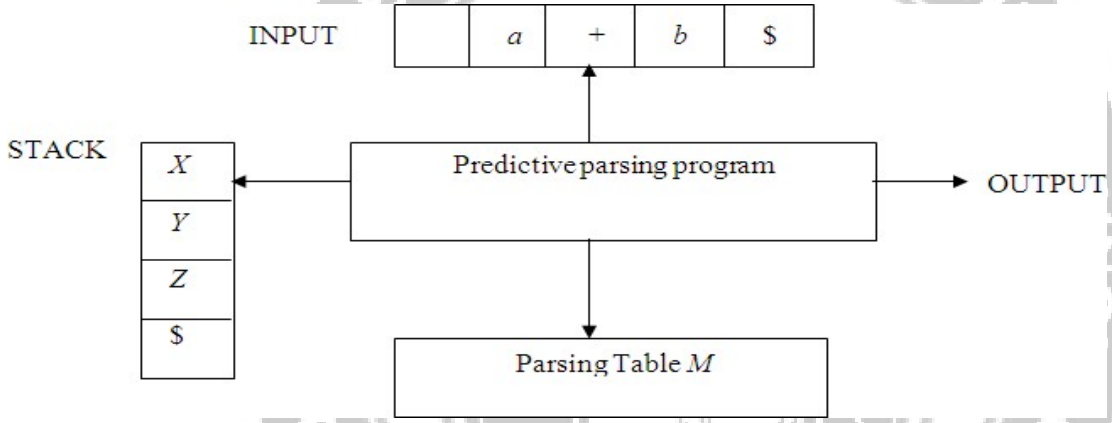
# PREDICTIVE PARSING

It is possible to build a non recursive predictive parser by maintaining a stack explicitly, rather than implicitly via recursive calls. The key problem during predictive parsing is that of determining the production to be applied for a non terminal . The non recursive parser in figure looks up the production to be applied in parsing table. In what follows, we shall see how the table can be constructed directly from certain grammars.




**Model of a nonrecursive predictive parser**

A table-driven predictive parser has an input buffer, a stack, a parsing table, and an output stream. The input buffer contains the string to be parsed, followed by $, a symbol used as a right endmarker to indicate the end of the input string. The stack contains a sequence of grammar symbols with $ on the bottom, indicating the bottom of the stack. Initially, the stack contains the start symbol of the grammar on top of $. The parsing table is a two dimensional array M[A,a] where A is a non terminal, and a is a terminal or the symbol $. The parser is controlled by a program that behaves as follows. The program considers X, the symbol on the top of the stack, and a, the current input symbol. These two symbols determine the action of the parser. There are three possibilities.

1 If X= a=$, the parser halts and announces successful completion of parsing.

2 If X=a!=$, the parser pops X off the stack and advances the input pointer to the next input symbol.

3 If X is a non terminal, the program consults entry M[X,a] of the parsing table M. This entry will be either an X-production of the grammar or an error entry. If, for example, M[X,a]={X->UVW}, the parser replaces X on top of the stack by WVU( with U on top). As output, we shall assume that the parser just prints the production used; any other code could be executed here. If M[X,a]=error, the parser calls an error recovery routine.

**Algorithm for Non recursive 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 in which it has $S on the stack with S, the start symbol of G on top, and w$ in the input buffer. The program that utilizes the predictive parsing table M to produce a parse for the input is shown in Fig.

```
set ip to point to the first symbol of w$.
repeat
   let X be the top stack symbol and a the symbol pointed to by ip. if X is
   a terminal of $ then
     if X=athen


        pop X from the stack and advance ip
     elseerror()
   else
     if M[X,a]=X->Y1Y2...Yk then begin
        pop X from the stack;
        push Yk,Yk-1...Y1 onto the stack, with Y1 on top;
        output the production X-> Y1Y2...Yk
     end
     else error()
untilX=$        /* stack is empty*/
```

**Predictive parsing table construction:**

The construction of a predictive parser is aided by two functions associated with a grammar G :

1. FIRST
2. FOLLOW

**Rules for first( ):**

1. If X is terminal, then FIRST(X) is{X}.
2. If X → ε is a production, then add ε toFIRST(X).
3. If X is non-terminal and X → aα is a production then add a toFIRST(X).
4. If X is non-terminal and X → Y1 Y2…Yk is a production, then place a in FIRST(X)if for some i, a is in FIRST(Yi), and ε is in all of FIRST(Y1),…,FIRST(Yi-1);that is, Y1,….Yi-1=> ε. If ε is in FIRST(Yj) for all j=1,2,..,k, then add ε to FIRST(X).

**Rules for follow( ):**

1. If S is a start symbol, then FOLLOW(S) contains$.
2. If there is a production A →αBβ, then every thing in FIRST(β)except ε is placed in follow(B).
3. If there is a production A → αB, or a production A → αBβ where FIRST(β) contains ε, then everything in FOLLOW(A) is in FOLLOW(B).

**Algorithm for construction of predictive parsing table:**

Input : Grammar G Output

: Parsing table M Method :

1. For each production A → α of the grammar, do steps 2 and3.
2. For each terminal a in FIRST(α), add A → α to M[A,a].
3. If ε is in FIRST(α), add A → α to M[A, b] for each terminal b in FOLLOW(A). If εis in FIRST(α) and $ is in FOLLOW(A) , add A → α to M[A,$].
4. Make each undefined entry of M beerror.

Example:

Consider the following grammar :

E→E+T|T

T→T*F|F

F→(E)|id

After eliminating left-recursion the grammar is E

→TE'

E' → +TE' | ε

T →FT'

T' → *FT' | ε

F → (E)|id

First( ) :

FIRST(E) = { ( ,id}

FIRST(E') ={+ , ε}

FIRST(T) = { ( ,id}

FIRST(T') = {*, ε}

FIRST(F) = { ( , id }

Follow( ): FOLLOW(E) = {

$, ) }

FOLLOW(E') = { $, ) }

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

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

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

**Predictive parsing Table**

| NON-TERMINAL | 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) | | |

| 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 |

**Stack Implementation**

**LL(1) grammar:**

Theparsingtableentriesaresingleentries.Soeachlocationhasnotmorethanoneentry.Thisty pe of grammar is called LL(1)grammar.

Consider this following grammar:

S→iEtS | iEtSeS| a

E→b

After eliminating left factoring, we have

S→iEtSS'|a

S'→ eS | ε

E→b

To construct a parsing table, we need FIRST() and FOLLOW() for all the non-terminals. FIRST(S) = { i, a }

FIRST(S') = {e, ε }

FIRST(E) = { b}

FOLLOW(S) = { $ ,e }

FOLLOW(S') = { $ ,e }

FOLLOW(E) = {t}

| NON-TERMINAL | a | b | e | i | t | $ |
|---|---|---|---|---|---|---|
| S | S→a | | | S→iEtSS' | | |
| S' | | | S' → eS<br>S' → ε | | | S' → ε |
| E | | E→b | | | | |

**Parsing table:**

Since there are more than one production, the grammar is not LL(1) grammar. Actions performed in predictive parsing:

1. Shift
2. Reduce
3. Accept
4. Error

**Implementation of predictive parser:**

1. Elimination of left recursion, left factoring and ambiguous grammar.
2. Construct FIRST() and FOLLOW() for all non-terminals.
3. Construct predictive parsing table.
4. Parse the given input string using stack and parsing table.