Non- Recursive Predictive Parsing

- Implemented using stack instead of recursive calls.
- Key problem- determining the production to be used for a non-terminal.

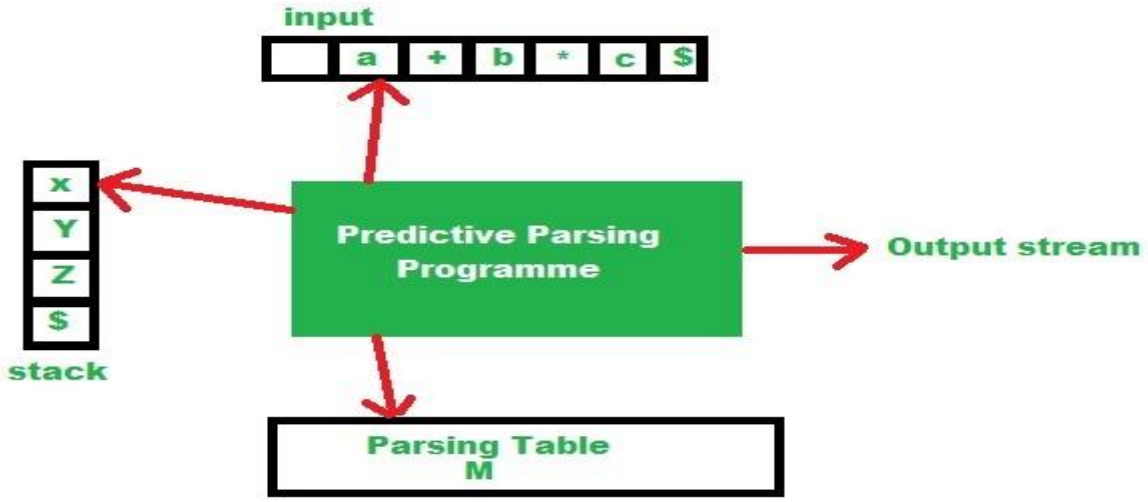


fig: Non Recursive Parser Model

(Source: https://www.geeksforgeeks.org/algorithm-for-non-recursive-predictive-parsing)

SYNTAX ANALYSIS

FIRST and FOLLOW Set in Parsing.

The construction of a predictive parser is aided by two functions associated with a grammar G. These functions, FIRST and FOLLOW, allow us to fill in the entries of a predictive parsing table for G, whenever possible.

Why FIRST set is required?

Back tracking in syntax analysis is really a complex process to implement. The easier process is as follows:

If the compiler would have come to know in advance, that what is the “first character of the string produced when a production rule is applied”, and comparing it to the current character or token in the input string it sees, it can wisely take decision on which production rule to apply.

Source: https://www.geeksforgeeks.org/why-first-and-follow-in-compiler-design/

- Let us consider the following grammar:

$$S \rightarrow cAd$$

$$A \rightarrow bc \mid a$$

Input string is “cad”.

In the above example, if it knew that after reading character ‘c’ in the input string and applying the rule: S → cAd, next character in the input string is ‘a’, then it would have ignored the production rule A->bc (because ‘b’ is the first character of the string produced by this production rule, not ‘a’), and directly use the production rule A->a.

Source:https://www.geeksforgeeks.org/why-first-and-follow-in-compiler-design/

- Hence it is validated that if the compiler/parser knows about first character of the string that can be obtained by applying a production rule, then it can wisely apply the correct production rule to get the correct syntax tree for the given Input string.
- Why FOLLOW set is required?

Let us consider the grammar:

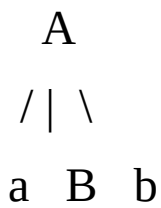
$$A \rightarrow aBb$$

$$B \rightarrow c \mid \varepsilon$$

Suppose the string to be parsed is “ab”.

Source: https://www.geeksforgeeks.org/why-first-and-follow-in-compiler-design/

- As the first character in the input is a, the parser applies the rule A→aBb.

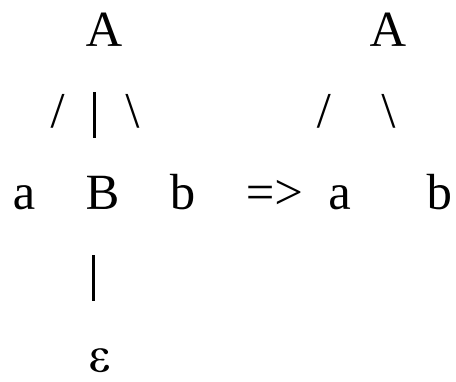


Now the parser checks for the second character of the input string which is b, and the Non-Terminal to derive is B, but the parser can’t get any string derivable from B that contains b as first character.

Source: https://www.geeksforgeeks.org/why-first-and-follow-in-compiler-design/

- But the Grammar does contain a production rule B -> ε, if that is applied then B will vanish, and the parser gets the input “ab” , as shown below. But the parser can apply it only when it knows that the character that follows B in the production rule is same as the current character in the input.
- In RHS of A -> aBb, b follows Non-Terminal B, i.e. FOLLOW(B) = {b}, and the current input character read is also b. Hence the parser applies this rule. And it is able to get the string “ab” from the given grammar.

- Source: https://www.geeksforgeeks.org/why-first-and-follow-in-compiler-design/



FIRST and FOLLOW sets for a given grammar, so that the parser can properly apply the needed rule at the correct position.

Source: https://www.geeksforgeeks.org/why-first-and-follow-in-compiler-design/

- Algorithm for finding FIRST(α).
- If a is any string of grammar symbols, let FIRST(a) be the set of terminals that begin the strings derived from a. If a=> ε then ε is also in FIRST(α).
- To compute FIRST(X) for all grammar symbols X, apply the following rules until no more terminals or ε can be added to any FIRST set:

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 A→ BC then FIRST (A) = FIRST (B) if FIRST (B) doesn't contain ε. If FIRST (B) contains ε then we have to substitute ε in place of B. We then have FIRST (A)= FIRST (B) U FIRST (C) .

- Source: https://www.geeksforgeeks.org/why-first-and-follow-in-compiler-design/

Example:

Find the FIRST set from the following grammar:

$E \rightarrow T\ E'$

$E' \rightarrow +\ T\ E' | \varepsilon$

$T \rightarrow F\ T'$

$T' \rightarrow *\ F\ T' | \varepsilon$

$F \rightarrow (\ E\ ) \mid id$

FIRST(E) = FIRST(T) = FIRST(F) = {(, id}

FIRST(E') = {+, ε}

FIRST(T') = {*, ε}

- Consider the production rule-

$$A \rightarrow abc / def / ghi$$

Calculate the FIRST set.

FIRST (A)= {a,d,g}

Source: https://www.gatevidyalay.com/first-and-follow-compiler-design/

- Calculate the FIRST set for the given grammar-

$S \rightarrow aBDh$
$B \rightarrow cC$
$C \rightarrow bC / \in$
$D \rightarrow EF$
$E \rightarrow g / \in$
$F \rightarrow f / \in$

- First(S) = { a } First(C) = { b , ∈ } First(F) = { f , ∈ }
- First(B) = { c } First(D) = { First(E) – ∈ } ∪ First(F) = { g , f , ∈ } First(E) = { g , ∈ }
- Source: https://www.gatevidyalay.com/first-and-follow-compiler-design/

- Calculate the FIRST set for the given grammar-

S→ Aa

A → BD

B → b | ε

D → d | ε

FIRST (S) ={b, d, a}

FIRST (A)={b, d, ε}

FIRST (B)={b, ε}

FIRST (D)={d, ε}

Calculation of the FOLLOW set

- Follow(α) is a set of terminal symbols that appear immediately to the right of α.

- **Rules For Calculating Follow Function-**

1. For the start symbol S, place $ in Follow(S).

2. For any production rule A → αB, we have:
   Follow(B) = Follow(A)
3. For any production rule A → αBβ, we have:

✓If ∈ ∉ First(β), then Follow(B) = First(β)
✓If ∈ ∈ First(β), then Follow(B) = { First(β) – ∈ } ∪ Follow(A)

- Note:
- ∈ may appear in the first function of a non-terminal.
- ∈ will never appear in the follow function of a non-terminal.
- Before calculating the first and follow functions, eliminate **Left Recursion** from the grammar, if present.
- We calculate the FOLLOW function of a non-terminal by looking where it is present on the RHS of a production rule.

Solved Problems:

1. Calculate the FOLLOW functions for the given grammar-

$$S \rightarrow aBDh$$
$$B \rightarrow cC$$
$$C \rightarrow bC / \in$$
$$D \rightarrow EF$$
$$E \rightarrow g / \in$$
$$F \rightarrow f / \in$$

- First(S) = { a }
- First(B) = { c }

- First(C) = { b , ∈ }
- First(D) = { First(E) – ∈ } ∪ First(F) = { g , f , ∈ }
- First(E) = { g , ∈ }
- First(F) = { f , ∈ }

- Follow(S) = { $ } Follow(B) = { First(D) – ∈ } ∪ First(h) = { g , f , h }
- Follow(C) = Follow(B) = { g , f , h } Follow(D) = First(h) = { h }

Follow(E) = { First(F) – ∈ } ∪ Follow(D) = { f , h }

Follow(F) = Follow(D) = { h }

2. Calculate the follow functions for the given grammar:

$$S \rightarrow A$$

$$A \rightarrow aB \,/\, Ad$$

$$B \rightarrow b$$

$$C \rightarrow g$$

We need to eliminate Left Recursion from the given grammar. After left recursion removal, we have the grammar as follows:

$$S \rightarrow A$$
$$A \rightarrow aBA'$$
$$A' \rightarrow dA' / \in$$
$$B \rightarrow b$$
$$C \rightarrow g$$

- First(S) = First(A) = { a }
- First(A) = { a }
- First(A') = { d , ∈ }
- First(B) = { b }
- First(C) = { g }

- Follow(S) = { $ } Follow(B) = { First(A’) – ∈ } ∪ Follow(A) = { d , $ }
- Follow(A) = Follow(S) = { $ }
- Follow(A’) = Follow(A) = { $ }

- Calculate the follow functions for the given grammar-

$$S \rightarrow ACB \,/\, CbB \,/\, Ba$$

$$A \rightarrow da \,/\, BC$$

$$B \rightarrow g \,/\, \in$$

$$C \rightarrow h \,/\, \in$$

- First(S) = { First(A) – ∈ } ∪ { First(C) – ∈ } ∪ First(B) ∪ First(b) ∪ { First(B) – ∈ } ∪ First(a) = { d , g , h , ∈ , b , a }
- First(A) = First(d) ∪ { First(B) – ∈ } ∪ First(C) = { d , g , h , ∈ }
- First(B) = { g , ∈ } First(C) = { h , ∈ }

- Follow(S) = { $ }
- Follow(A) = { First(C) – ∈ } ∪ { First(B) – ∈ } ∪ Follow(S) = { h , g , $ }
- Follow(B) = Follow(S) ∪ First(a) ∪ { First(C) – ∈ } ∪ Follow(A) = { $, a , h , g }
- Follow(C) = { First(B) – ∈ } ∪ Follow(S) ∪ First(b) ∪ Follow(A) = { g , $, b , h }.

LL(1) parsing table

Construction of LL(1) parsing talbes

For ever production rule A → ω, do the following :

1. For every a ∈ First (ω) and a ≠ λ, put A → ω into T [A, a]
2. If λ ∈ First (ω) , then for every a ∈ Follow (A), put A → ω into T [A, a]

1. S → A S b
2. S → C
3. A → a
4. C → c C
5. C → λ

First (ASb) = {a},
First (C) = {c, λ},
Follow (S) = {b, $},
First (a) = {a},
First (cC) = {c},
First (λ) = {λ},
Follow (C) = {b, $}

	a	b	c	$
S	1	2	2	2
A	3			
C		5	4	5

Example of LL(1) grammar

- Consider the grammar below:

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

- After eliminating Left recursion, we get:

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

- Calculation of FIRST and FOLLOW sets:

Production	FIRST	FOLLOW
E –> TE'	{ id, (}	{ $,) }
E' → +TE' | ε	{ +, ε }	{ $,) }
T → FT'	{ id, (}	{ +, $,) }
T' → *FT' | ε	{ *, ε }	{ +, $,) }
F → id | (E)	{ id, (}	{ *, +, $,) }

- Construction of Parsing table:

	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)		

Condition for being an LL(1) grammar:

For every $A \in N$ with $A \rightarrow \alpha$, $A \rightarrow \beta$ and $\alpha \neq \beta$

1. First $(\alpha) \cap$ First $(\beta) = \phi$.

2. If $\lambda \in$ First (β), then First $(\alpha) \cap$ Follow $(A) = \phi$.

Q: Can an LL(1) grammar be ambiguous? A: No.

Stack implementation of LL(1) parsing

- Input: An input string ‘w’ and a parsing table(‘M’) for grammar G.
- Output: If ‘w’ is in L(G), an LMD of ‘w’; otherwise an error indication.
- Algorithm:

Set input pointer to point to the first symbol of the string $;

repeat

let X be the symbol pointed by the stack pointer,
and **a** is the symbol pointed to by input pointer;

if X is a terminal or $

then if X=a then pop X from the stack and increment the input pointer;

else eror()

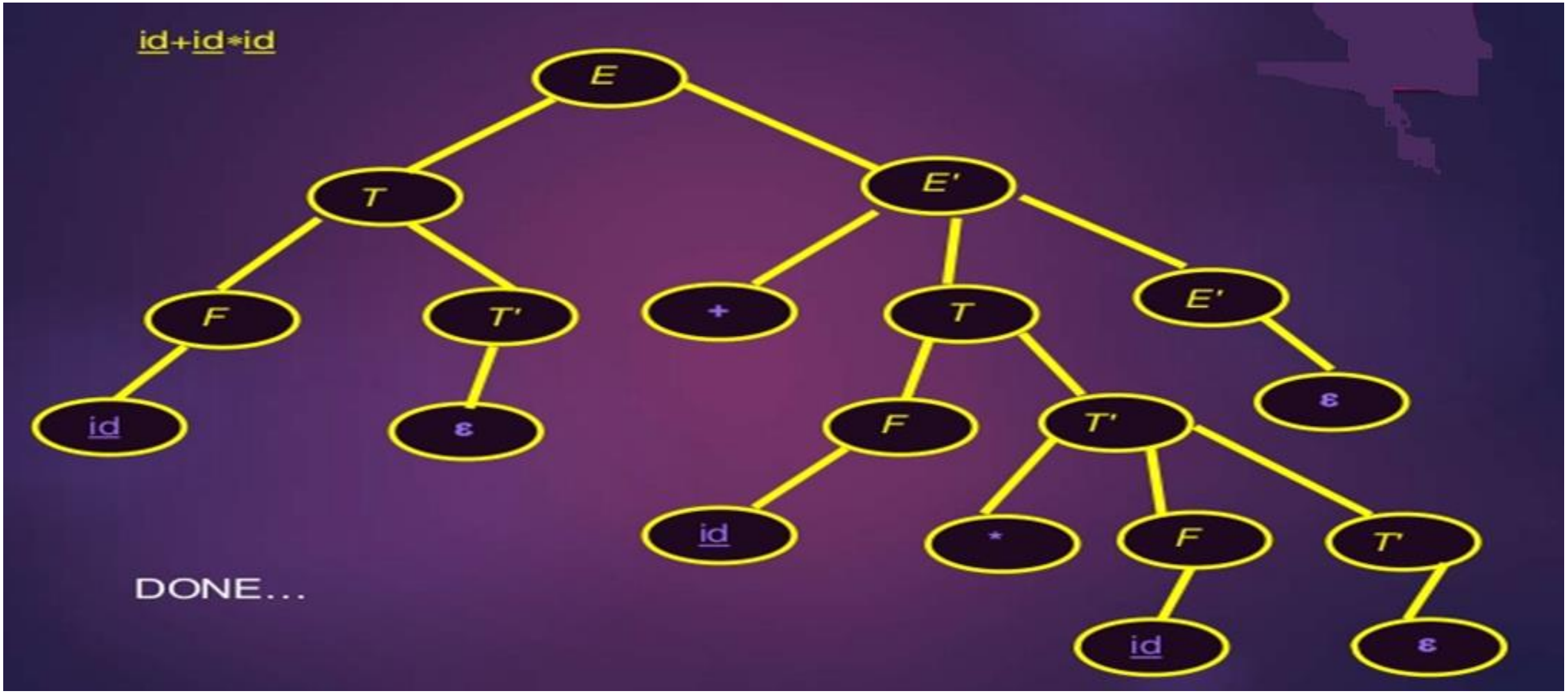
end if

```
else /*if X is a non terminal */
      if M[X,a]= X→y1 y2……yk  then
begin
   pop X from the stack;
   push y1 y2……yk onto the stack, with y1 on top;
    output the production X→y1 y2……yk
  end
        else error()
        end if
end if
until X=$ /* stack is empty */
```

Steps in processing the input string: id+id*id

MATCHED	STACK	INPUT	ACTION
	E$	id+id * id$	
	TE'$	id+id * id$	E->TE'
	FT'E'$	id+id * id$	T->FT'
	id T'E'$	id+id * id$	F->id
id	T'E'$	+id * id$	Match id
id	E'$	+id * id$	T'->Є
id	+TE'$	+id * id$	E'-> +TE'
id+	TE'$	id * id$	Match +
id+	FT'E'$	id * id$	T-> FT'
id+	idT'E'$	id * id$	F-> id
id+id	T'E'$	* id$	Match id
id+id	* FT'E'$	* id$	T'-> *FT'
id+id *	FT'E'$	id$	Match *
id+id *	idT'E'$	id$	F-> id
id+id * id	T'E'$	$	Match id
id+id * id	E'$	$	T'-> Є
id+id * id	$	$	E'-> Є

Parse tree for id+id*id



Left Factoring a grammar

- **<u>Grammar With Common Prefixes-</u>**

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

✓This kind of grammar creates a problematic situation for Top down parsers.

✓Top down parsers can not decide which production must be chosen to parse the string in hand.

✓To remove this confusion, we use left factoring.

(Source: https://www.gatevidyalay.com/tag/left-factoring-in-compiler-design-ppt/)

- In left factoring,

✓We make one production for each common prefixes.

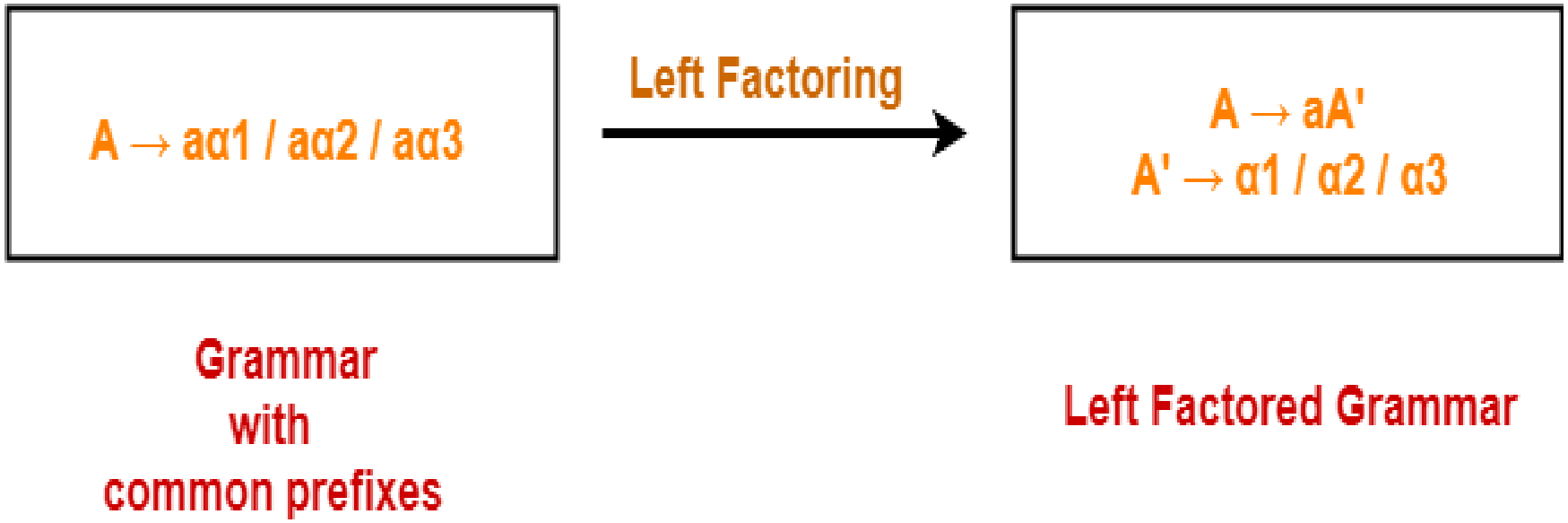
✓The common prefix may be a terminal or a non-terminal or a combination of both.

✓Rest of the derivation is added by new productions.

- The grammar obtained after the process of left factoring is called as **Left Factored Grammar**.

- (Source: https://www.gatevidyalay.com/tag/left-factoring-in-compiler-design-ppt/)

- **Example-**



(Source: https://www.gatevidyalay.com/tag/left-factoring-in-compiler-design-ppt/)

- Consider the grammar below:

- S→ iEtS | iEtSeS | a
- E → b

- Construct a parse table for the above grammar.

- Left factor the grammar:

Source: https://www.slideshare.net/ll1-parser-in-compilers

Source: https://www.slideshare.net/ll1-parser-in-compilers

TABLE:
FAST & FOLLOW

SYMBOL	FIRST	FOLLOW
S → iEtSS' | a	a , i	$, e
S' → eS |ε	e, ε	$, e
E → b	b	t

TABLE:
PARSING TABLE

Non Terminal	INPUT SYMBOLS					
	a	b	e	i	t	$
S	S→a					
S'						
E						

Source: https://www.slideshare.net/ll1-parser-in-compilers

TABLE:
FAST & FOLLOW

SYMBOL	FIRST	FOLLOW
S → iEtSS' | a	a , i	$, e
S' → eS |ε	e, ε	$, e
E → b	b	t

TABLE:
PARSING TABLE

Non Terminal	INPUT SYMBOLS					
	a	b	e	i	t	$
S	S→a			S→iEtSS		
S'						
E						

Source: https://www.slideshare.net/ll1-parser-in-compilers

TABLE:
FAST & FOLLOW

SYMBOL	FIRST	FOLLOW
S → iEtSS' | a	a , i	$, e
S' → eS |ε	e, ε	$, e
E → b	b	t

TABLE:
PARSING TABLE

Non Terminal	INPUT SYMBOLS					
	a	b	e	i	t	$
S	S→a			S→iEtSS		
S'			S' → eS			
E						

Source: https://www.slideshare.net/ll1-parser-in-compilers

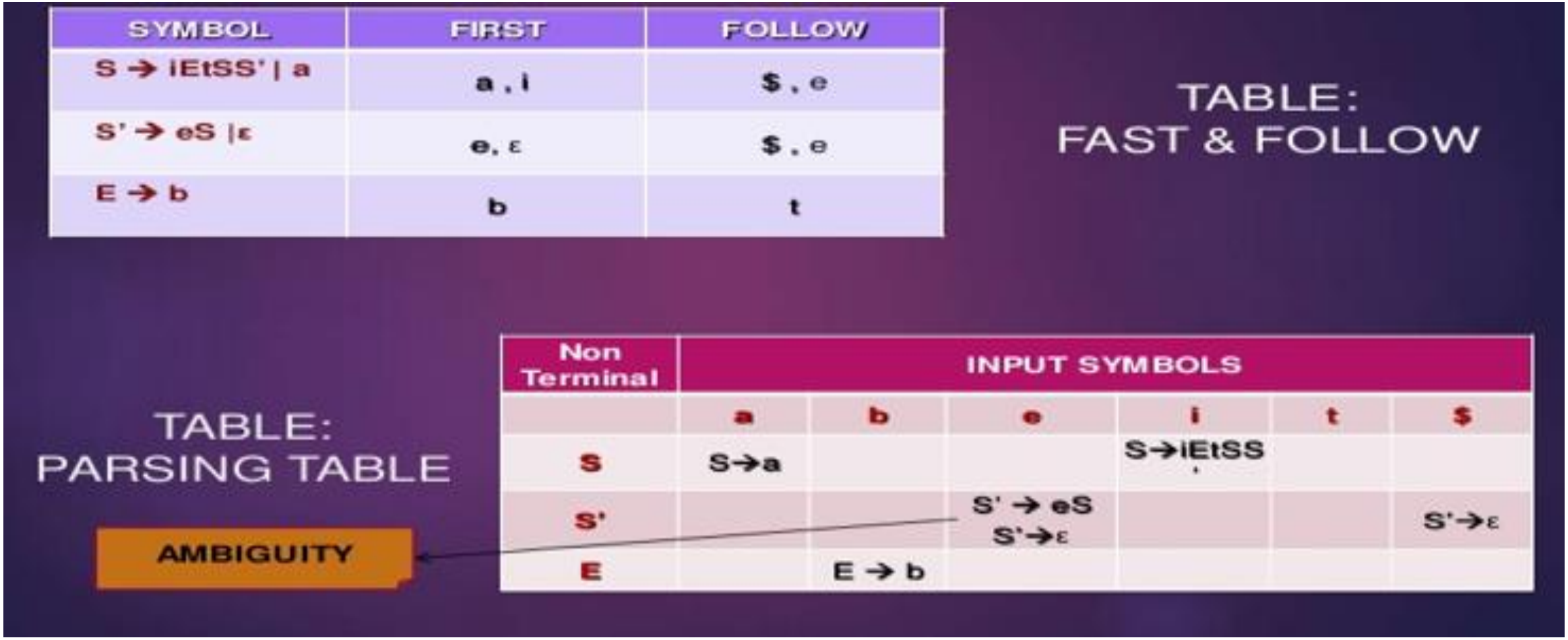
TABLE:
FAST & FOLLOW

SYMBOL	FIRST	FOLLOW
S → iEtSS' | a	a , i	$, e
S' → eS |ε	e, ε	$, e
E → b	b	t

TABLE:
PARSING TABLE

Non Terminal	INPUT SYMBOLS					
	a	b	e	i	t	$
S	S→a			S→iEtSS'		
S'			S' → eS S'→ε			S'→ε
E						

Source: https://www.slideshare.net/ll1-parser-in-compilers



Source: https://www.slideshare.net/ll1-parser-in-compilers

- The grammar is ambiguous and it is evident by the fact that we have two entries corresponding to M[S',e] containing $S' \rightarrow \varepsilon$ and $S' \rightarrow eS$.
- Note that the ambiguity will be solved if we use LL(2) parser, i.e. Always see for the two input symbols.
- LL(1) grammars have distinct properties.
  - No ambiguous grammar or left recursive grammar can be LL(1).
- Thus , the given grammar is not LL(1).

Source: https://www.slideshare.net/ll1-parser-in-compilers