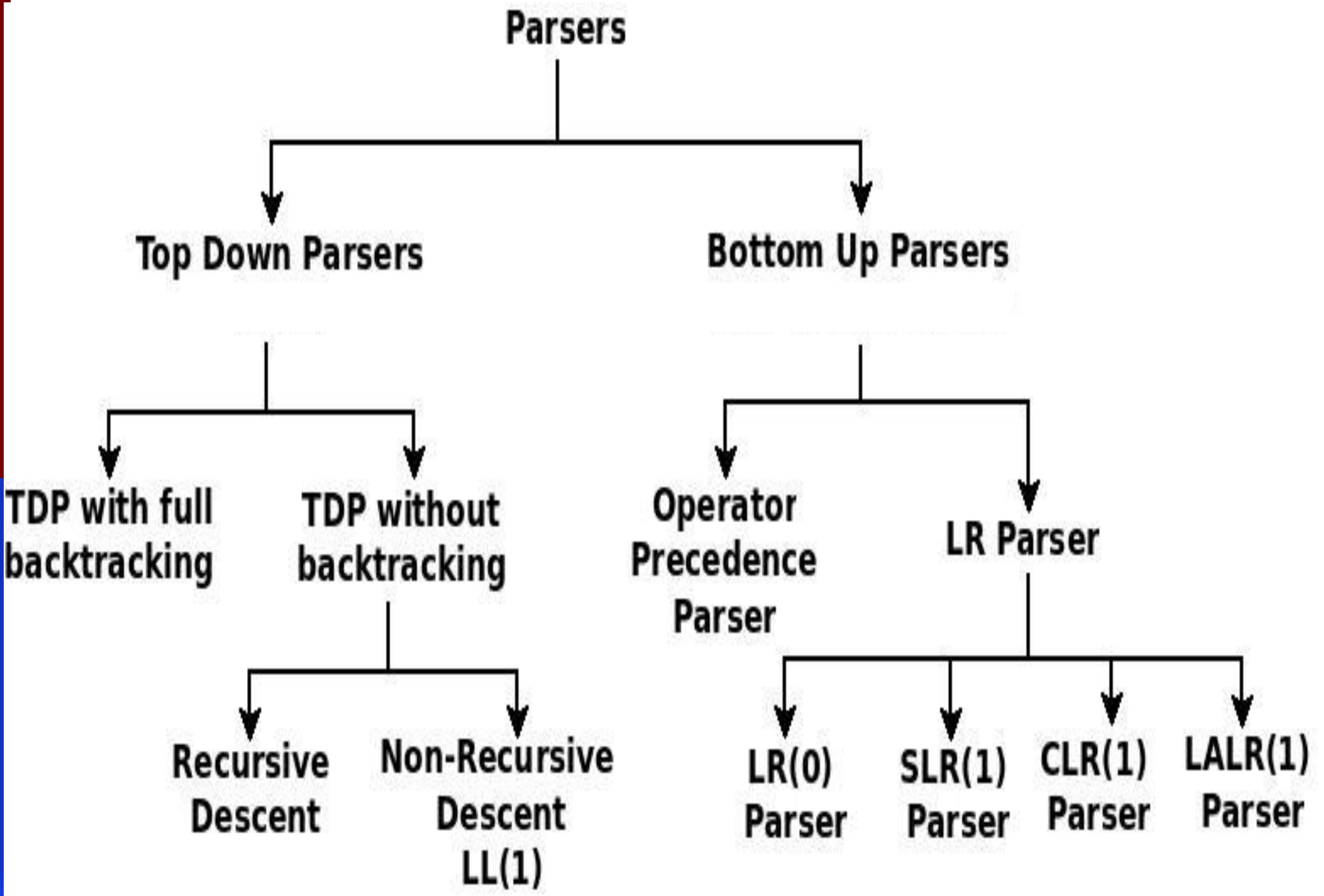
COMPILER DESIGN

UNIT 2

PRAGYA GAUR

- A Parser(syntax analyzer) is a program that takes the sequence of tokens generated by lexical analyzer as input and group them together into syntactic structure.
- Parser checks the syntactic validity of source string.


Parsers
Top Down Parsers
Bottom Up Parsers
TDP with full backtracking
TDP without backtracking
Operator Precedence Parser
LR Parser
Recursive Descent
Non-Recursive Descent LL(1)
LR(0) Parser
SLR(1) Parser
CLR(1) Parser
LALR(1) Parser

FIRST()

Rules For Calculating First Function-

Rule-01:

For a production rule $X \rightarrow \in$,
$First(X) = \{ \in \}$

Rule-02:

For any terminal symbol 'a',
$First(a) = \{ a \}$
$X \rightarrow ALPHA$

Rule-03:

For a production rule $X \rightarrow Y_1Y_2Y_3$,

Calculating First(X)

- If $\in \notin First(Y_1)$, then $First(X) = First(Y_1)$
- If $\in \in First(Y_1)$, then $First(X) = \{ First(Y_1) - \in \} \cup First(Y_2Y_3)$

Calculating First(Y_2Y_3)

- If $\in \notin First(Y_2)$, then $First(Y_2Y_3) = First(Y_2)$
- If $\in \in First(Y_2)$, then $First(Y_2Y_3) = \{ First(Y_2) - \in \} \cup First(Y_3)$

Similarly, we can make expansion for any production rule $X \rightarrow Y_1Y_2Y_3.....Y_n$.

$S \to aBDh$

$B \to cC$

$C \to bC \mid \in$

$D \to EF$

$E \to g \mid \in$

$F \to f \mid \in$

First(F) = { }

$F \rightarrow f$ Add f in FIRST(F)

$F \rightarrow \in$ Add $\in$ in FIRST(F)

First(F) = { f , ∈ }

First(E) = { }

$E \rightarrow g$ Add g in FIRST(E)

$E \rightarrow \in$ Add $\in$ in FIRST(E)

First(E) = { g , ∈ }

$S \rightarrow aBDh$

$B \rightarrow cC$

$C \rightarrow bC \, / \in$

$D \rightarrow EF$

$E \rightarrow g \, / \in$

$F \rightarrow f \, / \in$

First(D) = { }

$D \rightarrow EF$

First(D) = First(EF)

First(D) = { First(E) – ∈ } ∪ First(F)

= { g , f , ∈ }

First(C) = { }

$C \rightarrow bC$

$C \rightarrow \in$

First(C) = { b , ∈ }

$S \rightarrow aBDh$

$B \rightarrow cC$

$C \rightarrow bC \,/\, \in$

$D \rightarrow EF$

$E \rightarrow g \,/\, \in$

$F \rightarrow f \,/\, \in$

First(F) = { f , ∈ }

First(E) = { g , ∈ }

First(B) = { }

$B \rightarrow cC$

First(B)= First (cC) Add c in FIRST(B)

First(B) = {c }

First(S) = { }

$S \rightarrow aBDh$

First (S) = First (aBDh) Add a in FIRST(S)

First(S) = { a }

$S \rightarrow aBDh$

$B \rightarrow cC$

$C \rightarrow bC / \in$

$D \rightarrow EF$

$E \rightarrow g / \in$

$F \rightarrow f / \in$

$S \rightarrow aBDh$

$B \rightarrow cC$

$C \rightarrow bC \,/\, \in$

$D \rightarrow EF$

$E \rightarrow g \,/\, \in$

$F \rightarrow f \,/\, \in$

$\text{First}(F) = \{ f , \in \}$

$\text{First}(E) = \{ g , \in \}$

$\text{First}(D) = \{ \text{First}(E) - \in \} \cup \text{First}(F) = \{ g , f , \in \}$

$\text{First}(C) = \{ b , \in \}$

$\text{First}(B) = \{ c \}$

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

$$D \rightarrow ABC$$

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

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

$First(C) = \{d, \epsilon\}$

$FIRST(D) = FIRST(ABC)$

$= \{a, b\}$

$$S \rightarrow 1AB \mid \in$$

$$A \rightarrow 1AC \mid 0C$$

$$B \rightarrow 0S$$

$$C \rightarrow 1$$

$S \rightarrow 1AB \mid \in$

$A \rightarrow 1AC \mid 0C$

$B \rightarrow 0S$

$C \rightarrow 1$

FIRST (C)={ }

$C \rightarrow 1$

FIRST (C)={ 1}

FIRST (B)={ }

$B \rightarrow 0S$

FIRST (B)={ 0}

FIRST (A)={ }

$A \rightarrow 1AC$

$A \rightarrow 0C$

FIRST (A)={ 0,1}

FIRST (S)={ }

$S \rightarrow 1AB$

$S \rightarrow \in$

FIRST (S)={ 1, $\in$ }

$E \rightarrow TE'$

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

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \in$

$F \rightarrow (E)$

$F \rightarrow id$

$E \rightarrow TE'$

$E' \rightarrow + T E' \mid \in$

$T \rightarrow F T'$

$T' \rightarrow *F T' \mid \in$

$F \rightarrow (E)$

$F \rightarrow id$

FIRST (F) ={}

F → (E)

F → id

FIRST (F)= { (, id}

FIRST (T') ={}

T' → *F T'

T' → ∈

FIRST (T')= { *,∈ }

E → TE’

E’ →+ T E’ | ∈

T → F T’

T’ → *F T’ | ∈

F → (E)

F → id

FIRST (F)= { (, id}

FIRST (T’)= { *,∈ }

FIRST (T) ={}

T → F T’

FIRST (T)= FIRST(F T’)={ (, id}

FIRST (T) ={(,id}

FIRST (E’) ={}

E’ →+ T E’

E’ → ∈

FIRST (E’)= { +,∈ }

$E \rightarrow TE'$

$E' \rightarrow + T E' \mid \in$

$T \rightarrow F T'$

$T' \rightarrow *F T' \mid \in$

$F \rightarrow (E)$

$F \rightarrow id$

FIRST (F)= { (, id}

FIRST (T')= { *,∈ }

FIRST (T) ={(,id}

FIRST (E')= { +,∈ }

FIRST (E) ={}

$E \rightarrow TE'$

FIRST(E)= FIRST(TE')

={(, id}

FIRST (E) ={(,id}

$E \rightarrow TE'$

$E' \rightarrow + T E' \mid \in$

$T \rightarrow F T'$

$T' \rightarrow *F T' \mid \in$

$F \rightarrow (E)$

$F \rightarrow id$

FIRST (F)= { (, id}

FIRST (T')= { *, ∈ }

FIRST (T) ={(,id}

FIRST (E')= { +, ∈ }

FIRST (E) ={(,id}

$S| \rightarrow S\ \#$

$S \rightarrow qABC$

$A \rightarrow a \mid bbD$

$B \rightarrow a \mid \in$

$C \rightarrow b \mid \in$

$D \rightarrow c \mid \in$

$S \rightarrow iEtSS' \mid a$

$S' \rightarrow eS \mid epsilon$

$E \rightarrow b$

$$S \rightarrow (L) \mid a$$

$$L \rightarrow L , S \mid S$$

```
S -> ACB | Cbb | Ba
A -> da | BC
B -> g | ϵ
C -> h | ϵ
```

FOLLOW()

Follow(X) to be the set of terminals that can appear immediately

to the right of Non-Terminal X in some sentential form.

Example:

S ->Aa | Ac

A ->b

Here, FOLLOW (A) = {a, c}

Rules to compute FOLLOW ():

1) FOLLOW(S) = { $ }   // where S is the starting Non-Terminal

2) If A -> αBβ is a production, where α, B and β are any grammar symbols,
   then everything in FIRST(β) except Є is in FOLLOW(B).

3) If A->αB is a production, then everything in FOLLOW(A) is in FOLLOW(B).

4) If A->αBβ is a production and FIRST(β) contains Є,
   then FOLLOW(B) contains { FIRST(β) – Є } U FOLLOW(A)

Example 1:

Production Rules:

E -> TE′

E′ -> +T E′ |ϵ

T -> F T′

T′ -> *F T′ | ϵ

F -> (E) | id

FIRST (F)= { (, id}

FIRST (T’)= { *,∈ }

FIRST (T) ={(,id}

FIRST (E’)= { +,∈ }

FIRST (E) ={(,id}

FOLLOW(E) = { }

F -> (E)

A -> α B β

FOLLOW(B)= FIRST(β)

FOLLOW(E)= FIRST())= {)}

E IS START SYMBOL

SO

FOLLOW (E) = {$}

⇔ FOLLOW(E) = { $,) }

E -> TE′

E′ -> +T E′|Є

T -> F T′

T′ -> *F T′ | Є

F -> (E) | id

FIRST()

FIRST (F)= { (, id}

FIRST (T')= { *,∈ }

FIRST (T) ={(,id}

FIRST (E')= { +,∈ }

FIRST (E) ={(,id}

FOLLOW(E′) = { }

E -> T E′

A -> α B

FOLLOW(B)= FOLLOW(A)

FOLLOW(E′)= FOLLOW (E) ={$,) }

E′ -> +T E′

A -> α B

FOLLOW (E′) = FOLLOW (E′) x

⇔ FOLLOW(E′) = { $,) }

E -> TE′

E′ -> +T E′|Є

T -> F T′

T′ -> *F T′ | Є

F -> (E) | id

FIRST()

FIRST (F)= { (, id}

FIRST (T')= { *,∈ }

FIRST (T) ={(,id}

FIRST (E')= { +,∈ }

FIRST (E) ={(,id}

FOLLOW()

FOLLOW(E) = { $,) }

FOLLOW(T) = { }

E -> T E′

A -> α B β

FOLLOW(B)= FIRST(β)

FOLLOW(T)= FIRST(E′)

FIRST(E′) ={+,∈}

⇔ FIRST (E′) contains ∈

FOLLOW(B)={FIRST(β)−Є} U FOLLOW(A)

⇔ *FOLLOW(T)={FIRST(E′)−Є} U FOLLOW(E)*

⇔ FOLLOW(T)= {+,),$}

E -> TE′

E′ -> +T E′|Є

T -> F T′

T′ -> *F T′ | Є

F -> (E) | id

FIRST()

FIRST (F)= { (, id}

FIRST (T')= { *,∈ }

FIRST (T) ={(,id}

FIRST (E')= { +,∈ }

FIRST (E) ={(,id}

FOLLOW()

FOLLOW(E) = { $,) }

FOLLOW(E') = { $,) }

FOLLOW(T) = { }

E′ -> + T E′

A -> α B β

FOLLOW(B)= FIRST(β)

FOLLOW(T)= FIRST(E′)

NOTHING NEW CAN BE ADDED IN FOLLOW (T) BY THIS PRODUCTION

⇔ FOLLOW(T)= {+,),$}

E -> TE′

E′ -> +T E′|ϵ

T -> F T′

T′ -> *F T′ | ϵ

F -> (E) | id

FIRST()

FIRST (F)= { (, id}

FIRST (T')= { *,∈ }

FIRST (T) ={(,id}

FIRST (E')= { +,∈ }

FIRST (E) ={(,id}

FOLLOW()

FOLLOW(E) = { $,) }

FOLLOW(E') = { $,) }

FOLLOW(T′) = { }

T -> F T′

A -> α B

FOLLOW(B) = FOLLOW(A)

FOLLOW(T′) = FOLLOW (T) ={+,),$ }

⇔ FOLLOW(T′) = {+,),$}

T′ -> *F T′

A -> α B

FOLLOW (T′) = FOLLOW (T′) x

⇔ FOLLOW(T′) = { +,) ,$}

E -> TE′

E′ -> +T E′|ϵ

T -> F T′

T′ -> *F T′ | ϵ

F -> (E) | id

FIRST()

FIRST (F)= { (, id}

FIRST (T')= { *,∈ }

FIRST (T) ={(,id}

FIRST (E')= { +,∈ }

FIRST (E) ={(,id}

FOLLOW()

FOLLOW(E) = { $,) }

FOLLOW(E') = { $,) }

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

FOLLOW(F) = { }

T -> * F T′

A -> α B β

FOLLOW(B)= FIRST(β)

FOLLOW(F)= FIRST(T′)

FIRST(T′) ={*,∈}

⇔ FIRST (T′) contains ∈

FOLLOW(B)={FIRST(β)−Є} U FOLLOW(A)

⇔ *FOLLOW(F)={FIRST(T′)−Є} U FOLLOW(T)*

⇔ FOLLOW(F)= {*,+,),$}

E -> TE′

E′ -> +T E′|Є

T -> F T′

T′ -> *F T′ | Є

F -> (E) | id

FIRST()

FIRST (F)= { (, id}

FIRST (T')= { *,∈ }

FIRST (T) ={(,id}

FIRST (E')= { +,∈ }

FIRST (E) ={(,id}

FOLLOW()

FOLLOW(E) = { $,) }

FOLLOW(E') = { $,) }

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

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

E -> TE′

E′ -> +T E′ |ϵ

T -> F T′

T′ -> *F T′ | ϵ

F -> (E) | id

FIRST()

FIRST (F)= { (, id}

FIRST (T’)= { *,∈ }

FIRST (T) ={(,id}

FIRST (E’)= { +,∈ }

FIRST (E) ={(,id}

FOLLOW()

FOLLOW(E) = { $,) }

FOLLOW(E’) = { $,) }

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

FOLLOW(T’) = { +,) ,$}

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

EXAMPLE 1

S -> aBDh
B -> cC
C -> bC |Є
D -> EF
E -> g | Є
F -> f | Є

First(F) = { f , ∈ }

First(E) = { g , ∈ }

First(D)= { g , f , ∈ }

First(C) = { b , ∈ }

First(B) = { c }

First(S) = { a }

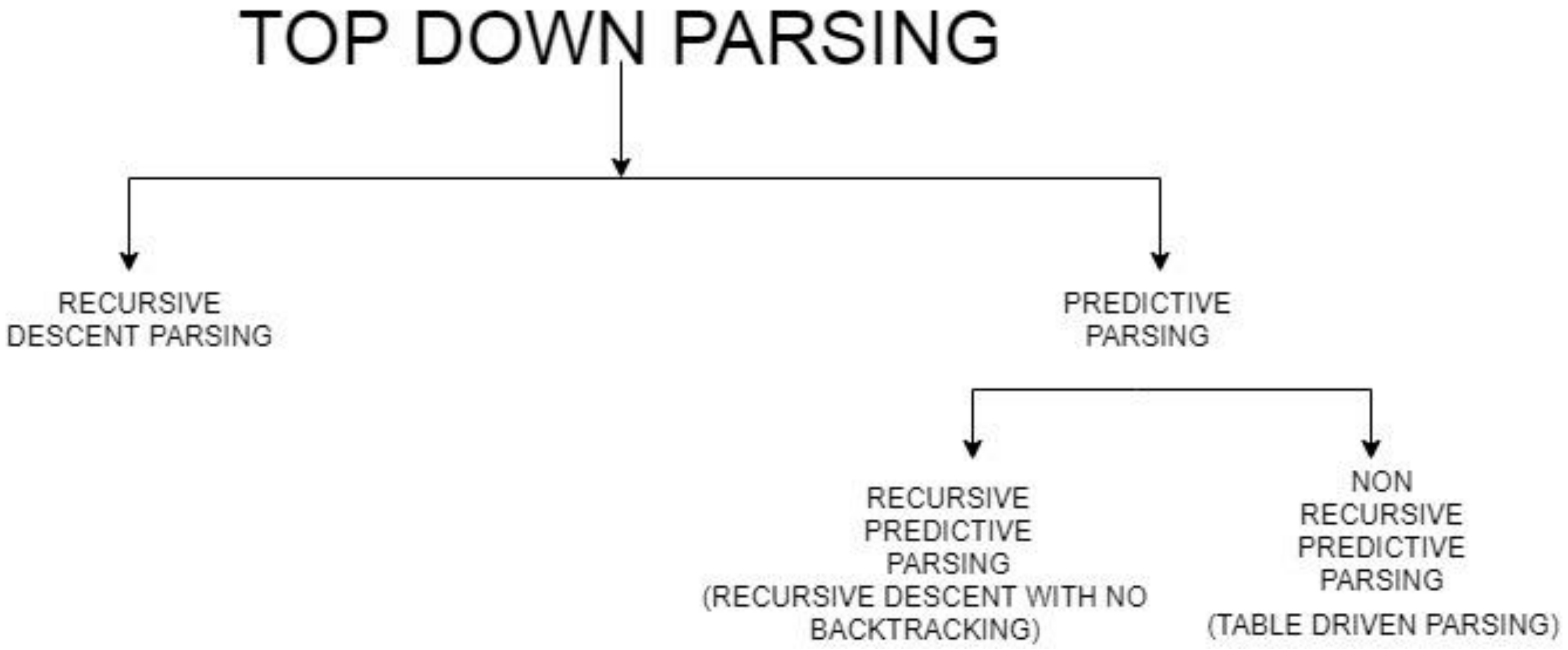
FOLLOW(S) = { $ }
FOLLOW(B) = { FIRST(D) – Є } U FIRST(h)
= { g , f , h }
FOLLOW(C) = FOLLOW(B)
= { g , f , h }
FOLLOW(D) = FIRST(h)
= { h }
FOLLOW(E) = { FIRST(F) – Є } U
FOLLOW(D)
= { f , h }
FOLLOW(F) = FOLLOW(D) = { h }

EXAMPLE 2

```
S -> ACB|Cbb|Ba
A -> da|BC
B-> g|Є
C-> h| Є
```

```
FIRST(C) = { h, Є }
FIRST(B) = { g, Є }
FIRST(A) = { d, g,h, Є }
FIRST(S) = { d, g, h, Є, b, a}

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


TOP DOWN PARSING
RECURSIVE DESCENT PARSING
PREDICTIVE PARSING
RECURSIVE PREDICTIVE PARSING (RECURSIVE DESCENT WITH NO BACKTRACKING)
NON RECURSIVE PREDICTIVE PARSING (TABLE DRIVEN PARSING)

Recursive Descent Parsing

- Recursive descent parsing is a top-down method of syntax analysis in which a set recursive procedures to process the input is executed.
- A procedure is associated with each nonterminal of a grammar.
- Top-down parsing can be viewed as an attempt to find a leftmost derivation for an input string.
- Equivalently, it attempts to construct a parse tree for the input starting from the root and creating the nodes of the parse tree in preorder.
- Recursive descent parsing involves backtracking.

Example (backtracking)

- Consider the grammar

  $S \rightarrow cAd$

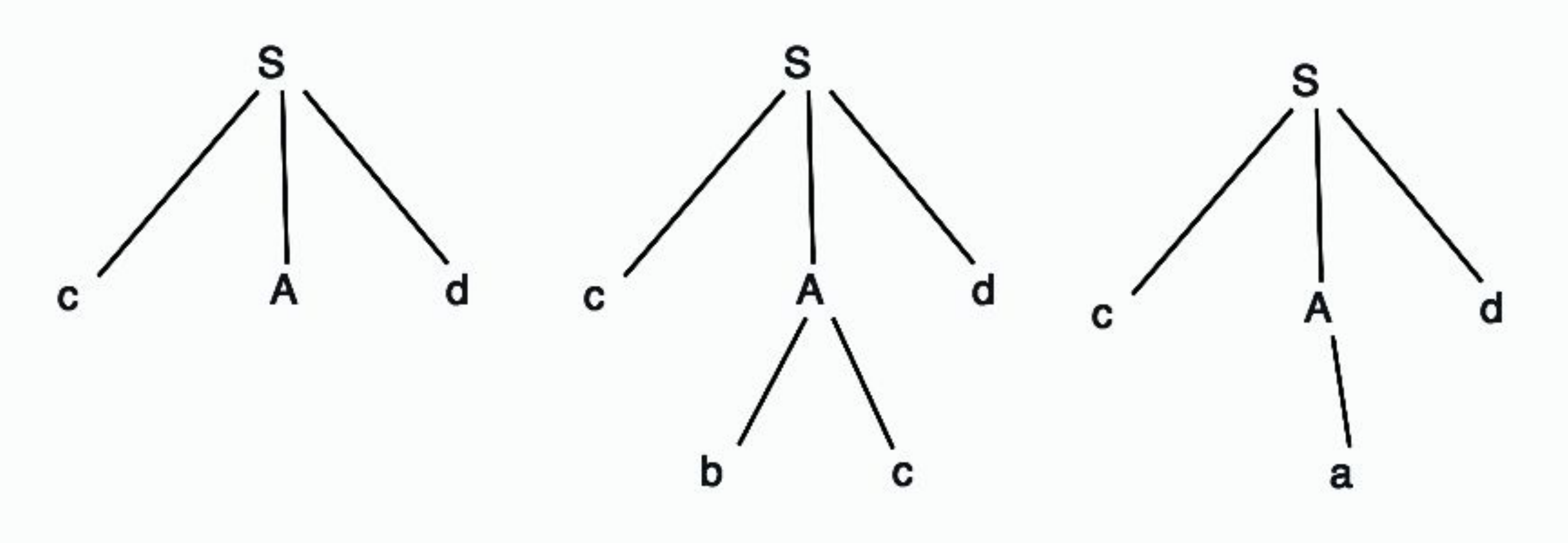
  $A \rightarrow ab \mid a$

  and the input string $w = cad$

- To construct a parse tree for this string using top-down approach, initially create a tree consisting of a single node labeled S.

S-> cAd
A -> ac |a

w= cad



Predictive Parser

A *Predictive Parser* is a special case of Recursive Descent Parser, where no Backtracking is required.

- Recursive Predictive Parser
- Non Recursive Predictive Parsing

Recursive Predictive Parser
(Recursive Descent Parser - no backtracking)

- Recursive Predictive parsing is a top-down method of syntax analysis in which a set recursive procedures to process the input is executed.
- A procedure is associated with each nonterminal of a grammar.

For terminals, create function parse_a

- If lookahead is a then parse_a consumes the lookahead by advancing to the next token and then returns
- Otherwise fails with a parse error if lookahead is not a

For each non terminal N, create a function parse_N

- Called when we're trying to parse a part of the input which corresponds to (or can be derived from) N
- parse_S for the start symbol S begins the parse

The body of parse_N for a nonterminal N does the following

- Let $N \to \beta_1 \mid \dots \mid \beta_k$ be the productions of N
    - øø Here β_i is the entire right side of a production- a sequence of terminals and nonterminals
- Pick the production $N \to \beta_i$ such that the lookahead is in First(β_i)
    - øø It must be that First(β_i) ∩ First(β_j) = ∅ for $i \neq j$
    - øø If there is no such production, but $N \to \varepsilon$ then return
    - øø Otherwise fail with a parse error
- Suppose $\beta_i = \alpha_1 \ \alpha_2 \dots \alpha_n$. Then call parse_$\alpha_1$(); ... ; parse_$\alpha_n$() to match the expected right-hand side, and return

EXAMPLE 1:

Given grammar :

$S \rightarrow xyz \mid abc$

First(xyz) = { x }

First(abc) = { a }

Parser

```
parse_S( )
{
    if (lookahead == “x”)
    {
        parse_x; parse_y; parse_z);  // S → xyz
    }
    else if (lookahead == “a”)
    {
        parse_a; parse_b; parse_c;    // S → abc

    }
    else error( );
}
```

EXAMPLE 2:

Given grammar

$S \to A \mid B$

$A \to x \mid y$

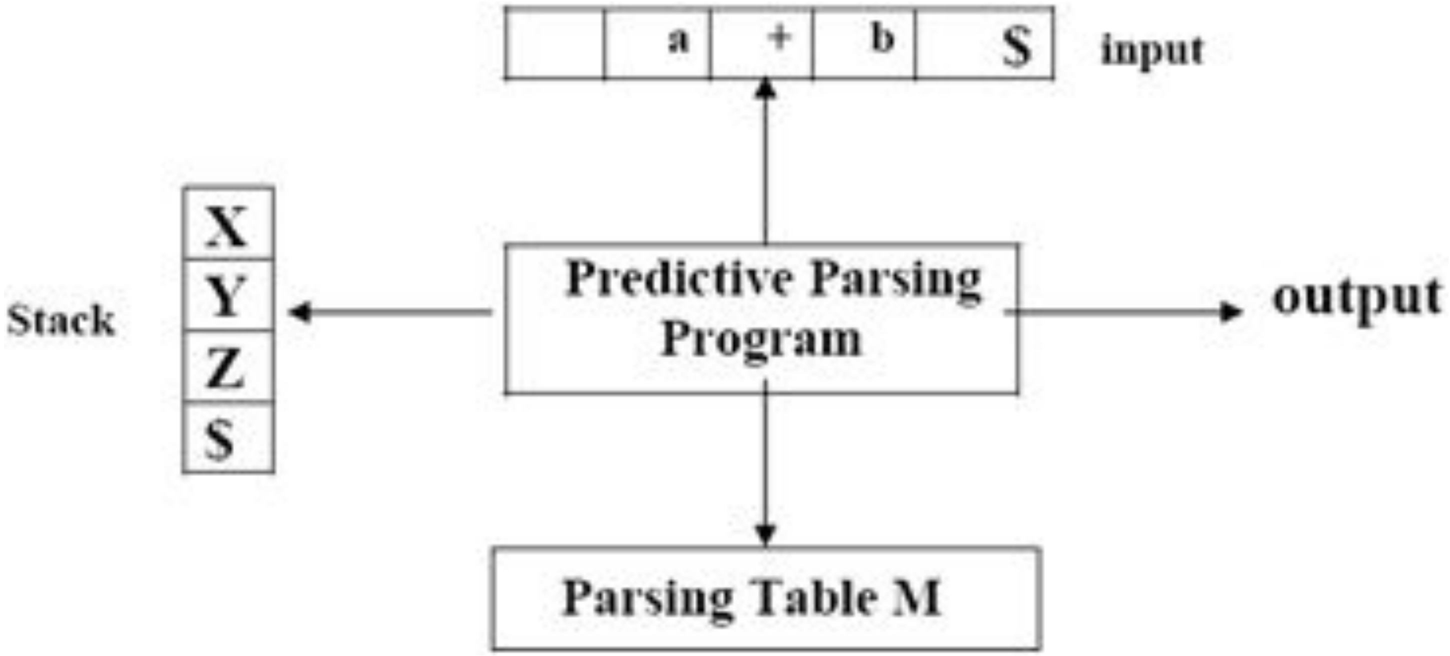
$B \to z$

First(A) = { x, y }

First(B) = { z }

```
parse_S( )
  { if ((lookahead == "x") || (lookahead == "y"))
              parse_A( );                      // S → A
    else if (lookahead == "z")
              parse_B( );                      // S → B
    else error( );
  }
parse_A( )
 { if (lookahead == "x")
              parse_x();                       // A → x
   else if (lookahead == "y")
              parse_y();                       // A → y
   else error( );
parse_B( )
 {  if (lookahead == "z")
              parse_z();                       // B → z
    else error( );
 }
```

Non Recursive Predictive Parser (Table Driven Parser)



•A table –driven predictive parser has an input buffer, stack, a parsing table, and an output stream.

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

•The stack contains a sequence of grammar symbols with $ on the bottom. Initially, the stack contains the start symbol of the grammar on top of $.

•The parsing table is a 2-dimensional array M[A, a] where A is a non-terminal and a is a terminal or $.

The parser is controlled by a program that behaves as follows:

X: top of stack

a: current input

1if x=a=$, the parser halts (successful completion)

2if x=a≠$, the parser pops x off the stack and advances the input pointer to the next input symbol.

3if x is a non-terminal, the program consults entry M[x, a] of the parsing table M.

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

Non-terminal	Input Symbol					
	id	+	*	(	)	$
E	E→TE`			E→TE`		
E`		E`→+TE`			E`→ε	E`→ε
T	T→FT`			T→FT`		
T`		T`→ε	T`→*FT`		T`→ε	T`→ε
F	F→id			F→(E)		

Parsing Table

Input: id +id * id

Stack	Input	Output
$E	id+id*id$	
$E`T	id+id*id$	E→TE`
$E`T`F	id+id*id$	T→FT`
$E`T`id	id+id*id$	F→id
$E`T`	+id*id$	
$E`	+id*id$	T`→ε
$E`T+	+id*id$	E`→+TE`
$E`T	id*id$	
$E`T`F	id*id$	T→FT`
$E`T`id	id*id$	F→id
$E`T`	*id$	
$E`T`F*	*id$	T`→*FT`
$E`T`F	id$	
$E`T`id	id$	F→id
$E`T`	$	
$E`	$	T`→ε
$	$	E`→ε

EXAMPLE 2

G :
S-> aBa
B-> bB
b-> ∈

w= abba

	a	b	$
S	S->aBa		
B	B->∈	B->bB	

Stack	I/P buffer	O/P
$ S	a b b a $	S-> aBa
$		
$		
$		
$		
$		
$		
$		
$		