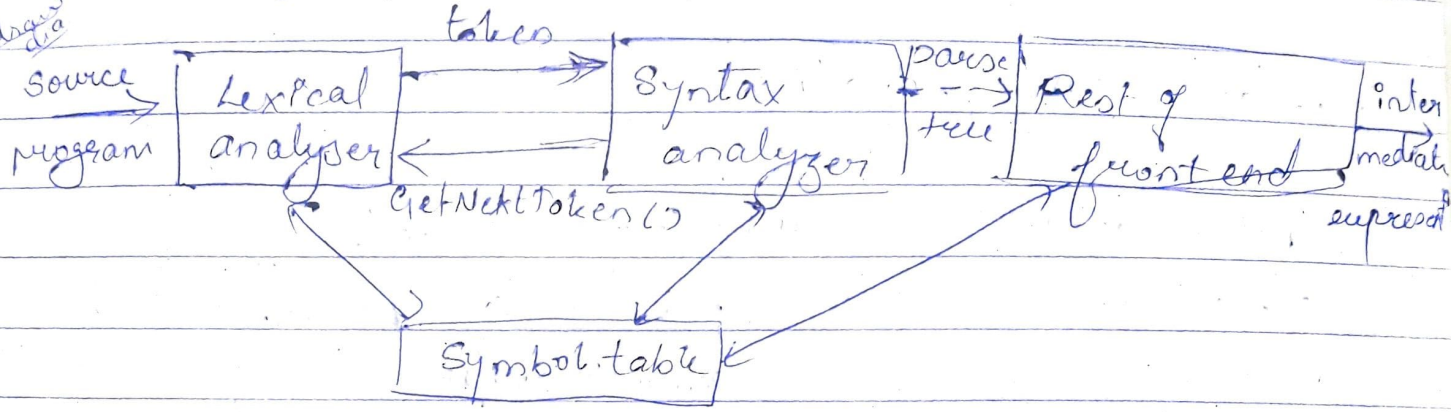
unit - 2

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

$u = id + id * id$.

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

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

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

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

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

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

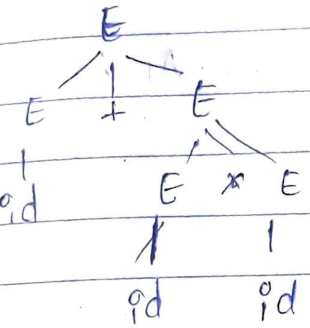
$E \overset{RMD}{\Rightarrow} E+E*E$

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

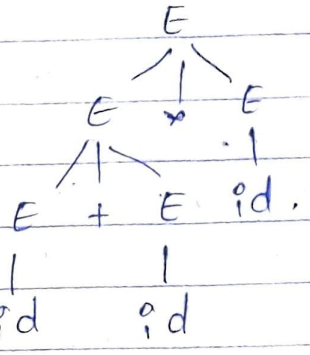
$E \overset{RMD}{\Rightarrow} id+id*E$

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

parse tree.



parse tree

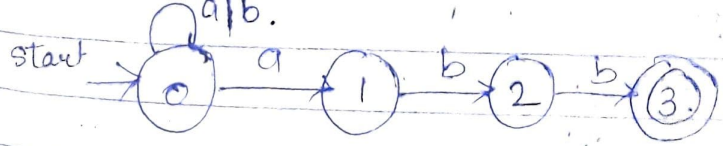


* Verifying the language generated by grammer.

* 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.

$S \rightarrow A_i$.

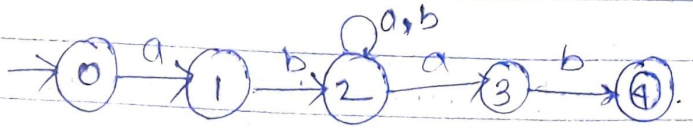
Grammer for above NFA

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"

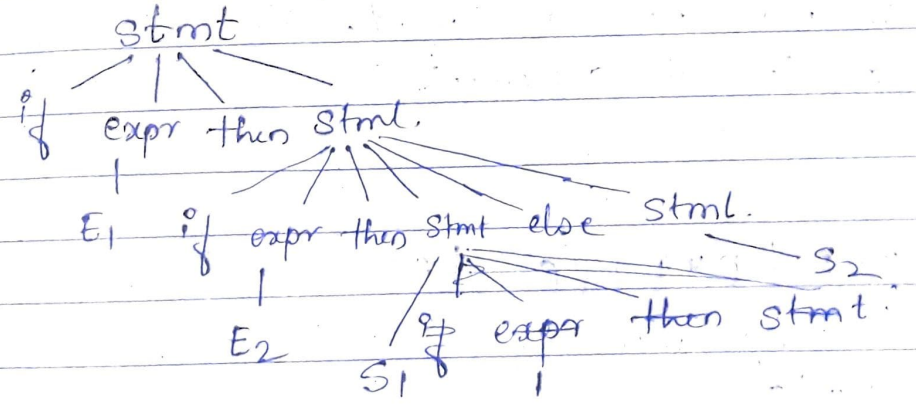
stmt → if expr then stmt

| 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

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

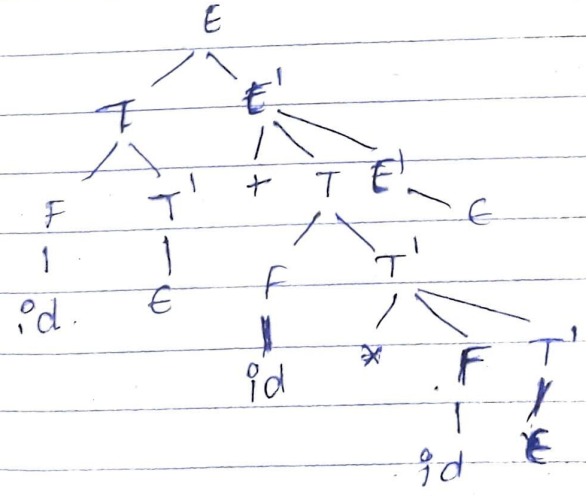
left recursive

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

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



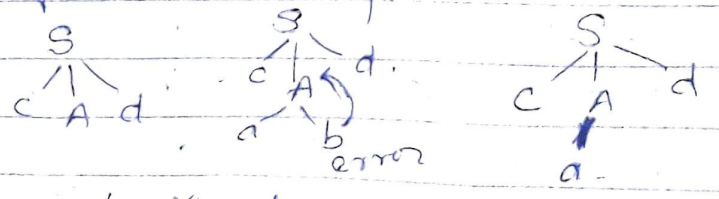
Recursive Descent parsing - General form of TDP

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	$A'\$$		