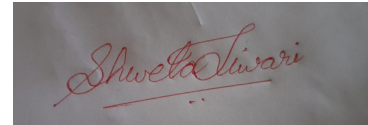


CD: COMPILER DESIGN
CD: UNIT-2 21/09/2022

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 PREDICTIVE PARSER/NON-RECURSIVE

DESCENT/NON-RECURSIVE PREDICTIVE/LL(1)

By SHWETA TIWARI

Under On: Basic Parsing Techniques

TOPIC On : UNIT-2 PREDICTIVE PARSER/NON-RECURSIVE DESCENT/NON-RECURSIVE PREDICTIVE/LL(1)

II. PREDICTIVE PARSER/NON-RECURSIVE DESCENT/NON-RECURSIVE PREDICTIVE/LL(1)

Predictive Parser:

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

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 Fig1 looks up the production to be applied in a parsing table.

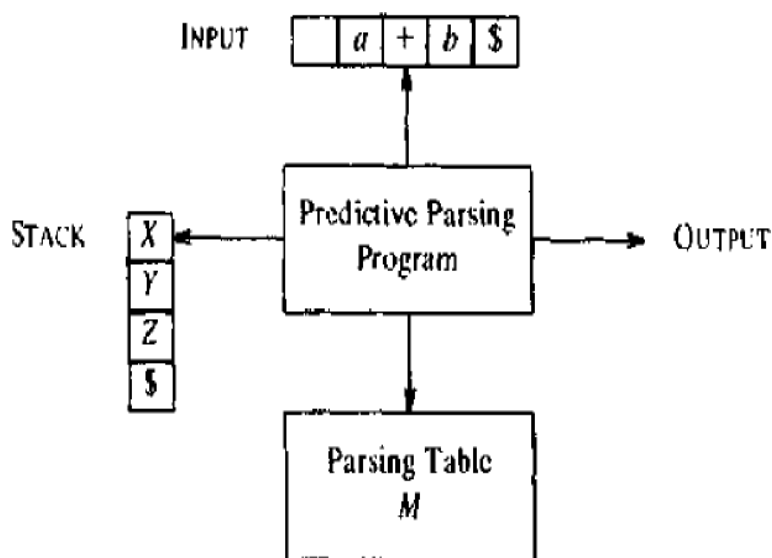


Fig 1 Model of a Non-recursive predictive parser

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 \$.

Construction of (LL1) parser:

There are five steps involved in this process:

- 1. Elimination of left recursion.*
- 2. Elimination of left factoring.*
- 3. Calculator of FIRST() And FOLLOW() of Grammar/Production.*
- 4. Construction of parsing table.*
- 5. Check whether the input string is accepted or not.*

STEP-1 Elimination of left recursion.

Eliminating Left Recursion:

A grammar is said to be left recursive if it has a non-terminal A such that there is a derivation $A \Rightarrow A\alpha$ 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

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' \mid \epsilon$$

*If there is a production $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid \beta_1 \mid \beta_2 \mid \dots$ it can be replaced with a sequence of two productions

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

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \epsilon$$

Without changing the set of strings derivable from A.

Example : Consider the following grammar for arithmetic expressions:

$$E \rightarrow E+T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid \text{id}$$

First eliminate the left recursion for E as

$$E \rightarrow TE'$$

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

Then eliminate for T as

$$T \rightarrow FT'$$

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

Thus the obtained grammar after eliminating left recursion is

$$E \rightarrow TE'$$

$$E' \rightarrow +TE' \mid \epsilon \quad T \rightarrow FT'$$

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

$F \rightarrow (E) \mid \text{id}$

Algorithm to eliminate left recursion:

1. Arrange the non-terminals in some order $A_1, A_2 \dots A_n$.
2. **for** $i := 1$ to n **do begin**
 for $j := 1$ to $i-1$ **do begin**
 replace each production of the form $A_i \rightarrow A_j \gamma$
 by the productions $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \dots \mid \delta_k \gamma$.
 where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$ are all the current A_j -productions;
 end
 eliminate the immediate left recursion among the A_i - productions
end

STEP-2 Elimination of left Factoring.

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 $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \dots \mid \gamma_1\gamma_2 \dots$, it can be rewritten as

$A \rightarrow \alpha A' \mid \gamma_1 \mid \gamma_2$

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

Consider the grammar,

$S \rightarrow \text{if} E \text{ then } S \mid \text{if} E \text{ then } S \text{ else } a$

$E \rightarrow b$

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

After Left factored, the grammar becomes

$S \rightarrow \text{if} E \text{ then } S S' \mid a$

$S' \rightarrow e S \mid \epsilon$

$E \rightarrow b$

STEP-3 Calculator of FIRST() And FOLLOW() of Grammar/Production.

These functions are FIRST() and FOLLOW().

Rules for FIRST():

1. If **X is terminal**, then **FIRST(X)** is **{X}**.
2. If **$X \rightarrow \epsilon$** is a production, then **add ϵ to FIRST(X)**.
3. If **X is non-terminal and $X \rightarrow a\alpha$** is a production then **add a to FIRST(X)**.
4. If X is non-terminal and **$X \rightarrow Y_1 Y_2 \dots Y_k$** 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, $Y_1, \dots, Y_{i-1} \Rightarrow \epsilon$. If ϵ is in FIRST(Y_j) for all $j=1,2,\dots,k$, 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 **$A \rightarrow \alpha B \beta$** , then everything in **FIRST(β) except ϵ** is placed in **FOLLOW(B)**.
3. If there is a production **$A \rightarrow \alpha B$** , or a production **$A \rightarrow \alpha B \beta$** where **FIRST(β) contains ϵ** , then everything in **FOLLOW(A)** is in **FOLLOW(B)**.

STEP-4 Construction of Parsing Table.

Predictive parsing table construction:

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

Algorithm for construction of predictive parsing table:

Input : Grammar G

Output : Parsing table M

Method :

1. For each production **$A \rightarrow \alpha$** of the grammar, do steps 2 and 3.
2. For each terminal a in FIRST(α), add **$A \rightarrow \alpha$** to **$M[A, a]$** .
3. If ϵ is in FIRST(α), add **$A \rightarrow \alpha$** to **$M[A, b]$** for each terminal b in FOLLOW(A). If ϵ is in FIRST(α) and \$ is in FOLLOW(A), add **$A \rightarrow \alpha$** to **$M[A, \$]$** .
4. Make each undefined entry of M be an error.

Construction of predictive parsing table(Rules in simple form)

Additional for understanding rules

For each production $A \rightarrow \alpha$

1. Find **FIRST(α)** and for each terminal in **FIRST(α)**. Make an entry **$[A \rightarrow \alpha]$** in the table.
2. If **FIRST(α) contains ϵ as terminal**, then **FOLLOW(A)** for each terminal in **FOLLOW(A)**. Make an entry **$[A \rightarrow \alpha]$** in the table.
3. If **FIRST(α) contains ϵ** and **FOLLOW(A) contains \$** terminal, then make an entry **$[A \rightarrow \alpha]$** in the table **for \$**.
4. Make each undefined entry of M be an error.

STEP-5 Check whether the input string is accepted or not.

Predictive Parser Model (Algorithm)

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 $X = a \neq \$$, 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, $M[X, a] = \{X \rightarrow UVW\}$, the parser replaces X on top of the stack by WVU (with U on top). If $M[X, a] = \text{error}$, the parser calls an error recovery routine.

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 nonterminal */

if $M[X, a] = X \rightarrow Y_1 Y_2 \dots Y_k$, **then**

begin

 pop X from the stack:

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

 output the production $X \rightarrow Y_1 Y_2 \dots Y_k$

end

else error()

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

Example:

Consider the following grammar:

$$E \rightarrow E+T \mid T$$

$$T \rightarrow T*F \mid F$$

$$F \rightarrow (E) \mid \text{id}$$

There are five steps involved in this process:

- 1. Elimination of left recursion.*
- 2. Elimination of left factoring.*
- 3. Calculator of FIRST() FOLLOW of Grammar/Production.*
- 4. Construction of parsing table.*
- 5. Check whether the input string is accepted or not.*

STEP-1 Elimination of left recursion.

- After eliminating left recursion the grammar is $E \rightarrow TE'$

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

$$T \rightarrow FT'$$

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

$$F \rightarrow (E) \mid \text{id}$$

STEP-2 Elimination of left factoring.

- There is no left factoring in above grammar.

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

$$F \rightarrow (E) \mid \text{id}$$

STEP-3 Calculator of FIRST() FOLLOW of Grammar/Production.

FIRST() :

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

$$F \rightarrow (E) \mid \text{id}$$

$$\text{FIRST}(E) = \{ (, \text{id} \}$$

$$\text{FIRST}(E') = \{ +, \varepsilon \}$$

$$\text{FIRST}(T) = \{ (, \text{id} \}$$

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

$\text{FIRST}(F) = \{ (, \text{id} \}$

FOLLOW():

$\text{FOLLOW}(E) = \{ \$,) \}$

$\text{FOLLOW}(E') = \{ \$,) \}$

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

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

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

STEP-4 Construction of parsing table.

- Construction the parsing table through “Construction of parsing table Algorithm(Rules)”.
Predictive parsing table for the given grammar is shown in Fig 2.

M[X,a]	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow \text{id}$			$F \rightarrow (E)$		

Fig 2 Parsing table

STEP-5 Check whether the input string is accepted or not.

STRING $w = \text{id} + \text{id} * \text{id}$

- Add \$ in the end of the given string w which will be taken as INPUT BUFFER and \$ must be STACK in bottom.
- Follow the Predictive Parser Model (Algorithm/Rules).
- With input $\text{id} + \text{id} * \text{id}$ the predictive parser makes the sequence of moves shown in Fig 3.

STACK	INPUT	OUTPUT
\$E	id+id*id\$	$E \rightarrow TE'$
\$E'T	id+id*id\$	$T \rightarrow FT'$
\$E'T'F	id+id*id\$	$F \rightarrow id$
\$E'T'id	id+id*id\$	pop
\$E'T'	+id*id\$	$T' \rightarrow \epsilon$
\$E'	+id*id\$	$E' \rightarrow +TE'$
\$E'T+	+id*id\$	pop
\$E'T	id*id\$	$T \rightarrow FT'$
\$E'T'F	id*id\$	$F \rightarrow id$
\$E'T'id	id*id\$	Pop
\$E'T'	*id\$	$T' \rightarrow *FT'$
\$E'T'F*	*id\$	Pop
\$E'T'F	id\$	$F \rightarrow id$
\$E'T'id	id\$	Pop
\$E'T'	\$	$T' \rightarrow \epsilon$
\$E'	\$	$E' \rightarrow \epsilon$
\$	\$	Accept

Fig 3 Moves made by predictive parser on input **id+id*id**

Syntax Tree of string (w=id+id*id) by using this above table.

*Final syntax tree of (w=id+id*id) string.*

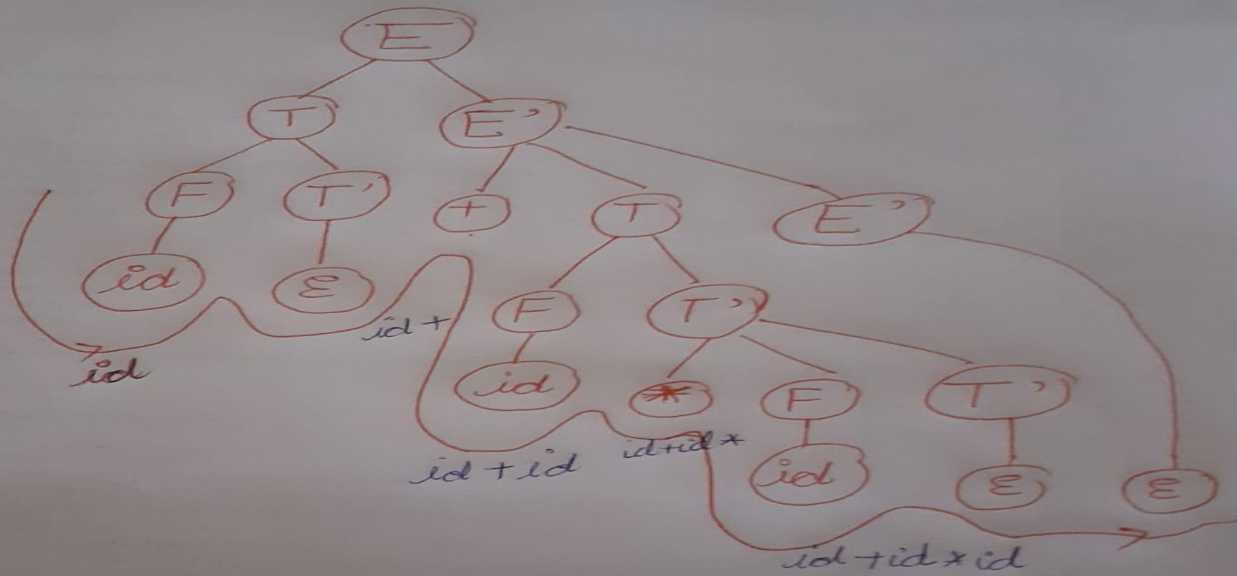


Fig 4 Syntax Tree

Question-1. : Construct parsing table of given grammar and check whether a string $w=id+id$ is accepted or not. if string is accepted, then draw a Syntax tree of string $w=id+id$. Grammar is:

$$E \rightarrow E+T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid id$$

Other Example

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:

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

$$E \rightarrow b$$

After eliminating left factoring, we have

$$S \rightarrow iEtSS' \mid a \quad S' \rightarrow eS \mid \varepsilon$$

$$E \rightarrow b$$

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

$$\text{FIRST}(S) = \{ i, a \}$$

$$\text{FIRST}(S') = \{ e, \varepsilon \}$$

$$\text{FIRST}(E) = \{ b \}$$

$$\text{FOLLOW}(S) = \{ \$, e \}$$

$$\text{FOLLOW}(S') = \{ \$, e \}$$

$$\text{FOLLOW}(E) = \{ t \}$$

Parsing Table for the grammar:

NON- TERMINAL	a	b	e	i	t	\$
S	$S \rightarrow a$			$S \rightarrow iEtSS'$		
S'			$S' \rightarrow eS$ $S' \rightarrow \varepsilon$			$S' \rightarrow \varepsilon$
E		$E \rightarrow b$				

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