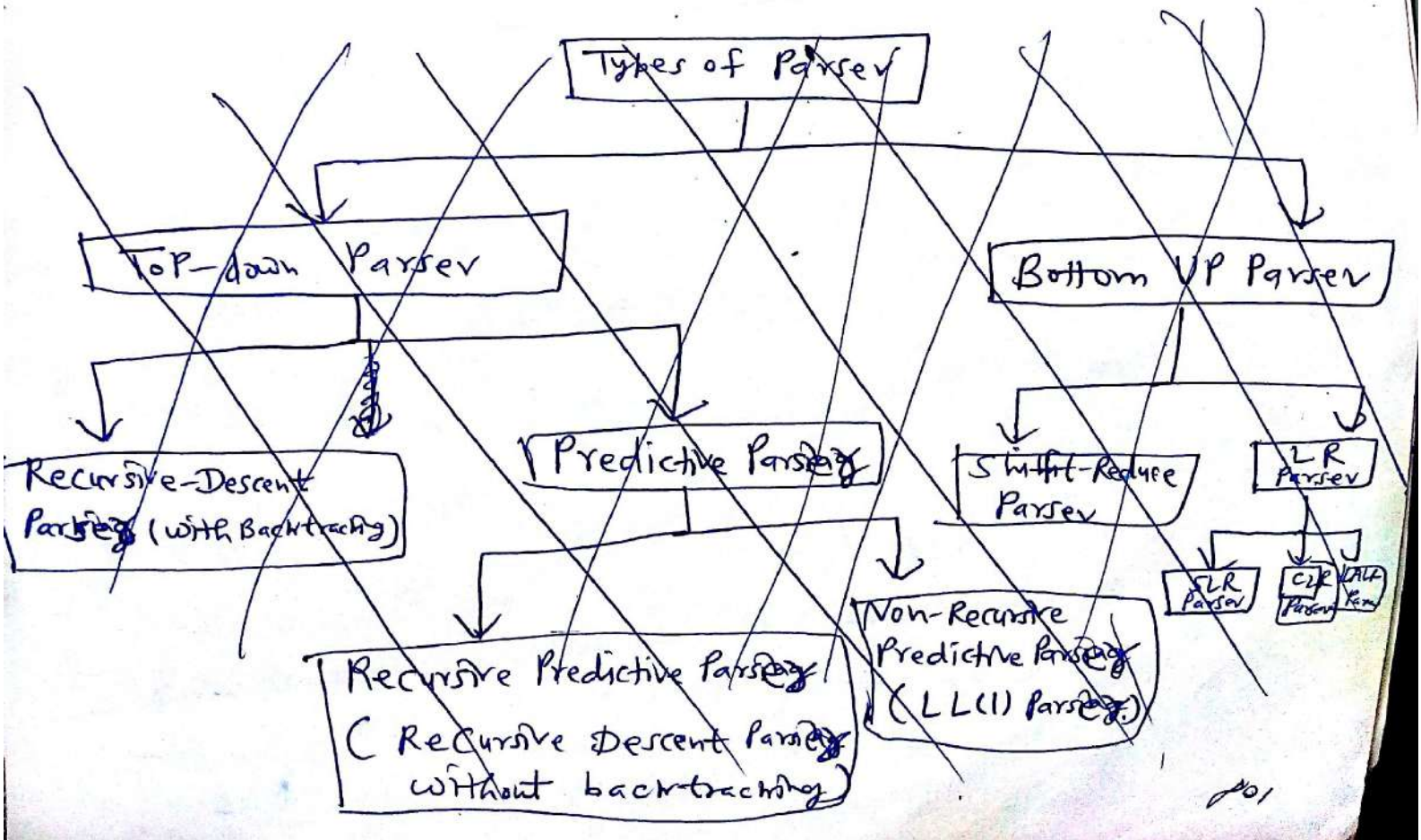
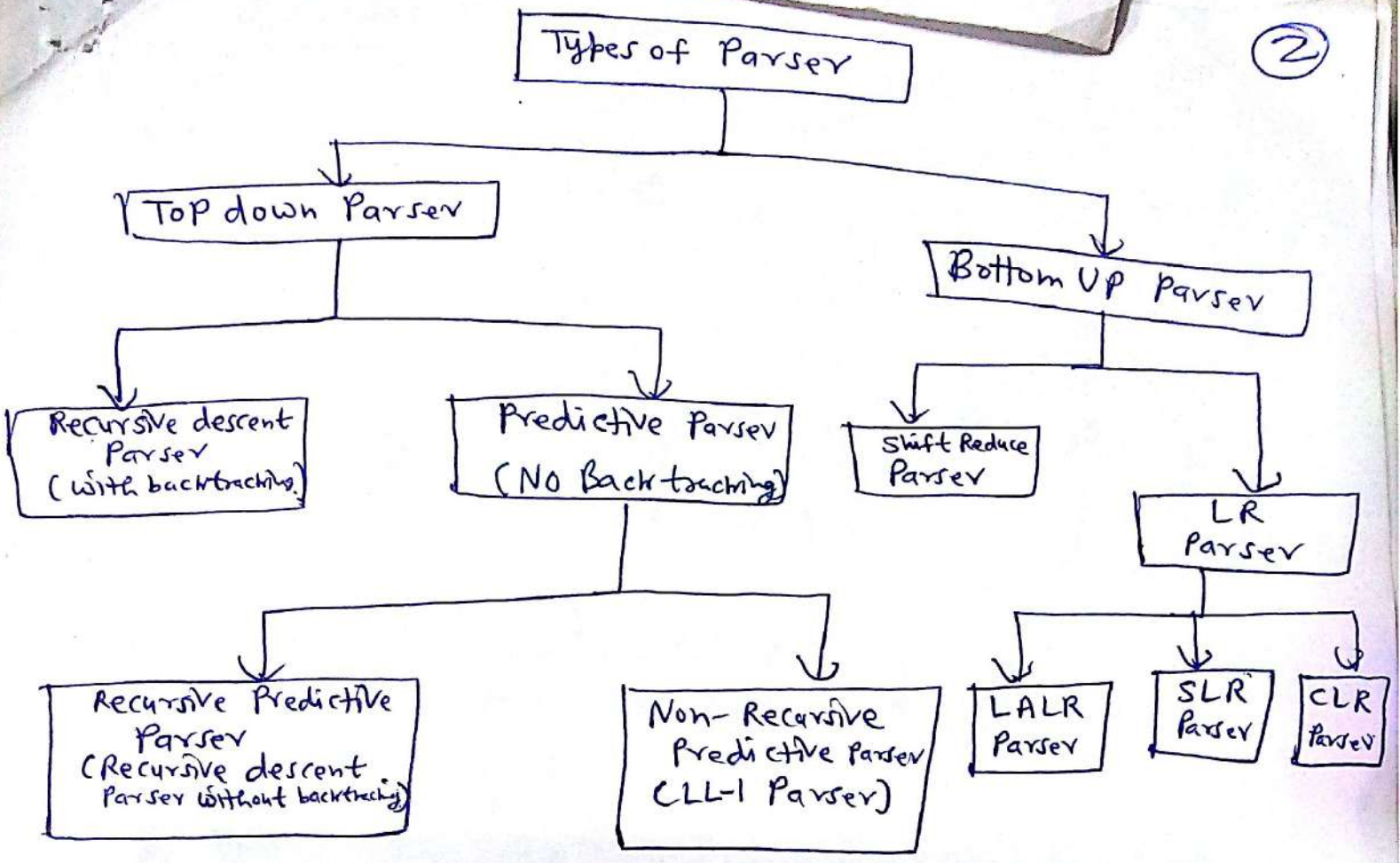
Parsing:- Parsing also known as syntax analysis can be defined as a process of analyzing a text which contains a sequence of tokens to determine its grammatical structure with respect to a given grammar.

Parser:- A parser is used to implement the parsing function. It receives a string of tokens from lexical analyzer and construct a parse tree to check whether the string of tokens can be generated by the grammar of the source language or not.

Types of Parser:-



Scanned by CamScanner



Types of Parser in Syntax Analysis

Top down Parsing:- In Top down Parsing, the parse tree is created from top to bottom using Top down Parser.

The Top down Parser is of following types:-

(1) **Recursive Descent Parser (with Backtracking)**

(a) Backtracking is needed (If a choice of a production rule does not work, we backtrack to try some other alternatives.

(b) It tries to find the left-most derivation

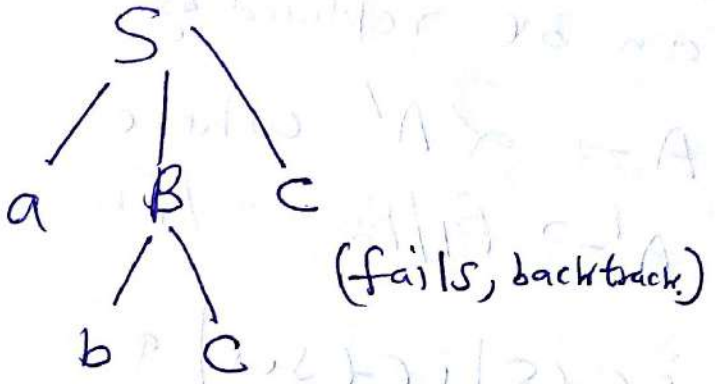
(c) It is a general parsing technique, but not widely used.

(d) Not Efficient.

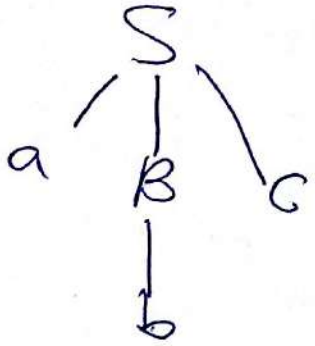
Scanned by CamScanner

Example:- $S \rightarrow aBc$
$B \rightarrow bc \mid b$

input string: abc



(fails, backtracks)



There are some problems associated with it:-

(1) **Left-Recursion Problem:-** A grammar of the form $A \rightarrow A\alpha$ is called left-Recursive, where $A \rightarrow$ Non-terminal
$\alpha \rightarrow$ any combination of terminal & Nonterminals
It can cause the recursive descent top down parser to go into infinite loop. So we need to eliminate left Recursion.

(2) **Backtracking Problem:-** If we make a sequence of erroneous expansions and subsequently discovers a mismatch, we may have to undo the semantic effect of making these erroneous expansions. for example, entries made in the symbol table might have to be removed.
In backtracking failure problem, it is very difficult to identify where the error actually occured.

Scanned by CamScanner

(3) Left factoring is also a problem associated with it. (4)

A production of the form

$A \to \alpha\beta_1 | \alpha\beta_2 -- | \alpha\beta_n$

can be replaced by

$A \to \alpha A'$ where

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

ex:- $S \to \underset{\alpha}{i\,c\,t\,s} | \underset{\alpha}{i\,c\,t\,s\,e\,s} | \underset{\beta_2}{a}$

$C \to b$

becomes

$S \to i\,c\,t\,s\,S' | a$

$C \to b$

$S' \to e\,s | \epsilon$

Scanned by CamScanner

$A \to Ac|Sd|\epsilon$ can be written as

$A \to Ac|Aad|bd|c$

$S \to Aa|b$

$A \to Ac|Aad|bd|c$

$S \to Aa|b$ does not have left recursion

$A \to Ac|Aad|bd|c$

for Rule

$A \to A\alpha_1|A\alpha_2 \cdots |\beta_1|\beta_2 \cdots$

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

$A \to \alpha_1 A'|\alpha_2 A'|\epsilon$

$A \to bdA'|cA'$

$A' \to cA'|adA'$

$S \to Aa|b$

Scanned by CamScanner

Example:- Recursive Descent Parser with backtracking!

$S \rightarrow aBc$

$B \rightarrow bc \mid b$

```
Procedure S();
Begin
    if input symbol = 'a' then
      begin
          Advance();
        if B() then
          if input symbol = 'c' then
            begin advance();
            return true end;
            else
              return false
              end

Procedure B()
Begin
    isave = input pointer
  if input symbol = 'b' then
      begin
          Advance();
          if input symbol = 'c' then
          begin advance();
          return true else return false
          end
    end
  input pointer = isave
    if input symbol = 'b' then
        begin advance();
              return true end
    else return false
end
```

Scanned by CamScanner

② Predictive Parser :- It is also of two types :-

(a) Recursive Predictive Parser (Recursive descent Parser without backtracking) :-

(b) Non-Recursive Predictive Parser (LL(1) Parser.

(a) Recursive Descent Parser without backtracking
(Implementation of Predictive Parser using recursive procedure) :-

A parser that uses a set of recursive procedures to recognize its input with no backtracking is called a recursive descent parser. Recursive descent Parser is easy to write and effecient if written in a language that implements procedure calls effectively.

for a recursive descent parser without backtracking there should be no left recursion & left factoring

ex:-

~~for S → E~~

Write the recursive descent procedure for the following grammar having no backtracking.

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

Ans

first we will eliminate left recursion

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

Scanned by CamScanner

Now write the code for it

```
procedure E();
begin
    T();
    E'()
end;
procedure E'();
if input symbol = '+' then
    begin
        Advance();
        T();
        E'()
    end;
Procedure T();
 begin
     F();
     T'()
 end;
procedure T'();
if input symbol = '*' then
 begin
     Advance();
     F();
     T'()
 end
procedure F();
if input symbol = 'id' then
    Advance()
else if input symbol = '(' then
 begin
     Advance();
     E();
     if input symbol = ')' then
         Advance()
     else Error();
end
```

Procedure Advance() moves the input pointer to the next input symbol.

Scanned by CamScanner

else Error()

fig: Recursive Descent Parser with out back tracking (Predictive Parser)

Transition Diagram for Recursive predictive Parser

To construct the transitive diagram of a predictive parser from a grammar, first eliminate left recursion from the grammar & then left factor the grammar. Then for each Nonterminal A do the following:-

(1) Create an initial & final state

(2) for each production $A \rightarrow X_1 X_2 \text{---} X_n$, create a path from the initial state to the final state with edge labled $X_1 X_2 \text{---} X_n$

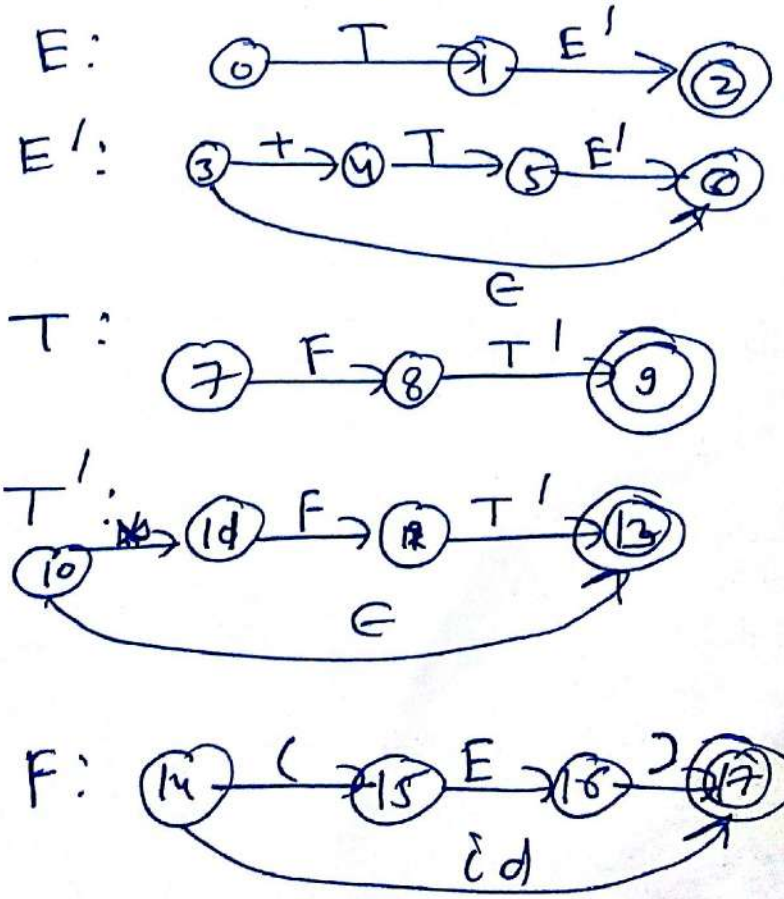
for (1) $E \rightarrow TE'$

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

$T \rightarrow FT'$

$T' \rightarrow FT' / \epsilon$

$F \rightarrow (E) / id$



Transition diagram for Grammar (1)

Scanned by CamScanner

Elimination of Left Recursion:-

If we have production of the form

$$A \to A\alpha \mid \beta$$

where β does not begin with A, then we can eliminate the left recursion by replacing this pair with

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

In General, for $A \to A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$

Can be replaced by

$$A \to \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_n A'$$
$$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A' \mid \epsilon$$

example:- Eliminate left recursion from it

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

Ans

$$E \to \underline{E}\underline{+T} / T \qquad T \to \underline{T}\underline{*F} \mid F$$
$$A \to \underline{A}\ \underline{\alpha} / \beta, \qquad A \to A\ \alpha\ \beta$$

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

(2) Eliminate left recursion from the following grammar:-

$$S \to Aa \mid b$$
$$A \to Ac \mid Sd \mid c$$

Scanned by CamScanner

Rx

Algorithm to Compute First()

First(): is a function which gives the set of terminals that begin the strings derived from the production rule.

(1) If $A \to a$, First(A) = [a]

(2) If $A \to aBC$, first(A) = [a], Take only First terminal

(3) If $A \to \epsilon$, First(A) = [ϵ]

(4) If $A \to BC$

First(A) = First(B), If First(B) $\neq \epsilon$

OR

= [First(B) − ϵ] ∪ First(C)

(5) If $A \to BCD$

First(A) = [first(B), If first(B) $\neq \epsilon$]

OR

= [[First(B) − ϵ] ∪ First(CD)]

= [[First(B) − ϵ] ∪ First(C), If first(C) $\neq \epsilon$

OR

First(A) = [[First(B) − ϵ] ∪ [first(C) − ϵ] ∪ first(D)]

Algorithm to Compute follow()

Follow(): is a function which gives the set of terminals that can appear immediately to the right of a given symbol.

(1) If $A \to \alpha B \beta$

follow(B) = first(β), If first(β) $\neq \epsilon$

OR

follow(B) = [first(β) − ϵ] follow(A)

Scanned by CamScanner

By

Date

~~Predictive Parsing Algorithm (Table)~~

Predictive Parsing Table Algorithm :

(1) Compute First and Follow for each of the Non-terminal of the grammar.

(2) For each Production $A \rightarrow \alpha$
Calculate First(α)
If first(α) does not Contain ϵ, then
for each terminal 'a' in First(α)
add Table[A, a] = $A \rightarrow \alpha$
(OR)
If first(α) Contains ϵ, then
Calculate follow(A) &
for each terminal b in follow(A),
add Table[A, b] = $A \rightarrow \alpha$

Scanned by CamScanner

① $S_1 \to S\#$ $S_1 \to$ is start symbol

$S \to QABC$

$A \to a \mid bbD$

$B \to a \mid \epsilon$

$C \to b \mid \epsilon$

$D \to c \mid \epsilon$

$first(S_1) = \{first(S) - \epsilon\} \cup first(\#)$

$= first(S)$ as $first(\#) \neq \epsilon$

OR

$first(S) = \{Q\}$

$first(A) \to \{a\}, \{b\} \Rightarrow \{a, b\}$

$first(B) = \{a, \epsilon\}$

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

$first(D) = \{c, \epsilon\}$

$S_1 \to \underline{\epsilon} S \#$

$A \to \underline{\alpha} B \underline{B}$

$follow(S) = first(B) = \{\#\}$ — (1)

$S \to \underline{Q} A B C$

$A \to \underline{\alpha} B B$

$follow(A) = \{first(BC) - \epsilon\}\ follow(S)$

$= \{first(B) - \epsilon\}\ first(C) - \epsilon\}\ follow(S)$

$follow(A) = \{a, b\}\ follow(S)$

$= \{a, b, \#\}$

$S \to \underline{Q} A \underline{B} \underline{C}$

$A \to \underline{\alpha} B \underline{B}$

$follow(B) = \{first(C) - \epsilon\}\ follow(S)$

$follow(B) = b\ follow(S)$

$= \{b, \#\}$

$S \to \underline{Q} A B C \underline{\epsilon}$

$A \to \underline{\alpha} B \underline{B}$

$follow(C) = follow(S)$

$= \{\#\}$

~~$A \to bbD$~~

$S \to \epsilon S \#$

$A \to \alpha B B$

$follow(B) = \{\#\}$

$A \to \underline{bb} \underline{D} \underline{\epsilon}$

$A \to \underline{\alpha} \underline{B} \underline{B}$

$follow(D) = follow(A)$

$= \{a, b, \#\}$

Scanned by CamScanner

(2)

$A \to Ad \mid aB$
$A \to A\alpha \mid \beta$

Date

$S \to A$
$A \to aB \mid Ad$
$B \to bBC \mid f$
$C \to g$

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

$S \to A$

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

$first(A) = \{a\}$
$first(B) = \{b\}, \{f\}$
$first(C) = \{g\}$

$follow(S) = \{\$\}$

~~$A \to \alpha B \beta$~~
~~follow(A)~~
~~follow(B)~~

$S \to \epsilon A \epsilon$
$A \to \alpha B \beta$
$follow(A) = \{first(\epsilon) - \epsilon\}$ follow(S)
$follow(A) = follow(S)$

$A \to aB\epsilon$
$A \to \alpha B \beta$
$follow(B) = \{first(\epsilon) - \epsilon\}$ follow(A)
$follow(B) = follow(A)$

$A \to \epsilon A d$
$A \to \alpha B \beta$
$follow(A) =$

$S \to iEtSS_1 \mid a$
$S_1 \to es \mid \epsilon$
$E \to b$

$S \to iEtSS_1 \mid a$
$first(S) = \{a, i\}$
$first(S_1) = \{e, \epsilon\}$
$first(E) = \{b\}$

$S \to iEtSS_1\epsilon$
$A \to \alpha B \beta$
$follow(S_1) = follow(S)$
$= \{e, \$\}$

$S \to iEtSS_1$
$A \to \alpha B \beta$
$follow(E) = \{t\}$
$follow(S) = \{\$\}$
$S \to iEtSS_1$
$\alpha B \quad B \beta$
~~follow(S) → first(S) ∈ follow~~
$follow(S) = \{first(S_1) - \epsilon\}$ follow(S)
$follow(S) = e$ follow(S)
$= \{e, \$\}$

Scanned by CamScanner

(3)

Date

$S_1 \rightarrow e\ S\ f$

$A \rightarrow \alpha\ B\ B$

follow(S) = follow(S_1) = {e, $}

(4)

$S \rightarrow a\ B\ D\ h$

$B \rightarrow c\ C$

$C \rightarrow b\ C \mid \epsilon$

$D \rightarrow E\ F$

$E \rightarrow g \mid \epsilon$

$F \rightarrow f \mid \epsilon$

first(S) = {a} ✓

first(B) = {c} ✓

first(C) = {b}, {ϵ} ✓

first(D) = {first(E) − ϵ} ∪ first(F)

= {g} first(F) = {g, f, ϵ}

first(D) = {g, f, ϵ} ✓

first(E) = {g, ϵ} ✓

first(F) = {f, ϵ} ✓

follow(S) = {$}

$S \rightarrow a\ B\ D\ h$

$A \rightarrow \alpha\ B\ \beta$

follow(B) = {first(D) − ϵ} follow(S)

= {g, f} follow(S)

follow(B) = {g, f, $}

$S \rightarrow a\ B\ D\ h$

$A \rightarrow \alpha\ B\ B$

follow(D) = {h}

$B \rightarrow c\ C$

$A \rightarrow \alpha\ B\ B$

follow(C) = {first(ϵ) − ϵ} follow(B)

follow(C) = follow(B) = {g, f, $}

Scanned by CamScanner

Date

a $ | a $
$ | $ Successful Parsing

(1) Construct the Predictive Parsing Table for the following grammar and parse the string "id+id*id".

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

Ans First check, whether it contains left recursion or left factoring. [Must condition for

Since it contains left recursion, so eliminate it

$\underline{E \to E+T} \mid T$
$A \to A\alpha \mid \beta$

So $E \to TE'$
$E' \to +TE' \mid \epsilon$ [After elimination of left Recursion]

$\underline{T \to T*F} \mid F$
$A \to A\alpha \mid \beta$

So $T \to FT'$
$T' \to *FT' \mid$ [After left Recursion elimination]

So After left Recursion elimination, CFG is

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

Now calculate First() & Follow() for each Non Terminal, $E \to TE'$

$First(E) = First(T)$
$T \to FT'$
$First(T) = First(F)$
$F \to (E), F \to id$
$First(F) = \{(, id\}$

since $First(F) \neq \epsilon$, so $\Rightarrow First(T) = First(F) = \{(, id\}$
since $First(T) \neq \epsilon$, so $First(E) = First(T) = \{(, id\}$

Scanned by CamScanner

$E' \to +TE' \mid \epsilon$ (6)

First$(E') = \{+, \epsilon\}$

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

First$(T') = \{*, \epsilon\}$

Now Calcalate follow(), follow$(E) = \{\$\}$

$$\underset{A}{E} \to \underset{\alpha}{T}\underset{B}{E'}\underset{\beta}{\epsilon}$$

follow$(E') = \{$first$(\beta) - \epsilon\}$ follow(E)

follow$(E') = \{\epsilon - \epsilon\}$ follow(E)

follow(E') = follow(E) — (1)

$$\underset{A}{E} \to \underset{\alpha}{\epsilon}\underset{B}{T}\underset{\beta}{E'}$$

follow$(T) = \{$first$(E') - \epsilon\}$ follow(E)

follow$(T) = \{+, \epsilon\}$ follow(E)

follow$(T) = \{+\}$ follow(E) — (2)

$$\underset{A}{E'} \to \underset{\alpha}{+}\underset{B}{T}\underset{\beta}{E'}$$

follow$(T) = \{$first$(E') - \epsilon\}$ follow(E')

follow$(T) = \{+\}$ follow(E') — (3)

$$\underset{A}{E'} \to \underset{\alpha}{+}\underset{B}{T}\underset{\beta}{E'}\underset{}{\epsilon}$$

follow$(E') = \{$first$(E') - \epsilon\}$ follow(E')

follow(E') = follow(E') — (4)

$$\underset{A}{T} \to \underset{\alpha}{F}\underset{B}{T'}\underset{\beta}{\epsilon}$$

follow$(T') = \{$first$(\epsilon) - \epsilon\}$ follow(T)

follow(T') = follow(T) — (5)

$$\underset{A}{T} \to \underset{\alpha}{\epsilon}\underset{B}{F}\underset{\beta}{T'}$$

follow$(F) = \{$first$(T') - \epsilon\}$ follow(T)

follow$(F) = \{*, \epsilon, \epsilon\}$ follow(T)

Scanned by CamScanner

α $ | α $
$ | $

Successful end of Parsing

(7)

follow(F) = {*} follow(T) — (6)

$T' \to *FT'$
$A \to \alpha B \beta$

follow(F) = {first(T') − ε} follow(T')

follow(F) = {*} follow(T') — (7)

$T' \to *FT' \mid \epsilon$
$A \to \alpha B \mid \beta$

follow(T') = {first(ε) − ε} follow(T')

follow(T') = follow(T') — (8)

$F \to (E)$
$A \to \alpha B \beta$

follow(E) = first())

follow(E) = {)} — (9)

since also follow(E) = {$}

So, follow(E) = {), $}

As from eqn (1)

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

from eqn (2)

follow(T) = {+} follow(E)

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

from eqn (5)

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

from eqn (6)

follow(F) = {*} follow(T) = {*, +,), $}

from eqn (3)

Scanned by CamScanner

Predictive Parsing Table

- $E \to TE'$

$A \to \alpha$

First(α) = first(TE') = first(T) = {(, id}

So Table(E, () = $E \to TE'$

Table(E, id) = $E \to TE'$

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

Predictive Parsing Table

- $E' \to +TE'$

$A \to \alpha$, First($+TE'$) = {+}

So Table[E', +] $\Rightarrow E' \to +TE'$

- $E' \to \epsilon$

$A \to \alpha$, First(ϵ) = {ϵ}

So finding follow(E') = {), $}

So Table[E',)] = $E' \to \epsilon$

Table[E', $] = $E' \to \epsilon$

- $T \to FT'$

$A \to \alpha$, first(FT') = First(F) = {(, id}

Table[T, (] = $T \to FT'$, Table[T, id] = $T \to FT'$

- $T' \to *FT'$

First($*FT'$) = {*}

Table[T', *] = $T' \to *FT'$

- $T' \to \epsilon$, first(α) = first(ϵ) = {ϵ}

So finding follow(T') = {+,), $}

Table[T', +] = Table[T',)] = Table[T', $] = $T' \to \epsilon$

- $F \to (E)$, First(α) = first(() = {(}

Table[F, (] = $F \to (E)$

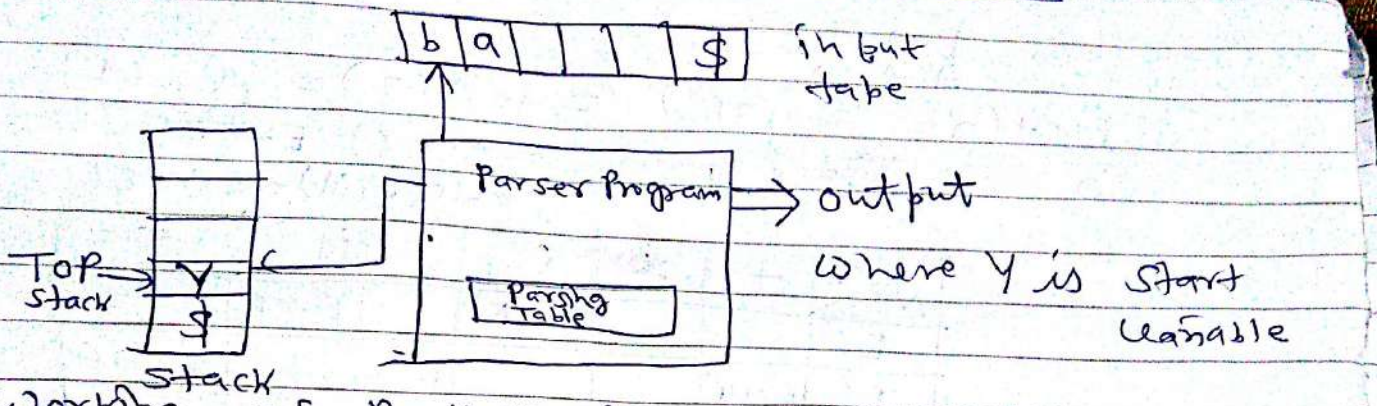
- $F \to id$, First(α) = first(id) = {id}

Table[F, id] = $F \to id$

Scanned by CamScanner

Date

Parsing Model of Predictive Parser



where Y is start variable

Working of Predictive Parser

1. If Y = b = $ (end of string), indicates the successful parsing & halts
2. If Y = b ≠ $, parser POP-off Y from stack, & advance pointer to next symbol
3. If Y is a terminal ≠ b, the parser indicates error.
4. If Y is nonterminal, then parser lookup in Table[Y, b] = Y → α, POP-off 'Y' from stack & replace it by 'α'.

Parsing of string "id+id*id"

Stack	Input string	Production Rule
E $	id+id*id $	
TE' $	id+id*id $	E → TE'
FT'E' $	id+id*id $	T → FT'
idT'E' $	id+id*id $	F → id
T'E' $	+id*id $	T' → ∈
E' $	+id*id $	
+TE' $	+id*id $	E' → +TE'
TE' $	id*id $	
FT'E' $	id*id $	T → FT'
idT'E' $	id*id $	F → id
T'E' $	*id $	T' → *FT'
*FT'E' $	*id $	
FT'E' $	id $	
idT'E' $	id $	F → id
T'E' $	$	T' → ∈
E' $	$	E' → ∈
$	$	

Successful Parsing of string.

Scanned by CamScanner

LL(1) Parser:- A predictive Parser, in which the parsing table has no multiple defined entries is said to be LL(1) Parser.

In LL(1) Parser, the First 'L' stands for 'Scanning the input from left to Right' and Second 'L' stands for 'Left Most Derivation' and '1' stands for using one input symbol at each step to make parsing decision.

Bottom-UP Parser:- Bottom-up Parser build parse tree from the bottom (leaves) to the TOP (Root).

The bottom-up parsing can be implemented by Shift Reduce Parser (SR Parser)
(SR Parser)

Shift Reduce Parser:- A shift reduce parser tries to reduce the given input string to the start symbol of the grammar by using rightmost derivation in reverse

input string $\xrightarrow{\text{Reduced to}}$ start symbol of Grammar.

At each step, a particular substring matching the right side of a production is replaced by the symbol on the left side of that production.

ex:- $S \to aABb$
$A \to aA|a$
$B \to bB|b$

$\Rightarrow a\underline{a}abb$
$\Rightarrow$
$\Rightarrow a\underline{aA}bb$
$aA\underline{bb}$
$\Rightarrow aA\underline{B}b$

Substring "aaabb"

~~aaabb~~
$\Rightarrow$ ~~aAbbb~~
$\Rightarrow$ ~~aAbBb~~
$\Rightarrow aABb$
$\Rightarrow S$

Scanned by CamScanner