**CD: COMPILER DESIGN**
**CD: UNIT-2 21/09/2022**

**21 SEPTEMBER 2022 / IT-3rd year, Vth semester**
**FALL SEMESTER, YEAR (Vth, 3rd)**
**FALL SESSION (2022-23)**
**(CD)**
**MS. SHWETA TIWARI**
**Published: SEPTEMBER 21, 2022**

**PREPARED FOR**
**Engineering Students**
**All Engineering College**

# CD: COMPILER DESIGN

# TOPIC On : UNIT-2 PREDUCTIVE PARSER

By SHWETA TIWARI

## Under On: Basic Parsing Techniques

# TOPIC On : UNIT-2 PREDUCTIVE PARSER

## II. PERDICTIVE PARSER

**Predictive Parser:**

A grammar after eliminating left recursion and left factoring can be parsed by a recursive descent parser that needs no backtracking is a called a predictive parser. Let us understand how to eliminate left recursion and left factoring.

**Eliminating Left Recursion:**

A grammar is said to be left recursive if it has a non-terminal A such that there is a derivation A=>Aα for some string α. Top-down parsing methods cannot handle left-recursive grammars. Hence, left recursion can be eliminated as follows:

If there is a production $A \rightarrow A\alpha \mid \beta$ it can be replaced with a sequence of two productions

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \varepsilon$

Without changing the set of strings derivable from A.

**Example :** Consider the following grammar for arithmetic expressions:

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

First eliminate the left recursion for E as

$E \rightarrow TE'$

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

Then eliminate for T as

$T \rightarrow FT'$

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

Thus the obtained grammar after eliminating left recursion is

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

**Algorithm to eliminate left recursion**:

1. Arrange the non-terminals in some order $A_1, A_2 \ldots A_n$.
2. **for** i := 1 to n do begin

    **for** j := 1 to i-1 do begin

        replace each production of the form $Ai \rightarrow Aj\ \gamma$

        by the productions $A_i \rightarrow \delta_1\ \gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\ \gamma$.

        where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ are all the current Aj-productions;

    **end**

    eliminate the immediate left recursion among the $A_i$- productions

**end**

**Left factoring:**

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing. When it is not clear which of two alternative productions to use to expand a non-terminal A, we can rewrite the A-productions to defer the decision until we have seen enough of the input to make the right choice.

If there is any production $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$, it can be rewritten as

$A \rightarrow \alpha A'$

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

Consider the grammar,

$S \rightarrow iEtS \mid iEtSeS \mid a$

$E \rightarrow b$

Here,***i,t,e*** stand for **if ,the**,and **else** and **E** and **S** for “expression” and “statement”.

After Left factored, the grammar becomes

$S \rightarrow iEtSS' \mid a$

$S' \rightarrow eS \mid \varepsilon$

$E \rightarrow b$

**Non-recursive 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 Fig 2.22 looks up the production to be applied in a parsing table.

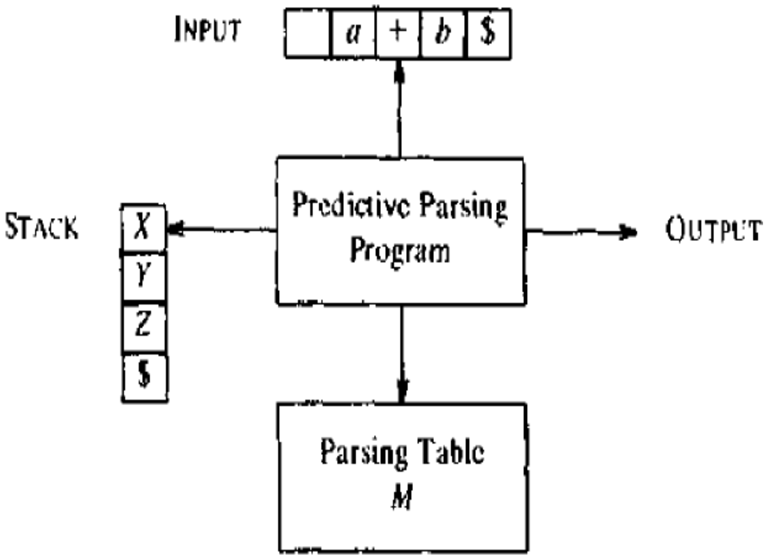


Fig 2.22 Model of a Non-recursive predictive parser

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 end marker 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 S. 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 program considers X, the symbol on 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 nonterminal, 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). If M[X, a] = error, the parser calls an error recovery routine.

**Predictive parsing table construction:**

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

G .These functions are FIRST and FOLLOW.

**Rules for FIRST():**

1. If X is terminal, then FIRST(X) is {X}.
2. If X → ε is a production, then add ε to FIRST(X).
3. If X is non-terminal and X → aα is a production then add a to FIRST(X).
4. If X is non-terminal and X → $Y_1 Y_2 \ldots Y_k$ is a production, then place a in FIRST(X) if for some i, a is in FIRST($Y_i$), and ε is in all of FIRST($Y_1$),…,FIRST($Y_{i-1}$); that is, $Y_1, \ldots Y_{i-1}$ => ε. If ε is in FIRST($Y_j$) 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 everything 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 and 3.
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 ε is in FIRST(α) and $ is in FOLLOW(A) , add A → α to M[A, $].
4. Make each undefined entry of M be error.

**Algorithm : 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 .

***Method:*** Initially, the parser is in a configuration in which it has $$ 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.

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 or $ **then**

**if** X = a **then**

pop X from the stack and advance ip

**else** error()

**else** /* X is a nomerminal */

**if** $M[X,a] = X \rightarrow Y_1Y_2 \ldots\ldots Y_k$, **then**

**begin**

pop X from the stack:

push $Y_k$ , $Y_{k-1}$ $Y_1$ , onto the stack, with Y1 on top;

output the production $X \rightarrow Y_1Y_2 \ldots\ldots Y_k$

**end**

**else** error()

**until** $X \neq \$$ /* stack is empty*/

**Example:**

Consider the following grammar:

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

After eliminating left recursion the grammar is

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

**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 for the given grammar is shown in Fig 2.23.

| M[X,a] | 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) | | |

Fig 2.23 Parsing table

With input **id+id*id** the predictive parser makes the sequence of moves shown in Fig 2,24.

| STACK | INPUT | OUTPUT |
|---|---|---|
| $E | **id+id*id$** | E →TE’ |
| $E’T | **id+id*id$** | T →FT’ |
| $E’T’F | **id+id*id$** | F →**id** |
| $E’T’**id** | **id+id*id$** | pop |
| $E’T’ | **+id*id$** | T’ →ε |
| $E’ | **+id*id$** | E’ →+TE’ |
| $E’T+ | **+id*id$** | pop |
| $E’T | **id*id$** | T →FT’ |
| $E’T’F | **id*id$** | F →**id** |
| $E’T’**id** | **id*id$** | Pop |
| $E’T’ | ***id$** | T’ →*FT’ |
| $E’T’F* | ***id$** | Pop |
| $E’T’F | **id$** | F →**id** |
| $E’T’**id** | **id$** | Pop |
| $E’T’ | **$** | T’ →ε |
| $E’ | **$** | E’ →ε |
| $ | **$** | Accept |

Fig 2.24 Moves made by predictive parser on input **id+id*id**

**LL(1) Grammars:**

For some grammars the parsing table may have some entries that are multiply-defined. For example, if G is left recursive or ambiguous , then the table will have at least one multiply-defined entry. A grammar whose parsing table has no multiply-defined entries is said to be LL(1) grammar.

**Example:** 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}

Parsing Table for the grammar:

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

Since there are more than one production for an entry in the table, the grammar is not LL(1) grammar.