, S) = I3
I3 S → (S.)S

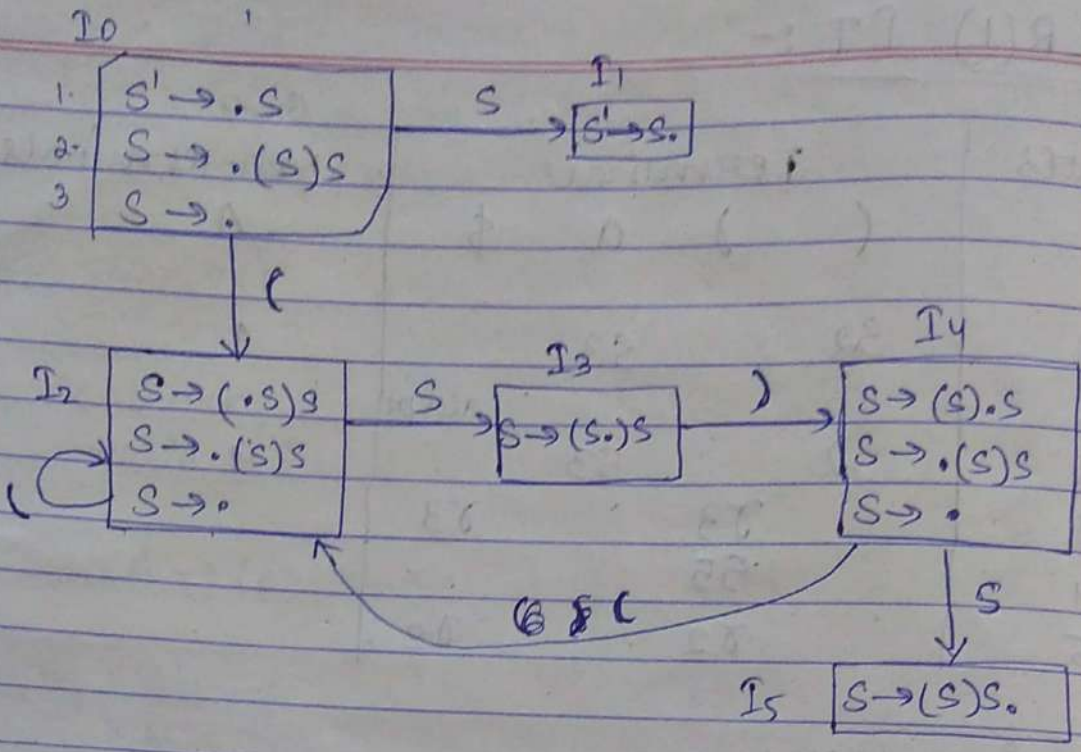
* Goto (I2, ( ) = I2
S → (.S)S
S → .(S)S
S → .

* Goto (I3,)) = I4
I4 S → (S).S
= S → .(S)S

* Goto (I4, S)
S → (S)S. I5

* Goto (I4, ( )
= I2

follow (S) = {$,)}



★ SLR(1) Parsing Table:

States	Terminals			Goto
	(	)	$	Non-terminals S
0	S2	r3	r3	1
1			accept	
2	S2	r3	r3	3
3		S4		
4	S2	r3	r3	5
5		r2	r2	

★ Parsing Stack:

Stack	I/p string	Action
$0	()()$	S2
$0(2	)()$	r3 S → .
$0(2S3	)()$	S4
$0(2S3)4	()$	S2
$0(2S3)4(2	)$	r3 S → .
$0(2S3)4(2S3	)$	S4
$0(2S3)4(2S3)4	$	r3 S → .
$0(2S3)4(2S3)4S5	$	r2 S → (S)S
	S	

$ 0(2s3) 4 5 5 $ 0 S 1	$ $	r2 S → .(S)S accept

Ques: S → (L) Parse (x,(x))
S → x
L → L, S | S

G': S' → S
S → (L)
S → x
L → L, S
L → S

I_0:
S' → .S
S → .(L)
S → .x
L → .L, S
L → S

SLR(1) Grammar or not :

- Grammar is SLR(1) iff, for any state 's' the following two conditions are satisfied:

① For any item, $A \to \alpha . x\beta$ in state 's' with x as a terminal, there is no complete item $B \to \gamma .$ in s with x in follow(B).

② For any two complete items, $A \to \alpha .$ and $B \to \beta .$ in 's', follow(A) $\cap$ follow(B) is empty.

- The violation of the 1st of these conditions represent Shift-Reduce Conflict.
- The violation of the 2nd condition, represent Reduce-Reduce Conflict.
- If either of these cond^{n} is not satisfied, then the grammar is not a SLR(1) grammar.

Imp

Ques Show that the given grammar is SLR(1).

Sol^{n} - Construct PT and show all have unique entries.

Ques :

stmt → call-stmt | assign-stmt
call-stmt → identifier
assign-stmt → var := exp
var → var [exp] | identifier
exp → var | number

Short Hand

S → C | A
C → id
A → V := E
V → V [E] | id
E → V | num

follow(C) = {$}
follow(V) = {$, [, :=}

Ex

Soln:

$S' \rightarrow .S$
$S \rightarrow .C$
$S \rightarrow .A$
$C \rightarrow .id$
$A \rightarrow .V := E$
$V \rightarrow .V[E]$
$V \rightarrow .id$

Here, ∵ Intersection follow(C) ∩ follow(V) is not empty. ∴ Under '$', it will produce reduce-reduce conflict.

Ex
$C \rightarrow id.$
$V \rightarrow id.$

↓
limitation of SLR(1).

* LR(1) parser | Canonical LR(1) CLR(1):

SLR(1) LR(0) items $[A \rightarrow .\alpha x \beta]$

↓

CLR(1) or LR(1) LR(1) items $[A \rightarrow .\alpha x \beta, a]$

$[A \rightarrow .\alpha x \beta$: LR(0) item; a : lookahead symbol; whole: LR(1) item.

- In CLR(1) we use lookahead symbols to make decision easier & thus no such conflicts will occur as in SLR(1).

Steps:

1). Add new production $S' \rightarrow S$ to the grammar.
2). Compute collection of LR(1) item sets.
3). Construct DFA of LR(1) items.
4). Construct LR(1) parsing table using algorithm.
5). Verify the valid i/p string using passing stack.

[A → α.Xγ, a] → spontaneously lookahead

lookahead for next items = first(γa)

Ques: G ≡ A → (A) | a

G' ≡ 1. A' → A
2. A → (A)
3. A → a

I_0 A' → .A, $ (for start var. $ is lookahead)

∴ first(γa) = $
∴ A → .(A), $
A → .a, $

DFA of LR(1) items (Propagating lookahead)

I_0
A' → .A, $
A → .(A), $
A → .a, $

A → I_1 A' → A., $ (same lookahead gets carried forward)

(→ I_2
A → (.A), $
A → .(A),)
A → .a,)

first(')') = ') spontaneous lookahead

a → I_3 A → a., $ (propagating lookahead)

A → I_4 A → (A.), $

) → I_7 A → (A).,

(→ I_5
A → (.A),)
A → .(A),)
A → .a,)

a → I_6 A → a.,)

A → I_8 A → (A.),)

) → I_9 A → (A).,)

Algorithm:

Let 's' be the current state (at the top of the parsing stack) then actions are defined as follows:

1). If state 's' contains any LR(1) item of the form $A \to \alpha . x\beta , a$ where x is a terminal & x is the next token in the i/p string then the action is to Shift the current i/p token onto the stack & the new state to be pushed on the stack containg the LR(1) item $A \to \alpha x . \beta , a$.

2). If state 's' contains the complete LR(1) item i.e. $A \to \alpha . , a$ & the next i/p token string is A then the action is reduce by the rule $A \to \alpha$.

A reduction by the rule $S' \to S .$, where S is the start symbol is equivalent to acceptance & this will happen only if the next i/p token is $.

3). If the i/p token is such that, neither the above two cases then it is an error.

General LR(1).

* Parsing Table :

States	input states (Term.)				Goto (V) (Non-ter
	(	)	a	$	A
0	S2		S3		1
1				accept	
2	S5		S6		4
3				r3	
4		S7			
5	S5		S6		8
6		r3			
7				r2	
8		S9			
9		r2			

* Parsing Stack :

For (a)

Parsing stack	input strip	action
$0	(a) $	S2
$0(2	↑a) $	S6
$0(2a6	)↑$	r3 A→a Pop 2 i
$0(2A4	)↑$	S7
$0(2A4)7	↑$	r2 A→(A) pop 6 items
$0A1	$↑	accept

Ques Construct LR(1) PT of :-

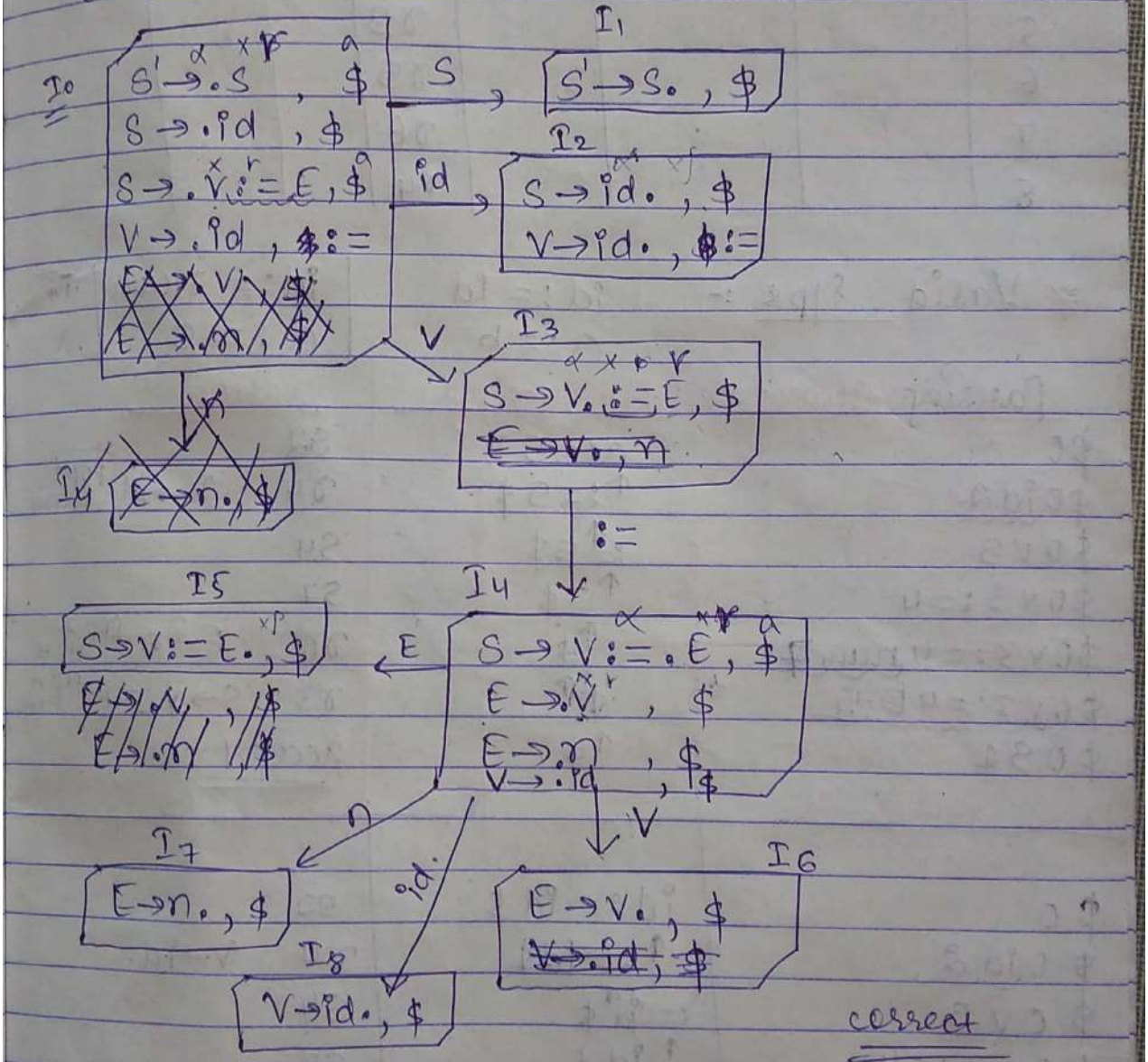
S → Id | V := E

V → id

E → V | n

:=

G' : 1. S' → S
2. S → id
3. S → V := E
4. V → id
5. E → V
6. E → n



correct

Valid Inputs :-

a := b
a := 50
a

id := id
id := num
id

★ LR(1) PT:-

States	i/p & terminals				Goto (Non Term		
	id	n	:=	$	S	V	E
0	S2				1	3	
1				accept			
2			r4	r2			
3			S4				
4	S8	S7				6	5
5				r3			
6				r5			
7				r6			
8				r4			

✳ Valid i/ps :- id := id | id := num
a := b | a := 5.

Parsing Stack	i/p string	action
$0	a := 5 $	S2
$0id2	↑ := 5 $	r4 V→id Pop 2 ite
$0V3	:= ↑ 5 $	S4
$0V3:=4	↑ 5 $	S7
$0V3:=4num7	↑ $	r6 E→n Pop 2 ite
$0V3:=4E5	$ ↑	r3 S→V:=E Pop
$0S1	↑ $	accept

$0	id := id $	
$0id2	↑ := id $	S2
$0V3	:= ↑ id $	r4 V→id.
$0V3:=4	↑ id $	S4
$0V3:=4id8	↑ $	S8
$0V3:=4V6	$ ↑	r4 V→id
$0S1 0V3:=4V6	$ ↑	r5 E→E
✳ Checked later	↑	accept

Non Terminals

V	E
3	
6	5

num | id
5. | a

→ id Pop 2 items

→ n Pop 2 items

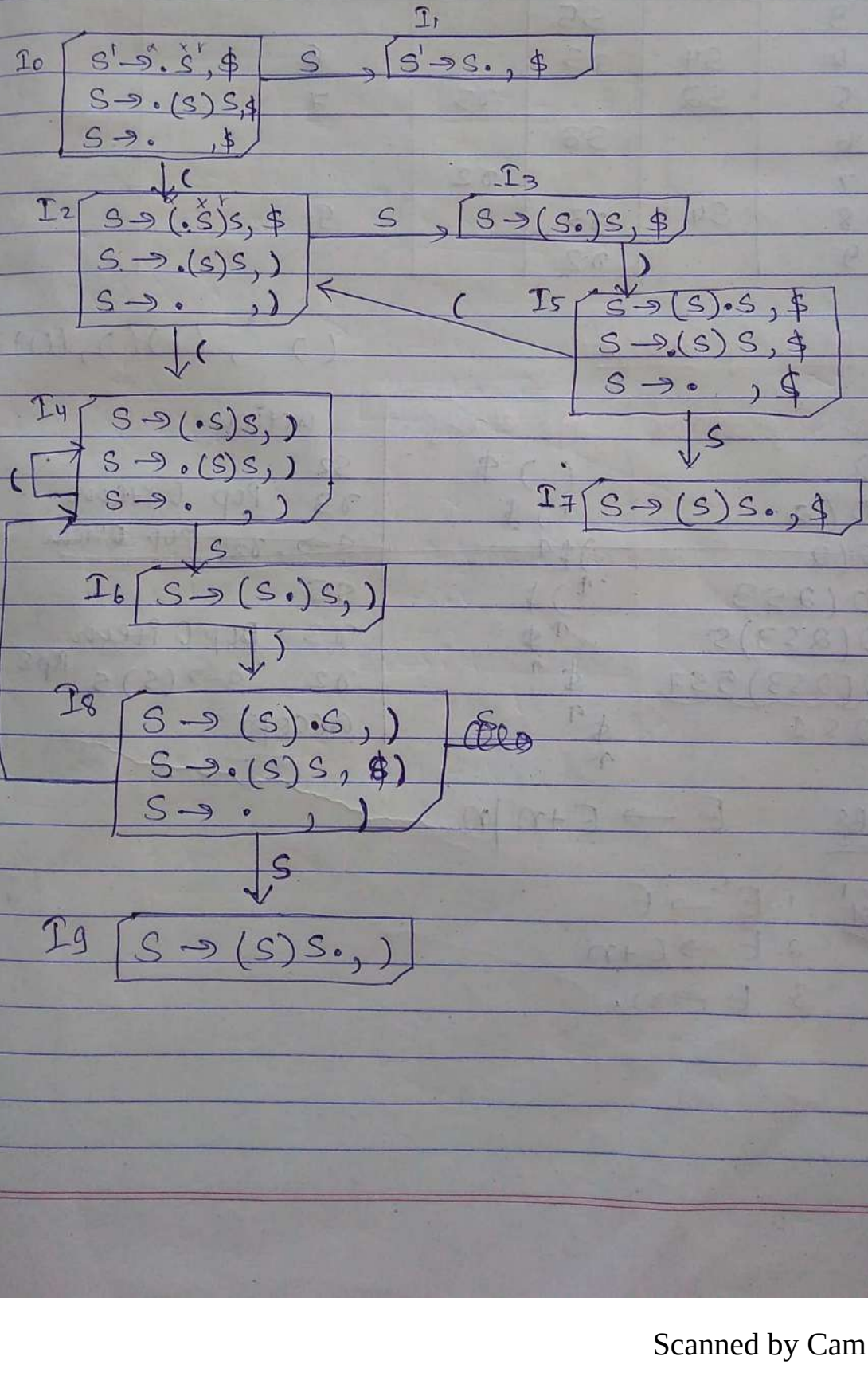
V := E Pop 6 items

id.

E → V

Ques $S \to (S)\,S \mid \epsilon$

G' 1. $S' \to S$
2. $S \to (S)\,S$
3. $S \to \epsilon$



* PT

States	Input (	)	$	GOTO S
0	S2		r3	1
1			accept	
2	S4	r3		3
3		S5		
4	S4	r3		6
5	S2		r3	7
6		S8		
7			r2	
8	S4	r3		9
9		r2		

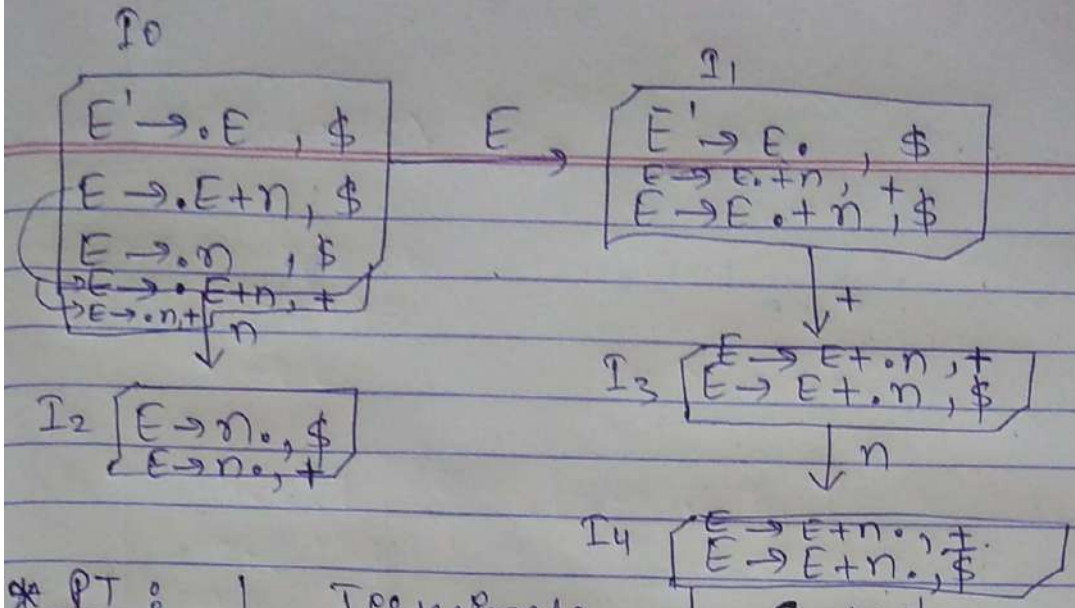
( ) , ( )( ), (())

* Parsing stack:

Parsing Stack	I/p String	action
$0	( ) $	S2
$0(2	↑) $	r3 Pop 0 item
~~$0(2~~	) $ ↑	~~S → . r3 pop 0 item~~
$0(2S3	↑) $	S5
$0(2S3)S	↑ $	r3 Pop 0 item
$0(2S3)5S7	$ ↑	r2 S → (S)S pop 8
$0S1	$ ↑	accept.

Ques E → E+n | n

G' 1. E' → E
2. E → E+n
3. E → n.



* PT :

States	Terminals			Goto
	+	n	$	E
0	.	S2		1
1	S3		accept	
2	r3		r3	
3		S4		
4	r2		r2	

* Stack : 2+3 =n+n.

Parsing Stack	I/p string	action
$0	2+3$	S2
$0n2	+3$	r3 E→n.
$0E1	+3$	S3
$0E1+3	3$	S4
$0E1+3n4	$	r2 E→E+n
$0E1	$	accept