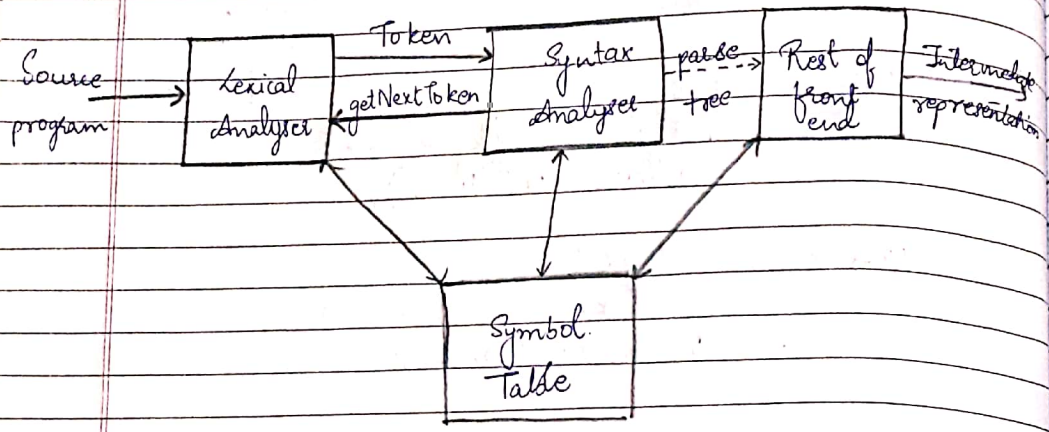
UNIT-II : SYNTAX ANALYSIS

* The role of Parser



* Representative Grammar

⇒ There are 3 types of parser / parsing techniques.
i) Universal parser
ii) Top-down parser
iii) Bottom-up parser

Grammar is unambiguous + non-recursive
- Grammar should be free from ambiguity

* Expression Grammar

$E \Rightarrow E+E \mid E*E \mid (E) \mid id$ Ambiguous

all are same

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

Unambiguous, Recursive

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

Free from recursion

* Syntax Error Handling

⇒ Types of errors :-
i) Lexical errors - spelling mistakes of keywords
ii) Syntax errors - missing semicolons
iii) Semantic Errors - For operators if there are different datatypes of operands
iv) Logical errors.
↳ instead of (a=b) we write (a-b) which is not a comparison in if statement.

* Error recovery Strategies

1. Panic mode recovery
2. Phrase level recovery.
3. Error productions.
4. Global correction

* CGF (Context Free Grammar)

1. Construct parse tree for the string id + id * id

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

Soln:

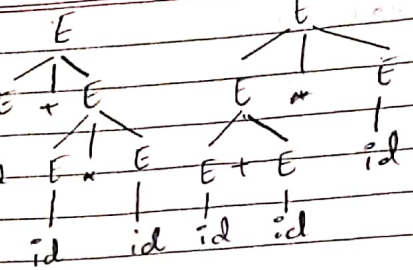
LMD:
$E \Rightarrow E+E$
$\Rightarrow id+E$
$\Rightarrow id+E*E$
$\Rightarrow id+id*E$
$\Rightarrow id+id*id$.

RMD:
$E \Rightarrow E+E$
$\rightarrow E+E*E$
$\Rightarrow E+E*id$.
$\Rightarrow E+id*id$.
$\Rightarrow id+id*id$.

LMD:
$E \Rightarrow E*E$
$\Rightarrow E+E*E$
$\rightarrow id+E*E$
$\rightarrow id+id*E$
$\rightarrow id+id*id$

RMD:
$E \Rightarrow E*E$
$\Rightarrow E*id$
$\Rightarrow E+E*id$
$\rightarrow E+id*id$
$\Rightarrow id+id*id$

Parse trees for E*E



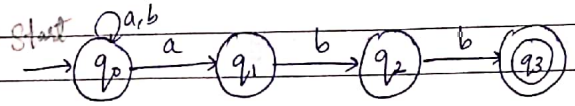
* Verifying the language Generated by a Grammar
* Context-free Grammar vs. Regular Expressions.

-> All regular languages are subset of context-free languages but vice-versa is not possible.

i) Draw the NFA for the language that ends with abb.

$L = \{ abb, aabb, babb \ldots \}$.

$R.E = (a+b)^*abb$.



-> Procedure for converting NFA to Grammar

Step1:- For each state 'i' of NFA, create a non-terminal A_i.

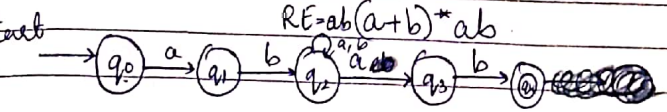
Step2:- If state 'i' has a transition to state 'j' on i/p 'a', add the production $A_i \rightarrow aA_j$.
If state 'i' goes to state 'j' on input ε, add the production $A_i \rightarrow A_j$

Step3:- If 'i' is an accepting (final) state, add $A_i \rightarrow \varepsilon$

Step4:- If 'i' is the start state, make A_i as the start symbol of the grammar

$A_0 \rightarrow aA_0 \mid bA_0 \mid aA_1$
$A_1 \rightarrow bA_2$
$A_2 \rightarrow bA_3$
$A_3 \rightarrow \varepsilon$

ii) Draw NFA for the language that begins and ends with ab.



Grammar:-
$A_0 \rightarrow aA_1$
$A_1 \rightarrow bA_2$
$A_2 \rightarrow aA_3 \mid aA_2 \mid bA_2$
$A_3 \rightarrow bA_4$
$A_4 \rightarrow \varepsilon$

* Writing a Grammar

I] Lexical vs. Syntactic Analysis.

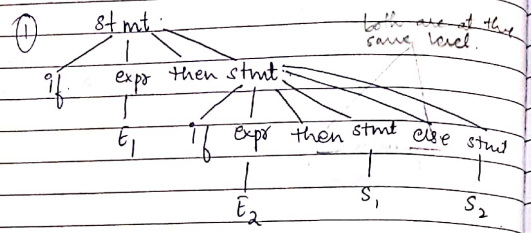
II] Eliminating Ambiguity.

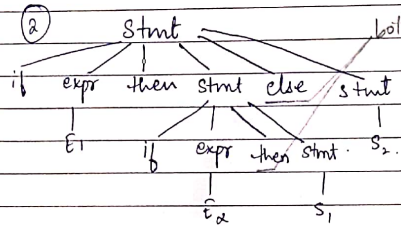
=> Grammar for Conditional Statement / "dangling else".

i)
stmt → if expr then stmt
| if expr then stmt else stmt
| other.
expr → $E_1 \mid E_2$.
stmt → $S_1 \mid S_2$.

Construct parse tree for "if E_1 then if E_2 then S_1 else S_2".

Stmt $\xrightarrow{LMD}$ if expr then stmt.
⇒ if E_1 then stmt.
⇒ if E_1 then if expr then stmt else stmt
⇒ if E_1 then if E_2 then stmt else stmt
⇒ if E_1 then if E_2 then S_1 else S_2





⇒ Match each else with the closest unmatched then, therefore the 1st parse tree is correct.

⇒ Unambiguous form of the Grammar.

Stmt → matched_stmt
| open_stmt

matched stmt → if expr then matched_stmt else matched stmt
| others

open_stmt → if expr then stmt
| if expr then matched_stmt else open_stmt

⇒ The idea is the statement appearing between then and else keyword must be matched i.e. the interior stmt must not end with an unmatched or open then. A matched stmt is either an if-then-else statement containing no open stmts or it is any other kind of unconditional stmt.

* Removing ambiguity from an arithmetic expression

Ambiguous E → E+E | E*E | (E) | id | E-E | E/E

Unambiguous Grammar: E → E+T | T | E-T
T → T*F | F | T/F
F → (E) | id

Take care of precedence & associativity

* Left factoring (Top-down Approach)

Eg:- i) S → ab | ac.

Rewritten as:- S → aS'
S' → b | c.

ii) S → abc | abd | abe | b

Rewritten as:- S → abS' | b
S' → c | d | e

Defn:- If A → $\alpha\beta_1 | \alpha\beta_2$ are two A-productions, and

The input begins with a non-empty string derived from α, we do not know whether to expand A to $\alpha\beta_1$ or $\alpha\beta_2$.

However we may expand differ the decision by expanding $A \to \alpha\hat{A}'$, the original productions become, $A \to \alpha A'$

$A' \to \beta_1 | \beta_2$

⇒ Algorithm - Left factoring a Grammar.

INPUT : Grammar G.

OUTPUT : An Equivalent Left Factored Grammar

METHOD : For each non-terminal A find the longest prefix α common to two or more of its alternatives. If $\alpha \neq \varepsilon$ (epsilon), replace all of A productions, $A \to \alpha\beta_1 | \alpha\beta_2 | \ldots | \alpha\beta_n | \gamma$ where γ represents all alternatives that do not begin with α, by $A \to \alpha A' | \gamma$.

$A' \to \beta_1 | \beta_2 | \ldots | \beta_n$

Here A' is

Repeatedly Apply this transformation until no two alternatives for a non-Terminal have a common prefix.

⇒ Examples :- Left Factor the following Grammars.

i) $A \to aAB | aBc | aAc$

Soln :- $A \to aA''$

$A' \to AB | Bc | Ac$.

$A' \to AA'' | Bc$

$A'' \to B | c$

Final: $A \to aA'$

$A' \to AA'' | Bc$

$A'' \to B | c$

ii) $S \to bSSaaS | bSSaSb | bSb | a$.

$S \to bSS' | a$

$S' \to SaaS | SaSb | b$

$S' \to SaS'' | b$

$S'' \to aS | Sb$

Final Grammar:-

$S \to bSS'$

$S' \to SaS'' | b$

$S'' \to aS | Sb$

iii) $S \to a | ab | abc | abcd$.

$S \to aS'$

$S' \to \varepsilon | b | bc | bcd$

$S' \to bS'' | \varepsilon$

$S'' \to c | cd | \varepsilon$

$S' \to \varepsilon | cS'''$

$S''' \to \varepsilon | d$

Final Grammar:

$S \to aS'$

$S' \to bS'' | \varepsilon$

$S'' \to cS''' | \varepsilon$

$S''' \to d | \varepsilon$

iv) $S \to aAd | aB$

$A \to a | ab$

$B \to ccd | ddc$

$S \to aS'$

$S' \to Ad | B$

$A \to aA'$

$A' \to b | \varepsilon$

$B \to ccd | ddc$

$A \xrightarrow{+} A\alpha$:- $A\alpha$ is derived from A in one or more derivation steps.

Elimination of Left Recursion

→ A Grammar is left recursive if it has a non-Terminal A such that there is a derivation.

$$A \xrightarrow{+} A\alpha$$

for some string α.

→ Immediate left recursion where there is a production of the form $A \to A\alpha$.

→ Rule to eliminate left recursion

I] The productions $A \to A\alpha \mid \beta$ could be replaced by the non-left recursive productions

$$\boxed{\begin{array}{l} A \to \beta A' \\ A' \to \alpha A' \mid \varepsilon \end{array}}$$ for immediate left recursion.

⇒ Examples : Eliminate left recursion for the following grammar:-

i) $E \to E+T \mid T$
$T \to T*F \mid F$
$F \to (E) \mid id$

Sol^n:

$E \to TE'$ $\quad$ $T \to FT'$
$E' \to +TE' \mid \varepsilon$ $\quad$ $T' \to *FT' \mid \varepsilon$

Final Grammar on eliminating left recursion:

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

gatevidyalay.com

Grammar with cycle : $S \to A \mid b$
$A \to S \mid a$

⇒ ALGORITM: Eliminating left recursion

INPUT: A Grammar with no cycles or epsilon productions

OUTPUT: An equivalent Grammar with no left recursion.

METHOD: Apply the following algorithm. Note that the resulting non-left recursive Grammar may have epsilon productions.

1. Arrange the non-terminals in some order $A_1, A_2 \ldots\ldots A_n$
2. for (each i from 1 to n) {
3. for (each j from 1 to i-1) {
4. Replace each production of the form $A_i \to A_j\gamma$ by the productions $A_i \to \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$ where $A_j \to \delta_1 \mid \delta_2 \mid \ldots\ldots \mid \delta_k$ are all current A_j productions.
5. }.
6. Eliminate the immediate left recursion among the A_i productions.
7. }

⇒ Examples - i) Eliminate left recursion from the following Grammar.

$A \to ABd \mid Aa \mid a$
$B \to Be \mid b$.

Sol^n:

$A \to aA'$ $\quad$ $B \to bB'$
$A' \to BdA' \mid aA' \mid \varepsilon$ $\quad$ $B' \to eB' \mid \varepsilon$

ii) $S \to (L) \mid a$
$L \to L, S \mid S$.

Terminals :- ( ) , a (4-total)
Nonterminals :- L S (2-Total)

$S \to (L) \mid a$
$L \to SL'$
$L' \to , SL' \mid \varepsilon$

iii) $S \to A$
$A \to Ad \mid Ae \mid aB \mid ac$
$B \to bBc \mid f$

$A \to acA' \mid aBA'$
$A' \to dA' \mid eA' \mid \varepsilon$.

final Grammar : $S \to A$
$A \to acA' \mid aBA'$
$A' \to dA' \mid eA' \mid \varepsilon$
$B \to bBc \mid f$

iv) $A \to AAa \mid \beta$

Soln: $A \to \beta A'$
$A' \to AaA' \mid \varepsilon$

v) $S \to Aa \mid b$
$A \to Ac \mid Sd \mid \varepsilon$

$S \to Aa \mid b$.

Soln: $A \to Ac \mid Aad \mid bd \mid \varepsilon$
β

$A \to bdA'$
$A' \to cA' \mid adA' \mid \varepsilon$

Example I) Eliminate left recursion from the following grammar:-

i) ① $A \to Ba \mid Aa \mid c$ } Indirect left recursion
② $B \to Bb \mid Ab \mid d$

Soln: $A \to BaA' \mid cA'$ } eliminating immediate left recursion
$A' \to aA' \mid \varepsilon$

$B \to Bb \mid BaA'b \mid cA'b \mid d$ (Replace A with its body).

$B \to cA'bB' \mid dB'$
$B' \to bB' \mid aA'bB' \mid \varepsilon$

Final Grammar :- $A \to BaA' \mid cA'$
$A' \to aA' \mid \varepsilon$.
$B \to cA'bB' \mid dB'$
$B' \to bB' \mid aA'bB' \mid \varepsilon$

ii) ① $X \to XSb \mid Sa \mid b$
② $S \to Sb \mid Xa \mid a$

$X \Rightarrow Sa$
$\Rightarrow Xaa$ } Indirect left recursion

Soln: Step-I
$X \to SaX' \mid bX'$
$X' \to SbX' \mid \varepsilon$
$S \to Sb \mid Xa \mid a$

Step-I (body | X)
$S \to Sb \mid SaX'a \mid bX'a \mid a$
$S \to bX'aS' \mid aS'$
$S' \to bS' \mid aX'aS' \mid \varepsilon$.

iii) $S \to Aa \mid b$
$A \to Ac \mid Sd \mid \varepsilon$

Soln: $S \to Aa \mid b$.
$A \to Ac \mid Aad \mid bd \mid \varepsilon$ ← Replace by body S as it is indirect left recursion.
$A \to bdA' \mid A'$
$A' \to cA' \mid adA' \mid \varepsilon$

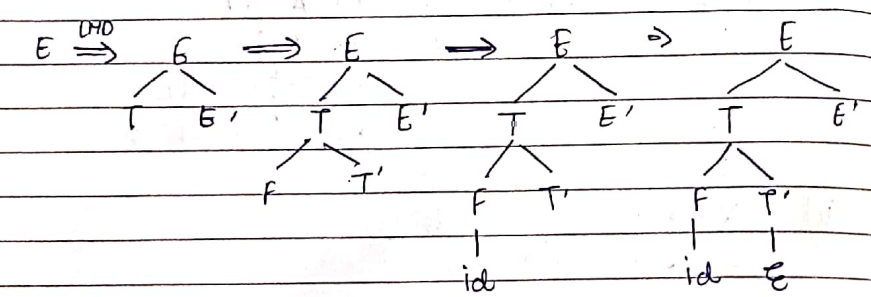
Top-Down Parser – Parse tree starts from the root and ends at leaves.

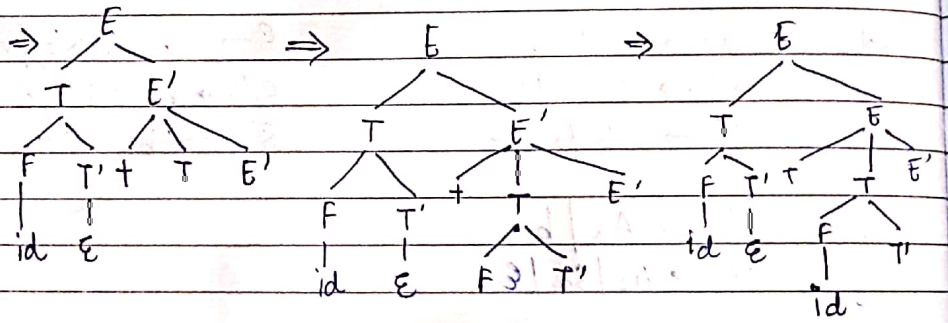
⇒ Requirement :- The Grammar should be:
- Unambiguous
- Left factored
- Eliminate left recursion

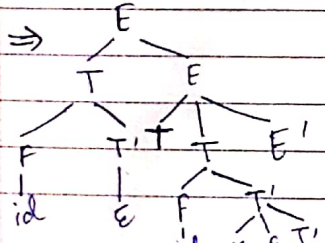
- Construct parse tree for the following grammar:-

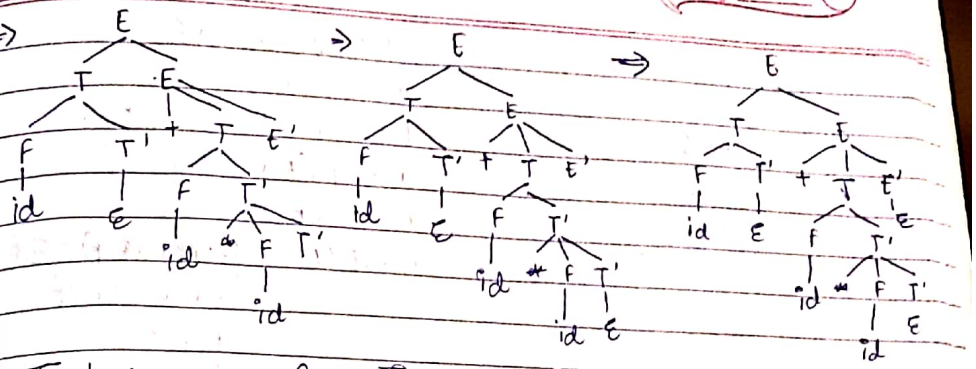
$E \to TE'$
$E' \to +TE' \mid \varepsilon$
$T \to FT'$
$T' \to *FT' \mid \varepsilon$
$F \to (E) \mid id$, for $\omega = id + id * id$

Soln:









Techniques under Top-Down Parsing.

1. Recursive-Descent Parsing
2. First & Follow

→ First

α be any ~~string~~ Grammar symbol.

FIRST(α) = set of terminals which are derived from α & those should be first symbol.

For eg:- i) $S \to abc \mid da \mid pq$. ii) $S \to \varepsilon$
FIRST(S) = {a, d, p} FIRST(S) = {ε}

Examples:

i) Compute FIRST from the following Grammar:

$A \to abc \mid def \mid ghi$

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

ii) $S \to aBDh$
$B \to cC$
$C \to bC \mid \varepsilon$
$D \to EF$
$E \to g \mid \varepsilon$
$F \to f \mid \varepsilon$

Soln:
FIRST(S) = {a}
FIRST(B) = {c}
FIRST(C) = {b, ε}
FIRST(E) = {g, ε}
FIRST(F) = {f, ε}
FIRST(D) = {FIRST(E) − ε} ∪ {FIRST(F)}
= {g, f, ε}

iii) $S \to A$
$A \to aB \mid Ad$
$B \to b$
$C \to g$

(eliminate left recursion)

$S \to A$
$A \to$

iv) $S \to (L) \mid a$
$L \to SL'$
$L' \to ,SL' \mid \varepsilon$

Soln:
FIRST(S) = {(, a}
FIRST(L) = FIRST(S)
= {(, a}
FIRST(L') = {,, ε}

v) $S \to AaAb \mid BbBa$
$A \to \varepsilon$
$B \to \varepsilon$

FIRST(A) = {ε}
FIRST(B) = {ε}
FIRST(S) = {FIRST(A) − ε} ∪ FIRST(a) ∪ {FIRST(B) − ε} ∪ FIRST(b)
= {a, b}

* If FOLLOW(E) is a non-terminal, then we take FIRST(F) & if we get an ε in FIRST(F) then we take union of {FIRST(F) − ε} ∪ FOLLOW(LHS non-terminal)

→ **Follow(A):-** Set of terminals that appears exactly after A.

Eg:- i) $S \to aAb$
Soln FOLLOW(A) = {b}

ii) $S \to Aa \mid bBC$
$A \to a \mid b$
Soln: FOLLOW(A) = {a}
FOLLOW(B) = {C}

→ To compute FOLLOW(A) for all non-terminals A, apply the following rules, until nothing can be added to any FOLLOW set.

1. Place $ in FOLLOW(S), where S is the start symbol, and $ is the input right endmarker

2. If there is a production $A \to \alpha B\beta$, then everything in FIRST(B) except ε is in FOLLOW(B).

3. If there is a production $A \to \alpha B$, or a production $A \to \alpha B\beta$, where FIRST(β) contains ε, then everything in FOLLOW(A) is in FOLLOW(B).

Examples:-

→ When there's no symbol then the follow is same as LHS i.e D

i) $S \to aBDh$
$B \to cC$
$C \to bC \mid \varepsilon$
$D \to EF$
$E \to g \mid \varepsilon$
$F \to f \mid \varepsilon$

Soln: FIRST
FIRST(S) = {a}
FIRST(B) = {c}
FIRST(C) = {b, ε}
FIRST(E) = {g, ε}
FIRST(F) = {f, ε}
FIRST(D) = {g, f, ε}

FOLLOW (by default)
FOLLOW(S) = {$}
FOLLOW(D) = FIRST(h) = {h}
FOLLOW(F) = FOLLOW(D) = {h}
*FOLLOW(E) = {FIRST(F) − ε} ∪ FOLLOW(D) = {f, h}
FOLLOW(B) = {FIRST(D) − ε} ∪ FIRST(h) = {g, f, h}

* Follow of any terminal, we dont get epsilon.

FOLLOW(C) = FOLLOW(B) = {g, f, h} we considered only $E \to c($ and not $C \to bC$ bcoz in the second its the same terminal in L.H.S & R.H.S

ii) Compute first & follow

$S \to A$
$A \to aB \mid Ad$
$B \to b$
$c \to g$.

Soln: Eliminate left recursion.

$S \to A$
$A \to aBA'$
$A' \to dA' \mid \varepsilon$
$B \to b$
$C \to g$

FIRST(A) = {a}
FIRST(A') = {d, ε}
FIRST(B) = {b}
FIRST(C) = {g}
FIRST(S) = {a}

We check for terminals FIRST(a) = {a}, FIRST(b) = {b}, FIRST(d) = {d}, FIRST(g) = {g}

FOLLOW(S) = $, FOLLOW(A) = FOLLOW(S) = {$}.
FOLLOW(A') = FOLLOW(A') = {$}.
FOLLOW(B) = {FIRST(A') − ε} ∪ FOLLOW(A)
= {d} ∪ {$} = {d, $}
FOLLOW(C) = NA

iii) $S \to (L) \mid a$
$L \to SL'$
$L' \to , SL' \mid \varepsilon$

Soln:
FIRST(S) = {(, a}
FIRST(L') = {,, ε}
FIRST(L) = FIRST(S) = {(, a}

FOLLOW(L) = FIRST( ) = {) }
FOLLOW(L') = FOLLOW(L) = {) }
FOLLOW(S) = {$} ∪ {FIRST(L') − ε} ∪ FOL

iv) $S \to AaAb \mid BbBa$
$A \to \varepsilon$
$B \to \varepsilon$

Soln: FIRST(A) = {ε}
FIRST(B) = {ε}
FIRST(S) = {a, b}

v) $E \to E+T \mid T$
$T \to T * F \mid F$
$F \to (E) \mid id$

Soln:- Eliminating left recursion.
$E \to TE'$
$E' \to +TE' \mid \varepsilon$
$T \to FT'$
$T' \to *FT' \mid \varepsilon$
$F \to (E) \mid id$

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

FOLLOW(E) = {$} ∪ {)} (by default)
= {$,)}

FOLLOW(T) = {FIRST(E') − ε} ∪ FOLLOW(E) ∪ FOLLOW(E')
= {+, $,)}

FOLLOW(E') = FOLLOW(E)
= {$,)}

FOLLOW(T') = FOLLOW(T)
= {+, $,)}

FOLLOW(F) = {FIRST(T') − ε} ∪ FOLLOW(T) ∪ FOLLOW(T')
= {*, +, $,)}

vi) $S \to ACB \mid CbB \mid Ba$
$A \to da \mid BC$
$B \to g \mid \varepsilon$
$C \to h$

Soln: FIRST(S) = {{FIRST(A)} ∪ {FIRST(C) − ε} ∪ {FIRST(B)}} ∪ {{FIRST(C) − ε} ∪ FIRST(b)} ∪ {FIRST(B) − ε} ∪ FIRST(a)}
= {{d, g, h} ∪ {h} ∪ {g}} ∪ {h, b} ∪ {g, a} = {g, h, d, b, a}

FIRST (B) = {g, ε}
FIRST (C) = {h, ε}
FIRST (A) = {d} ∪ {FIRST(B) − ε} ∪ FIRST (C)
= {d, g, h, ε}

FOLLOW(S) = {$}
FOLLOW (A) = {FIRST(C) − ε} ∪ {FIRST(B) − ε} ∪ FOLLOW(S)
= {h, g, $}

FOLLOW (C) = {{FIRST(B) − ε} ∪ FOLLOW(S)} ∪ FIRST(b)
∪ {FOLLOW(A)}
= {g, $} ∪ {b} ∪ {h, g, $}.
= {h, g, b, $}

FOLLOW (B) = {FOLLOW(S) ∪ FIRST(a)} ∪ {{FIRST (C) − ε} ∪ FOLLOW(A)}
= {$, a} ∪ {h, g, $}
= {h, g, a, $}.

Top Down Parsing

↓

Recursive Descent parsing

To select correct production

- Without Backtracking → Predictive parsing (One of recursive descent parsing)
  - ↳ Construction of Predictive parsing table
  - Table → 2D-array: M[A, a] — A: non-terminal; a: terminal or $.
- With Backtracking (less efficient, more time)

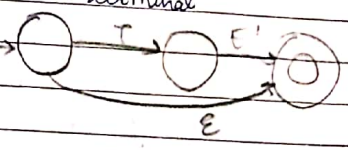
* Predictive Parsing

Requirements: i) Unambiguous ii) Left recursion removed iii) Left factoring done

⇒ Accepts LL(1) Grammar class
- 1 → Symbol of lookahead
- L → left-most derivation
- L → Left to right

⇒ Transition diagram for each non-terminal

for eg: E → TE' | ε



⊙ LL(1) Grammar: A Grammar G is LL(1) if and only if whenever A → α | β are two distinct productions of G, the following conditions hold:

1. For no terminal a do both α & β derive beginning with a
2. At most one of α & β can derive the empty string.
3. If $\beta \overset{*}{\Rightarrow} \varepsilon$, then α does not derive any string beggining with a terminal in FOLLOW(A). Likewise if $\alpha \overset{*}{\Rightarrow} \varepsilon$, then β does not derive any string beginning with a terminal in FOLLOW(A).

* Algorithm: Construction of a predictive passing table

INPUT: Grammar G

OUTPUT: Passing table M

METHOD: For each production A → α of the grammar, do the following:

1. For each terminal a in FIRST(α), add $A \to \alpha$ to M[A,a]
2. If ε is in FIRST(α), then for each terminal b in FOLLOW(A), add $A \to \alpha$ to M[A,b]. If ε is in FIRST(α) and $ is in FOLLOW(A), add $A \to \alpha$ to M[A,$] as well.

If, after performing the above, there is no production at all in M[A,a], then set M[A,a] to error (which we normally represent by an empty entry in the table).

Example: 1) $E \to TE'$
$E' \to +TE' \mid \varepsilon$
$T \to FT'$
$T' \to *FT' \mid \varepsilon$
$F \to (E) \mid id$

Construct predictive parsing table.

Soln:

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

① $E \to TE'$ $\quad \{A \to \alpha \mid \text{First}(\alpha)\}$
FIRST(TE') = {(, id}

② $E' \to +TE'$ $\quad E' \to \varepsilon$
FIRST(+TE') = {+} $\quad$ FOLLOW(E') = {), $}

③ $T \to FT'$
FIRST(FT') = {(, id}

④ $T' \to *FT'$ $\quad T' \to \varepsilon$
FIRST(*FT') = {*} $\quad$ FOLLOW(T') = {+,), $}

⑤ $F \to (E)$ $\quad F \to id$
FIRST((E)) = {(} $\quad$ FIRST(id) = {id}

2) Construct predictive parsing table for the following grammar.

$S \to aBDh$
$B \to cC$
$C \to bC \mid \varepsilon$
$D \to EF$
$E \to g \mid \varepsilon$
$F \to f \mid \varepsilon$

Soln:

Non-terminal	Input Symbol						
	a	b	c	h	g	f	$
S	$S \to aBDh$						
B			$B \to cC$				
C		$C \to \varepsilon$		$C \to bC$	$C \to \varepsilon$	$C \to \varepsilon$	
D		$D \to EF$			$D \to EF$	$D \to EF$	
E		$E \to \varepsilon$			$E \to g$	$E \to \varepsilon$	
F		$F \to \varepsilon$				$F \to f$	

① $S \to aBDh$
FIRST(aBDh) = {a}

② $B \to cC$
FIRST(cC) = {c}

③ $C \to bC$ $\quad C \to \varepsilon$
FIRST(bC) = {b} $\quad$ FOLLOW(C) = {g, f, h}

④ $D \to EF$

FIRST(EF) = {g, f, ε}

FOLLOW(D) = {h} { ∵ FIRST(α) produces ε }

⑤ $E \to g$

FIRST(g) = {g}

$E \to \varepsilon$

FOLLOW(E) = {f, h}

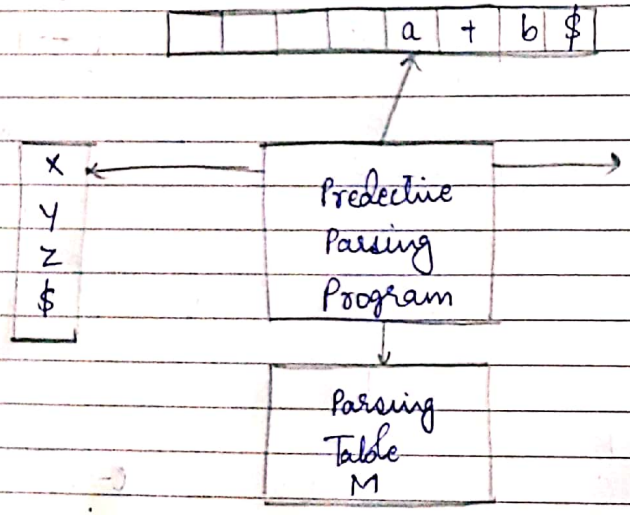
⑥ $F \to f$

FIRST(f) = {f}

$F \to \varepsilon$

FOLLOW(F) = {h}

* Model of a table-driven predictive parser



⇒ Predictive parsing Algorithm: (see SS)

METHOD : Initially

Example: Construct the LMD for id + id * id for the following grammar:-

$E \to TE'$

$E' \to +TE' \mid \varepsilon$

$T \to FT'$

$T' \to *FT' \mid \varepsilon$

$F \to (E) \mid id$

Soln.

$E \overset{lmd}{\Rightarrow} TE'$ — $E \to TE'$

$\Rightarrow FT'E'$ — $T \to FT'$

$\Rightarrow id\,T'E'$ — $F \to id$

$\Rightarrow id\,\varepsilon\,E'$ — $T' \to \varepsilon$

$\Rightarrow id + TE'$ — $E' \to +TE'$

$\Rightarrow id + FT'E'$ — $T \to FT'$

$\Rightarrow id + id\,T'E'$ — $F \to id$

$\Rightarrow id + id * FT'E'$ — $T' \to *FT'$

$\Rightarrow id + id * id\,T'E'$ — $F \to id$

$\Rightarrow id + id * id\,\varepsilon\,E'$ — $T' \to \varepsilon$

$\Rightarrow id + id * id$ — $E' \to \varepsilon$

According to predictive parsing algo: Moves made by predictive parser on input id + id * id

Matched	Stack	Input	Action ← According to predictive parser table
	E$	id + id * id$	Look at E row & id column
	TE'$	id + id * id$	output $E \to TE'$
	FT'E'$	id + id * id$	output $T \to FT'$
	id T'E'$	id + id * id$	output $F \to id$
id	T'E'$	+ id * id$	match id
id	E'$	+ id * id$	output $T' \to \varepsilon$
id	+ TE'$	+ id * id$	Output $E \to +TE'$
id +	TE'$	id * id$	match +
id +	FT'E'$	id * id$	output $T \to FT'$
id +	id T'E'$	id * id$	output $F \to id$

id+id	T'E'$	* id $	match id
id+id	*FT'E'$	* id$	output T' → *FT'
id+id*	FT'E'$	id$	match *
id+id*	idT'E'$	id$	output F → id
id+id*id	T'E'$	$	match id
id+id*id	E'$	$	output T' → ε
id+id*id	$	$	output E' → ε

Example2: S → A Input String = abdd$
A → aBA'
A' → dA' | ε
B → b
C → g

Soln:- Step1: Construct the table

Input → / Non-Terminals	a	d	b	g	$
S	S → A				
A	A → aBA'				
A'		A' → dA'			A' → ε
B					
C					

(i) S → A
FIRST(A) = {a}

ii) A → aBA'
FIRST(aBA') = {a}

iii) A' → dA' , A' → ε
FIRST(dA') = {d} FOLLOW(A') = {$}

iv) B → b
FIRST(b) = {b}

v) C → g
FIRST(g) = {g}

Step2: LMD

	output
S → A	S → A
⇒ aBA'	A → aBA'
⇒ abA'	B → b
⇒ abdA'	A' → dA'
⇒ abddA'	A' → dA'
⇒ abdd	A' → ε

Step 3: Moves made by a predictive parser on input abdd$.

Matched	Stack	Input	Action