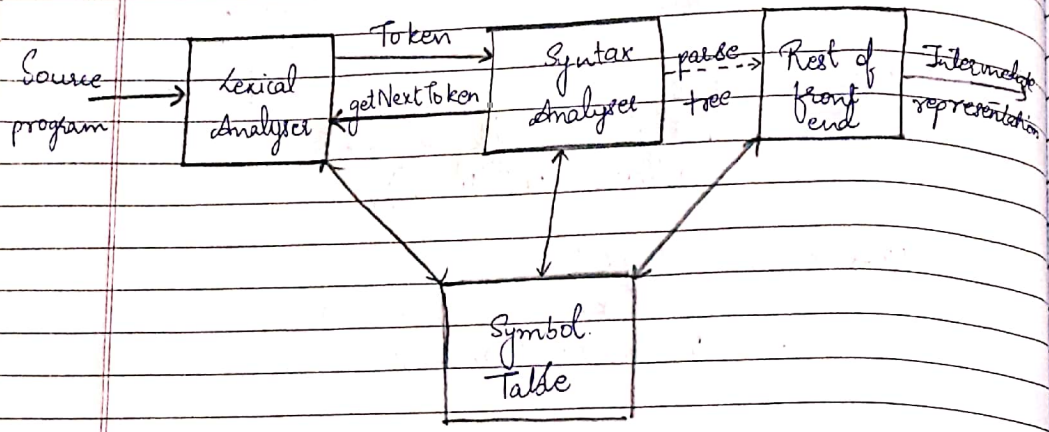
UNIT-II : SYNTAX ANALYSIS

* 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:

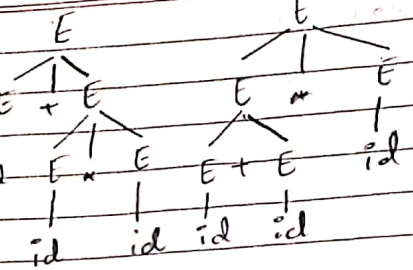
LMD:
$E \Rightarrow E+E$
$\Rightarrow id+E$
$\Rightarrow id+E*E$
$\Rightarrow id+id*E$
$\Rightarrow id+id*id$.

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



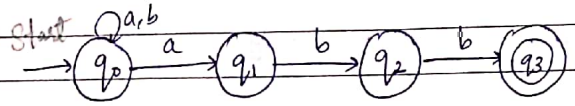
* Verifying the language Generated by a Grammar
* Context-free Grammar vs. Regular Expressions.

-> 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.

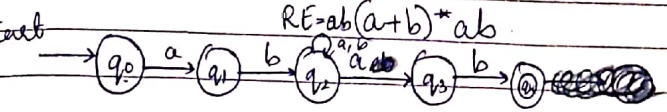
Step2:- If state 'i' has a transition to state 'j' on i/p 'a', add the production $A_i \rightarrow aA_j$.
If state 'i' goes to state 'j' on input ε, add the production $A_i \rightarrow A_j$

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.

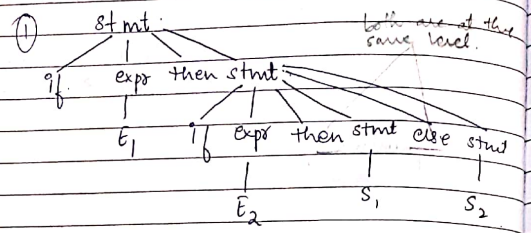
II] Eliminating Ambiguity.

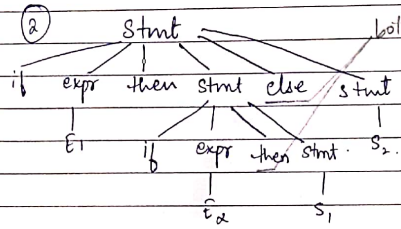
=> Grammar for Conditional Statement / "dangling else".

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

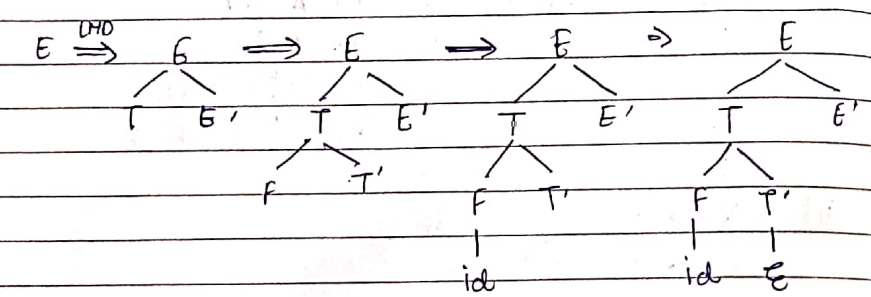
Top-Down Parser – Parse tree starts from the root and ends at leaves.

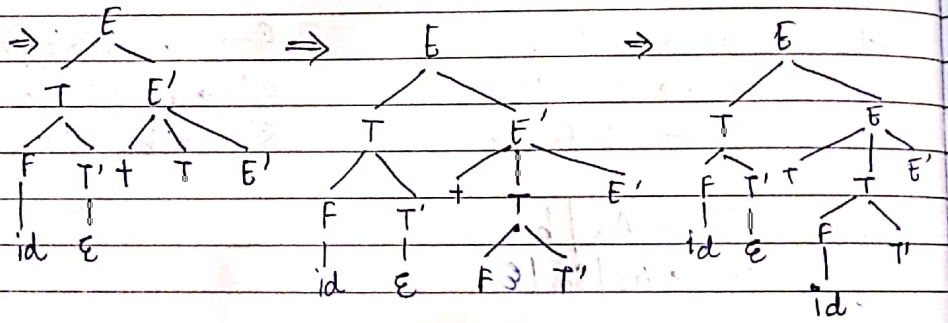
⇒ Requirement :- The Grammar should be:
- Unambiguous
- Left factored
- Eliminate left recursion

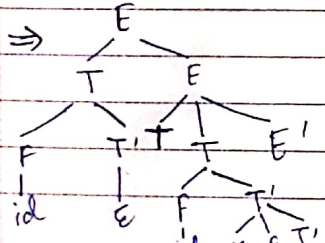
- Construct parse tree for the following grammar:-

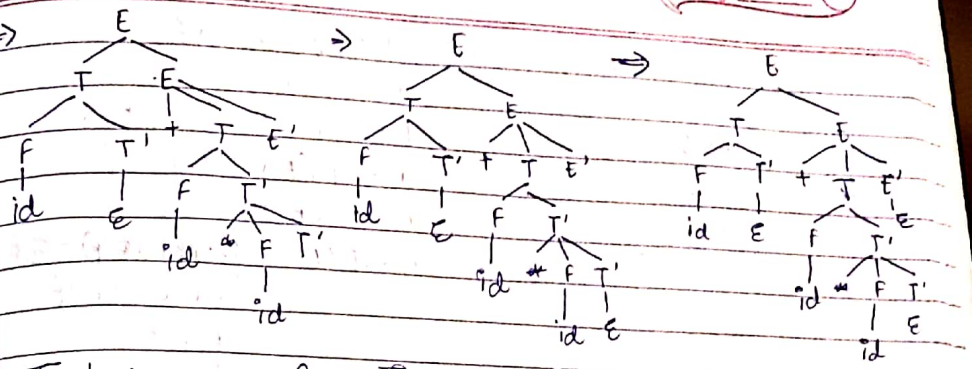
$E \to TE'$
$E' \to +TE' \mid \varepsilon$
$T \to FT'$
$T' \to *FT' \mid \varepsilon$
$F \to (E) \mid id$, for $\omega = id + id * id$

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)

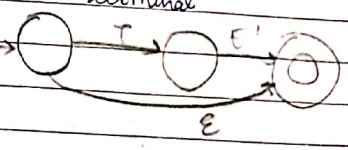
* Predictive Parsing

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}

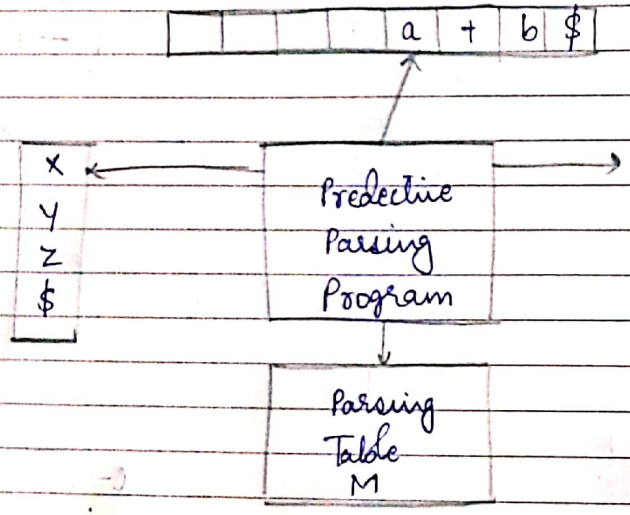
⑥ $F \to f$

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

UNIT 2 : SYNTAX ANALYSIS-1

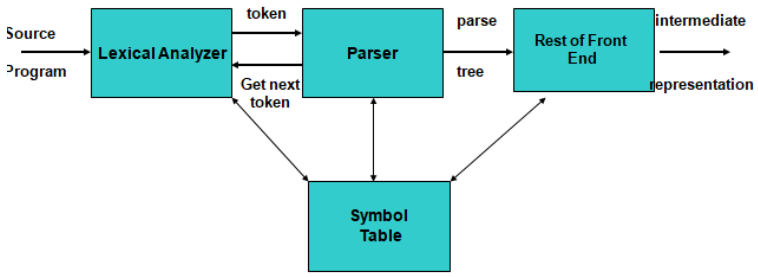
❖ Introduction :

- Syntax Analyser determines the structure of the program.
- The tokens generated from Lexical Analyser are grouped together and checked for valid sequence defined by programming language.
- Syntax Analyser uses context free grammar to define and validate rules for language construct.
- Output of Syntax Analyser is parse tree or syntax tree which is hierarchical / tree structure of the input.
- There is a need of mechanism to describe the structure of syntactic units or syntactic constructs of programming language. So we use Context free grammars.

❖ Role of a parser:

- The stream of tokens is input to the syntax analyzer.
- The job of the parser is:
  - To identify the valid statement represented by the stream of tokens as per the syntax of the language. If it is a valid statement, it will be represented by a parse tree.
  - If it is not a valid statement, then a suitable error message is displayed, so that the programmer is able to correct the syntax error.

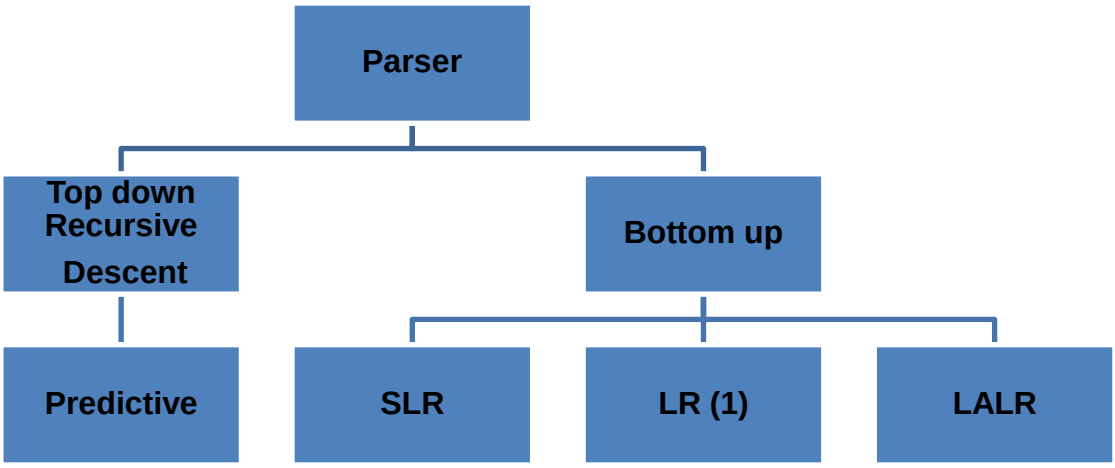
Position of Parser and role in Compiler model



- Usually the semantic analysis and intermediate code generation can interspersed with parsing. Hence, in addition to the validation of the programming statements parser also performs the following tasks :
  - Type-checking and providing the semantic consistency to the source programs.

- Execution of semantic actions that are attached with grammar and responsible for generating the required intermediate form for the source program that facilitates some kind of code optimization

❖ Classification of Parser:



❖ We categorize the parsers into two groups:

1. **Top-Down Parser**

   The parse tree is created top to bottom, starting from the root.

2. **Bottom-Up Parser**

   The parse is created bottom to top, starting from the leaves

- Both top-down and bottom-up parsers scan the input from left to right (one symbol at a time).
- Efficient top-down and bottom-up parsers can be implemented only for sub-classes of context-free grammars.
  - LL for top-down parsing
  - LR for bottom-up parsing

❖ Representative Grammars :

The following grammar treats + and * alike, so it is useful for illustrating techniques for handling ambiguities during parsing:

$$E \rightarrow E + E \mid E * E \mid (E) \mid \mathbf{id}$$

Associativity and precedence are captured in the following grammar. E represents expressions consisting of terms separated by + signs, T represents terms consisting of factors separated by * signs, and F represents factors that can be either parenthesized expressions or identifiers:

E -> E+T | T

T -> T*F | F

F -> (E) | id

This expression grammar belongs to the class of LR grammars that are suitable for bottom-up parsing. This grammar can be adapted to handle additional operators and additional levels of precedence. However, it cannot be used for top-down parsing because it is left recursive. The following non-left-recursive variant of the expression grammar will be used for top-down parsing:

E -> TE'

E' -> +TE' | ϵ

T -> FT'

T'-> *FT' | ϵ

F -> (E) | id

❖ Syntax Error Handling:

Common programming errors can occur at many different levels.

Lexical errors include misspellings of identifiers, keywords, or operators - e.g., the use of an identifier elipsesize instead of ellipsesize - and missing quotes around text intended as a string.

Syntactic errors include misplaced semicolons or extra or missing braces; that is, '((" or ")." As another example, in C or Java, the appearance of a case statement without an enclosing switch is a syntactic error (however, this situation is usually allowed by the parser and caught later in the processing, as the compiler attempts to generate code).

Semantic errors include type mismatches between operators and operands. An example is a return statement in a Java method with result type void.

Logical errors can be anything from incorrect reasoning on the part of the programmer to the use in a C program of the assignment operator = instead of the comparison operator ==. The program containing = may be well formed; however, it may not reflect the programmer's intent.

The error handler in a parser has goals that are simple to state but challenging to realize:

- Report the presence of errors clearly and accurately.
- Recover from each error quickly enough to detect subsequent errors.
- Add minimal overhead to the processing of correct programs.

❖ Error-Recovery Strategies:

1. Panic-Mode Recovery :

With this method, on discovering an error, the parser discards input symbols one at a time until one of a designated set of synchronizing tokens is found. The synchronizing tokens are usually delimiters, such as semicolon or }, whose role in the source program is clear and unambiguous. The compiler designer must select the synchronizing tokens appropriate for the source language.

- **Advantage:**

simple, and is guaranteed not to go into an infinite loop.

- **Disadvantage:**

panic-mode correction often skips a considerable amount of input without checking it for additional errors.

2. Phrase-Level Recovery :

On discovering an error, a parser may perform local correction on the remaining input; that is, it may replace a prefix of the remaining input by some string that allows the parser to continue. A typical local correction is to replace a comma by a semicolon, delete an extraneous semicolon, or insert a missing semicolon. The choice of the local correction is left to the compiler designer.

- **Advantage:**

Phrase-level replacement has been used in several error-repairing compilers, as it can correct any input string.

- **Disadvantage:**

Its major drawback is the difficulty it has in coping with situations in which the actual error has occurred before the point of detection.

3. Error Productions :

By anticipating common errors that might be encountered, we can augment the grammar for the language at hand with productions that generate the erroneous constructs. A parser constructed from a grammar augmented by these error productions detects the anticipated errors when an error production is used during parsing. The parser can then generate appropriate error diagnostics about the erroneous construct that has been recognized in the input.

4. Global Correction :

Given an incorrect input string x and grammar G, these algorithms will find a parse tree for a related string y, such that the number of insertions, deletions, and changes of tokens required to transform x into y is as small as possible. Unfortunately, these methods are in general too costly to implement in terms of time and space, so these techniques are currently only of theoretical interest. Do note that a closest correct program may not be what the programmer had in mind. Nevertheless, the notion of least-cost correction provides a yardstick for evaluating error-recovery techniques, and has been used for finding optimal replacement strings for phrase-level recovery.

❖ Context-Free Grammars:

- Inherently recursive structures of a programming language are defined by a context-free grammar.
- In a context-free grammar-G={ V, T, S, P },

  we have:

  V : A finite set of non-terminals (syntactic-variables)

  T : A finite set of terminals (in our case, this will be the set of tokens or lexical units)

  S : A start symbol (one of the non-terminal symbol)

  P : A finite set of productions rules in the following form

  $A \rightarrow \alpha$ where A is a non-terminal and α is a string of terminals and non-terminals (including the empty string)

- Example:

  $E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid - E$

  $E \rightarrow (E)$

  $E \rightarrow id$

❖ Derivations:

- $E \Rightarrow E+E$ i.e., E+E derives from E, which means that we can replace E by E+E
- To able to do this, we have to have a production rule $E \rightarrow E+E$ in our grammar.
- $E \Rightarrow E+E \Rightarrow id+E \Rightarrow id+id$
- A sequence of replacements of non-terminal symbols is called a **derivation** .
- In general a derivation step is $\alpha A \beta \Rightarrow \alpha\gamma\beta$ if there is a production rule $A \rightarrow \gamma$ in our grammar where α and β are arbitrary strings of terminal and non-terminal symbols

  $\alpha_1 \Rightarrow \alpha_2 \Rightarrow ... \Rightarrow \alpha_n$ (α_n derives from α_1 or α_1 derives α_n)

  $\Rightarrow$: derives in one step

  *

  $\Rightarrow$: derives in zero or more steps

  +

  $\Rightarrow$: derives in one or more steps

- Example:
  $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(id+E) \Rightarrow -(id+id)$
  OR
  $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(E+id) \Rightarrow -(id+id)$
- At each derivation step, we can choose any of the non-terminals in the sentential form of G for the replacement.
- If we always choose the left-most non-terminal in each derivation step, this derivation is called as **left-most derivation**.

  Ex. : $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(id+E) \Rightarrow -(id+id)$
- If we always choose the right-most non-terminal in each derivation step, this derivation is called as **right-most derivation**.

  Ex. :$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(E+id) \Rightarrow -(id+id)$
- We will see that the top-down parsers try to find the left-most derivation of the given source program.

❖ We will see that the bottom-up parsers try to find the right-most derivation of the given source program in the reverse order

❖ Parse Tree:

- Inner nodes of a parse tree are non-terminal symbols.
- The leaves of a parse tree are terminal symbols.
- A parse tree can be seen as a graphical representation of a derivation.

Ex. : Consider the grammar

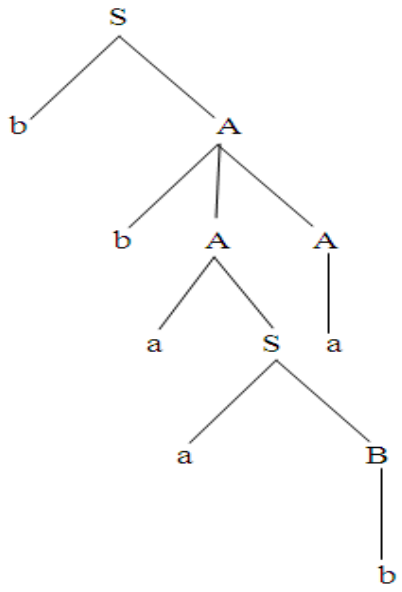
S → b A | a B

A → b A A | a S | a

B → a B B | b S | b

Writing leftmost and rightmost derivation for the sentence **bbaaba** along with Parse tree.

i) bbaaba



ii) bbbaaaba (Home Work)

Leftmost Derivation

S ⇒ b A

⇒ b b A A

⇒ b b a S A

⇒ b b a a B A

⇒ b b a a b A

⇒ b b a a b a

Rightmost derivation

S ⇒ b A

⇒ b b A A

⇒ b b A a

⇒ b b a S a

⇒ b b a a B a

⇒ b b a a b a

Fig. 3.5 Parse Tree for the string bbaaba

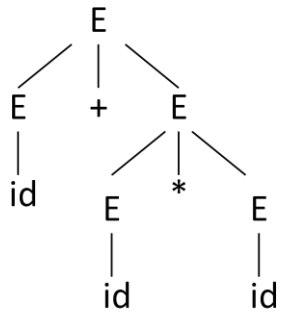
❖ Ambiguity:

A grammar produces more than one parse tree for a sentence is called as an ***ambiguous*** grammar.

Example:

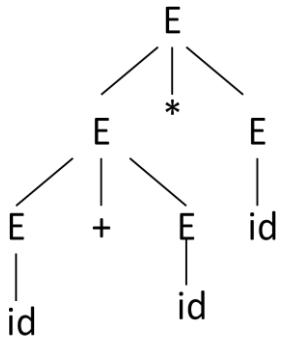
1. $E \Rightarrow E+E \Rightarrow id+E \Rightarrow id+E*E$

$\Rightarrow id+id*E \Rightarrow id+id*id$



$E \Rightarrow E*E \Rightarrow E+E*E \Rightarrow id+E*E$

$\Rightarrow id+id*E \Rightarrow id+id*id$



2. A → B C | a a C

   B → a | B a

   C → b

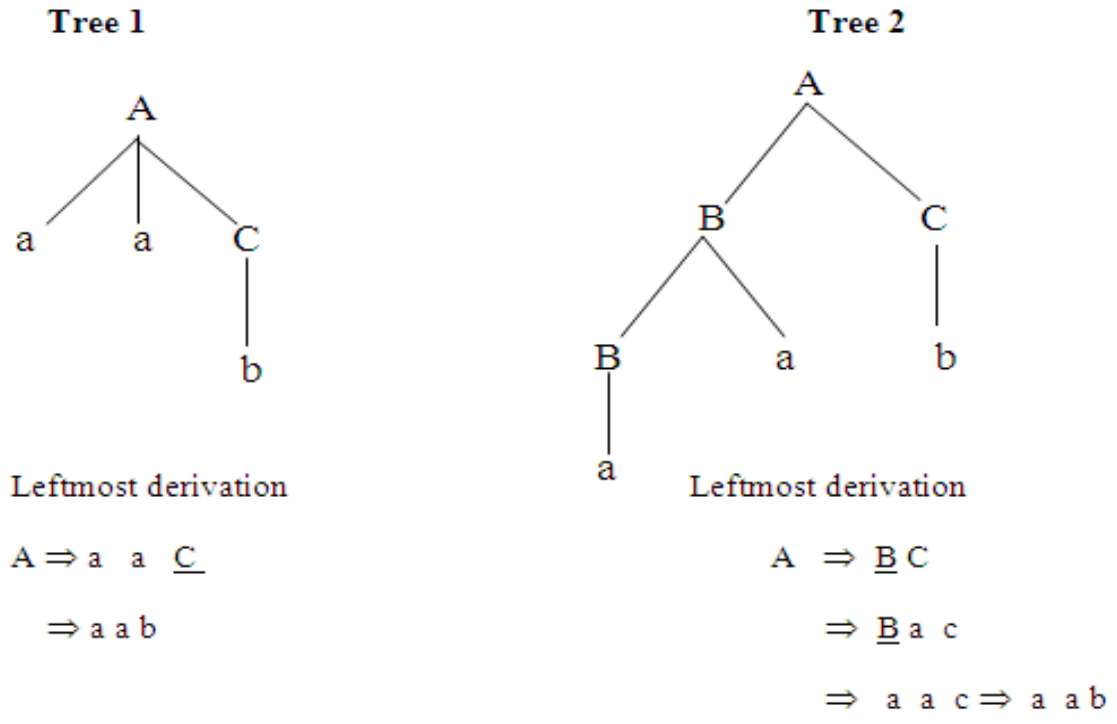


Fig. 3.20 Two leftmost derivation for string a a b

- For the most parsers, the grammar must be unambiguous.
- unambiguous grammar implies unique selection of the parse tree for a sentence.
- We have to prefer one of the parse trees of a sentence (generated by an ambiguous grammar) to disambiguate that grammar to restrict to this choice.

❖ Ambiguity – Operator Precedence:

Let us consider the grammar

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

At each step we begin by introducing One Non terminal - NT for each precedence level.

Priority levels (High to low)

Exponentiation **^** - F is NT (right associative rule)

Multiplicative operator **(*, /)** - T is NT (right associative rule)

Additive operator **(+ ,-)** - E is NT (right associative rule)

A subexp 'E' that is indivisible is either an identifier or parenthesized expression which is written as

G → id |(E) where G is New NT

- To write next rule we take New NT for the next highest priority level and this is connected with o Zero or more instance of next highest priority operator with previous level NT

  F → G^F | G

- Ambiguous grammars (because of ambiguous operators) can be disambiguated according to the precedence and associativity rules.

  Ex.:

  1. disambiguate the grammar E → E+E | E-E | E*E | E/E | E^E | id | (E)

     precedence: ^ (right to left)

     * (left to right)

     + (left to right)

  Ans.: E → E+T | E-T | T

  T → T*F | TF | F

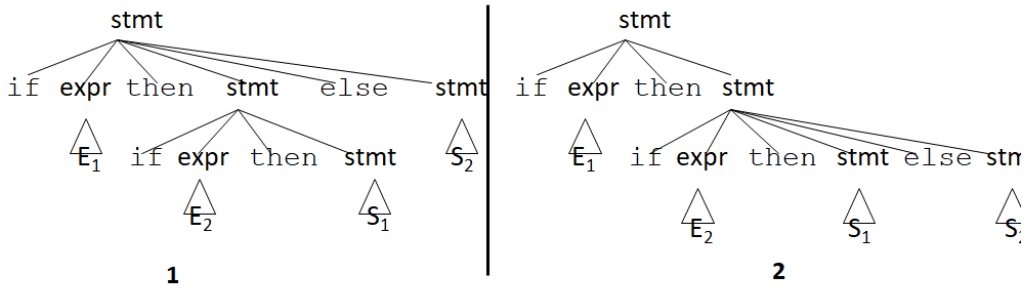
  F → G^F | G

  G → id | (E)

  2. Disambiguating the problem of danglilng else

stmt → **if** expr **then** stmt |
if expr **then** stmt **else** stmt | otherstmts

if E_1 then if E_2 then S_1 else S_2



We prefer the second parse tree (else matches with closest then). So, we have to disambiguate our grammar to reflect this choice. The unambiguous grammar will be:

stmt $\rightarrow$ matchedstmt

| unmatchedstmt

matchedstmt $\rightarrow$ **if** expr **then** matchedstmt **else** matchedstmt

| otherstmts

unmatchedstmt $\rightarrow$ **if** expr **then** stmt

| **if** expr **then** matchedstmt **else** unmatchedstmt

❖ Left Recursion:

- A grammar is ***left recursive*** if it has a non-terminal A such that there is a derivation, $A \Rightarrow A\alpha$ for some string α
- Top-down parsing techniques **cannot** handle left-recursive grammars.So, we have to convert our left-recursive grammar into an equivalent grammar which is not left-recursive.
- The left-recursion may appear in a single step of the derivation (*immediate left-recursion*), or may appear in more than one step of the derivation.

Immediate Left-Recursion:

$A \rightarrow A\alpha \mid \beta$ where β does not start with A

$\Downarrow$ eliminate immediate left recursion

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \varepsilon$ an equivalent grammar

In general,

$A \rightarrow A\alpha_1 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \ldots \mid \beta_n$ where $\beta_1 \ldots \beta_n$ do not start with A

$\Downarrow$ eliminate immediate left recursion

$A \rightarrow \beta_1 A' \mid \ldots \mid \beta_n A'$

$A' \rightarrow \alpha_1 A' \mid \ldots \mid \alpha_m A' \mid \varepsilon$ an equivalent grammar

Example:

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow id \mid (E)$

$\Downarrow$ eliminate immediate left recursion

$E \rightarrow T\,E'$

$E' \rightarrow +T\,E' \mid \varepsilon$

$T \rightarrow F\,T'$

$T' \rightarrow *F\,T' \mid \varepsilon$

$F \rightarrow id \mid (E)$

Left-Recursion – Problem:

A grammar cannot be immediately left-recursive, but it still can be left-recursive. By just eliminating the immediate left-recursion, we may not get a grammar which is not left-recursive.

$S \rightarrow Aa \mid b$

$A \rightarrow Sc \mid d$ This grammar is not immediately left-recursive,

but it is still left-recursive.

$\underline{S} \Rightarrow Aa \Rightarrow \underline{S}ca$ or

$\underline{A} \Rightarrow Sc \Rightarrow \underline{A}ac$ causes to a left-recursion

So, we have to eliminate all left-recursions from our grammar

- **Eliminate Left-Recursion – Algorithm:**

- Arrange non-terminals in some order: $A_1 \ldots A_n$

- for i **from** 1 **to** n **do** {

- **for** j **from** 1 **to** i-1 **do** {

replace each production

$A_i \rightarrow A_j \gamma$

by

$A_i \rightarrow \alpha_1 \gamma \mid \ldots \mid \alpha_k \gamma$

where $A_j \rightarrow \alpha_1 \mid \ldots \mid \alpha_k$

}

- eliminate immediate left-recursions among A_i productions

}

Example:1

$S \rightarrow Aa \mid b$

$A \rightarrow Ac \mid Sd \mid f$

- Order of non-terminals: S, A

for S:

- we do not enter the inner loop.

- there is no immediate left recursion in S.

for A:

- Replace $A \rightarrow Sd$ with $A \rightarrow Aad \mid bd$

So, we will have $A \rightarrow Ac \mid Aad \mid bd \mid f$

- Eliminate the immediate left-recursion in A

$A \rightarrow bdA' \mid fA'$

$A' \rightarrow cA' \mid adA' \mid \varepsilon$

So, the resulting equivalent grammar which is not left-recursive is:

$S \rightarrow Aa \mid b$

$A \rightarrow bdA' \mid fA'$

$A' \rightarrow cA' \mid adA' \mid \varepsilon$

Example:2

$S \rightarrow Aa \mid b$

$A \rightarrow Ac \mid Sd \mid f$

- Order of non-terminals: A, S

for A:

- we do not enter the inner loop.

- Eliminate the immediate left-recursion in A

$A \rightarrow SdA' \mid fA'$

$A' \rightarrow cA' \mid \varepsilon$

for S:

- Replace $S \rightarrow Aa$ with $S \rightarrow SdA'a \mid fA'a$

So, we will have $S \rightarrow SdA'a \mid fA'a \mid b$

- Eliminate the immediate left-recursion in S

$S \rightarrow fA'aS' \mid bS'$

$S' \rightarrow dA'aS' \mid \varepsilon$

So, the resulting equivalent grammar which is not left-recursive is:

$S \rightarrow fA'aS' \mid bS'$

$S' \rightarrow dA'aS' \mid \varepsilon$

$A \rightarrow SdA' \mid fA'$

$A' \rightarrow cA' \mid \varepsilon$

❖ Left-Factoring:

A predictive parser (a top-down parser without backtracking) insists that the grammar must be *left-factored*.

grammar ➔ a new equivalent grammar suitable for predictive parsing

stmt $\rightarrow$ if expr then stmt else stmt

|if expr then stmt

when we see if, we cannot immediately decide which production rule to choose to expand *stmt* in the derivation.In general,

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

Here α is non-empty and the first symbols of β_1 and β_2 (if they have one)are different. while processing if the input begins string derived from α we do not know or decide whether to expand

A to $\alpha\beta_1$ or A to $\alpha\beta_2$

However we can defer the decision by first expanding A to $\alpha A'$ and then after seeing the i/p derived from α and look ahead symbol, we expand A' to β_1 or β_2. This is left factored and the production are re-written for the grammar and is as follows:

$A \rightarrow \alpha A'$

$A' \rightarrow \beta_1 \mid \beta_2$ so, we can immediately expand A to $\alpha A'$

Left-Factoring – Algorithm:

Input : Grammar G

Output : An equivalent left factored grammar

Method :

1. For each Non-terminal A find the longest prefix α common to two or more alternatives (production rules).

2. If α <> ε then replace all of A-productions

$A \rightarrow \alpha\beta_1 \mid ... \mid \alpha\beta_n \mid \gamma_1 \mid ... \mid \gamma_m$

where γi represents all alternatives that do not begin with α

by

$A \rightarrow \alpha A' \mid \gamma_1 \mid ... \mid \gamma_m$ Here A' is a new Non-terminal

$A' \rightarrow \beta_1 \mid ... \mid \beta_n$

3. Step 1 and 2 are repeated until no two alternatives for a Non-terminal have a common prefix.

Example:1

$A \rightarrow \underline{a}bB \mid \underline{a}B \mid cdg \mid cdeB \mid cdfB$

$\Downarrow$

$A \rightarrow aA' \mid \underline{cd}g \mid \underline{cd}eB \mid \underline{cd}fB$

$A' \rightarrow bB \mid B$

$\Downarrow$

$A \rightarrow aA' \mid cdA''$

$A' \rightarrow bB \mid B$

$A'' \rightarrow g \mid eB \mid fB$

Example:2

$A \rightarrow ad \mid a \mid ab \mid abc \mid b$

$\Downarrow$

$A \rightarrow aA' \mid b$

$A' \rightarrow d \mid \varepsilon \mid b \mid bc$

$\Downarrow$

$A \rightarrow aA' \mid b$

$A' \rightarrow d \mid \varepsilon \mid bA''$

$A'' \rightarrow \varepsilon \mid c$

❖ Parsing:

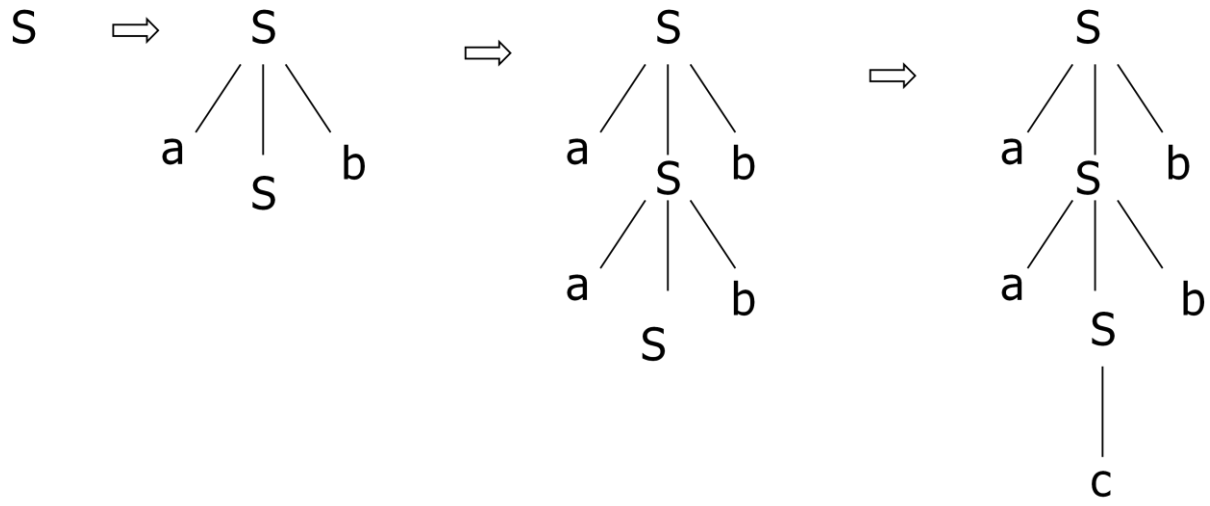
- **Top down parsing**

In top down parsing we start from the start symbol of the grammar and by choosing the production judiciously we try to derive the given sentence.

- **Bottom-up parsing**

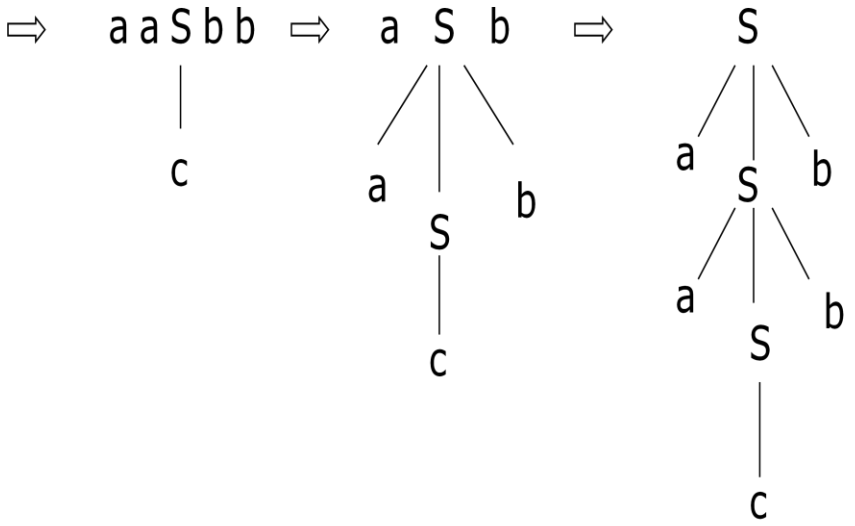
In bottom-up parsing we start from the given sentence and using various production, we try to reach the start symbol.

Consider the grammar S -> aSb | c for the string aacbb



Top down parsing

aacbb=>



Bottom-up parsing

❖ Design strategy for Top down Parser:

Basically Top down parsing can be viewed as an attempt to find leftmost derivation for an input string and constructs a parse tree from root to leaves. The following steps may be followed.

1. Given a Non-terminal (Initially Start symbol) which is to be expanded, the first alternative (production rule) is used for expansion.

2. Within the newly expanded string, the substring of terminals from left are compared with input string. If found to be match then next left most Non terminal is selected for expansion and the step 2 is repeated.

3. Otherwise the current alternative structure(production rule) selected is incorrect, hence undo the previous expansion and use the next alternative structure of the Non-terminal for expansion and step 2 is repeated.

4. In the process of step 2 and 3 if No alternative structure for a Non-terminal to be tried then process is backed up by undoing all the previous expansion. In the process of backtracking, If we reach start symbol and no alternative structure to be tried then input is invalidated. Otherwise if no Non-terminal are left for expansion then input is validated.

Recursive-Descent Parsing (uses Backtracking):

- In order to find the correct production the general form of top down parser uses **backtracking** (via recursive calls) and is called is **Recursive Descent parser**.
- It consists of **set of procedures**, one for each Non-terminal and looks for a substring of input string and if it is found it returns **TRUE**, otherwise, it returns **FALSE.**
- Execution begins with a call to procedure for **start symbol** which **halts and announces successful parsing** if its procedure body scans the entire input string, otherwise **it announces unsuccessful parsing.**
- The pseudo-code for a typical Non-terminal may be written as follows :

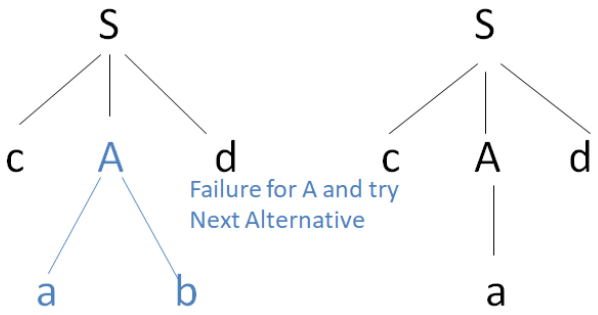
Recursive-Descent Parsing (uses Backtracking)

- Example:

$S \rightarrow cAd$

$A \rightarrow ab \mid a$

input: cad



Pseudo Code for implementation:

```
Main()
{
    i=1;   /* index pointing to input string */
    read input
     if (S() and input[i] == $)
          print(" String is valid ");
     else
          print(" String is Invalid ");
}
```

```
int S()
{
  If input[i] ==‘c’ then
  {
     i=i+1;
      If A() then
      {
          if input[i]==‘d’
          {
               i=i+1;
               return 1;
          }
      }
      else
               return 0;
  }
   else
        return 0;
}
```

```
Int A()
{
  isave=i;
  if input[i] == ‘a’ then
  {
     i=i+1;
     if input[i] == ‘b’ then
     {
        i=i+1;
        return 1;
     }
  }
     i=isave;
     if input[i]==‘a’ then
     {
        i=i+1;
        return 1;
      }
  else
   {
      return 0;
   }
}
```

Pseudo-code for Non-terminal in Recursive-descent Parser (RDP):

Void A()

{

Choose an **A-production A→$X_1X_2X_3….X_k$**

for (i=1 to k)

{

if (**X_i is Non-terminal**)

call procedure X_i()

else if (**Xi equals the current input symbol ‘a’**)

advance the input to the next symbol

else

error() /* error and try next alternative for A-production */

}

}

Difficulties of Recursive Descent parser:

1. A left recursive grammar creates top down parser to go into an infinite loop. i.e if A→Aα is a A-production then, when we try to expand A, we may find ourselves again trying to expand A without having consumed any input.
2. A second problem concerns backtracking. If we make a sequence of erroneous expansion, we may have to undo the semantic action taken. This slows the process of parsing hence backtracking must be avoided.
3. The order in which the alternative structure (production rules) are selected for Non-terminal would affect the language accepted.

Prerequisites for Predictive topdown parsers:

- Elimination of Left-recursion
- Left Factoring
- First Set
- Follow Set

First and follow sets:

- The implementation of both Top-down parser and bottom parser is aided by two function name **FIRST** and **FOLLOW** sets associated with **grammar G**.
- These two sets allow us choose which production to be selected based on the next input symbol
- FIRST(α) : It is defined to be the set terminals that begin string derived from α.
- How it is used ?

Consider two A-production

A → α |β where FIRST(α) and FIRST(β) are disjoint sets.

Let us consider the terminal 'a' to be first symbol which is either in FIRST(α) or FIRST(β) but not in both. When choosing A-production we see the look-ahead symbol 'a' from the input. If 'a' in FIRST(α) then select A production as A → α or If 'a' in FIRST(β) then select A production as A → β

FIRST set Computation:

FIRST(**X**) for all Grammar symbols **X** can be computed by applying the following rules until no more **terminals** or **ε can be added to any FIRST set.**

1. **IF X is a terminal then** FIRST(**X**)= { **X** }

2. **IF X = ε or X→ε then** FIRST(**X**) = {**ε** }

3. **IF X is a Non-Terminal and X→$Y_1Y_2Y_3$ ¨ Y_k then**

FIRST(**X)=**FIRST(**$Y_1Y_2Y_3$ …. Y_k)**

= FIRST**(Y_1) → if FIRST(Y_1) does not derive any empty string ε**

FIRST(**X)=**FIRST(**$Y_1Y_2Y_3$ ¨ Y_k)**

= FIRST**(Y_1) – {ε } U** FIRST**(Y_2Y_3…..Y_k)**

→ if Y_1 derive an empty string ε.

FIRST**(Y_2Y_3…Y_k) =** FIRST**(Y2)**

→ if Y_2 does not derive an empty string ε.

FIRST**(Y_2Y_3…. Y_k) =**FIRST**(Y2) –{ε }U** FIRST**(Y_3…Y_k)**

→ if Y_2 derive an empty string ε.

This is repeated for each Yi until no more terminals or ε can be added

//examples to solve:

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

2. $E \rightarrow T E'$
   $E' \rightarrow +T E' \mid \varepsilon$
   $T \rightarrow F T'$
   $T' \rightarrow *F T' \mid \varepsilon$
   $F \rightarrow id \mid (E)$

3. $S \rightarrow ACB \mid CbB \mid Ba$

4. . $S \rightarrow AaAb \mid BbBa$
   $A \rightarrow \varepsilon$
   $B \rightarrow \varepsilon$

$A \rightarrow da \mid BC$

$B \rightarrow g \mid \varepsilon$

$C \rightarrow h \mid \varepsilon$

FOLLOW computation:

- If the grammar is **ε-free** then **FIRST** symbols are used in selecting the appropriate production for some Non-terminal and these gets added to Parsing table
- But when the grammar is **not ε-free**, the FIRST symbols cannot be used to decide the appropriate productions, as these are not added to parsing table. i.e If there is production **A→ε** in the grammar then when **A** is replaced by ε cannot be decided by the FIRST symbols and hence additional information is required to decide when **A→ε** is to be used so that it can be added in the table. Here we need **FOLLOW** symbols to take the decision.
- FOLLOW(A) : It is defined to be the set of terminals 'a' that can appear immediately to the right of A in some sentential form
- $S \Rightarrow \alpha Aa\beta$
- To Compute FOLLOW(**A**) for all Non-terminals, apply the following rules until nothing can be added to any follow Set

1. Place **$** in FOLLOW(**S**), where **S** is the start symbol **$** is the input right end-marker.

2. If there is a production A→α**B**β then everything in FIRST**(β**) except **ε** is in FOLLOW(**B**).

3. If there is production A→α**B or** a production A→α**B**β where FIRST(β) contains **ε** then everything in FOLLOW(**A**) is FOLLOW(**B**). i.e FOLLOW(**B**) = FOLLOW(**A**)

LL(1) Parsers:

A grammar such that it is possible to choose the correct production with which to expand a given nonterminal, looking only at the next input symbol, is called LL(1). These grammars allow us to construct a predictive parsing table that gives, for each nonterminal and each lookahead symbol, the correct choice of production. Error correction can be facilitated by placing error routines in some or all of the table entries that have no legitimate production.

The first "L" in LL(1) stands for scanning the input from left to right, the second "L" for producing a leftmost derivation, and the "1" for using one input symbol of lookahead at each step to make parsing action decisions.

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 and β derive strings beginning with a.

2. At most one of a and β can derive the empty string.

3. If $\mathcal{B} \Rightarrow \mathcal{E}$, then a does not derive any string beginning with a terminal in FOLLOW(A). Likewise, if $\alpha \Rightarrow \mathcal{E}$ then $\mathcal{B}$ does not derive any string beginning with a terminal in FOLLOW(A).

ALGORITHM:construction of predictive parsing table

Consider the Grammar- G:

For each production **A → α** do the following:

a) Find **FIRST (α)** – **call set as { S1 }**

and **FOLLOW (A)** - **call set as { S2 }**

b) For all symbols in **{S1}** make entries in the table as

TABLE[**A, a**] = **A → α** , where **a is S1**

c) if **ε** is in **{S1}** then make the entries in the table as

TABLE**[A, b] = A → α.** where **b is S2**

Ex.: for this grammar,

E → T E'

E' → + T E' | **ε**

T → F T'

T' → * F T' | **ε**

F → (E) | id

We have first and follow symbols as->

	FIRST	**FOLLOW**
E	**(id**	**) $**
E'	**+ , ε**	**) $**
T	**(id**	**+) $**
T'	*** , ε**	**+) $**
F	**(id**	**+ *) $**

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

And this is the parsing table that can be obtained.

Ex.: for this grammar,

S->AaAb | BbBa

A-> **ε**

B-> ε

We have first and follow symbols as->

	FIRST
S	**a, b**
A	**ε**
B	**ε**

M	a	b	$
S	S -> AaAb	S->BbBa	
A	**A → ε**	**A → ε**	
B	**B → ε**	**B → ε**	

And this is the parsing table that can be obtained.

Tracing for input string ba:

matched	**stack**	**input**	**action**
	S$	ba$	
	BbBa $	ba$	Consult table M[S,b] i.e., S->BbBa pushed onto stack
	bBa $	ba$	Consult table M[B,b] i.e., **B → ε** pushed onto stack
b	Ba $	a$	Match b
	a $	a$	Consult table M[B,a] i.e., **B → ε** pushed onto stack
a	$	$	Match a

As we have $ on top of stack and also input pointer , string is successfully parsed. And parse tree can be generated for this string.

Nonrecursive Predictive Parsing:

A nonrecursive predictive parser can be built by maintaining a stack explicitly, rather than implicitly via recursive calls.

If w is the input that has been matched so far, then the stack holds a sequence of grammar symbols a such that

*

S=> wα

lm

The table-driven parser has an input buffer, a stack containing a sequence of grammar symbols, a parsing table constructed by Algorithm, and an output stream.

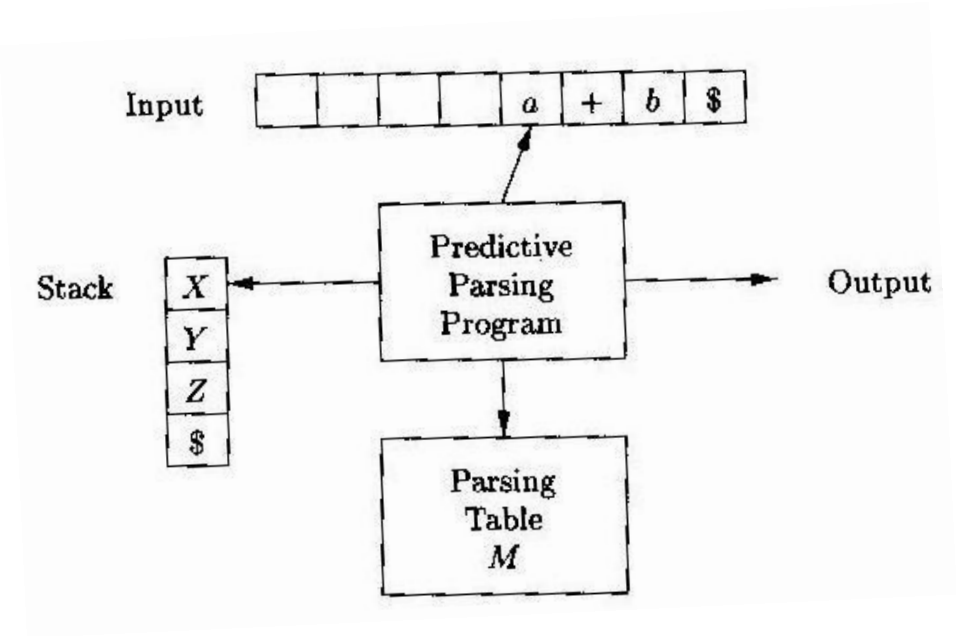
The input buffer contains the string to be parsed, followed by the endmarker $.

We reuse the symbol $ to mark the bottom of the stack, which initially contains the start symbol of the grammar on top of $.

The parser is controlled by a program that considers X, the symbol on top of the stack, and a, the current input symbol.

If X is a nonterminal, the parser chooses an X-production by consulting entry M[X, a] of the parsing table .

Otherwise, it checks for a match between the terminal X and current input symbol a.



Algorithm : Table-driven 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 is in a configuration with w$ in the input buffer and the start symbol S of G on top of the stack, above $.*/

set ip to point to the first symbol of w;

```
 set X to the top stack symbol;
while ( X!= $ )
 {
 /* stack is not empty */
 if ( X is a )
        pop the stack and advance zp;
else if ( X is a terminal )
        error();
 else if ( M[X, a] is an error entry )
        error();
else if ( M[X,a] = X -+ Y1Y2 Yk )
{
        output the production X -+ YlY2 -. Yk;
pop the stack; push Yk, Yk-1,. . . , Yl onto the stack, with Yl on top;
}
 set X to the top stack symbol;
}
```

Example : On input id + id * id, the nonrecursive predictive parser of Algorithm makes the sequence of moves.

MATCHED	STACK	INPUT	ACTION
	$E\$$	id + id * id$	
	$TE'\$$	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 $T' \to \epsilon$
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 $T' \to * FT'$
id + id *	$FT'E'\$$	id$	match *
id + id *	id $T'E'\$$	id$	output $F \to$ id
id + id * id	$T'E'\$$	$	match id
id + id * id	$E'\$$	$	output $T' \to \epsilon$
id + id * id	$	$	output $E' \to \epsilon$

Error Recovery in Predictive Parsing:

An Error is detected when :

- TRM on top of stack does not match with next i/p symbol.
- TABLE[A, a] is error i.e. table entry is empty.
- **1. PANIC MODE OF ERROR RECOVERY:**
- Skipping the symbol on the i/p until a token in selected set of synchronizing tokens appears and popping the current Non-terminal from the stack
- SYNC- TOKEN (A) = FOLLOW (A)
- If we add symbols in FIRST (A) to Synchronizing set of non TRM A, then it may be possible to resume parsing according to A if a symbol in FIRST (A) appears in the i/p.
- If A→ ε ; this can be used so that some error detection may be postponed, but cannot cause error to be missed.
- If TRM cannot be matched, pop the terminal and issue an message saying that TRM was inserted and continue parsing.
- Ex.: parsing table above can be modified as ->

NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$	synch	synch
E'		$E \to +TE'$			$E \to \epsilon$	$E \to \epsilon$
T	$T \to FT'$	synch		$T \to FT'$	synch	synch
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to$ id	synch	synch	$F \to (E)$	synch	synch

And now the nonrecursive predictive parser of Algorithm makes the sequence of moves

STACK	INPUT	REMARK
E $	) id * + id $	error, skip)
E $	id * + id $	id is in FIRST(E)
TE' $	id * + id $	
$FT'E'$ $	id * + id $	
id $T'E'$ $	id * + id $	
$T'E'$ $	* + id $	
* $FT'E'$ $	* + id $	
$FT'E'$ $	+ id $	error, $M[F, +] =$ synch
$T'E'$ $	+ id $	F has been popped
E' $	+ id $	
$+TE'$ $	+ id $	
TE' $	id $	
$FT'E'$ $	id $	
id $T'E'$ $	id $	
$T'E'$ $	$	
E' $	$	
$	$	

2. PHRASE-LEVEL ERROR RECOVERY:

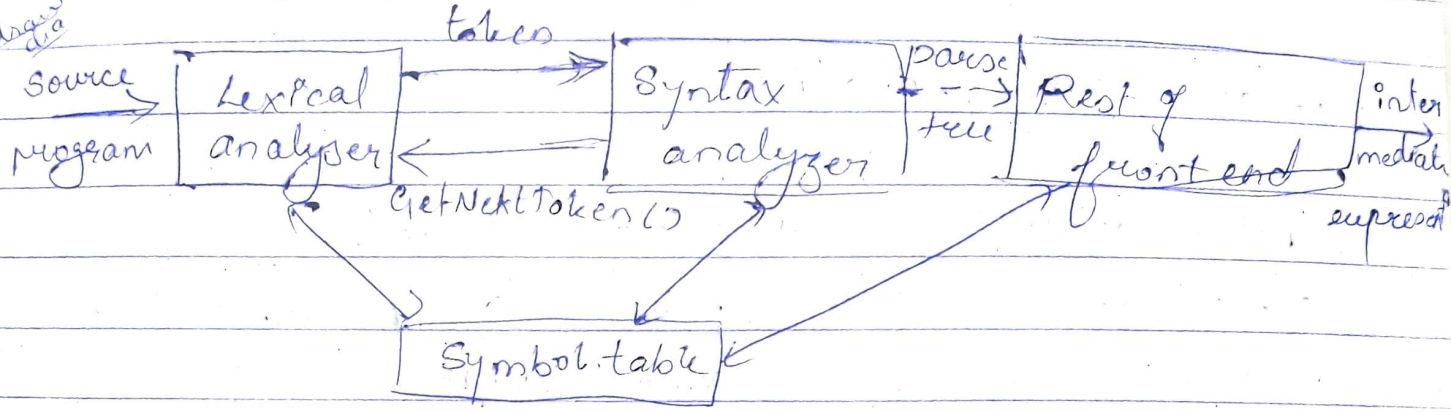
- This is implemented by filling the blank entries in the parsing table with pointers to error routines. These routines may change, insert or delete symbols in the input or STACK and issue appropriate error messages.

unit - 2

Syntax analysis - I

Role of parser

1) Produce parse tree
2) check syntax



what is parser -

there are 3 types of parser / parsing technique

1) Top-down parsing (unambiguous, non recursive) root to leaves. Representative grammer.
2) bottom up parsing leaves to root (unambiguous)
3) Universal ↓ can pass any grammer (costly).

Expression grammer

$E \rightarrow E+E \mid E*E \mid (E) \mid id$. - Ambigous grammer

unambigious grammer

$E \rightarrow E+T \mid T$.

$T \rightarrow T*F \mid F$.

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

Two ways to reduce ambiguity.

1) use terminal

free form recursion

$E \rightarrow TE'$

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

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \epsilon$.

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

Syntax error handling

1) lexical errors → handle by lexer. misspelling
2) syntax errors → " " parser. missing bracket, semicolon
3) semantic errors → " " semantic analyser.
4) Logical errors → ex a=b. instead of a==b

parser is imp phase in compiler

Error recovery strategies

1) Panic mode recovery

2) phrase level recovery

3) Error productions - append errors in grammer as productions.

4) global correction

Context free grammer (CFG).

Type 2

Regular language can be described by CFG

V, T, PS

derivation - RMD, LMD.

ambiguous

ex $E \Rightarrow E+E \mid E*E \mid (E) \mid id$.

$u = id + id * id$.

$E \overset{LMD}{\Rightarrow} E+E$

$E \overset{LMD}{\Rightarrow} id+E$

$E \overset{LMD}{\Rightarrow} id+E*E$

$E \overset{LMD}{\Rightarrow} id+id*E$

$E \overset{LMD}{\Rightarrow} id+id*id$.

$E \overset{RMD}{\Rightarrow} E*E$

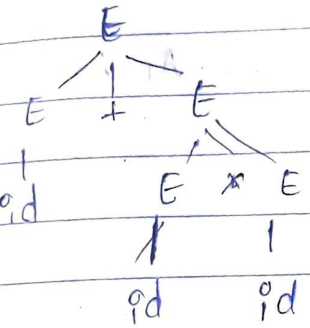
$E \overset{RMD}{\Rightarrow} E+E*E$

$E \overset{RMD}{\Rightarrow} id+E*E$

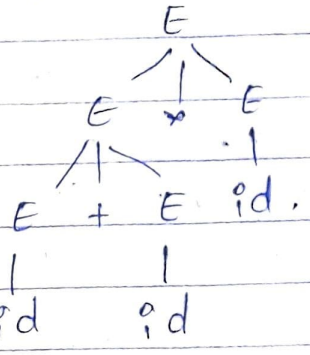
$E \overset{RMD}{\Rightarrow} id+id*E$

$E \overset{RMD}{\Rightarrow} id+id*id$.

parse tree.



parse tree

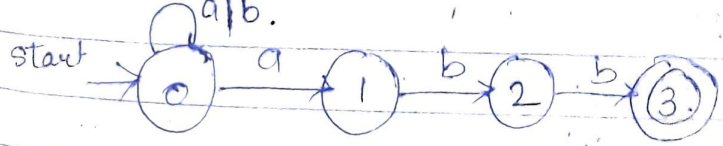


* Verifying the language generated by grammer.

* Context free grammer vs regular expression.
all RE (regular languages) subset of context free language (CFL)
vice-versa not possible.

Not imp
1) Draw the NFA for the language that ends with abb.

⇒ RE = $(a+b)^* abb$



Procedure to Convert NFA to grammer.
1) For each state i of NFA create a non terminal A_i.

here A_0, A_1, A_2, A_3

2) If state i has a transition to state j on input a, add the production.

$A_i \rightarrow aA_j$.

If state i goes to state j on input ϵ, add it recursion.

$A_i \rightarrow A_j$.

3) If i is an accepting state add ϵ.

$A_i \rightarrow \epsilon$.

4) If i is the start state make A_i be the start symbol of the grammer.

$S \rightarrow A_i$.

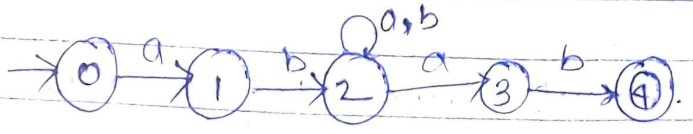
Grammer for above NFA

state 0 — A_0
state 1 — A_1
state 2 — A_2
state 3 — A_3.

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

2) Draw NFA for the language that begins & ends with ab.

$\Rightarrow$ RE = $ab(a+b)^* ab$.



$A_0 \rightarrow aA_1$
$A_1 \rightarrow bA_2$
$A_2 \rightarrow aA_2 \mid bA_2 \mid aA_3$
$A_3 \rightarrow bA_4$
$A_4 \rightarrow \epsilon$

expr → $t_1 | t_2$

stmt → $s_1 | s_2$

Writing a grammer

1) Lexical vs syntactic analysis.

same abefore

2) eliminating ambiguity

* grammer for conditional statement.

"dangling else"

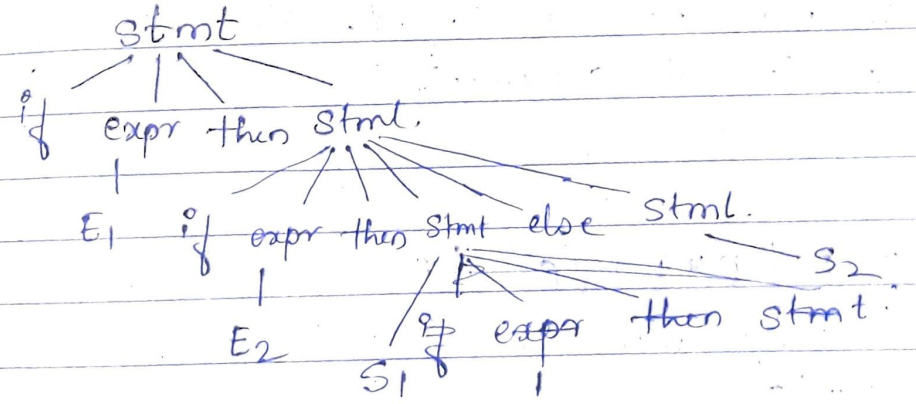
stmt → if expr then stmt

| if expr then stmt else stmt.

| other.

if E_1 then if E_2 then S_1 else S_2

construct parse tree.



Match each 'else' with the closest unmatched 'then' (both should be in some level)

Associat^{n} , Precedence

stmt → matched_stmt
| open_stmt

matched_stmt → if expr then matched_stmt else matched_stmt
| other

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

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

Open_stmt

E → E+E | E*E | (E) | id

unambiguous grammar

E → E + T | T

T → T * F | F

F → (E) | id.

Least priority operator near start symbol

ex E → E+E | E-E

* Left Factoring

Ex ① $S \to ab \mid ac$

simplifying :-

$S \to aS'$

$S' \to b \mid c$

② $S \to abc \mid abd \mid abe \mid b.$

rewriting the grammer.

$S \to abS' \mid b.$

$S' \to c \mid d \mid e.$

If $A \to \alpha B_1 \mid \alpha B_2$ are 2 A-productions, & the input begins with a non empty string derived from α we do not know whether to expand A to αB_1 or αB_2. However we may defer the decision by expanding A to $\alpha A'$ the original production becomes

$A \to \alpha A'$

$A' \to B_1 \mid B_2$

Algorithm.

Left factoring a grammer.

Input : a grammer G

output : An equivalent left factor grammer.

method :

For each non terminal A find the longest prefix α common to 2 or more of its alternatives. If $\alpha \neq \epsilon$ replace all of A productions $A \to \alpha B_1 \mid \alpha B_2 \mid \dots \mid \alpha B_n \mid \gamma$ where γ represents all productn (alternatives) that do not begin with α. by

$A \to \alpha A' \mid \gamma$

$A' \to B_1 \mid B_2 \mid \dots \mid B_n$

A' is new non terminal

Repeatedly apply this operation until no two alternatives have the common prefix.

5m

* Do left factoring for following grammar

(1) $A \rightarrow aAB \mid aBc \mid aAc$

$\Rightarrow$ $A \rightarrow aA'$

$A' \rightarrow AB \mid Bc \mid Ac$

$A' \rightarrow AP \mid Bc$

$P \rightarrow B \mid c$

(2) $S \rightarrow bSSaaS \mid bSSaSb \mid bSb \mid a$

$S \rightarrow bSS' \mid a$

$S' \rightarrow SaaS \mid SaSb \mid b$

$S' \rightarrow SaP \mid b$

$P \rightarrow aS \mid Sb$

(3) $S \rightarrow a \mid ab \mid abc \mid abcd$

$S \rightarrow aA'$

$A' \rightarrow \epsilon \mid b \mid bc \mid bcd$

$A' \rightarrow \epsilon \mid bP$

$P \rightarrow \epsilon \mid c \mid cd$

$P \rightarrow \epsilon \mid cQ$

$Q \rightarrow \epsilon \mid d$

resulting grammar

$S \rightarrow aA'$

$A' \rightarrow \epsilon \mid bP$

$P \rightarrow \epsilon \mid cQ$

$Q \rightarrow \epsilon \mid d$

(4) $S \rightarrow aAd \mid aB$

$A \rightarrow a \mid ab$

$B \rightarrow ccd \mid ddc$

$\Rightarrow$ $S \rightarrow aS'$

$S' \rightarrow Ad \mid B$

$A \rightarrow aA'$

$A' \rightarrow \epsilon \mid b$

$B \rightarrow ccd \mid ddc$

* Elimination of left recursion.

$E \to E+T$ → left recursion.
↳ left side. (Immidiate left recursion)

* $A \overset{+}{\Rightarrow} A\alpha$ → left recursion.

$E \to T+E$ → right recursion.
↳ right side.

A grammer is left recursive if it has a non terminal A such that there is a derivation $A \overset{+}{\Rightarrow} A\alpha$ for some string α.

Ex $S \to Ab \mid \epsilon$
$A \to Sa \mid b$.

Immidiate left recursion where there is production of the form $A \to A\alpha$.

only for Immidiate left recursive grammer
* A productions $A \to A\alpha \mid \beta$ could be replaced by the non left recursive productions.

$A \to \beta A'$
$A' \to \alpha A' \mid \epsilon$

* Eliminate left recursion

① $E \to E+T \mid T$
$T \to T*F \mid F$
$F \to (E) \mid id.$

⇒ for $E \to E+T \mid T$ resultant grammer
$E \to TE'$ $E \to TE'$
$E' \to +TE' \mid \epsilon$ $E' \to +TE' \mid \epsilon$
for $T \to T*F \mid F$ $T \to FT'$
$T \to FT'$ $T' \to *FT' \mid \epsilon$
$T' \to *FT' \mid \epsilon$. $F \to (E) \mid id.$

getvidhylay.com → examples.

(cycle $A \xRightarrow{+} A$)

(2) Algorithm:

Eliminating left recursion

Input :- A grammer G with no cycles or ε production

output :- An equivalent grammer with no left recursion.

Method:

a Apply the following algorithm. Note that the resulting non left recursive grammer may have ε production

1) Arrange the non terminals in some order $A_1, A_2 \dots A_n$.
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 $A_i \to \delta_1\gamma \mid \delta_2\gamma \mid \dots \mid \delta_k\gamma$

where $A_j \to \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$ all current A_j prod^n

5) }
6) eliminate the immidiate left recursion among the A_i productions.
7) }

* Eliminate left recursion from the following gram

(1) $A \to ABd \mid Aa \mid a$
$B \to Be \mid b$

$\Rightarrow$ $A \to aA'$
$A' \to BdA' \mid aA' \mid \epsilon$
$B \to bB'$
$B' \to eB' \mid \epsilon$

(2) $S \to (L) \mid a$

$L \to L, S \mid S$.

$\Rightarrow$ $L \to SL'$

$L' \to ,L' \mid \epsilon$

(3) $S \to A$

$A \to Ad \mid Ae \mid aB \mid ac$

$B \to bBc \mid f$

$\Rightarrow$ $S \to A$

$A \to aBA' \mid acA'$

$A' \to dA' \mid eA' \mid \epsilon$

$B \to bBc \mid f$

(4) $A \to Aa \mid B$

$\Rightarrow$ $A \to BA'$

$A' \to aA' \mid \epsilon$

5) $S \to Ab \mid b$ (indirect recursion)

$A \to Ac \mid Sd \mid \epsilon$

$\Rightarrow$ $S \to Ab \mid b$

~~$A \to Sd A' \mid \epsilon$~~

$A' \to Ac \mid Abd \mid bd \mid \epsilon$

$A \to bdA' \mid \epsilon$

$A' \to cA' \mid bdA' \mid \epsilon$

6) $A \to Ba \mid Aa \mid c$ (indirect left recursion)

$B \to Bb \mid Ab \mid d$

$\Rightarrow$ i) remove immidiate left recursion of 1st NT (if present)

$A \to BaA' \mid cA'$

$A' \to aA' \mid \epsilon$

$B \to Bb \mid Ab \mid d$.

replace A with its body (new A)

$B \to Bb \mid BaA'b \mid cA'b \mid d$.

$B \to cA'bB' \mid dB'$ (remove immidiate left recursion)

$B' \rightarrow bB' \mid aA'bB' \mid \epsilon$

Resulting grammer is

$A \rightarrow BaA' \mid cA'$

$A' \rightarrow$

7) i) $X \rightarrow XSb \mid Sa \mid b$

ii) $S \rightarrow Sb \mid Xa \mid a.$

$\Rightarrow X \Rightarrow Sa. \Rightarrow Xaa.$ (indirect left recursion)

Step 1 $X \rightarrow XSb \mid Sa \mid b.$

remove immidiate left recursion from 1st

$X \rightarrow SaX' \mid bX'$

$X' \rightarrow SbX' \mid \epsilon$

Step 2 $S \rightarrow Sb \mid Xa \mid a.$

replace X with body

$S \rightarrow Sb \mid SaX'a \mid bX'a \mid a.$

remove immidiate left recu

$S \rightarrow bX'aS' \mid aS'$

$S' \rightarrow bS' \mid aX'aS' \mid \epsilon.$

Step 3 Resulting grammer :-

$X \rightarrow SaX' \mid bX'$

$X' \rightarrow SbX' \mid \epsilon$

$S \rightarrow bX'aS' \mid aS'$

$S' \rightarrow bS' \mid aX'aS' \mid \epsilon$

8) i) $S \rightarrow Aa \mid b$

ii) $A \rightarrow Ac \mid Sd \mid \epsilon.$

$\Rightarrow$ Step 1: 1st is as it Ps (No immidiate left recu)

step 2: $A \rightarrow Ac \mid Aad \mid bd \mid \epsilon$

$A' \rightarrow bdA' \mid A'$

$A' \rightarrow cA' \mid adA' \mid \epsilon$

Recalling grammer

$S \rightarrow Aa \mid b$.

$A \rightarrow bdA' \mid A'$

$A' \rightarrow cA' \mid adA' \mid \epsilon$.

general form of TDP → recursive descent parsing

⑨ Top-down Parsing - equivalent to LMD.

* Cond^n To apply top-down parsing.

1) Grammer should unambiguous
2) left factored grammer.
3) free from left recursion.

ambiguous

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

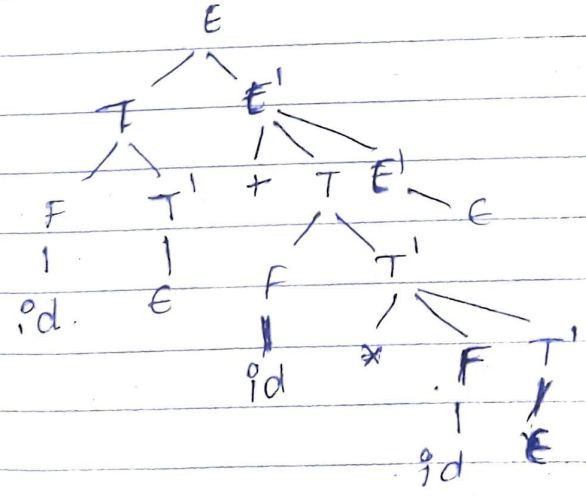
left recursive

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

free from left.

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

id + id * id. parse tree



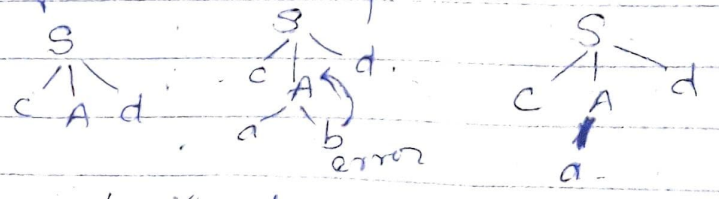
Recursive Descent parsing - General form of TDP

requires backtracking to find current A's prod

$S \rightarrow cAd$

$A \rightarrow ab \mid a$.

$w = cad$



while backtracking store input ~~varia~~ pointer

predictive parsing → special case of recursive descent parsing where no backtracking is required.
looks ahead at i/p a fixed no. of symbols to chose correct production

First and follow. A's

says which production to be applied based on next input α → any string

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

ex ① $S \to abc \mid da \mid pq$

FIRST(S) = {a, d, p}.

② $S \to \epsilon$

first(S) = {ϵ}

if α is non terminal

$S \to \underset{Y_1}{A}\underset{Y_2}{B}\underset{Y_3}{C}$ $B \to b$

$A \to a$ $C \to c$

first(A) = {a}

first(B) = {b}

first(C) = {c}

first(S) = first(A) = {a}

Compute "First" from the following grammer

⇒ 1) $A \to abc \mid def \mid ghi$

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

2) $S \to aBDh$ First(S) = {a}

$B \to bcC$ First(b) = {c}

$C \to bC \mid \epsilon$ First(C) = {b, ϵ}

$D \to EF$ contains NT.

$E \to gE$ first(E) = {g, ϵ}

$F \to f \mid \epsilon$. first(F) = {f, ϵ}

if t contains ϵ then take F.

$First(D) = \{First(t) - \epsilon\} \cup \{First(F)\}$

$= \{g, f, \epsilon\}$

(3) $S \to A$

$A \to aB \mid Ad$

$B \to b$

$C \to g$

⇒ first eliminate left recursion

$S \to A$

$A \to aBA'$

$A' \to dA' \mid \epsilon$

$B \to b$

$C \to g$

$First(A) = \{a\}$

$First(A') = \{d, \epsilon\}$

$First(B) = \{b\}$

$First(C) = \{g\}$

$First(S) = First(A) = \{a\}$

(4) $S \to (L) \mid a$

$L \to SL'$

$L' \to ,SL' \mid \epsilon$

⇒ $First(S) = \{(, a\}$

$First(L) = First(S) = \{(, a\}$

~~$First(L) = \{First(S) - \epsilon\} \cup First$~~

$First(L) = \{,, \epsilon\}$

5) $S \to AaAb \mid BbBa$

$A \to \epsilon$

$B \to \epsilon$

⇒ $First(A) = \{\epsilon\}$

$First(B) = \{\epsilon\}$

See previous

$First(S) = \{First(A) - \epsilon\} \cup \{First(a)\} \cup \{First(B) - \epsilon\} \cup \{First(b)\}$

$= \{a, b\}$

FOLLOW

define only on non terminal

* $S \to aAb$

  FOLLOW (A) = {b}

  terminal followed by that terminal

* $S \to Aa \mid bBc$

  $A \to a \mid b$.

  FOLLOW (A) = {a}

  FOLLOW (B) = {c}

1) follow of start symbol must contain $

   $S \to aAb$

   FOLLOW (S) = {$}

2) $A \to \alpha B \beta$ then everything in First (β) except ϵ is in FOLLOW (B)

3) $A \to \alpha B$ or $A \to \alpha B \beta$ where FIRST (β) contain ϵ, then everything in FOLLOW (A) is in ~~FOLLOW~~ FOLLOW(B)

* Compute first & FOLLOW

(1) $S \to aBDh$

$B \to cC$

$e \to bC \mid \epsilon$

$D \to EF$

$E \to g \mid \epsilon$

$F \to f \mid \epsilon$

First (S) = {a}

FIRST (B) = {c}

FIRST (C) = {b, ϵ}

FIRST (E) = {g, ϵ}

FIRST (F) = {f, ϵ}

FIRST (D) = ~~{g, ϵ}~~

{first(E) - ϵ} ∪ FIRST(

{g, f, ϵ}

FOLLOW

FOLLOW (S) = {$}

FOLLOW (D) = ~~{h}~~ FIRST (h) = {h}

FOLLOW (B) = {FIRST (D) - ϵ} ∪ FOLLOW (S)

= {g, f, $}

FOLLOW (C) = FOLLOW (B) = {g, f, h}

FOLLOW (E) = {FIRST (F) - ϵ} ∪ {~~FOLLOW~~ FIRST (D)} ③

② {f, h}

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

from ③

② $S \to A$

$A \to aB \mid Ad$

$B \to b$

$C \to g$

eliminate left recursion

$S \to A$

$A \to aBA'$

$A' \to \epsilon \mid dA'$

$B \to b$

$C \to g$

FIRST (A) = {a}

FIRST (B) = b.

FIRST (A') = {ϵ, d}

FIRST (C) = {g}.

FIRST (S) = FIRST (A) = {a}

FOLLOW (S) = {$}

FOLLOW (A) = ~~{d}~~ FOLLOW (S) = {$}

FOLLOW (A') = FOLLOW (A) = {$}

FOLLOW (B) = ~~FOLLOW~~ {FIRST (A') - ϵ} ∪ FOLLOW (A)

= {d, $}

FOLLOW (C) = NA.

③ $S \to (L) \mid a$

$L \to SL'$

$L' \to ,SL' \mid \epsilon$

FIRST(S) = {(, a}

FIRST(L') = {,, ε}

FIRST(L) = FIRST(S) = {(, a}

FOLLOW(L) = first()) = {)}

FOLLOW(L') = FOLLOW(L) = {)}

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

{First(L') − ε} ∪ FOLLOW(L')

= {$, ,,)}

④ $S \to AaAb \mid BbBa$

$A \to \epsilon$

$B \to \epsilon$

FIRST(A) = {ε}

FIRST(B) = {ε}

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

= {a, b}

FOLLOW(S) = {$}

FOLLOW(A) = FIRST(a) ∪ FIRST(b)

= {a, b}

FOLLOW(B) = FIRST(b) ∪ FIRST(a)

= {a, b}

⑤ $E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid id$

eliminate left

$E \to TE'$

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

$T \to FT'$

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

$F \to (E) \mid id$

FIRST(E') = {+, ε}

FIRST(T') = {*, ε}

FIRST(F) = {(, id}

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

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

FOLLOW(E) = {$} ∪ FIRST(')') = {$,)}

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

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

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

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

(6) S → ACB | CbB | Ba
A → da | BC
B → g | ε
C → h | ε

⇒ FIRST(B) = {g, ε}

FIRST(C) = {h, ε}

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

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

FOLLOW(S) = {$}

FOLLOW(A) = {FIRST(C) - ε} ∪ FOLLOW(S)
= {h, $}

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

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

Top Down parsing

↓

Recursive decent parsing.

Predictive parser

LL(1) grammer. → 1 symbol of look ahead
↓
Left to Right → Left most derivation

predictive parser.
transition diagram for each non terminal

LL(1) → no left recursive / ambigous grammer

A grammer G is LL(1) if & only if whenever $A \to \alpha \mid \beta$ are two distinct production (G), the following cond^n hold.

1) For no terminal a do both α & β derive strings beginning with a.
2) At most
(First(α) & first(β) should be different from ① & ②.)

③ if $\beta \overset{*}{=} \epsilon$ FIRST(α) $\neq$ FOLLOW(A)
if $\alpha \overset{*}{\Rightarrow} \epsilon$ FIRST(β) $\neq$ FOLLOW(A).

Predictive parsing (do not have backtracking)
↓
unambigous.
↓
Left recursion
↓
Left factoring

table M[A][a]. (a: terminal, A: NT) Dynamic predictive

predictive parsing → construct a parsing table M[A

Construction of a predictive parsing table.

INPUT : Grammer G

OUTPUT : parsing table M.

method : For each production $A \to \alpha$ of the grammer do 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) & \$ is in follow(A), add $A \to \alpha$ to M(A, \$] as well.

$E \to TE'$

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

$T \to FT'$

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

$F \to (E) / id$

⇒ terminals +, *, (,), \$, id

~~FIRST~~ (E) =

take productions.

① $E \to TE'$ a $A \to \alpha$. FIRST(α)

FIRST(TE') = { (, id } (FIRST(T)) if first(T) has ϵ then take first(E')

② $E' \to +TE'$ $E' \to \epsilon$

FIRST($+TE'$) = { + } FOLLOW(E') = {), \$ }

③ $T \to FT'$

FIRST(FT') = { (, id }

④ $T' \to *FT'$ $T' \to \epsilon$

FIRST($*FT'$) = { * } FOLLOW(T') = { +,), \$ }

⑤ $F \to (E)$ $F \to id$

FIRST((E)) = { (} FIRST(id) = { id }

Input symbol

Non Terminal	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	$F \to id$			$F \to (E)$		

(Q) $S \to aBDh$

$B \to cC$

$C \to bC \mid \epsilon$

$D \to EF$

$E \to g \mid \epsilon$

$F \to f \mid \epsilon$

Input symbol

NT	a	b	c	f	g	h	$
S	$S \to aBDh$						
B			$B \to cC$				
C		$C \to bC$		$C \to \epsilon$	$C \to \epsilon$	$C \to \epsilon$	
D				$D \to EF$	$D \to EF$	$D \to EF$	
E					$E \to g$		
F							

$A \to \alpha$

(1) $S \to aBDh$.

FIRST $(aBDh) = \{a\}$

(2) $B \to cC$

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

(3) $C \to bC$

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

$C \to \epsilon$

FOLLOW (C) = FOLLOW (B) = (FIRST (D) $\cup$ FIRST $(h) = \{g, f, h\}$

(4) $D \to EF$

FIRST (EF) = {FIRST $(E) - \epsilon$} $\cup$ FIRST (F)

= $\{g, f, \epsilon\}$ $\cup$ FOLLOW $(D) = \{g, f, h\}$

(5) $\epsilon \to \epsilon$

FIRST(ϵ) = {ϵ}

$E \to \epsilon$

FOLLOW(E) = { FIRST(F) - ϵ }

$E \to TE'$

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

$T \to FT'$

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

$F \to (E) \mid id.$

id + id * id

$E \overset{LM}{\Rightarrow} TE' \Rightarrow FT'E' \Rightarrow idT'E' \Rightarrow id\epsilon E'$

$\Rightarrow id + TE' \Rightarrow id + FT'E' \Rightarrow id + idT'E'$

$\Rightarrow id + id\epsilon E' \Rightarrow id + id.$

$E \overset{LM}{\Rightarrow} TE'$ $\quad E \to TE'$

$\Rightarrow FT'E'$ $\quad T \to F$

$\Rightarrow idT'E'$ $\quad F \to id.$

$\Rightarrow id\epsilon E'$ $\quad T' \to \epsilon.$

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

$\Rightarrow id + FT'E'$ $\quad T \to FT'.$

$\Rightarrow id + idT'E'$ $\quad F \to id.$

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

$\Rightarrow id + id * idT'E'$ $\quad F \to id.$

$\Rightarrow id + id * id\epsilon E'$ $\quad T' \to \epsilon$

$\Rightarrow id + id * id$ $\quad E' \to \epsilon.$

LMD

Matched	stack	Input	action

$S \to A$

$A \to aBA'$

$A' \to dA' / \epsilon$

$B \to b$

$C \to g$

parsing table

NT	a	b	d	g	$
S	$S \to A$				
A	$A \to aBA'$				
A'			$A' \to dA'$		$A' \to \epsilon$
B		$B \to b$			
C				$C \to g$	

input symbol

$A' \to [\epsilon]$: follow (A')

follow (A) = follow $(\$) = \$$

$w = abdd$

$S \overset{lm}{\Rightarrow} A$ output $S \to A$

$\Rightarrow aBA'$ o/p $A \to BA'$

$\Rightarrow abA'$ $B \to b$

$\Rightarrow abdA'$ $A' \to dA'$

$\Rightarrow abddA'$ $A' \to dA'$

$\Rightarrow abdd$ $A' \to \epsilon$

matched	stack	i/p	action
	$S\$$	abdd	
	$A\$$	abdd	output $S \to A$
	$aBA'\$$	abdd	$A \to aBA'$
a	$BA'\$$	bdd	match a
	$bA'\$$	bdd	$B \to b$
b	$A'\$$	dd	match b
	$dA'\$$	dd	$A' \to dA'$
d	$A'\$$	d	match d
	$dA'\$$	d	$A' \to dA'$
d	$A'\$$		