PARSING

- The process of deriving the string from the given grammar is known as parsing (derivation).
- Depending upon how parsing is done we have two types of parser :
  - Top Down Parser
    - → Back Tracking
    - → Predictive Parser
  - Bottom Up Parser
    - → Shift - Reduce Parser
    - → LR Parser

L L (1) PARSER


scanning the input from left to right
leftmost derivation
using only one input symbol at a time

Steps to convert LL(1) parser

- Firstly check if the grammars contain -
  - Left Recursion
  - Left Factoring
- Then go for -
  - FIRST
  - FOLLOW
  - Predictive Parsing Table
  - String (if given)

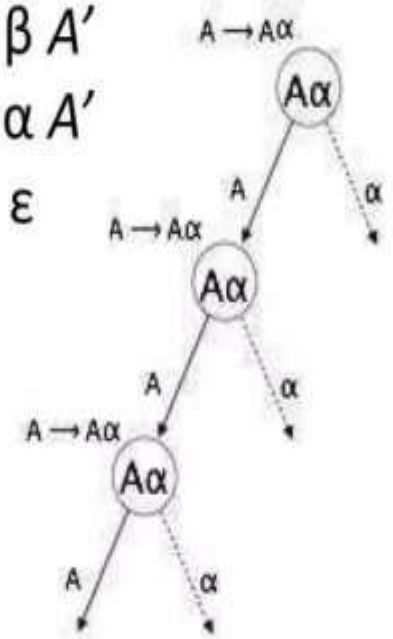
Left Recursion

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

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A'$

$\mid \varepsilon$

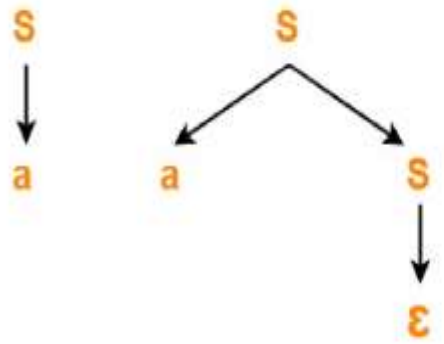


Left Factoring

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$

$A \rightarrow \alpha A'$

$A' \rightarrow \beta_1 \mid \beta_2$



EXAMPLE

- E → TA
- A → +TA / ε
- T → VB
- B → *VB / ε
- V → id / (E)

- NOTE :- Here we can see that there is no left recursion or left factoring in this example so now we will find first() .

FIRST

- FIRST is applied to the R.H.S. of a production rule :
- If first symbol is terminal then put into first(non-terminal).
- If non-terminal then go to that non-terminal production and continue above step.
- If ε → directly then put in first(non-terminal).
  → indirectly then put & check again.

$E \rightarrow TA$

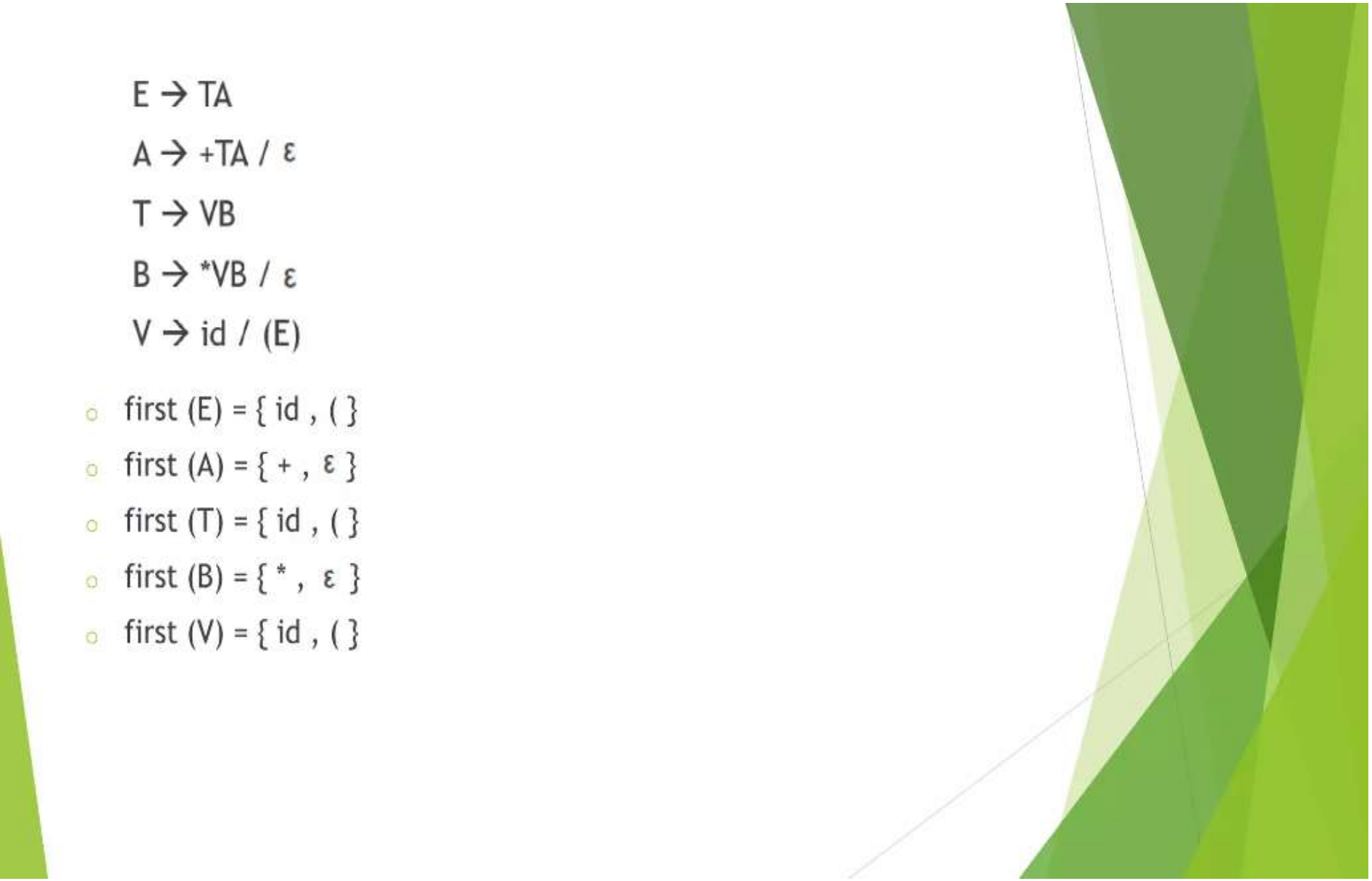
$A \rightarrow +TA \mid \varepsilon$

$T \rightarrow VB$

$B \rightarrow *VB \mid \varepsilon$

$V \rightarrow id \mid (E)$

- first (E) = { id , (}
- first (A) = { + , ε }
- first (T) = { id , (}
- first (B) = { * , ε }
- first (V) = { id , (}



FOLLOW

- For starting symbol put always $.
- Find the non-terminal in R.H.S. whose follow has to be found in the grammar.
- If its's next element is
  - ➔ terminal then put into follow(non-terminal).
  - ➔ no terminal then copy follow(non-terminal) from which it is found.

    ie. E → TE' follow(E')=follow(E)
  - ➔ non-terminal then check the value of first(next)
    - -> if it is terminal then put into follow(non-terminal).
    - -> if ε then put back & check again.

E → TA

A → +TA / ε

T → VB

B → *VB / ε

V → id / (E)

- first (E) = { id , (}
- first (A) = { + , ε }
- first (T) = { id , (}
- first (B) = { * , ε }
- first (V) = { id , (}

- follow(E) = { $,) }
- follow(A) = { $,) }
- follow(T) = { + , $,) }
- follow(B) = { + , $,) }
- follow(V) = { * , + , $,) }

Predictive Parsing Table

- Form a table whose
  - → row1 contain all terminal from grammar set including $ and excluding ε .
  - → column1 contains all non-terminals from grammar set.
- For non-terminal ie. n1 (n1→A) , if A is -
  - → terminal then put that production in row(terminal),column(n1).
  - → non-terminal then check first(n1) & put production rule in that symbol.
  - → ε then check follow(n1) & put into that symbol.
- If in any cell , we get two production then that grammar set will not be parsed by LL(1) grammar & so it should be solved by **Recursive Decent Parsing.**

E → TA	first (E) = { id , (}	follow(E) = { $,) }
A → +TA / ε	first (A) = { + , ε }	follow(A) = { $,) }
T → VB	first (T) = { id , (}	follow(T) = { + , $,) }
B → *VB / ε	first (B) = { * , ε }	follow(B) = { + , $,) }
V → id / (E)	first (V) = { id , (}	follow(V) = { * , + , $,) }

Symbol	id	(	)	+	*	$
E	E→TA	E→TA				
A			A→ ε	A→+TA		A→ ε
T	T→VB	T→VB				
B			B→ ε	B→ ε	B→*VB	B→ ε
V	V→id	V→(E)				

Parse Input String
(optional)

- Parse given string from left to right.
- Start from $ and starting symbol.
- Replace symbol with it's production, from which we can form the given input string.
- Also write the action which we performed in stack.
- Continue till we are left with $ in stack and input.

String :- id * id

E → TA
A → +TA / ε
T → VB
B → *VB / ε
V → id / (E)

Stack	Input	Action
$ E	id * id $	
$ A T	id * id $	E → TA
$ A B V	id * id $	T → VB
$ A B id	id * id $	V → id
$ A B	* id $	POP
$ A B V *	* id $	B → *VB
$ A B V	id $	POP
$ A B id	id $	V → id
$ A B	$	POP
$ A	$	B → ε
$	$	A → ε