SNS COLLEGE OF TECHNOLOGY

Coimbatore-36.
An Autonomous Institution

Accredited by NBA – AICTE and Accredited by NAAC – UGC with ‘A+’ Grade
Approved by AICTE, New Delhi & Affiliated to Anna University, Chennai

COURSE NAME : 19CSB301&Automata Theory and Compiler Design

III YEAR/ V SEMESTER

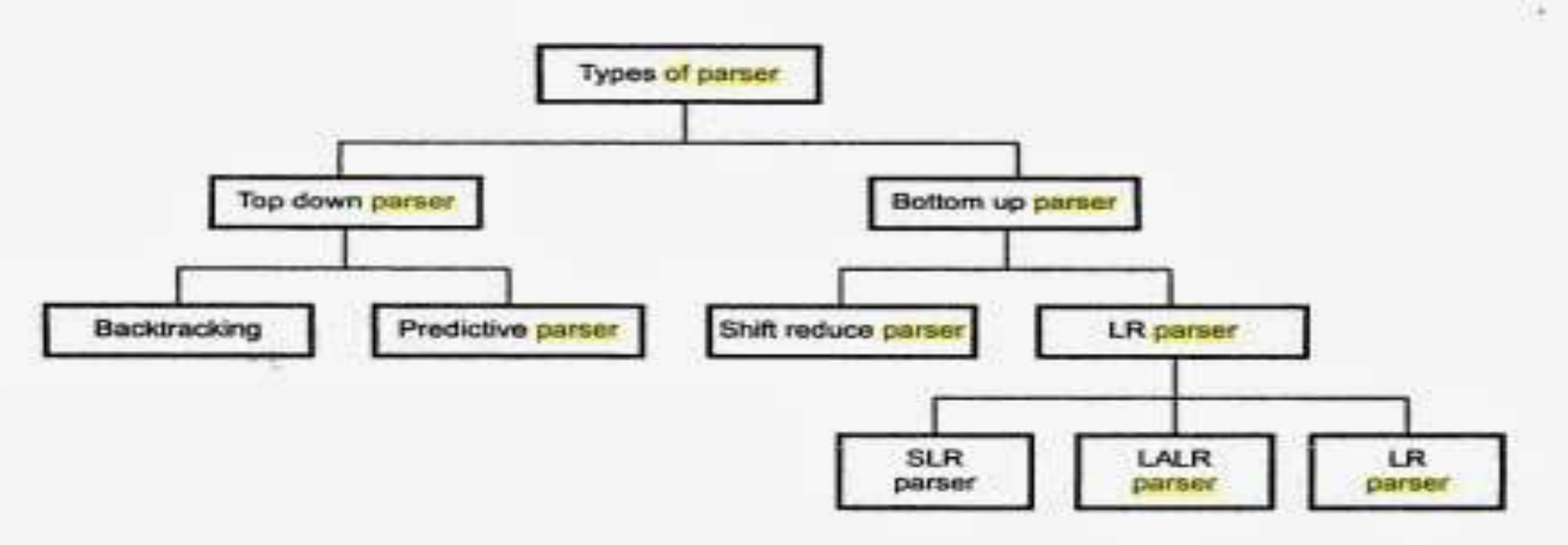
UNIT – III SYNTAX ANALYSIS AND SEMANTIC ANALYSIS

Topic: Top Down Parsing

Dr.B.Vinodhini
Associate Professor
Department of Computer Science and Engineering

Top Down Parsing – Definition & Types

Attempt to find a left-most derivation for an input string or an attempt to construct a parse tree for the input starting from the root to the leaves.



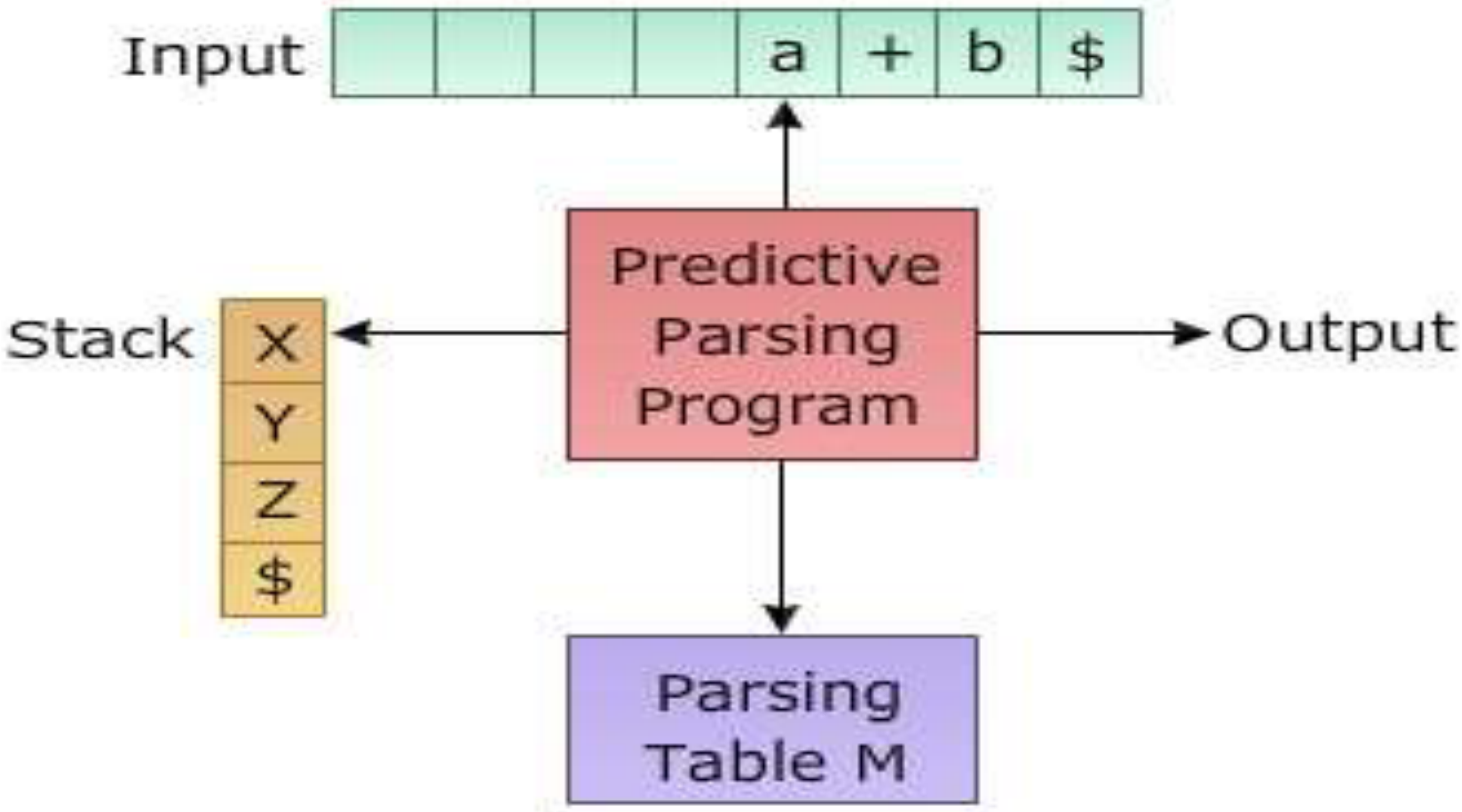
PREDICTIVE PARSING

LL(1) Parser or Predictive Parser or Non recursive descent Parser

- The first L indicates that the input is read from left to right.
- The second L says that it produces a left-to-right derivation.
- And the 1 says that it uses one lookahead token. (Some parsers look ahead at the next 2 tokens, or even more than that.)

PREDICTIVE PARSING

No backtracking



PREDICTIVE PARSING

Construction of LL(1) Parser

1.Elimination of Left Recursion

2.Elimination of Left Factoring

3.Calculation of First and Follow

4.Construction of Parsing Table

5.Check Whether i/p String accepted by Parser or Not

Predictive parsing program

Parsing table: It is a two-dimensional array M[A, a], where ‘A’ is a non-terminal and ‘a’ is a terminal.

Predictive parsing program:

1. If **X = a = $,** the parser halts & successful completion.

2. If **X = a ≠ $,** the parser pops X off the stack and advances the input pointer to the next input symbol.

3. If X is a non-terminal , the program consults **entry M[X, a]** of the parsing table M..
If M[X, a] = {X →UVW},the parser replaces X on top of the stack by UVW

4.If **M[X, a] = error**, the parser calls an error recovery routine.

Algorithm for non recursive predictive parsing

Algorithm for nonrecursive predictive parsing:

Input : A string w and a parsing table M for grammar G.
Output : If w is in L(G), a leftmost derivation of w; otherwise, an error indication.
Method : Initially, the parser has $S on the stack with S, the start symbol of G on top, and w$ in the input buffer.

```
repeat
let X be the top stack symbol and a the symbol pointed to by ip;
        if X is a terminal or $ then
                if X = a then
                pop X from the stack and advance ip
                else error()
        else        /* X is a non-terminal */
        if M[X, a] = X →Y1Y2 ... Yk then begin
                pop X from the stack;
                push Yk, Yk-1, ... ,Y1 onto the stack, with Y1 on top;
                output the production X →Y1 Y2 . . . Yk
        end
        else error()
until X = $
```

Predictive parsing table Construction

Predictive parsing table construction:

Compute FIRST and FOLLOW

Rules for first( ):

1. If X is terminal, then FIRST(X) is {X}.
2. If X →ε is a production, then add ε to FIRST(X).
3. If X is non-terminal and X →aα is a production then add a to FIRST(X)
4. If X is non-terminal and X → Y1 Y2...Yk is a production, then place a in FIRST(X) if for some i, a is in FIRST(Yi), and ε is in all of FIRST(Y1),...,FIRST(Yi-1); that is, Y1,....Yi-1 => ε. If ε is in FIRST(Yj) for all j=1,2,..,k, then add ε to FIRST(X).

Rules for follow( ):

1. If S is a start symbol, then FOLLOW(S) contains $.
2. If there is a production A →αBβ, then everything in FIRST(β) except ε is placed in follow(B).
3. If there is a production A →αB, or a production A →αBβ where FIRST(β) contains ε, then everything in FOLLOW(A) is in FOLLOW(B).

Example

Given Grammar

E → E+T
E → T
T → T*F
T →F
F →(E)
F →id

Not an LL(1) Grammar

Eliminate Left Recursion

LL(1) Grammar

E → TE’
E’ → +TE’ |ε
T → FT’
T’ → *FT’ | ε
F → (E) | id

First( ) :

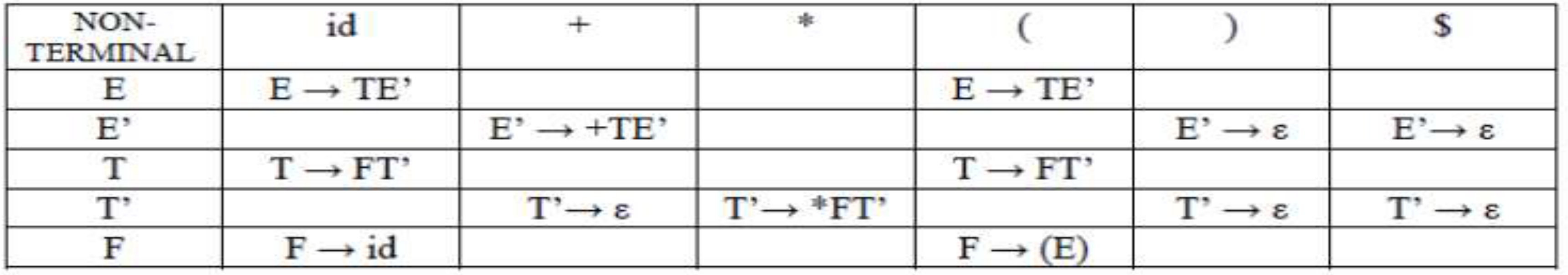
FIRST(E) = { (, id}
FIRST(E’) ={+ ,ε }
FIRST(T) = { (, id}
FIRST(T’) = {*, ε }
FIRST(F) = { (, id }

Follow( ):

FOLLOW(E) = { $,) }
FOLLOW(E’) = { $,) }
FOLLOW(T) = { +, $,) }
FOLLOW(T’) = { +, $,) }
FOLLOW(F) = {+, * , $,) }

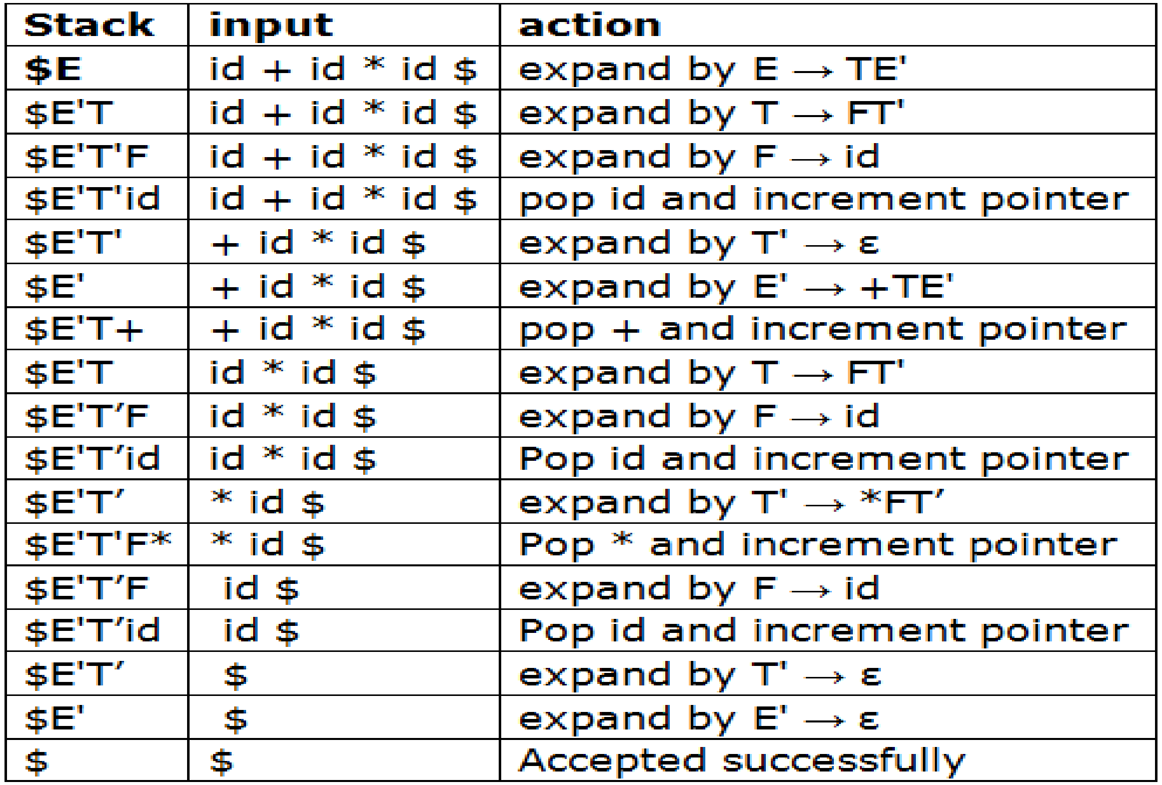
Put the entries in the table according to the algorithm

Predictive parsing table :

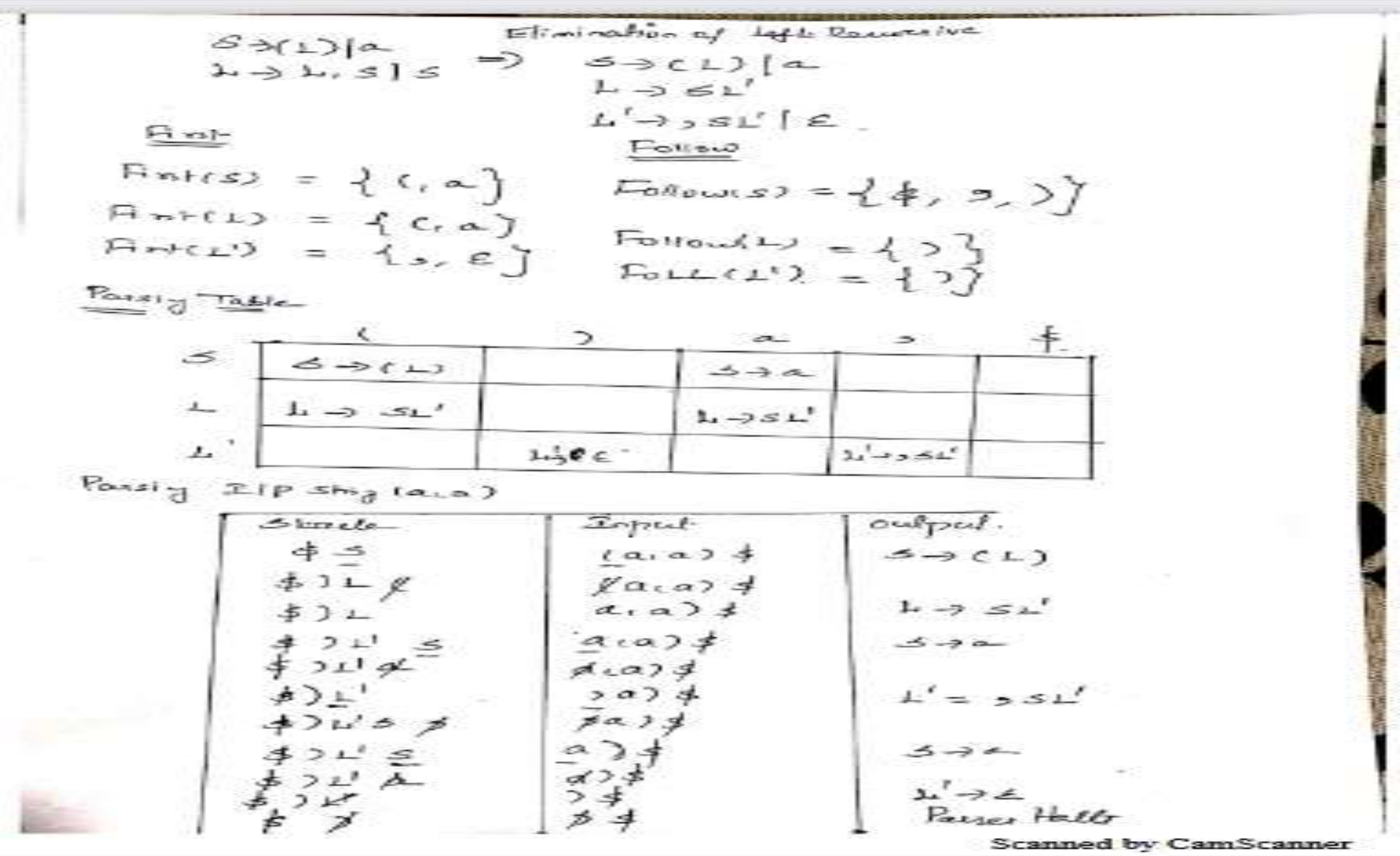


NON-TERMINAL	id	+	*	(	)	$
E	E → TE’			E → TE’		
E’		E’ → +TE’			E’ → ε	E’→ ε
T	T → FT’			T → FT’		
T’		T’→ ε	T’→ *FT’		T’ → ε	T’ → ε
F	F → id			F → (E)		

Check whether the given input string id+id*id is accepted by grammar or not



Stack	input	action
$E	id + id * id $	expand by E → TE'
$E'T	id + id * id $	expand by T → FT'
$E'T'F	id + id * id $	expand by F → id
$E'T'id	id + id * id $	pop id and increment pointer
$E'T'	+ id * id $	expand by T' → ε
$E'	+ id * id $	expand by E' → +TE'
$E'T+	+ id * id $	pop + and increment pointer
$E'T	id * id $	expand by T → FT'
$E'T’F	id * id $	expand by F → id
$E'T’id	id * id $	Pop id and increment pointer
$E'T’	* id $	expand by T' → *FT’
$E'T'F*	* id $	Pop * and increment pointer
$E'T’F	id $	expand by F → id
$E'T’id	id $	Pop id and increment pointer
$E'T’	$	expand by T' → ε
$E'	$	expand by E' → ε
$	$	Accepted successfully



S → (L) | a
L → L, S | S ⇒

Elimination of left Recursive

S → (L) | a
L → SL'
L' → ,SL' | ε

First

First(S) = { (, a }
First(L) = { (, a }
First(L') = { ,, ε }

Follow

Follow(S) = { $, ,,) }
Follow(L) = {) }
Foll(L') = {) }

Parsing Table

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

Parsing I/P string (a,a)

Stack	Input	output.
$S	(a,a)$	S → (L)
$)L~~(~~	~~(~~a,a)$	
$)L	a,a)$	L → SL'
$)L'S	a,a)$	S → a
$)L'~~a~~	~~a~~,a)$	
$)L'	,a)$	L' = ,SL'
$)L'S~~,~~	~~,~~a)$	
$)L'S	a)$	S → a
$)L'~~a~~	~~a~~)$	
$)~~L'~~	)$	L' → ε
$~~)~~	~~)~~$	Parser Halts

Thank You