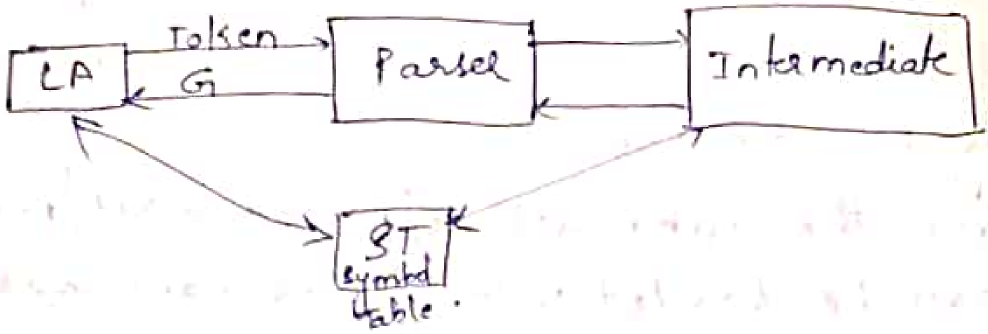
14-3-2024

UNIT - 2

Syntax Analysis.

Role of Parser :-

→ To check syntactic errors.



Any recursive structure of any programming lang can be ~~pr~~ represented by Context Free grammar

CFG = (VTPS)

- V → set of Non Terminals.
- T → set of Terminals.
- P → Production Rules
- S → starting symbol.

For $A \rightarrow \alpha$ ~~=~~ ↳ sentinal form?

Parse Tree :-

A tree which is having derivations is calle a 'parse tree'.

$\alpha A \beta$ can be written as $\alpha \gamma \beta$ if there is a production rule $A \rightarrow \gamma$ in our grammar. This is called as derivation.

Derivations ~~are~~ of 2 types :- (i) Left Most derivation
(ii) Right Most Derivation

$E \rightarrow E+E \mid E-E \mid E*E \mid E|E \mid -E$.

$E \rightarrow (E)$
$E \rightarrow id$

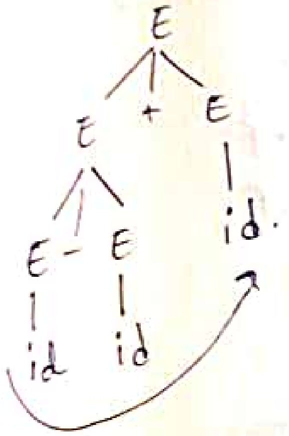
LMD:-

$E \rightarrow E+E$
$E \rightarrow E-E+E$
$\rightarrow id-E+E$
$\rightarrow id-id+E$
$\rightarrow id-id+id$.

RMD:-

$E \rightarrow E-E$
$\rightarrow E-E+E$
$\rightarrow E-E+id$.
$\rightarrow E-id+id$.
$\rightarrow id-id+id$.

Parse Tree → Internal nodes → Non Terminals
→ Leaf nodes → Terminals



(id-id+id)

Ambiguity:- If any grammar produces more than one parse tree, that grammar is ambiguious grammar & that parse tree is ambiguious parse tree

Eg:- LMD

$E \rightarrow E+E$
$\rightarrow id+E$
$\rightarrow id+E*E$
$\rightarrow id+id*id$.

(2 parse Trees)

LMD

$E \rightarrow E*E$
$\rightarrow E+E*E$
$\rightarrow id+id*id$

Un ambiguous :- E → E + T / T

T → T * F / F

F → F ^ G / G

G → (E) / id

15/3/2022

→ Topdown Parser cannot handle Left Recursion.

Eliminating Left Recursion :-

If any production is of the form A → Aα/β, then it is said to be in LR.

→ It is eliminated by the following rule :-

$\boxed{A \rightarrow A\alpha / \beta}$ is turned into $\boxed{A \rightarrow \beta A'}$ $\boxed{A' \rightarrow \alpha A' / \epsilon}$

Eg:- production rules :-

$$\left.\begin{matrix} E \rightarrow E + T / T \\ T \rightarrow T * F / F \end{matrix}\right\} \underline{LR}$$

F → (E) / id

A = E	A = T
α = +T	α = *F
β = T	β = F

E → E+T/T

E → TE'

E' → +TE'/ε

T → T*F/F.

T → FT'

T' → *FT'/ε

After eliminating LR, the production rules :-

E → TE'

E' → +TE'/ε

T → FT'

T' → *FT'/ε

F → (E) / id

If there are multiple production rules in LR,

Eg:- $A \to A\alpha_1 / A\alpha_2 / A\alpha_3 ---- A\alpha_n / \beta_1 / \beta_2 -- \beta_n$

then after elimination of LR,

$$A \to \beta_1 A' / \beta_2 A' ----$$

$$A' \to \alpha_1 A' / \alpha_2 A' ---- / \epsilon.$$

Eg:- $S \to Aa/b$

$A \to Ac/ Sd/\epsilon$

$A \to Ac/ Aad/ bd/ \epsilon$

$A \to bdA' / \epsilon A'$

$A' \to cA'/adA'/\epsilon$

Final production rules after eliminating LR:-

$S \to Aa/b.$

$A \to bdA'/A'$

$A' \to cA'/adA'/\epsilon.$

Elimination of Left Factoring:-

→ Because decision taking is difficult

If we have $A \to \underline{\alpha}\beta_1/\underline{\alpha}\beta_2$

starting with same symbols in the same production rule.

The idea is that when it is not clear which of 2 alternative productions are used to expand a non-terminal A. so, we may be able to rewrite A production to take a decision.

If $\boxed{A \to \alpha\beta_1 / \alpha\beta_2}$ are 2 productions & i/p begins with a non-empty string derived α, we do not know which A should either $\alpha\beta_1$ or $\alpha\beta_2$. Then, rewrite it as:

$$\boxed{A \longrightarrow \alpha A'}$$
$$\boxed{A' \to \beta_1 / \beta_2}$$

Eg:- $S \to iEtS / iEtSeS / a$
$E \to b$

$\alpha = iEtS$
$\beta_1 = \epsilon$
$\beta_2 = eS$

$$\left.\begin{array}{l} S \to iEtS\,S' / a \\ S' \to \epsilon / eS \\ E \to b \end{array}\right\}$$ After elimin Left Facto

Left Recursion Problems :-

① $A \to ABd / Aa / a$
$B \to Bc / b$

② $E \to E+E / E*E / a$

③ $S \to (L) / a$
$L \to L,S / S$

④ $S \to S_0 S_1 S / 01$

Left Factoring

① $A \to aAB / aBc / aAc$

② $S \to bSSaaS / bSS$
$/ bSb / a$

③ $S \to a / ab / abc / a$

Left Recursions :-

① $A \rightarrow ABd / Aa / a$

$A \rightarrow aA' / A'$

$A' \rightarrow BdA' / \epsilon / aA'$

② $E \rightarrow E+E / E*E / a$

$E \rightarrow aE' / E'$

$E' \rightarrow +EE' / *EE' / \epsilon$

③ $S \rightarrow (L) / a$

$L \rightarrow L,S / S$

$L \rightarrow SL'$

$L' \rightarrow ,SL' / \epsilon$

$S \rightarrow (L) / a$

④ $S \rightarrow S0S,S / 01$

$S \rightarrow 01S'$

$S' \rightarrow 0S,SS' / \epsilon$

Left Factoring :-

① $A \rightarrow a\underline{AB} / a\underline{BC} / a\underline{AC}$.

$A \rightarrow aA'$

$A' \rightarrow AB / BC / AC$.

② $S \rightarrow \underline{bS}Saa S / \underline{bS}SaSb / \underline{bS}b / \underline{a}$

$A \rightarrow bSA' / a$

$A' \rightarrow \underline{Sa}aS / \underline{Sa}Sb / b$

$A' \rightarrow SaA'' / b$.

$A'' \rightarrow aS / Sb$

Final Production rules:

$A \rightarrow bSA' / a$

$A' \rightarrow SaA'' / b$

$A'' \rightarrow aS / Sb$.

③ $S \rightarrow a / ab / abc / abcd$

$S \rightarrow aS'$

$S' \rightarrow \epsilon / b / bc / bcd$.

$S' \rightarrow \epsilon / bS''$

$S'' \rightarrow \epsilon / c / cd$

$S'' \rightarrow \epsilon / cS'''$

$S''' \rightarrow \epsilon / d$.

Final rules :-

$S \rightarrow aS'$

$S' \rightarrow \epsilon / bS''$

$S'' \rightarrow \epsilon / cS'''$

$S''' \rightarrow \epsilon / d$

17/3/2022

Top down Parsing

YACC is a parser generator for bottom-up Parsing
JAVACC is a parser used for top-down Parsi

Approaches:-

① Recursive Descent parsing technique:-

→ It is also called as LL(K) parsing technique

LL(K)

L → Left to Right parsing

L → Left Most Derivation

K → look ahead, Symbol.

→ Also called as LL(1) parser.

Now it is Non-Recursive Descent parsing technique

used to create a Predictive Parser

Be cause only 1 look ahead symbol is used.

→ To generate a parse table, we need to know FIRST and FOLLOW.

→ This parse table is used to build a parse tree

→ There are 3 rules to find out FIRST and 3 rules to find FOLLOW

FIRST :-

1. If X is a terminal, FIRST $(x) = \{x\}$

2. If X is a non-terminal & $x \rightarrow \epsilon$ is a production, FIRST$(x) = \{\epsilon\}$

3. If X is a non-terminal & $x \rightarrow y_1 y_2 \cdots y_k$

FIRST(x) = FIRST(y_1)

if $y_1 \rightarrow \epsilon$, FIRST(x) = FIRST(y_2)

$y_2 \rightarrow \epsilon$, FIRST(x) = FIRST(y_3)

$\vdots$

$y_k \rightarrow \epsilon$, FIRST$(x) = \{\epsilon\}$.

Q1) Find out the FIRST (S), FIRST (A), FIRST (B), FIRST(

FIRST (D), FIRST (E).

$S \rightarrow ABCDE$

$A \rightarrow a / \epsilon$

$B \rightarrow b / \epsilon$

$C \rightarrow c$

$D \rightarrow d / \epsilon$

$E \rightarrow e / \epsilon$

FIRST (S) = $\{a, b, c\}$

FIRST (A) = $\{a, \epsilon\}$

FIRST (B) = $\{b, \epsilon\}$

FIRST (C) = $\{c\}$

FIRST (D) = $\{d, \epsilon\}$

FIRST (E) = $\{e, \epsilon\}$

② $S \rightarrow ACB / CbB / Ba$

$A \rightarrow da / BC$

$B \rightarrow g / \epsilon$

$C \rightarrow h / \epsilon$

FIRST(S) = $\{d, g, h, \epsilon, b, a\}$

FIRST (A) = $\{d, g, h, \epsilon\}$

FIRST (B) = $\{g, \epsilon\}$

FIRST (C) = $\{h, \epsilon\}$

FOLLOW :-

1. FOLLOW (S) = $\{\$\}$, If S is start symbol

2. If $A \rightarrow \alpha B \beta$, then FOLLOW(B) = FIRST(β) except ϵ

3. If $A \rightarrow \alpha \beta$ or $A \rightarrow \alpha B \beta$, when FIRST ($\beta$) contains ϵ

FOLLOW(B) = FOLLOW(A)

Q1) S → ABCDE
A → a/ε
B → b/ε
C → c
D → d/ε
E → e/ε

Follow(S) = {$}
Follow(A) = {b, c}
Follow(B) = {c}
Follow(C) = {d, e, $}
Follow(D) = {e, $}
Follow(E) = {$}

FIRST(S) = {a, b, c}
FIRST(A) = {a, ε}
FIRST(B) = {b, ε}
FIRST(C) = {c}
FIRST(D) = {d, ε}
FIRST(E) = {e, ε}

Q2) S → ACB | cbB | Ba
A → da | BC
B → g | ε
C → h | ε

Follow(S) = {$}
Follow(A) = {h, g, $, b}
Follow(C) = {g, $, b}
Follow(B) = {a, b, $, g}

22/3/2022

* steps for constructing predictive Parser :-

1. Eliminate Left Recursion.
2. Eliminate Left Factoring.
3. Find FIRST & FOLLOW
4. Construct predictive Parse table / Parse table
5. Parse the i/p statement

Q) $E \rightarrow E + T / T$ उदा

$T \rightarrow T * F / F$

$F \rightarrow (E) / id$

① eliminate LR

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) / id$

② Eliminate LF

No Left Factoring

③

FIRST(E) = {id, (}

FIRST(E') = {+, ∈}

FIRST(T) = {id, (}

FIRST(T') = {*, ∈}

FIRST(F) = {(, id}

Follow(E) = {$,)}

Follow(E') = {$,)}

Follow(T) = {+, $,)}

Follow(T') : ~~{+, $,)~~ {+, $,)}

Follow(F) = {*, +, $,)}

? WTF

24|3|2022

Parse table using FIRST and FOLLOW :-

Rows - Non - Terminals
columns - Terminals

	+	*	(	)	id	$
E			$E\to TE'$		$E\to TE'$	
E'	$E'\to +TE'$			$E'\to \epsilon$		$E'\to \epsilon$
T			$T\to FT'$		$T\to FT'$	
T'	$T'\to \epsilon$	$T'\to *FT'$		$T'\to \epsilon$		$T'\to \epsilon$
F			$F\to (E)$		$F\to id$	

Rules to fill the parse table :-

~~1. If the production does not contain ε, then find First(~~

① $M[A, a] = A \to \alpha$

a is FIRST(α)

② $M[A, b] = A \to \alpha$.

If ε is in FIRST(α), then consider Follow

Parsing any input string :-

$w = id * id + id$

Here we use the concept of stack.

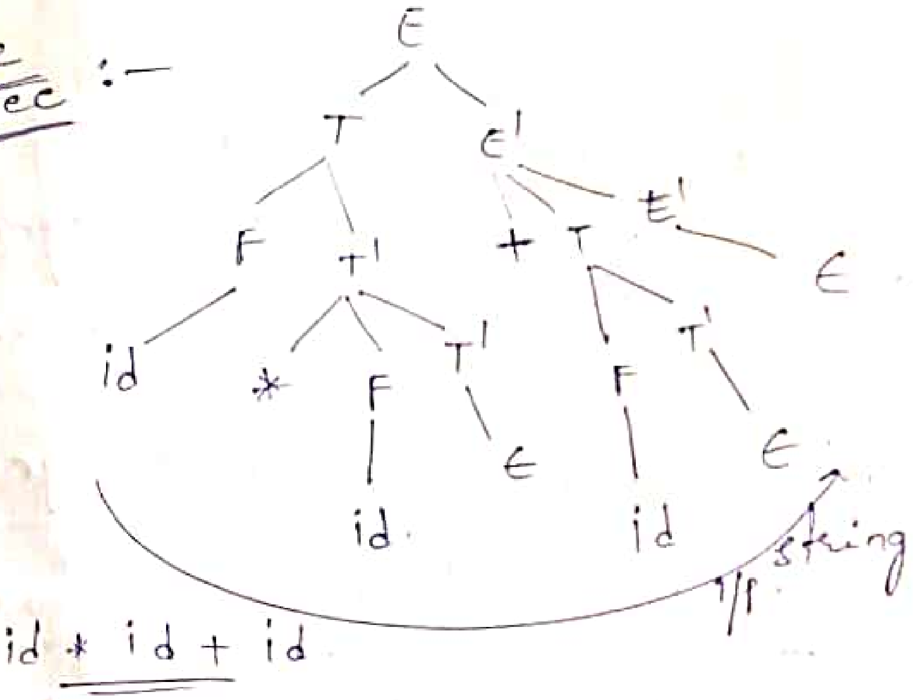
stack	i/p	o/p
$E	id * id + id$	
$E'T	id * id + id$	E → TE'
$E'T'F	id * id + id$	T → FT'
$E'T'id	id * id + id$	F → id.
$E'T'	* id + id$ ·	T' → *FT'
$E'T'F*	* id + id$ pop	
$E'T'F	id + id $	~~F → id~~
$E'T'id	id + id $ pop	F → id.
$E'T'	+ id $	
$E'	+ id $.	T' → ε
$E'T+	+ id $ pop	E' → +TE'
$E'T	id $	
$E'T'F	id $	T → FT'
$E'T'id	id $ pop	F → id.
$E'T'	$	φ
$E'	$	T' → ε ·

$E'	$	E' → ε
$	$	

pop

Passed

Parse Tree :-



31/3/2022

Error recovery in Predictive parsing :-

when errors occurs :-

1. when multiple entries in parsing table create an error

2. when top of the stack doesn't match i/p

3 when top of stack matches, but i/p is empty in parsing table

Error Recovery :- (Panic Mode)

Panic Mode Rules :-

Place all symbols in FOLLOW(A) into sync

1. If parser looks up an entry, M [A, a] and finds it blank, then the i/p symbol a is

skipped.

2. If the entry is sync bit, then non-termi
on top of stack is popped.

3. If the token not matches the top of the st
then pop the token in the stack.

	+	*	(	)	id	$
E			E→TE'	sync	E→TE'	sync
E'	E'→+TE'			E'→ε		E'→ε
T	sync		T→FT'	sync	T→FT'	sync
T'	T'→ε	T'→*FT'		T'→ε		T'→ε
F	sync	sync	F→(E)	sync	F→id	sync

I/P =) id * + id $

	stack	i/p	o/p
skip	$E	id*+id$	sync,
	$E'T	id*+id$	E→TE'
	$E'T'F	id*+id$	T→FT'
	$E'T'id	id*+id$	F→id
	$E'T'	*+id$	
	$E'T'F*	*+id$	T→*FT' [illegible]
	$E'T'F	+id$	———
	$E'T'	+id$	T'→ε

$ E'	+ id $	.
$ E'T+	+ id $	E' -> FTE'
$ E' T	id $	
$ E'T' F	id $	T -> FT'
$ E'T' id	id $	
$ ~~E'T'~~	$	
$	$	