

SYNTAX ANALYSIS (PARSING) :Top Down Parsing

Ref: Aho, Sethi & Ullman, "Compilers: Principles, Techniques and Tools", Pearson Education

4.1 ROLE OF THE PARSER (SYNTAX ANALYSER) :

Parser for any grammar is program that takes as input string w (obtain set of strings tokens from the lexical analyser) and produces as output either a parse tree for w , if w is a valid sentences of grammar or error message indicating that w is not a valid sentences of given grammar. The goal of the parser is to determine the syntactic validity of a source string is valid, a tree is built for use by the subsequent phases of the computer. The tree reflects the sequence of derivations or reduction used during the parser. Hence, it is called parse tree. If string is invalid, the parse has to issue diagnostic message identifying the nature and cause of the errors in string. Every elementary subtree in the parse tree corresponds to a production of the grammar.

There are two ways of identifying an elementary subtree:

1. By deriving a string from a non-terminal or
2. By reducing a string of symbol to a non-terminal.

The two types of parsers employed are:

- a. Top down parser: which build parse trees from top(root) to bottom(leaves)
- b. Bottom up parser: which build parse trees from leaves and work up the root.

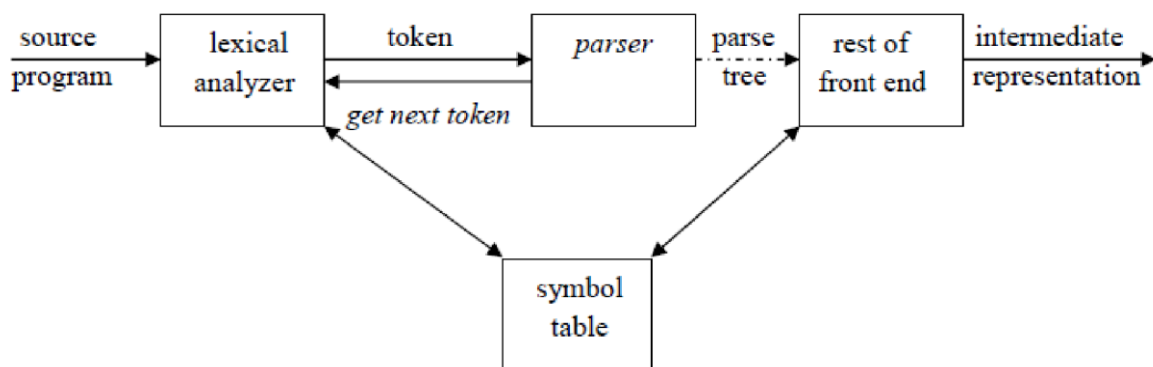


Fig. 4.1: position of parser in compiler model.

4.2 CONTEXT FREE GRAMMARS

Inherently recursive structures of a programming language are defined by a context-free Grammar. In a context-free grammar, we have four triples $G(V,T,P,S)$.

Here, V is finite set of terminals (in our case, this will be the set of tokens)

T is a finite set of non-terminals (syntactic-variables)

P is a finite set of productions rules in the following form

$A \rightarrow \alpha$ where A is a non-terminal and α is a string of terminals and non-terminals (including the empty string)

S is a start symbol (one of the non-terminal symbol)

$L(G)$ is the language of G (the language generated by G) which is a set of sentences.

A sentence of $L(G)$ is a string of terminal symbols of G . If S is the start symbol of G then ω is a sentence of $L(G)$ iff $S \Rightarrow \omega$ where ω is a string of terminals of G . If G is a context free grammar, $L(G)$ is a context-free language. Two grammar G_1 and G_2 are equivalent, if they produce same grammar.

Consider the production of the form $S \Rightarrow \alpha$, If α contains non-terminals, it is called as a sentential form of G . If α does not contain non-terminals, it is called as a sentence of G .

4.2.1 Derivations

In general a derivation step is $\alpha A \beta \Rightarrow \alpha \gamma \beta$ is sentential form and if there is a production rule $A \rightarrow \gamma$ in our grammar. where α and β are arbitrary strings of terminal and non-terminal symbols $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow \alpha_n$ (α_n derives from α_1 or α_1 derives α_n). There are two types of derivation

- 1 At each derivation step, we can choose any of the non-terminal in the sentential form of G for the replacement.
- 2 If we always choose the left-most non-terminal in each derivation step, this derivation is called as left-most derivation.

Example:

$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid - E$

$E \rightarrow (E) E$

$\rightarrow id$

Leftmost derivation :

$E \rightarrow E + E$

$\rightarrow E * E + E \rightarrow id * E + E \rightarrow id * id + E \rightarrow id * id + id$

The string is derive from the grammar $w = id * id + id$, which is consists of all terminal symbols

Rightmost derivation

$E \rightarrow E + E$

$\rightarrow E + E * E \rightarrow E + E * id \rightarrow E + id * id \rightarrow id + id * id$

Given grammar $G : E \rightarrow E + E \mid E * E \mid (E) \mid - E \mid id$

Sentence to be derived : $-(id + id)$

LEFTMOST DERIVATION

$E \rightarrow - E$

$E \rightarrow - (E)$

$E \rightarrow - (E + E)$

$E \rightarrow - (id + E)$

$E \rightarrow - (id + id)$

RIGHTMOST DERIVATION

$E \rightarrow - E$

$E \rightarrow - (E)$

$E \rightarrow - (E + E)$

$E \rightarrow - (E + id)$

$E \rightarrow - (id + id)$

- String that appear in leftmost derivation are called left sentinel forms.
- String that appear in rightmost derivation are called right sentinel forms.

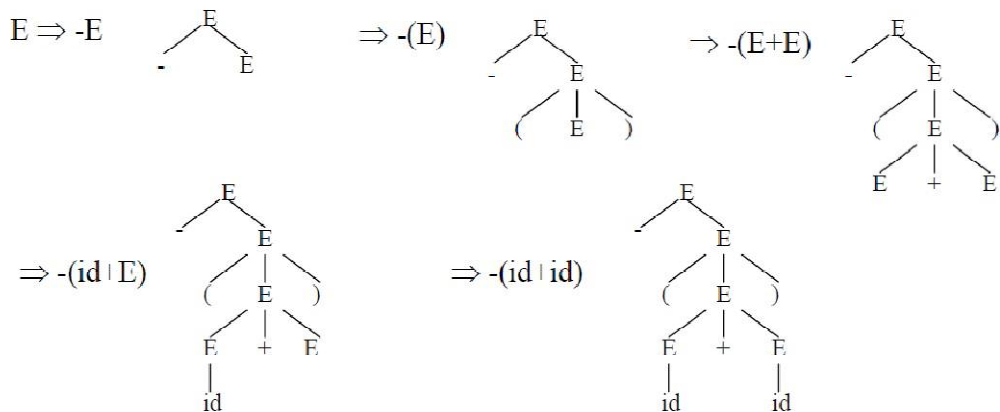
Sentinels:

- Given a grammar G with start symbol S , if $S \rightarrow \alpha$, where α may contain non-terminals or terminals, then α is called the sentinel form of G . Yield or frontier of tree:
 - Each interior node of a parse tree is a non-terminal. The children of node can be a terminal or non-terminal of the sentinel forms that are read from left to right. The sentinel form in the parse tree is called yield or frontier of the tree.

4.2.2 PARSE TREE

- Inner nodes of a parse tree are non-terminal symbols.
- The leaves of a parse tree are terminal symbols.
- A parse tree can be seen as a graphical representation of a derivation.

Example:



Ambiguity:

A grammar that produces more than one parse for some sentence is said to be ambiguous grammar.

Example: Given grammar $G : E \rightarrow E+E \mid E^*E \mid (E) \mid -E \mid id$ The sentence $id+id*id$ has the following two distinct leftmost derivations:

$E \rightarrow E + E$	$E \rightarrow E^* E$
$E \rightarrow id + E$	$E \rightarrow E + E^* E$
$E \rightarrow id + E^* E$	$E \rightarrow id + E^* E$
$E \rightarrow id + id^* E$	$E \rightarrow id + id^* E$
$E \rightarrow id + id^* id$	$E \rightarrow id + id^* id$

The two corresponding parse trees are:



Example:

To disambiguate the grammar $E \rightarrow E+E \mid E^*E \mid E^{\wedge}E \mid id \mid (E)$, we can use precedence of operators as follows:

^ (right to left)
/,* (left to right) -

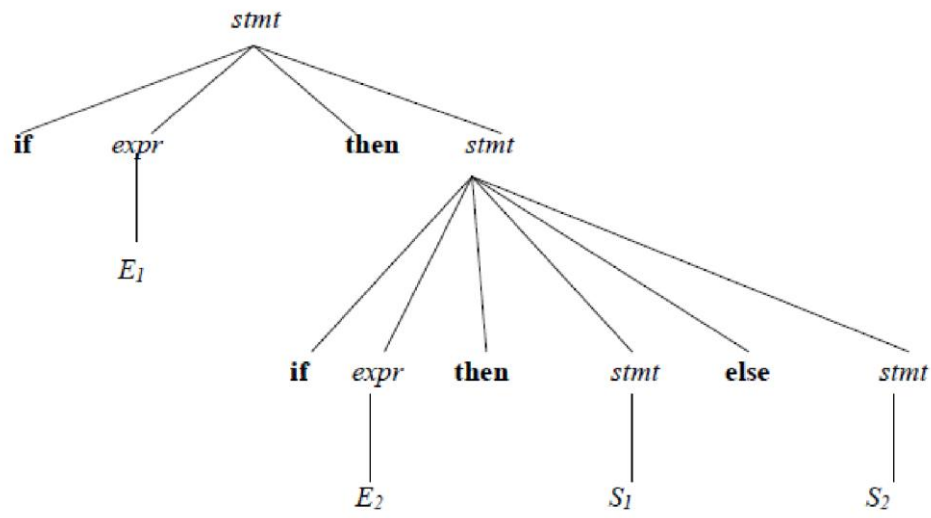
,+ (left to right) We get the following unambiguous grammar:

$E \rightarrow E+T \mid T$
 $T \rightarrow T^*F \mid F$
 $F \rightarrow G^{\wedge}F \mid G$
 $G \rightarrow id \mid (E)$

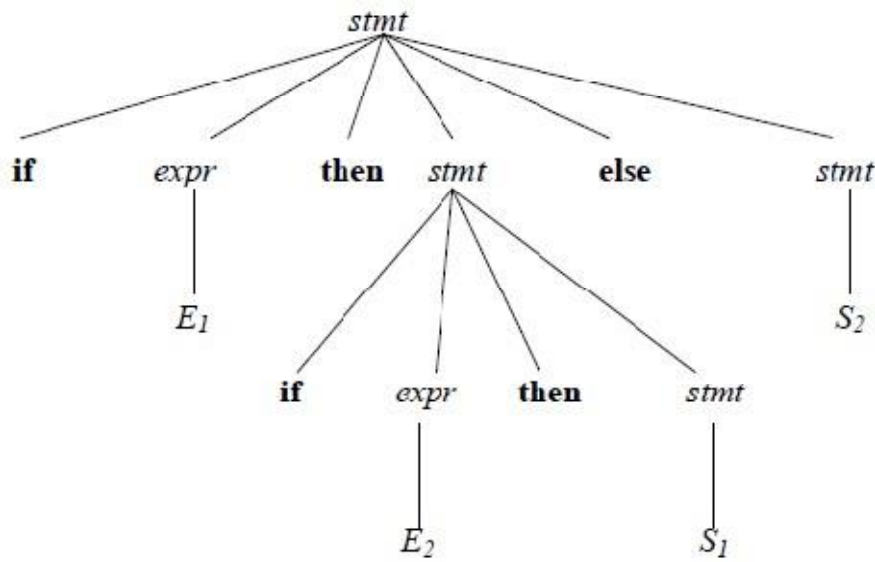
Consider this example, $G: stmt \rightarrow if\ expr\ then\ stmt \mid if\ expr\ then\ stmt\ else\ stmt \mid other$
 This grammar is ambiguous since the string $if\ E1\ then\ if\ E2\ then\ S1\ else\ S2$ has the following

Two parse trees for leftmost derivation:

1.



2.



To eliminate ambiguity, the following grammar may be used:

$stmt \rightarrow matched_stmt \mid unmatched_stmt$

$matched_stmt \rightarrow if\ expr\ then\ matched_stmt\ else\ matched_stmt \mid other$

$unmatched_stmt \rightarrow if\ expr\ then\ stmt \mid if\ expr\ then\ matched_stmt\ else\ unmatched_stmt$

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

$$A \rightarrow \beta A'$$

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

Without changing the set of strings derivable from A.

Example: Consider the following grammar for arithmetic expressions: E

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

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

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

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

$$E \rightarrow TE'$$

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

$$\rightarrow FT'$$

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

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

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

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

$$\rightarrow \alpha A'$$

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

Consider the grammar, $G : S \rightarrow iEtS \mid iEtSeS \mid a$

$$E \rightarrow b$$

Left factored, this grammar becomes

$$S \rightarrow iEtSS' \mid a$$

$$S' \rightarrow eS \mid \epsilon$$

$$E \rightarrow b$$

TOP-DOWN PARSING

It can be viewed as an attempt to find a left-most derivation for an input string or an attempt to construct a parse tree for the input starting from the root to the leaves.

Types of top-down parsing:

1. Recursive descent parsing
2. Predictive parsing

1. RECURSIVE DESCENT PARSING

Recursive descent parsing is one of the top-down parsing techniques that uses a set of recursive procedures to scan its input.

This parsing method may involve backtracking, that is, making repeated scans of the input.

Example for backtracking:

Consider the grammar $G : S \rightarrow cAd$

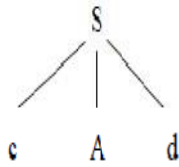
$$A \rightarrow ab \mid a$$

and the input string $w=cad$.

The parse tree can be constructed using the following top-down approach:

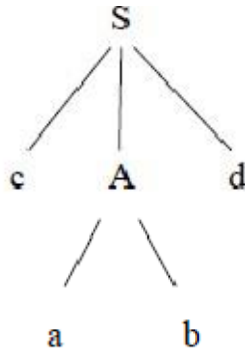
Step1:

Initially create a tree with single node labeled S. An input pointer points to 'c', the first symbol of w. Expand the tree with the production of S.



Step2:

The leftmost leaf 'c' matches the first symbol of w, so advance the input pointer to the second symbol of w 'a' and consider the next leaf 'A'. Expand A using the first alternative.



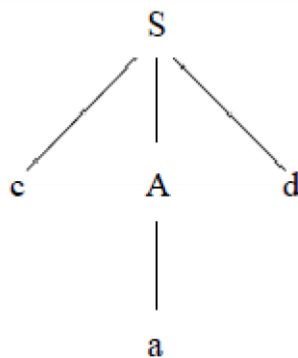
Step3:

The second symbol 'a' of w also matches with second leaf of tree. So advance the input pointer to third symbol of w 'd'. But the third leaf of tree is b which does not match with the input symbol d.

Hence discard the chosen production and reset the pointer to second position. This is called backtracking.

Step4:

Now try the second alternative for A.



Now we can halt and announce the successful completion of parsing.

Example for recursive decent parsing:

A left-recursive grammar can cause a recursive-descent parser to go into an infinite loop.

Hence, elimination of left-recursion must be done before parsing.

Consider the grammar for arithmetic expressions

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

After eliminating the left-recursion the grammar becomes,

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

Now we can write the procedure for grammar as follows:

Recursive procedure:

Procedure E() begin

 T();

 EPRIME();

End

Procedure EPRIME() begin

 If input_symbol='+' then

 ADVANCE();

 T();

 EPRIME(); end

Procedure T() begin

 F();

 TPRIME();

End

Procedure TPRIME() begin

 If input_symbol='*' then

 ADVANCE();

 F();

 TPRIME(); end

Procedure F() begin

 If input-symbol='id' then ADVANCE(
);

```

else if input-symbol='(' then
  ADVANCE( );
E( ); else if input-symbol=')'
then
  ADVANCE( );
end else ERROR( );

```

Stack implementation:

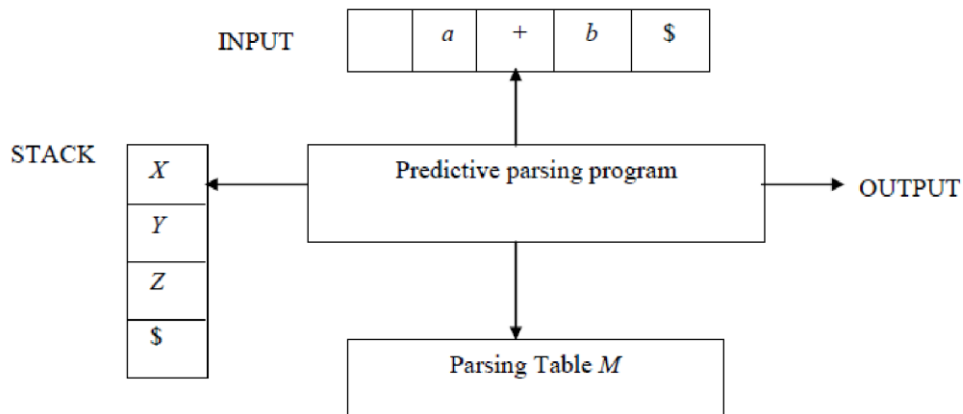
PROCEDURE	INPUT STRING
E()	<u>i</u> d+id*id
T()	<u>i</u> d+id*id
F()	<u>i</u> d+id*id
ADVANCE()	id+ <u>i</u> d*id
TPRIME()	id+ <u>i</u> d*id
EPRIME()	id+ <u>i</u> d*id
ADVANCE()	id+i <u>d</u> *id
T()	id+i <u>d</u> *id
F()	id+i <u>d</u> *id
ADVANCE()	id+i <u>d</u> *id
TPRIME()	id+i <u>d</u> *id
ADVANCE()	id+i <u>d</u> *id
F()	id+i <u>d</u> *id
ADVANCE()	id+i <u>d</u> *id
TPRIME()	id+i <u>d</u> *id

2. PREDICTIVE PARSING

Predictive parsing is a special case of recursive descent parsing where no backtracking is required.

The key problem of predictive parsing is to determine the production to be applied for a non-terminal in case of alternatives.

Non-recursive predictive parser



The table-driven predictive parser has an input buffer, stack, a parsing table and an output stream.

Input buffer:

It consists of strings to be parsed, followed by \$ to indicate the end of the input string.

Stack:

It contains a sequence of grammar symbols preceded by \$ to indicate the bottom of the stack.

Initially, the stack contains the start symbol on top of \$.

Parsing table:

It is a two-dimensional array $M[A, a]$, where 'A' is a non-terminal and 'a' is a terminal.

Predictive parsing program:

The parser is controlled by a program that considers X, the symbol on top of stack, and a, the current input symbol. These two symbols determine the parser action. 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 non-terminal, the program consults entry $M[X, a]$ of the parsing table M. This entry will either be an X-production of the grammar or an error entry.

If $M[X, a] = \{X \rightarrow UVW\}$, the parser replaces X on top of the stack by UVW

If $M[X, a] = \text{error}$, the parser calls an error recovery routine.

Algorithm for nonrecursive 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 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 as follows:

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 non-terminal */ 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 $X = \$$

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 $X \rightarrow \epsilon$ is a production, then add ϵ to $FIRST(X)$.
3. If X is non-terminal and $X \rightarrow \alpha a$ 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 $FIRST(Y_1), \dots, FIRST(Y_{i-1})$; 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 $FIRST(\beta)$ 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 $FIRST(\beta)$ 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 $A \rightarrow \alpha$ of the grammar, do steps 2 and 3.
2. For each terminal a in $\text{FIRST}(\alpha)$, add $A \rightarrow \alpha$ to $M[A, a]$.
3. If ϵ is in $\text{FIRST}(\alpha)$, add $A \rightarrow \alpha$ to $M[A, b]$ for each terminal b in $\text{FOLLOW}(A)$. If ϵ is in $\text{FIRST}(\alpha)$ and $\$$ is in $\text{FOLLOW}(A)$, add $A \rightarrow \alpha$ to $M[A, \$]$.
4. Make each undefined entry of M be error.

Example:

Consider the following grammar :

$E \rightarrow E+T \mid T$

$T \rightarrow T * F \mid F$

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

After eliminating left-recursion the grammar is

$E \rightarrow TE'$

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

$\rightarrow FT'$

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

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

First() :

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

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

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

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

$\{ (, \text{id} \}$

Follow():

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

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

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

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

$*, \$,) \}$

Predictive parsing table :

NON-TERMINAL	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	$F \rightarrow id$			$F \rightarrow (E)$		

Stack implementation:

stack	Input	Output
\$E	id+id*id \$	
\$E'T	id+id*id \$	$E \rightarrow TE'$
\$E'T'F	id+id*id \$	$T \rightarrow FT'$
\$E'T'id	id+id*id \$	$F \rightarrow id$
\$E'T'	+id*id \$	
\$E'	+id*id \$	$T' \rightarrow \epsilon$
\$E'T+	+id*id \$	$E' \rightarrow +TE'$
\$E'T	id*id \$	
\$E'T'F	id*id \$	$T \rightarrow FT'$
\$E'T'id	id*id \$	$F \rightarrow id$
\$E'T'	*id \$	
\$E'T'F*	*id \$	$T' \rightarrow *FT'$
\$E'T'F	id \$	
\$E'T'id	id \$	$F \rightarrow id$
\$E'T'	\$	
\$E'	\$	$T' \rightarrow \epsilon$
\$	\$	$E' \rightarrow \epsilon$

LL(1) grammar:

The parsing table entries are single entries. So each location has not more than one entry. This type of grammar is called LL(1) grammar.

Consider this following grammar:

$S \rightarrow iEtS \mid iEtSeS \mid aE$
 $\rightarrow b$

After eliminating left factoring, we have

$S \rightarrow iEtSS' \mid aS' \rightarrow$

$eS \mid \epsilon$

$E \rightarrow b$

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:

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

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.