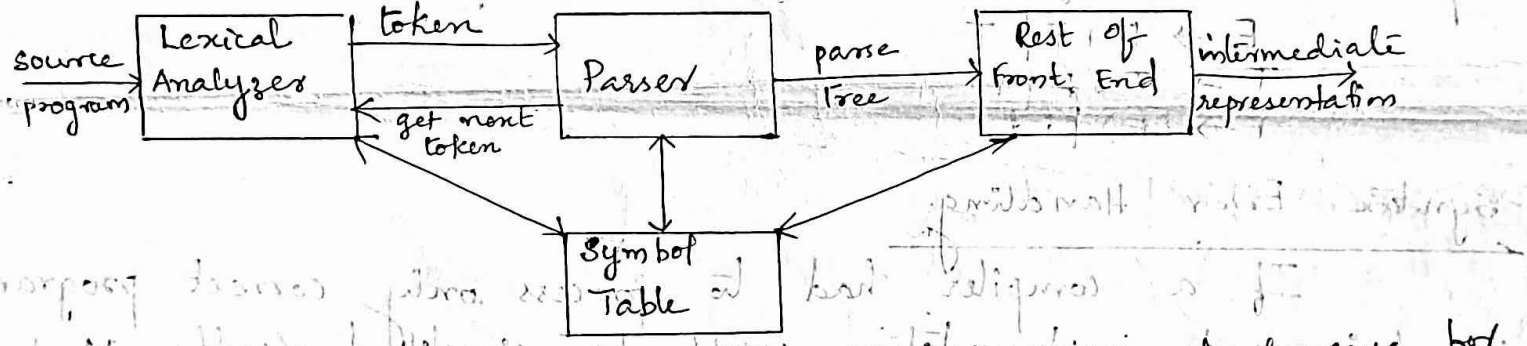
Analysis:-

Since programs may contain syntactic errors, the methods is required for recovery from common errors. syntax of programming language constructs can be specified content-free grammars or BNF (Backus-Naur form) notation.

The Role of the Parser

The Parser obtains a string of tokens from the lexical analyzer and verifies that the string of token names can be generated by the grammar for the source language. We expect the parser to report any syntax errors and to recover from commonly occuring errors to continue processing the remainder of the program for well-formed programs, the parser constructs a parse tree and passes it to the rest of the compiler for further processing.



There are three general types of parsers for grammars: universal, top-down and bottom-up. Universal passing methods such as Cocke-Younger-Kasami (CYK) algorithm and Earley's algorithm can parse any grammar.

The methods commonly used in compilers can be classified as being either top-down or bottom-up.

Top-down methods build parse trees from the top (root) to the bottom (leaves), while bottom-up methods start from leaves and work their way up to the root.

Parsing is the process of determining if a string of terminals can be generated by a grammar.

Representative Grammars :-

Constructs that begin with keywords like 'while' or 'int' are relatively easy to parse, because the keyword guides the choice of the grammar production that must be applied to match the input. Concentration is needed more on expressions, because of the associativity and precedence of operators.

For describing expressions, terms and factors E, T and F respectively. E represents expressions consisting of terms separated by + signs, T represents terms consisting of factors separated by * signs, and F represents factors that can be either parenthesized expressions or identifiers.

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

Syntax Error Handling

If a compiler had to process only correct programs its design and implementation would be simplified greatly. Most programming language specifications do not describe how a compiler should respond to errors; error handling is left to the compiler designer. Planning the error handling right from start can both simplify the structure of a compiler and improve its handling of errors.

Common programming errors can occur at different levels.

- Lexical errors :- include misspellings of identifiers, keywords or operators.

...ma by a semicolon, delete an extraneous semicolon or insert a missing semicolon. The choice of the local correction is left to the compiler designer.

Drawback :- i) Careful while choosing replacement that don't lead to infinite loops.

ii) Difficulty in coping with situations in which the actual error has occured before the point of detection.

Advantage :- less cost

- Error Productions :-

By anticipating common errors that might be encountered, we can augment the grammar for the language at hand with productions that generate the erroneous constructs.

- Global Correction :-

These are algorithms for choosing a minimal sequence of changes to obtain a globally least-cost correction.

Limitation :- Too costly to implement interms of time and space

So mainly designers use phrase level recovery or panic mode

Context Free Grammars :-

Noam Chomsky's findings made the construction of compilers considerably easier. Chomsky's study led to the classification of languages according to the complexity of their grammars. The Chomsky's hierarchy consists of four levels of grammars, called the type 0, type 1, type 2 and type 3 grammars

The type 2 or context-free grammars proved to be the most useful for programming languages and today they are the standard way to represent the structure of programming languages

- Syntactic errors :- include misplaced semicolons or extra missings brackets (braces) that is '{' or '}'. In C or Java case statement without switch is a syntactic error.
- Semantic errors :- include type mismatches between operators and operands.
  eg :- return stmt in Java method with result type void
- Logical errors :- can be anything from incorrect reasoning on the part of the programmer eg :- instead of '==' using '='.

The error handler in a parser has goals like
- Report the presence of errors clearly and accurately
- Recover from each error quickly enough to detect subsequent errors
- Add minimal overhead to the processing of correct programs.

ERROR - RECOVERY STRATEGIES

There are 4 error recovery strategies

- Panic - Mode Recovery :-

With this method on discovering an error, the parser discards input symbols one at a time until one of a designated set of synchronizing tokens is found. The synchronizing tokens are usually delimiters, such as semicolon or }, whose role in the source program is clear and unambiguous. Panic mode correction usually skips a considerable amount of input without checking it for additional errors.

Advantage :- Simplicity & do not go into an infinite loop.

- Phrase - Level Recovery :-

On discovering an error, a parser may perform local correction on the remaining input i.e, it may replace a prefix of the remaining input by some string that allows the parser to continue. A typical local correction is to replace a

Definition of Grammars

A context free grammar has four components

1) A set of terminal symbols sometimes referred to as "tokens". Terminals are the first components of the tokens output by the lexical analyzer.

2) Nonterminals are syntactic variables that denote sets of strings. The set of strings denoted by nonterminals help to define the language generated by the grammar. Nonterminals impose a hierarchical structure on the language that is key to syntax analysis and translation.

3) In a grammar, one nonterminal is distinguished as the start symbol and the set of strings it denotes is the language generated by the grammar. The productions for the start symbol are listed first.

4) The productions of a grammar specify the manner in which the terminals and nonterminals can be combined to form strings. Each production consists of
   a) A nonterminal called the head or left side of the production; this production defines some of the strings denoted by the head.
   b) The symbol $\rightarrow$. Sometimes ::= has been used in place of arrow.
   c) A body or right side consisting of zero or more terminals and nonterminals.

Notational Conventions:-

1) These symbols are terminals:
   a) Lowercase letters early in the alphabet such as a, b, c.
   b) Operator symbols such as +, * and so on
   c) Punctuation symbols such as parentheses, comma and so on
   d) The digits 0, 1, ... 9
   e) Boldface strings such as **if**, **id** or **else** of which represents a single terminal symbol.

2, These symbols are nonterminals:
a, Uppercase letters early in the alphabet such as A, B, C.
b, The letter S, which, when it appears, is usually the start symbol.
c, Lowercase, italic names such as expr or stmt.
d, Uppercase letters may be used to represent nonterminals for the constructs. eg:- E, T and F.

3, Uppercase letters late in the alphabet such as X, Y, Z represent grammar symbols i.e either nonterminals or terminals

4, Lowercase letters late in the alphabet u, v, ... z represent strings of terminals (possibly empty)

5, Lowercase Greek letters α, β, γ represent (possibly empty) strings of grammar symbols.

6, A set of productions $A \to \alpha_1, A \to \alpha_2, \ldots, A \to \alpha_k$ with a common head A (call them A-productions), may be written $A \to \alpha_1 | \alpha_2 | \ldots | \alpha_k$. Call $\alpha_1, \alpha_2, \ldots, \alpha_k$ the alternatives for A.

7, Unless stated otherwise, the head of the first production is the start symbol.

G :- $E \to E+T \mid E-T \mid T$
$T \to T*F \mid T/F \mid F$
$F \to (E) \mid id$

Nonterminals :- E, T and F
Terminals :- +, −, *, /, (,), id
Start symbol :- E

Derivations :-

A grammar derives strings by beginning with the start symbol and, repeatedly replacing a nonterminal by the body of a production for that nonterminal. The terminal strings that can be derived from the start symbol form the language defined by the grammar.

Top-down parsing is related to a class of derivations known as "leftmost" derivations, in which the leftmost nonterminal is rewritten at each step.

Bottom-up parsing is related to a class of derivations known as "rightmost" derivations, in which the rightmost nonterminal is rewritten at each step.

Consider the following grammar, with a single nonterminal, E which adds a production $E \to -E$ to the grammar.

$$E \to E+E \mid E*E \mid -E \mid (E) \mid id$$

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(id)$$

We call such a sequence of replacements a derivations of -(id) from E. The derivation provides a proof that the string -(id) is one particular instance of an expression.

The symbol $\Rightarrow$ means, "derives in one step".
Likewise $\overset{+}{\Rightarrow}$ means, "derives in one or more steps".
The symbol $\overset{*}{\Rightarrow}$ means, "derives in zero or more steps".

Leftmost Derivations:-

The leftmost nonterminal in each sentential is chosen. If $\alpha \Rightarrow \beta$ is a step in which the leftmost nonterminal in β is replaced, we write $\alpha \underset{lm}{\Rightarrow} \beta$.

$$E \underset{lm}{\Rightarrow} -E \underset{lm}{\Rightarrow} -(E) \underset{lm}{\Rightarrow} -(E+E) \underset{lm}{\Rightarrow} -(id+E) \underset{lm}{\Rightarrow} -(id+id)$$

Left sentential form

Rightmost Derivations:-

In rightmost derivations, the rightmost nonterminal is always chosen, we write $\alpha \underset{rm}{\Rightarrow} \beta$.

$$E \underset{rm}{\Rightarrow} -E \underset{rm}{\Rightarrow} -(E) \underset{rm}{\Rightarrow} -(E+E) \underset{rm}{\Rightarrow} -(E+id) \underset{rm}{\Rightarrow} -(id+id)$$

right sentential form

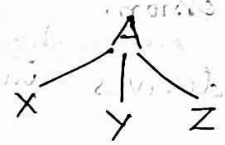
If $S \overset{*}{\Rightarrow} \alpha$, then we say that α is a left sentential

Analogous definitions hold for rightmost derivations. Rightmost derivations are sometimes called canonical derivations.

Parse Trees

A parse tree is a graphical representation of a derivation that filters out the order in which productions are applied to replace nonterminals. Each interior node of a parse tree represents the application of a production.

if non terminal A has a production $A \to XYZ$, then a parse tree may have an interior node labeled A with 3 children labeled X, Y and Z from left to right

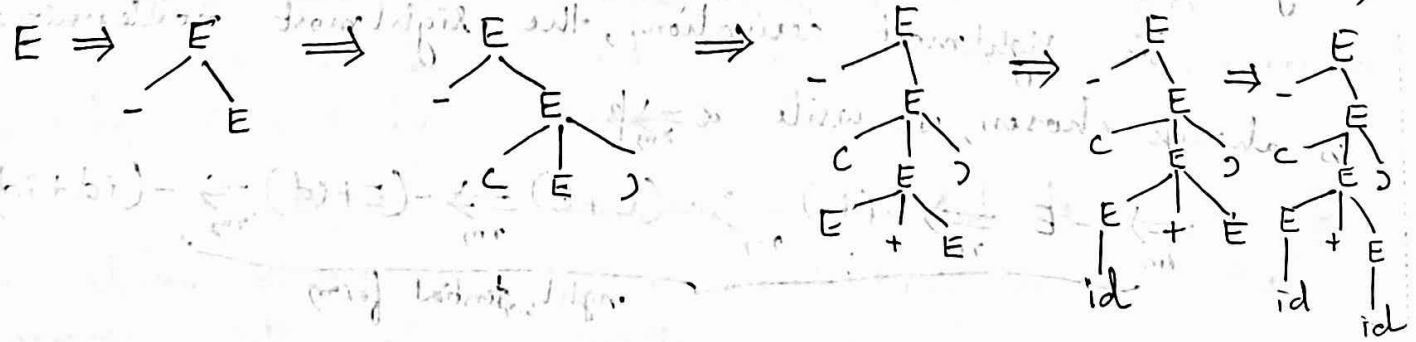


A parse tree according to the grammar is a tree with the following properties

a) The root is labeled by the start symbol
b) Each leaf is labeled by a terminal or by ε
c) Each interior node is labeled by a nonterminal
d) If A is the nonterminal labeling some interior node and $X_1, X_2, \ldots X_n$ are the labels of the children of that node from left to right, then there must be a production $A \to X_1 X_2 \ldots X_n$.

$G :- E \to E+E \mid E*E \mid -E \mid (E) \mid id$

Parse tree for $-(id+id)$ (Sequence of parse trees for derivation)



Ambiguity :-

A grammar that produces more than one parse tree for some sentence is said to be ambiguous. An ambiguous grammar is one that produces more than one leftmost derivation or more than one rightmost derivation for the same sentence.

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

The above grammar permits two distinct leftmost derivations for the sentence id+id*id

$E \Rightarrow E+E$
$\Rightarrow id + E$
$\Rightarrow id + E*E$
$\Rightarrow id + id * E$
$\Rightarrow id + id * id$

$E \Rightarrow E*E$
$\Rightarrow E+E*E$
$\Rightarrow id+E*E$
$\Rightarrow id+id*E$
$\Rightarrow id+id*id$

Associativity of Operators :-

9+5+2 is equivalent to (9+5)+2 and 9-5-2 is equivalent to (9-5)-2.

Precedence of Operators :-

Consider the expression 9+5*2. There are two possible interpretations of this expression : (9+5)*2 or 9+(5*2). The associativity rules for + and * apply to occurrences of the same operator.

Verifying the Language Generated by a Grammar

A grammar G generates a language L has two parts : show that every string generated by G is in L, and every string in L can be generated by G.

$S \rightarrow (S)S \mid \epsilon$

This simple grammar generates all strings of balanced parentheses and only such strings.

Context-Free Grammars Versus Regular Expressions

Every regular language is a context-free language but not vice-versa.

The regular expression $(a|b)^*abb$ and the grammar

$$A_0 \rightarrow aA_0 \mid bA_0 \mid aA_1$$
$$A_1 \rightarrow bA_2$$
$$A_2 \rightarrow bA_3$$
$$A_3 \rightarrow \epsilon$$

describe the same language, the set of strings of a's and b's ending in abb.

Writing a Grammar

Grammars are capable of describing most, but not all, of the syntax of programming languages. The requirement that identifiers be declared before they are used, cannot be described by a context free grammar. Therefore, the sequences of tokens accepted by a parser form a superset of the programming language.

Lexical Versus Syntactic Analysis

Q:= Why use regular expressions to define the lexical syntax of a language?

a) Separating the syntactic structure of a language into lexical and non-lexical parts provides a convenient way of modularizing the front end of a compiler into two manageable sized components.

b) The lexical rules of a language are frequently quite simple, and to describe them we do not need a notation as powerful as grammars.

c) Regular expressions generally provide a more concise and easier-to-understand notation for tokens than grammars.

d) More efficient lexical analyzers can be constructed automatically from regular expressions than from arbitrary grammars.

Eliminating Ambiguity

Sometimes an ambiguous grammar can be rewritten to eliminate the ambiguity.

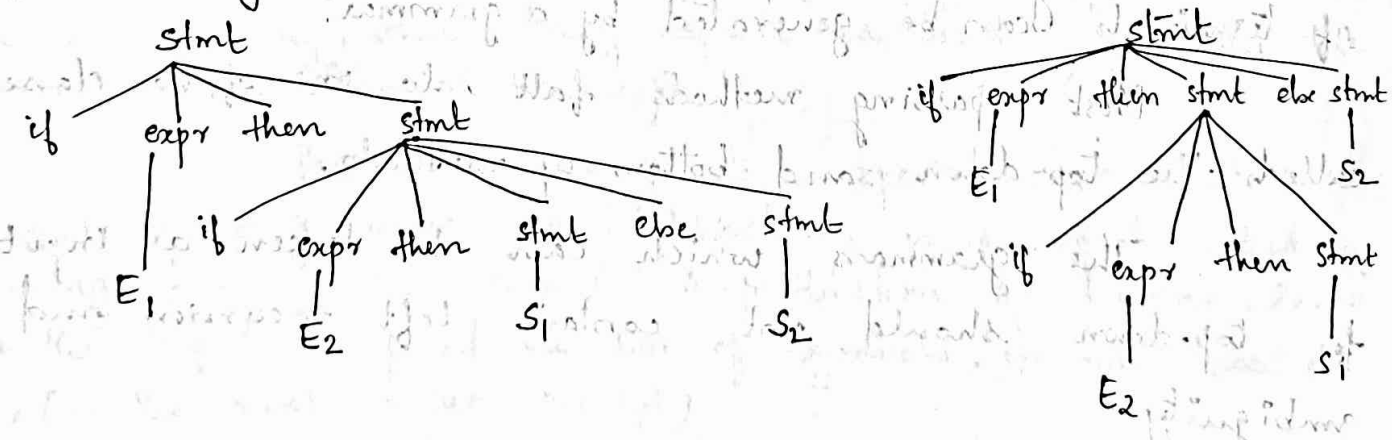
We can eliminate the ambiguity from the following "dangling-else" grammar.

```
stmt → if expr then stmt
     | if expr then stmt else stmt
     | other
```

Here other stands for any other statement.

For the string, if E_1 then if E_2 then S_1 else S_2, "dangling-else" is there, i.e with 2 if statements only one else statement, the passer will be in confusion what to do.

The string "if E_1 then if E_2 then S_1 else S_2" is ambiguous.



Two parse trees for the above ambiguous sentence

We can rewrite the dangling-else grammar. The idea that a statement appearing between a then and an else must be matched; i.e the interior statement must not end with an unmatched or open then.

A matched statement is either an if-then-else statement containing no open statements or it is any other kind of unconditional statement.

Unambiguous grammar for if-then-else statements

```
stmt → matched_stmt
     | open_stmt

matched_stmt → if expr then matched_stmt else matched_stmt
             | other

open_stmt → if expr then stmt
          | if expr then matched_stmt else open_stmt
```

Parsing :-

Parsing is the process of determining how a string of terminals can be generated by a grammar.

Most parsing methods fall into one of two classes, called the top-down and bottom-up methods.

The grammars which can be taken as input for top-down should not contain Left recursion and ambiguity.

The grammar which is taken as i/p to top-down parsing is called LL(1) grammar.

Programming-language parsers almost make a single left-to-right scan over the input, looking ahead one terminal at a time and constructing pieces of the parse tree.

Top-Down Parsing

→ 2 types
- → Recursive-Descent Parsing
- → Predictive Parsing

Top down parsing can be viewed as the problem of constructing a parse tree for the input string, starting from the root and creating the nodes of the parse tree in preorder (depth first). Equivalently, top-down parsing can be viewed as finding a leftmost derivation for an input string.

At each step of a top-down parse, the key problem is that of determining the production to be applied for a nonterminal, say A. Once an A-production is chosen, the rest of the parsing process consists of "matching" the terminal symbols in the production body with the input string.

General form of top-down parsing, called recursive-descent parsing, which may require backtracking to find the correct A-production to be applied.

Predictive parsing is a special case of recursive-descent parsing, where no backtracking is required. Predictive parsing chooses the correct A-production by looking ahead at the input a fixed number of symbols, we may look only at one (i.e the next input symbol)

Top-down parsing while scanning the input from left

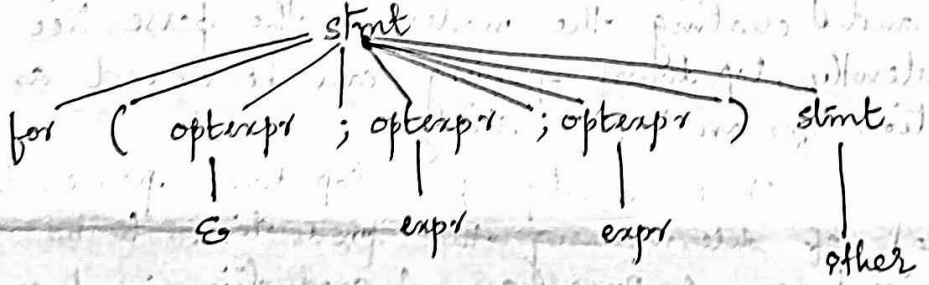
```
stmt → expr ;
     | if (expr) stmt
     | for (optexpr ; optexpr ; optexpr) stmt
     | other

optexpr → ε
        | expr
```

A grammar for some statement in C and java

INPUT :- for(; expr ; expr) other

Parse tree :-



Recursive - Descent Parsing

```
void A() {
    Choose an A-production, A → X1 X2 ... Xk;
    for (i=1 to k) {
        if (Xi is a nonterminal)
            call procedure Xi();
        else if (Xi equals the current input symbol a)
            advance the input to the next symbol;
        else /* an error has occurred */ ;
    }
}
```

Typical procedure for a nonterminal in a top-down parser

A recursive-descent parsing program consists of a set of procedures, one for each nonterminal. Execution begins with the procedure for the start symbol, which halts and announces success if its procedure body scans the entire input string.

General recursive-descent may require backtracking; that is, it may require repeated scans over the input.

eg:- Consider the grammar

$S \rightarrow cAd$

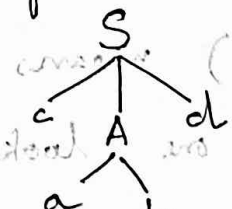
$A \rightarrow ab \mid a$

To construct a parse tree top-down for the input string $w = cad$, begin with a tree consisting of a single node labeled S, and the input pointer pointing to c, the first symbol of w.

S has only one production, so we use it to expand S and obtain the tree

The leftmost leaf, labeled c, matches the first symbol of input w, so we advance the input pointer to a, the second symbol of w, and consider the next leaf, labeled A.

Now, we expand A using the first alternative $A \rightarrow ab$ to obtain the tree



We have a match for the second input symbol, a, so we advance the input pointer to d, the third input symbol and compare d against the next leaf, labeled b.

Since b doesnot match d, we report failure and go back to see whether there is another alternative for A that has not been tried, but that might produce a match.

In going back to A, we must reset the input pointer to position 2, the position it had when we first came to A, which means that the procedure for A must store the input pointer in a local variable.

The second alternative for A produces the tree

The leaf a matches the second symbol of w and the leaf d matches the third symbol. Since we have produced a parse tree for w, we halt and announce successful completion of parsing.

Recursive-descent parsing is a top down method of syntax analysis in which a set of recursive procedures is used to process the input.

A simple form of recursive-descent parsing, called predictive parsing, is in which the lookahead symbol unambiguously determines the flow of control through the procedure body for each nonterminal.

Predictive Parsing take input grammar from LL(k) classes.

LL(1) means left to right scan, leftmost derivation and one lookahead character.

For a grammar to be taken as input for predictive parsing, Eliminate the left Recursion and do Left Factoring.

back has no production

Procedure stmt executes code corresponding to the

stmt → for (optexpr ; optexpr ; optexpr) stmt

In the code for the production body - that is the for case of procedure stmt - each terminal is matched with the lookahead symbol and each nonterminal leads to a call of its procedure, in the following sequence of calls :

```
match (for) ; match ('(');
optexpr (); match (';'); optexpr (); match (';'); optexpr ();
match (')'); stmt ();
```

Pseudocode for a predictive parser

```
void stmt () {
    switch (lookahead) {
    case expr :
        match (expr); match (';'); break;
    case if :
        match (if); match ('('); match (expr); match (')'); stmt ();
        break;
    case for :
        match (for); match ('(');
        optexpr (); match (';'); optexpr (); match (';'); optexpr ();
        match (')'); stmt (); break;
    case other :
        match (other); break;
    default :
        report ("syntax error");
    }
}
```

```
void optexpr() {
    if (lookahead == expr) match (expr);
}
void match (terminal t) {
    if (lookahead == t) lookahead = nextTerminal;
    else report ("syntax error");
}
```

FIRST (stmt) = {expr, if, for, other}
FIRST (expr;) = {expr}

Elimination of Left Recursion

Why to eliminate left Recursion ?

It is possible for a recursive-descent parser to loop forever. A problem arises with "left-recursive" productions like

expr → expr + term

where the leftmost symbol of the body is the same as the nonterminal at the head of the production.

~~Suppose the lookahead symbol changes only when a terminal in the body is matched, no change~~

Suppose the procedure for expr decides to apply this production. The body begins with expr so the procedure for expr is called recursively. Since the lookahead symbol changes only when a terminal in the body is matched, no change to the input took place between recursive calls of expr. As a result, the second call to expr does exactly what the first call did, which means a third call to expr and so on forever.

A left-recursive production can be eliminated by rewriting the offending production.

Consider a nonterminal A with two productions

$A \rightarrow A\alpha \mid \beta$, where α and β are sequences of terminals and nonterminals that donot start with A.

A grammar is left recursive if it has a nonterminal A such that there is a derivation $A \overset{+}{\Rightarrow} A\alpha$ for some string α. Top-down passing methods cannot handle left-recursive grammars, so a transformation is needed to eliminate left recursion.

Immediate Left Recursion:-

for example in

expr → expr + term | term

nonterminal A = expr ; string α = + term and string β = term

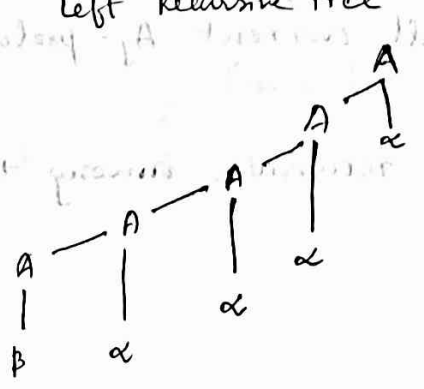
The nonterminal A and its production are said to be left recursive, because the production $A \rightarrow A\alpha$ has A itself as the leftmost symbol on the right side. Repeated application of this production builds up a sequence of α's to the right of A. When A is finally replaced by β, we have a β followed by a sequence of zero or more α's.

replace $A \rightarrow A\alpha \mid \beta$ by $A \rightarrow \beta A'$

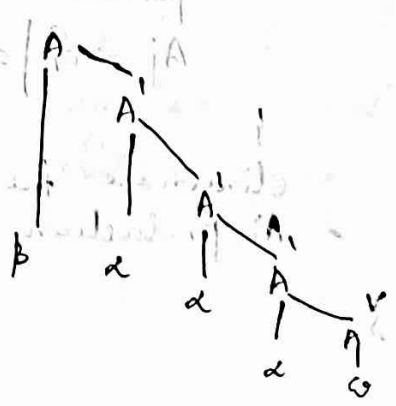
$A' \rightarrow \alpha A' \mid \varepsilon$.

Left Recursive tree

∴



replaced by ⟹



The new grammar is right recursive, because the production has A' itself as the last symbol on right side. But TDP methods don't care about right recursive grammar.

Left recursive Grammar	Non left recursive grammar
$E \to E+T \mid T$	$E \to TE'$
$T \to T*F \mid F$	$E' \to +TE' \mid \epsilon$
$F \to (E) \mid id$	$T \to FT'$
	$T' \to *FT' \mid \epsilon$
	$F \to (E) \mid id.$

If indirect Left Recursion is there, that also should be eliminated.

Algorithm to eliminate left recursion (immediate & indirect) from a grammar

Input :- Grammar G with no cycles or ϵ-productions

Output :- An equivalent grammar with no left recursion.

Method :-

1) Arrange the nonterminals in some order $A_1, A_2 \ldots A_n$
2) for (each i from 1 to n') {
   for (each j from 1 to i-1) {
      replace each production of the form $A_i \to A_j\gamma$ by the productions $A_i \to \delta_1\gamma \mid \delta_2\gamma \mid \cdots \delta_k\gamma$ where
      $A_j \to \delta_1 \mid \delta_2 \mid \cdots \mid \delta_k$ are all current A_j-productions.
   }
   eliminate the immediate left recursion among the A_i-productions.
}

Top-Down Parsing

TDP constructs a parse tree for the i/p string starting from the root and creating the nodes of the parse tree in preorder.

Recursive-Descent Parsing (RDP)

A recursive-descent parsing program consists of a set of procedures, one for each nonterminal. Execution begins with the procedure for the start symbol which halts and announces success if its procedure body scans the entire input string.

Procedure

```
void A() {
    Choose an A-production, A → X1 X2 ... Xn
    for (i=1 to k) {
        if (Xi is a nonterminal)
            call procedure Xi()
        else if (Xi equals the current-i/p symbol a)
            advance the i/p to the next symbol;
        else /* an error has occured */;
    }
}
```

General recursive-descent may require backtracking; that is, it may require repeated scans over the input.

ST and FOLLOW

The construction of both top-down and bottom-up parsers is aided by 2 functions, FIRST & FOLLOW.

Define FIRST(α), where α can be any string of grammar symbols. Define FOLLOW(A) for nonterminal A

To compute FIRST(X) for all grammar symbols X, apply the following rules until no more terminals or ε can be added to any FIRST set.

1. If X is a terminal, then FIRST(X) = $\{X\}$
2. If X is a nonterminal and $X \to Y_1 Y_2 \ldots Y_k$ is a production for some $k \geq 1$, 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}) i.e $Y_1 \ldots Y_{i-1} \overset{*}{\Rightarrow} \varepsilon$. If ε is in FIRST(Y_j) $\forall j = 1, 2, \ldots k$, then add ε to FIRST(X). For example everything in FIRST(Y_1) is surely in FIRST(X). If Y_1 doesnot derive ε, then we add nothing more to FIRST(X) but if $Y_1 \overset{*}{\Rightarrow} \varepsilon$, then we add FIRST($Y_2$) and so on.
3. If $X \to \varepsilon$ is a production, then add ε to FIRST(X)

If to find FIRST(X), where $X \to X_1 X_2 \ldots X_n$

FIRST(X) = FIRST(X_1) if FIRST(X_1) contains ε, add FIRST(X_2) and so on. If all FIRST($X_1 X_2 \ldots X_n$) contain ε add ε to FIRST(X)

To compute FOLLOW(A) for all nonterminals, A

1. Place \$ in FOLLOW(S), where S is the start symbol and \$ is the input right endmarker.

2. if there is a production $A \to \alpha B\beta$, then every one, except ε is in FIRST(β) is in FOLLOW(B).

3. If there is a production $A \to \alpha B$ or a production $A \to \alpha B\beta$ where FIRST(β) contains ε, then everything in FOLLOW(A) is in FOLLOW(B).

Consider G :-

$E \to TE'$ ——— a

$E' \to +TE' \mid \varepsilon$ ——— b

$T \to FT'$ ——— c

$T' \to *FT' \mid \varepsilon$ ——— d

$F \to (E) \mid id$ ——— e

FIRST(E) = FIRST(T) ⟶ Rule (2)

Check whether FIRST(T) contains ε.

FIRST(T) = FIRST(F) ⟶ Rule (2)

FIRST(F) = FIRST('(') ⟶ from (e)

FIRST('(') ⟶ By Rule 1 first(terminal) = Terminal.

so FIRST('(') = {'('}

$\Rightarrow F \to id$

FIRST(F) = FIRST(id) = {id}

∴ FIRST(E) = FIRST(T) = FIRST(F) = {'(', id}

Since FIRST(T) not contains ε, no need for checking FIRST(E') from (a)

(1) Grammars

Predictive parsers, that is recursive-descent parsers needing no backtracking, can be constructed for a class of grammars called LL(1). The first L in LL(1) stands for scanning the input from left to right, the second L for producing a leftmost derivation and the 1 for using one input symbol of lookahead at each step to make parsing action decisions.

A grammar G is LL(1) if and only if whenever $A \to \alpha \mid \beta$ are two distinct productions of G, the following conditions hold.

i, For no terminal a do both α and β derive strings beginning with a.

ii, At most one of α and β can derive the empty string

iii, if $\beta \overset{*}{\Rightarrow} \varepsilon$, then α doesnot derive any string beginning with a terminal in FOLLOW(A).

Likewise, if $\alpha \overset{*}{\Rightarrow} \varepsilon$, β doesnot derive any string beg with a terminal in FOLLOW(A)

eg:- (i) $E' \to +TE' \mid \varepsilon$, here $\alpha = +TE'$, $\beta = \varepsilon$.

if FOLLOW(E') contains +, not LL(1)

other wise LL(1) grammar.

Here FOLLOW(E') = {), $}

So this Grammar is LL(1)

By rule ③,

$T' \rightarrow *FT'$

$A \rightarrow \alpha B \beta$, if FIRST(β) contain ε, then everything in FOLLOW(A) is in FOLLOW(B)

FIRST(T') = $\{+, \varepsilon\}$

So ~~FOF~~ FOLLOW(T') $\Rightarrow$ FOLLOW(F) — (vii 1)

FOLLOW(T') = $\{+, \$\}$ — (viii)

FOLLOW(F) = $\{*, +, \$\}$ — from (vi) & (vii)

6, $F \rightarrow CE)$

$A \rightarrow \alpha B \beta$, A = F, α = C, B = E, β =)

By rule ②, FIRST(β) = FIRST()) = $\{)\}$

FOLLOW(E) = $\{)\}$ — ~~viii~~ (ix)

By rule ③

$F \rightarrow (E)$ in the form

$A \rightarrow \alpha B \beta$ but FIRST(β) not contain ε, so no need to do rule ③

$\therefore$ FOLLOW(E) = $\{), \$\}$ from (i) & ~~iii~~ ix

FOLLOW(E') = $\{), \$\}$ from (ii)

FOLLOW(T) = $\{+,), \$\}$ from (iii) & (iv)

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

FOLLOW(F) = $\{+, *,), \$\}$ from (vii) & (vii 1)

(ii) $S' \rightarrow eS \mid \varepsilon$

Here FOLLOW(S') = {e, $}

i.e Here $\alpha = eS$

$\beta = \varepsilon$.

FIRST(α) = {e}

e is there in FOLLOW(S')

So not an LL(1) grammar

Construction of a predictive parsing table

i/p :- Grammar G

o/p :- Parsing table M

Method :- for each production $A \rightarrow \alpha$

1. for each terminal a in FIRST(α), add $A \rightarrow \alpha$ to M[A, a]

2. If ε is in FIRST(α), then for each terminal b in FOLLOW(A), add $A \rightarrow \alpha$ to M[A, b]. If ε is in FIRST(α) and $ in FOLLOW(A), add $A \rightarrow \alpha$ to M[A, $] as well.

$E \rightarrow TE'$

FIRST(TE') = FIRST(T) = {C, id}

From Rule (1) in M[E, c] and M[E, id] write $E \rightarrow TE'$

$E' \rightarrow +TE'$ in form $A \rightarrow \alpha$

FIRST(α) = FIRST(+TE') = FIRST(+) = {+}

in M[E', +} write $E' \rightarrow +TE'$

$E' \to \varepsilon$; in form $A \to \alpha$

FIRST(α) = FIRST(ε) = ε

So FIND FOLLOW(E') = {), \$ }

in M[E',)] write $E' \to \varepsilon$

M[E', \$] write $E' \to \varepsilon$

$T \to FT'$

$A \to \alpha$

FIRST(α) = FIRST(FT') = FIRST(F) = { (, id }

in M[T, (] and M[T, id] $\Longrightarrow T \to FT'$

$T' \to *FT'$

$A \to \alpha$

FIRST(α) = FIRST($*FT'$) = FIRST($*$) = { * }

in M[T', *] write $T' \to *FT'$

$T' \to \varepsilon$ in $A \to \alpha$ form

FIRST(α) = FIRST(ε) = ε

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

in M[T', +] and M[T',)] and M[T', \$] write

$T' \to \varepsilon$

$F \to (E)$ in the form $A \to \alpha$

FIRST(α) = FIRST((E)) = FIRST(( ) = { (}

in M[F, (] write $F \to (E)$

$F \to id$ in the form $A \to \alpha$; FIRST(α) = FIRST(id) = {id}

In M[F, id] write $F \to id$

NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to id$			$F \to (E)$		

A grammar whose parsing table has no multiply defined entries is said to be LL(1). For the above Parsing table no multiply-defined entries.

So LL(1) grammar.

$S \to iEtSS' \mid a$

$S' \to eS \mid \varepsilon$

$E \to b$

FIRST

S	{i, a}
S'	{e, ε}
E	{b}

FOLLOW

S	{e, $}
S'	{e, $}
E	{t}

$S \to iEtSS'$

$FIRST(S) = FIRST(iEtSS') = FIRST(i) = \{i\}$

$S \to a$

$FIRST(S) = FIRST(a) = \{a\}$

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

$\therefore S' \to eS$

$FIRST(S') = FIRST(eS) = FIRST(e) = \{e\}$

$S' \to \varepsilon$

$FIRST(S') = FIRST(\varepsilon) = \varepsilon$

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

$E \to b$

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

FOLLOW

$FOLLOW(S) = \{\$\}$ { S is start symbol } ——— (1)

$S \to iEtSS'$

$A \to \alpha B \beta$; $A = S$; $\alpha = iEt$, $B = S$, $\beta = S'$

By rule (2).

$FIRST(\beta) = FIRST(S') = \{e, \varepsilon\}$ ——— (2)

everything except ε in $FOLLOW(S)$

$\therefore FOLLOW(S) = \{e\}$ ——— (3)

$S \rightarrow iEtSS'$

$A \rightarrow \alpha B \beta$

By rule (3)

if FIRST(s') contains ε

everything in FOLLOW(S) $\Rightarrow$ FOLLOW(s') — (4)

$S \rightarrow iEtSS'$

$A \rightarrow \alpha B \beta$; $A = S,\ \alpha = i,\ B = E,\ \beta = tSS'$

By rule (2)

FIRST(β) = FIRST(tSS') = FIRST(t) = $\{t\}$

everything except ε in FIRST(β) in FOLLOW(B)

$\therefore$ FOLLOW(E) = $\{t\}$ — (5)

Rule (3) can't be applied because FIRST(β) $\neq \varepsilon$

$S \rightarrow iEtSS'$

$A \rightarrow \alpha B$

Rule (3) ; $\beta \overset{*}{\Rightarrow} \varepsilon$

$\therefore$ FOLLOW(S) $\Rightarrow$ FOLLOW(s') — (6)

$\therefore$ FOLLOW(s') = $\{e, \$\}$

$S' \rightarrow eS$

$A \rightarrow \alpha B$, $\beta \overset{*}{\Rightarrow} \varepsilon$

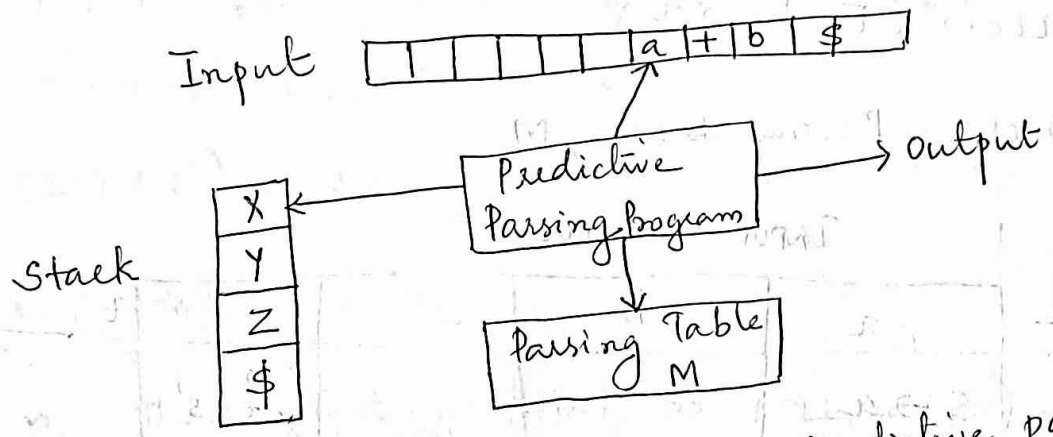
FOLLOW(s') $\Rightarrow$ FOLLOW(S) — (7)

Predictive Parsing table M

Non-Terminal	Input symbol					
	a	b	e	i	t	$
S	$S \to a$			$S \to iEtSS'$		
S'			$S' \to eS$ $S' \to \varepsilon$			$S' \to \varepsilon$
E		$E \to b$				

Here M have multiply defined entries. So. not LL(1) Grammar. If G is left recursive or ambiguous, then M will have at least one multiply defined entry.

Nonrecursive Predictive Parsing



Model of a table-driven predictive parser

Predictive parsing Algorithm

set ip to point to the first symbol of w;
set X to the top stack symbol;
while (X ≠ $) { /* stack is not empty */
if (X is a) pop the stack and advance ip;
else if (X is a terminal) error();
else if (M[X,a] is an error entry) error();
else if (M[X,a] = $X \rightarrow Y_1 Y_2 \cdots Y_k$) {
o/p the production $X \rightarrow Y_1 Y_2 \cdots Y_k$;
pop the stack;
push $Y_k, Y_{k-1} \cdots, Y_1$ onto the stack,
with Y_1 on top;
}

From ① & ③

∴ FOLLOW(S) = {e, $}

From ④ & ⑥

FOLLOW(S') = {e, $}

From ⑤

FOLLOW(E) = {t}

Constructing Parsing table, M

NON TERMINAL	INPUT SYMBOL a	b	e	i	t	$
S	S → a			S → iEtSS'		
S'			S' → eS S' → ε			S' → ε
E		E → b				

✓ $S \xrightarrow[A]{} iEtSS' \xrightarrow{\alpha}$

FIRST(α) = FIRST(iEtSS')
= FIRST(i)
= {i}

So M[S, i] ; S → iEtSS'

✓ S → a

A → α

FIRST(α) = FIRST(a) = {a}

M[S, a] ; S → a

✓ S' → eS

A → α

M[S', e] ; S' → eS

✓ S' → ε

A → α

FIRST(α) = FIRST(ε)

find FOLLOW(S') = {e, $}

In M[S', e] and M[S', $] write S' → ε

✓ E → b

A → α

FIRST(α)
= FIRST(b)
= {b}

In M[E, b] write E → b

$RST(E') = FIRST(+TE') = FIRST(+)$

$= \{+\}$

From $E' \rightarrow \varepsilon$

$FIRST(E') = FIRST(\varepsilon) = \{\varepsilon\}$

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

$FIRST(T') = FIRST(*FT') = FIRST(*) = \{*\}$

from $T' \rightarrow \varepsilon$, $FIRST(T') = FIRST(\varepsilon) = \{\varepsilon\}$

$\therefore FIRST(T') = \{+, \varepsilon\}$

FOLLOW (NonTerminals)

1, E is the start symbol, so FOLLOW $(E) = \{\$\}$ — (i)

2, $E \rightarrow TE'$ in the form

$A \rightarrow \alpha B$, where $A = E$, $\alpha = T$, $B = E'$

By rule (3), everything in FOLLOW(A) in FOLLOW(B)

$\therefore$ So FOLLOW $(E) \Longrightarrow$ FOLLOW (E') — (ii)

$E \rightarrow TE'$ can be

$A \rightarrow \alpha B \beta$, where $\beta \overset{*}{\Rightarrow} \varepsilon$, By rule (3)

$\therefore FOLLOW(E) \Rightarrow FOLLOW(E')$

3, $E' \rightarrow +TE'$ in form

$A \rightarrow \alpha B \beta$, where $A = E'$, $\alpha = +$, $B = T$, $\beta = E'$

By Rule (2)

Everything except ε in $FIRST(\beta)$ in $FOLLOW(B)$

FIRST(E') = {+, ε}

∴ FOLLOW(T) = {+} ⟹ (iii)

E → +TE'
A → αBβ , By rule (3)

β = E' ; FIRST(β) = FIRST(E') contains ε, so.

FOLLOW(A) ⟹ FOLLOW(B)
FOLLOW(E) ⟹ FOLLOW(T) —— (iv)

FOLLOW(T) = {+, $} , By (i) & (iii) —— (v)

4, T → FT'
A → αB , A = T, α = F, B = T'

By rule (3)
Everything in FOLLOW(A) in FOLLOW(B)

FOLLOW(T) ⟹ FOLLOW(T') —— (vi)

So. FOLLOW(T') = {+, $} — From (v)

T → FT'
By rule (3) , β ⇒ ε
FOLLOW(T) ⇒ FOLLOW(T')

5, T' → *FT'
A → αBβ , where A = T', α = *, B = F, β = T'

By rule (2) , everything in FIRST(β) except ε, in FOLLOW(B)

FIRST(T') = {*, ε}
FOLLOW(F) = {*} —— (vii)

X to the top stack symbol;

Consider G :- $E \to TE'$
$E' \to +TE' \mid \epsilon$
$T \to FT'$
$T' \to *FT' \mid \epsilon$
$F \to (E) \mid id$

On input id + id * id, leftmost derivation

$$E \underset{lm}{\Rightarrow} TE' \underset{lm}{\Rightarrow} FT'E' \underset{lm}{\Rightarrow} idT'E' \underset{lm}{\Rightarrow} idE' \underset{lm}{\Rightarrow} id+TE' \underset{lm}{\Rightarrow} id+FT'E'$$
$$\underset{lm}{\Rightarrow} id+idT'E' \underset{lm}{\Rightarrow} id+id*FT'E' \underset{lm}{\Rightarrow} id+id*idT'E'$$
$$\underset{lm}{\Rightarrow} id+id*idE' \underset{lm}{\Rightarrow} id+id*id$$

Moves made by a predictive parser on i/p id + id * id

Matched	Stack	Input	Action
	E $	id+id*id $	o/p :- $E \to TE'$
	TE' $	id+id*id $	o/p :- $T \to FT'$
	FT'E' $	id+id*id $	o/p :- $F \to id$
	idT'E' $	id+id*id $	match id
id	T'E' $	+id*id $	o/p :- $T' \to \epsilon$
id	E' $	+id*id $	o/p :- $E' \to +TE'$
id	+TE' $	+id*id $	match +
id+	TE' $	id*id $	o/p :- $T \to FT'$
id+	FT'E' $	id*id $	o/p :- $F \to id$
id+id	idT'E' $	id*id $	match id
id+id	T'E' $	*id $	o/p :- $T' \to *FT'$
id+id*	*FT'E' $	*id $	match *
id+id*id	FT'E' $	id $	o/p :- $F \to id$
id+id*id	idT'E' $	id $	match id
id+id*id	T'E' $	$	o/p :- $T' \to \epsilon$
id+id*id	E' $	$	o/p :- $E' \to \epsilon$
	$	$	

Error Recovery in Predictive Parsing

Panic mode error recovering:-

Synchronizing set of tokens :- Find FOLLOW(A), where A is each nonterminal in G, In M[A, b] where no G is in parsing table M put synch where b is each terminal in FOLLOW(A)

	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$	synch	synch
E'		$E' \to +TE'$			$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to FT'$	synch		$T \to FT'$	synch	synch
T'		$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to id$	synch	synch	$F \to (E)$	synch	synch

FOLLOW(E) = {), $} no relⁿ in M so put synch

FOLLOW(E') = {), $ } already relⁿ is there in M

So need of synch

FOLLOW(T) = {+,), $} , put synch in M[T, {+,), $}]

FOLLOW(T') = {+,), $}, already relⁿ in M

FOLLOW(F) = {+, *,), $}, put synch in

M [F, {+, *,), $}]

... of parsing and error recovery made by the predictive parser.

Stack	input	remark
E$	) id * + id $	M(E,)) synch so error, skip)
E$	id * + id $	M[E,id], id in first(E) so reduce to TE'
TE'$	id * + id $	id in FIRST(T) so reduce T → FT'
FT'E'$	id * + id $	id in FIRST(F) so reduce F → id
id T'E'$	id * + id $	Match id in i/p and stack, pop() and advance ip.
T'E'$	* + id $	* in first(T') so reduce T' → * FT'
* FT'E'$	* + id $	Matches * in i/p & stack, pop() & advances ip.
FT'E'$	+ id $	M[F, +] = syn error, F has been poped

Stack	I/P	
$T'E'\$$	$+ id \$$	First(+ , so T' → ε.
$E'\$$	$+ id \$$	+ is in FIRST(E') reduce $E' \to +TE'$
$+TE'\$$	$+ id \$$	Matches + in i/p & stack, pop() & advance ip.
$TE'\$$	$id \$$	id in FIRST(T) reduce $T \to FT'$
$FT'E'\$$	$id \$$	id in FIRST(F) reduce $F \to id$
$id\ T'E'\$$	$id \$$	Matches id pop & advance ip.
$T'E'\$$	$\$$	$m[T', \$] = T' \to \varepsilon$ reduce $T' \to \varepsilon$
$E'\$$	$\$$	$m[E', \$] = E' \to \varepsilon$ reduce $E' \to \varepsilon$.
$\$$	$\$$	Error recovered.

$S \rightarrow (L) \mid a$

$L \rightarrow L, S \mid S$

Eliminating Left Recursion

$S \rightarrow (L) \mid a$

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

New G:-

$S \rightarrow (L) \mid a$

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

$L' \rightarrow , S\ L' \mid \varepsilon$

Now the G is suitable for Predictive Parsing

Computing FIRST

$S \rightarrow (L) \mid a$

FIRST (S) = FIRST ((L)) and FIRST (a)

$= \{(, a\}$

FIRST (L) $= \{(, a\}$

FIRST (L') $= \{,, \varepsilon\}$

Computing FOLLOW

① $S \to (L)$ in the form

$A \to \alpha B \beta$ where $\alpha = (\,,\ B = L$ and $\beta =)$

By rule 2

First(β) − ε is in follow (B)

∴ First ( ) is in follow (L) ——— ①

② $S \to a$ not in any form $A \to \alpha B$ or $A \to \alpha B \beta$.

③ $L \to (L) L'$ in the form

$A \to \alpha B \beta$ where $\alpha = (\,,\ B = L\,;\ \beta =) L'$

By rule 2

∴ First ( ) L') in follow (L) ——— ②

$L \to (L)\ L'$ in the form

$A \to \alpha\ B$

By rule 3

∴ Everything in follow (L) in follow (L') ——— ③

④ $L \to a L'$ in the form

$A \to \alpha B$ where $\alpha = a,\ B = L'$

By rule 3

∴ Everything in Follow (L) is in follow (L') ——— ④

⑤ $L' \to , S L'$ in the form

$A \to \alpha B \beta$ where $\alpha = ,\ ;\ B = S\,;\ \beta = L'$

By rule 2

First (L') − ε is in follow (S) ——— ⑤

since First (L') contains ε, Apply rule 3

By rule 3

Everything in Follow (L') is in follow (S) ——— ⑥

$L' \to , S L'$ in the form

$A \to \alpha B$

By rule 3.

Everything in follow(L') is in follow(L') — (7)

(6) $L' \to \varepsilon$ is not in any of the forms $A \to \alpha B$ or $A \to \alpha B \beta$

∴ FOLLOW(S) contains $ since S is the start symbol

∴

	FIRST	FOLLOW
S	(, a	$, , ,)
L'	, , ε	)
L	(, a	)

Predictive parsing table

$S \to (L)$ in the form

$A \to \alpha$

first(α)

ie first((L)) = {(}

M[S, (] write $S \to (L)$

$L \to (L) L'$ in the form.

$A \to \alpha$

find first(α) ie first((L)L')

in M[L, (] write $L \to (L) L'$

$S \to a$ in the form

$A \to \alpha$

find first(α) i.e first(a)

in M[S, a] write

$S \to a$

$L \to a L'$ in the form

$A \to \alpha$

And first(α) i.e first(aL')

in M[L, a] write

$L \to a L'$

$L' \to , S L'$ in the form
$A \to \alpha$

Find first(α) i.e first($, SL'$)

i.e in M[L', ,] write

$L' \to , S L'$

$L' \to \varepsilon$ in the form
$A \to \alpha$

first(α) = first(ε)

Since first(α) contains ε

find follow(A)

i.e follow(L') = {) }

∴ in M[L',)] write

$L' \to \varepsilon$

Predictive Parsing table (M)

Non Terminals	Input symbols: (	)	,	a	$
S	$S \to (L)$			$S \to a$	
L	$L \to (L) L'$			$L \to a L'$	
L'		$L' \to \varepsilon$	$L' \to , S L'$		

Input string :- (a, a, a, (a, a))

Matched string	Stack	Input	Action
	S	(a, a, a, (a, a))$	
	(L)$	(a, a, a, (a, a))$	o/p S → (L)
(	L)$	a, a, a, (a, a))$	match (
(a	aL')$	a, a, a, (a, a))$	o/p L → aL'
(a	L')$	, a, a, (a, a))$	match a
	,SL')$	, a, a, (a, a))$	o/p L' → ,SL'
(a,	SL')$	a, a, (a, a))$	match ,
	aL')$	a, a, (a, a))$	o/p S → a
(a, a	L')$	, a, (a, a))$	match a
	,SL')$	, a, (a, a))$	o/p L' → ,SL'
(a, a,	SL')$	a, (a, a))$	match ,
	aL')$	a, (a, a))$	o/p S → a
(a, a, a	L')$	, (a, a))$	match a
	,SL')$	, (a, a))$	o/p L' → ,SL'
(a, a, a,	SL')$	(a, a))$	match ,
	(L)L')$	(a, a))$	o/p S → (L)
(a, a, a, (	L)L')$	a, a))$	match (

Matched string	Stack	Input	Action
	$aL')L')\$$	$a,a))\$$	o/p $L \to aL'$
$(a$	$L')L')\$$	$,a))\$$	match a
$(a,a$	$,SL')L')\$$	$,a))\$$	o/p $L' \to ,SL'$
$(a,a,$	$SL')L')\$$	$a))\$$	match ,
$(a,a,a$	$aL')L')\$$	$a))\$$	o/p $S \to a$
$(a,a,a,$	$L')L')\$$	$))\$$	match a
$(a,a,a,a$	$)L')\$$	$))\$$	o/p $L' \to \varepsilon$
(a,a,a,a)	$L')\$$	$)\$$	match)
(a,a,a,a)	$)\$$	$)\$$	o/p $L' \to \varepsilon$
$(a,a,a,(a,a))$	$\$$	$\$$	match)

$S \rightarrow CC$

$C \rightarrow cC \mid d$

First (S) = First (C) = {c, d}

∴ First (S) = First (C) = {c, d}

Computing Follow

Follow (S) contains $, since S is the start symbol.

① $S \rightarrow CC$ is in the form

$A \rightarrow \alpha B$

By rule 3

Everything in Follow (S) is in follow (C) —— ①

$S \rightarrow CC$ is in the form.

$A \rightarrow \alpha B \beta$

By rule 2

Everything in first (β) except ε is in follow (C) —— ②

Since first (β) doesnot contains ε rule 3 cannot be applied.

② $C \rightarrow cC$ is in the form

$A \rightarrow \alpha B$

By rule 3

Everything in follow (A) is in follow (B)

Everything in follow (C) is in follow (C) —— ③

	First	Follow
S	c, d	$
C	c, d	$, c, d

Predictive Parsing Table

i, $S \to CC$ is in the form.
$A \to \alpha$, where $\alpha = CC$

First(α) = First(CC) = First(C) = {c, d}

∴ In M[S, c] and M[S, d] write

$S \to CC$

ii, $C \to cC$ is in the form
$A \to \alpha$

First(α) = First(cC) = First(c) = {c}

∴ In M[C, c] write $C \to cC$

iii, $C \to d$ is in the form.
$A \to \alpha$

First(α) = First(d) = {d}

∴ In M[C, d] write $C \to d$.

Non terminals	Input Symbols c	d	$
S	S → CC	S → CC	
C	C → cC	C → d	

Input string cccdd

Matched string	Stack	Input	Action
	$ S $	ccccdd $	
	C C $	ccccdd $	o/p S → CC
	c C C $	ccccdd $	o/p C → cC
	C C $	cccdd $	match c
c	c C C $	cccdd $	o/p C → cC
c	C C $	ccdd $	match c
cc	c C C $	ccdd $	o/p C → cC
	C C $	cdd $	match c
ccc	c C C $	cdd $	o/p C → cC
	C C $	dd $	match c
cccc	d C $	dd $	o/p C → d
	C $	d $	match d
cccd	d $	d $	o/p C → d
	$	$	match d
cccdd			

(3) $S \rightarrow aBa$

$B \rightarrow bB \mid \varepsilon$

$S \rightarrow aBa$

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

$First(B) = First(bB) \cup First(\varepsilon) = First(b) \cup \{\varepsilon\}$

$= \{b, \varepsilon\}$

Computing Follow

$S \rightarrow aBa$

Follow(S) contains $\$$; since S is the start symbol

(1) $S \rightarrow aBa$

$A \rightarrow \alpha B \beta$

By rule 2

First(β) $- \{\varepsilon\}$ is in follow(B) ——— (1)

(2) $B \rightarrow bB$

$A \rightarrow \alpha B$

By rule 3

Everything in follow(B) is in follow(B)

	First	follow
S	$\{a\}$	$\{\$\}$
B	$\{b, \varepsilon\}$	$\{a\}$

Predictive Parsing table

$S \to aBa$ in the form $A \to \alpha$ $\therefore$ First(α) = {a}

$\therefore M[S, a] = S \to aBa$

$B \to bB$ in the form $A \to \alpha$ $\therefore$ First(α) = {b}

$\therefore M[B, b] = B \to bB$

$B \to \varepsilon$ in the form $A \to \alpha$ First(α) = ε

Since First(α) = ε find follow(B)

$\therefore M[B, a]$ write $B \to \varepsilon$

Non Terminal	Input Symbol		
	a	b	$
S	$S \to aBa$		
B	$B \to \varepsilon$	$B \to bB$	

Input string abb bbb a

Matched string	Stack	Input	Action
	S $	abbbbb a $	
	aBa $	abbbbba $	o/p $S \to aBa$
a	Ba $	bbbbba $	match a
	bBa $	bbbbba $	o/p $B \to bB$
ab	Ba $	bbbba $	match b
	bBa $	bbbba $	o/p $B \to bB$
abb	Ba $	bbba $	match b
	bBa $	bbba $	o/p $B \to bB$
abbb	Ba $	bba $	match b
	bBa $	bba $	o/p: $B \to bB$
abbbb	Ba $	ba $	match b
	bBa $	ba $	o/p $B \to bB$
abbbbb	Ba $	a $	match b
	a $	a $	o/p $B \to \varepsilon$
abbbbba	$	$	match a

(1) $S \to aBDh$
$B \to cC$
$C \to bC \mid \varepsilon$
$D \to EF$
$E \to g \mid \varepsilon$
$F \to f \mid \varepsilon$

Computing first

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

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

$First(C) = First(bC) \cup First(\varepsilon\}) = \{b\} \cup \{\varepsilon\} = \{b, \varepsilon\}$

$First(D) = First(EF)$

$First(E) = \{g, \varepsilon\}$

$First(F) = \{f, \varepsilon\}$

$First(D) = First(EF) = First(E)$ ~~$\{g$~~ $\cup First(F) \cup \{\varepsilon\}$

Since First(E) and First(F) contains ε.

$= \{g, f, \varepsilon\}$

Computing follow :. Since S is start symbol $Follow(S) = \$$

(1) $S \to aBDh$ is in the form
$A \to \alpha B \beta$; where $\alpha = a$; $B = B$; $\beta = Dh$

By rule 2
Everything in $First(\beta)$ except ε is in $Follow(B)$.
$First(\beta) = First(Dh) = \{g, f, h\}$
:. Follow(B) contains $= \{g, f, h\}$ ——— (1).

→ a B D h is the form of
$A \rightarrow \alpha \cdot B \beta$ where $\alpha = aB$, $\cdot B = D$, $\beta = h$

By rule 2

Everything in First (β) is in Follow (D)

∴ Follow (D) = {h} ——— (2)

(2) $B \rightarrow cC$ is in the form of
$A \rightarrow \alpha B$

By rule 3

Everything in Follow (B) is in Follow (C) — (3)

(3) $C \rightarrow bC$ is in the form of
$A \rightarrow \alpha B$ By rule 3

Everything in Follow (C) in Follow (C) ——— (4)

(4) $D \rightarrow EF$ in the form of
$A \rightarrow \alpha B$
By rule 3
Everything in follow (D) is in follow (F) — (5)

$D \rightarrow Ef$ in the form
$A \rightarrow \alpha B \beta$, where $\alpha = \varepsilon$, $B = E$, $\beta = f$

By rule 2
Everything in First (β) except ε is Follow (E) —
Since first (f) contains ε
By rule 3
Everything in Follow (D) is in Follow (E) — (7)

	First	follow
S	{ a }	{ $ }
B	{ c }	{ g, f, h }
C	{ b, ε }	{ g, f, h }
D	{ g, f, ε }	{ h }
E	{ g, ε }	{ h, f }
F	{ f, ε }	{ h }

Passing table

Non Terminals	Input symbols: a	b	c	f	g	h	$
S	S→aBDh						
B			B→cC				
C		C→bC		C→ε	C→ε	C→ε	
D				D→EF	D→EF	D→EF	
E				E→ε	E→g	E→ε	
F				F→f		F→ε	

$S \rightarrow ABa \mid bCA$

$A \rightarrow CBCD \mid \varepsilon$

$B \rightarrow CdA \mid ad$

$C \rightarrow eC \mid \varepsilon$

$D \rightarrow bSf \mid a$

Computing first

$first(S) = first(ABa) \cup first(bCA)$

$first(A) = first(CBCD) \cup first\{\varepsilon\}$

$first(B) = first(CdA) \cup first(ad)$

$first(C) = \{e, \varepsilon\}$

$first(D) = \{b, a\}$

$first(CdA) = first(C) \cup \{d\}$ // since $first(C) = \{\varepsilon\}$

$= \{e, d\}$

$\therefore first(B) = \{e, d\} \cup \{a\} = \{a, d, e\}$

$first(A) = \{a, d, e, \varepsilon\}$ // since $first(C) = \varepsilon$ and $first(B)$ also.

$first(S) = \{a, d, e, b\}$

Computing Follow

(1) $S \rightarrow ABa$

Follow(S) contains $\$$, since S is the start symbol (1)

$S \rightarrow ABa$ in the form
$A \rightarrow \alpha B \beta$

First(a) is in Follow(B) ——— (2)

$S \rightarrow ABa$ in the form
$A \rightarrow \alpha B \beta$, where $\alpha = \varepsilon$, $B = A$, $\beta = Ba$

First(Ba) − ε is in Follow(A) ——— (3)

(2) $S \rightarrow bCA$ is in the form
$A \rightarrow \alpha B \beta$

Everything in First(β) − ε is in Follow(B)
By rule 2

First(A) − ε is in Follow(C) ——— (4)
Since First(A) contains ε, by rule 3

Everything in Follow(S) is in Follow(C) ——— (5)

$S \rightarrow bCA$ is in the form
$A \rightarrow \alpha B$

Everything in Follow(S) is in Follow(A) ——— (6)

(3) $A \rightarrow CBCD$ is in the form
$A \rightarrow \alpha B \beta$, where $\alpha = C$, $B = B$, $\beta = CD$
By rule 2
First(CD) − ε is in Follow(B) ——— (7)

$A \rightarrow CBCD$ is in the form
$A \rightarrow \alpha B \beta$, where $\alpha = CB$, $B = C$, $\beta = D$

Rule 2
Everything in first(β) - ε is in follow (C)
(8)

$A \rightarrow CBCD$ is in the form
$A \rightarrow \alpha B \beta$, where $\alpha = \varepsilon$, $B = C$, $\beta = BCD$

By rule 2
Everything in First (BCD) - ε is in Follow (C)
(9)

$A \rightarrow CBCD$ is in the form
$A \rightarrow \alpha B$, where $\alpha = CBC$, $B = D$

By rule 3
∴ Everything in follow (A) is in follow (D) —— (10)

$B \rightarrow CdA$ is in the form
$A \rightarrow \alpha B$, where $\alpha = Cd$, $B = A$

By rule 3
Everything in follow (B) is in follow (A) —— (11)

$B \rightarrow CdA$ is in the form
$A \rightarrow \alpha B \beta$, where $\alpha = \varepsilon$, $B = C$, $\beta = dA$

By rule 2
Everything in first (dA) - ε is in follow (C) —— (12)

$C \rightarrow eC$ is in the form
$A \rightarrow \alpha B$; By rule 3 Everything in follow (C) in follow (C)

$D \rightarrow bSf$ is in the form
$A \rightarrow \alpha B \beta$
By rule 2
Everything in first (f) is in follow (S) —— (13)

	First	follow
S	{a, d, e, b}	{$, f}
A	{a, d, e, ε}	{a, d, e, $, b, f}
B	{a, d, e}	{a, e, b}
C	{e, ε}	{a, d, e, $, b, f}
D	{a, b}	{a, d, e, $, b, f}

Predictive Parsing Table

Non-Terminals	Input Symbols a	b	d	e	f	$
S	S→ABa	S→bCA	S→ABa	S→ABa		
A	A→CBCD A→ε	A→ε	A→ABCD A→ε	A→CBCD A→ε	A→ε	A→ε
B	B→ad		B→CdA	B→CdA		
C	C→ε	C→ε	C→ε	C→eC C→ε	C→ε	C→ε
D	D→a	D→bSf				

Since multiple entries are there in the Parsing table the above G is not LL(1)

$S \rightarrow AB \mid eDa$

$A \rightarrow ab \mid C$

$B \rightarrow dC$

$C \rightarrow eC \mid \varepsilon$

$D \rightarrow fD \mid \varepsilon$

Computing FIRST

$First(S) = First(AB) \cup First(eDa)$

$= First(AB) \cup First(e)$

$First(A) = First(ab) \cup First(C)$

$= First(a) \cup First(C)$

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

$First(C) = \{e, \varepsilon\}$

$First(D) = \{f, \varepsilon\}$

$First(A) = \{a\} \cup \{e, \varepsilon\} = \{a, e, \varepsilon\}$

$First(S) = \{a, e, d\}$

Computing Follow

① $S \rightarrow AB$ is in the form
$A \rightarrow \alpha B$

By rule 3

Everything in Follow(S) is in Follow(B) —— ①

Follow(S) = {$} since S is start symbol

$S \rightarrow AB$ is in the form
$A \rightarrow \alpha B \beta$
By rule 2
Everything in First(B) − ε is in Follow(A) — (2)

(2) $S \rightarrow eDa$ is in the form
$A \rightarrow \alpha B \beta$
By rule 2
First(a) is in Follow(D) — (3)

(3) $A \rightarrow C$ is in the form
$A \rightarrow \alpha B$
By rule 3
Everything in Follow(A) is in Follow(C) — (4)

(4) $B \rightarrow dC$ is in the form.
$A \rightarrow \alpha B$
By rule 3
Everything in Follow(B) is in Follow(C) — (5)

(5) $C \rightarrow eC$ is in the form
$A \rightarrow \alpha B$
By rule 3
Everything in Follow(C) is in Follow(C)

(6) $D \rightarrow fD$ is in the form
$A \rightarrow \alpha B$
By rule 3
Everything in Follow(D) is in Follow(D)

	First	Follow
S	{a, d, e}	{$}
A	{a, e, ε}	{d}
		{$}
B	{d}	
C	{e, ε}	{d, $}
D	{f, ε}	{a}

Predictive Parsing Table

Non Terminals	Input Symbols: a	b	d	e	f	$
S	S→AB		S→AB	S→AB S→eDa		
A	A→ab		A→C	A→C		A→C
B			B→dC			
C			C→ε	C→eC		C→ε
D	D→ε				D→fD	

④ $S \rightarrow L, S \mid a$

$L \rightarrow AB \mid b$

$A \rightarrow a \mid \varepsilon$

$B \rightarrow b$

$First(S) = First(L) \cup First(a)$

$First(L) = First(AB) \cup First(b)$

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

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

$First(S) = First(L) \cup \{a\}$

$First(L) = First(AB) \cup \{b\}$

$= \{a, b\}$

$First(S) = \{a, b\}$

Computing follow

$Follow(S) = \{\$\}$; Since S is the start symbol

① $S \rightarrow L, S$ is in the form
$A \rightarrow \alpha B \beta$, where $\alpha = \varepsilon$, $B = L$, $\beta = , S$
By rule 2
Everything in $First(\beta)$ is in $Follow(L)$ ——— ①

② $L \rightarrow AB$ is in the form
$A \rightarrow \alpha B$ where $\alpha = A$, $B = B$
Everything in $Follow(L)$ is in $Follow(B)$ ——— ②

$A \to \alpha B \beta$ is in the form
$A \to \alpha B \beta$, where $\alpha = \varepsilon$, $B = A$, $\beta = B$

By rule 2

Everything in first(β) $- \varepsilon$ is in follow(A) — (3)

	first	follow
S	{a, b}	{$}
L	{a, b}	{,}
A	{a, ε}	{b}
B	{b}	{,}

Predictive Parsing Table

Non Terminals	Input Symbol a	b	,	$
S	S → a			
L	L → AB	L → AB L → b		
A	A → a	A → ε		
B		B → b		

Since multiple entries are there in the parsing table. The grammar is not LL(1)

(5) $S \to aA \mid AB$

$A \to Ab \mid c$

$B \to e \mid f$

Eliminating Left Recursion

$A \to Ab \mid c$ is changed to

$A \to cA'$

$A' \to bA' \mid \varepsilon$

New G :-

$S \to aA \mid AB$

$A \to cA'$

$A' \to bA' \mid \varepsilon$

$B \to e \mid f$

$First(S) = \{a\} \cup First(AB)$

$First(A) = \{c\}$

$First(A') = \{b, \varepsilon\}$

$First(B) = \{e, f\}$

$\therefore First(S) = \{a, c\}$

ng follow

$S \to aA$ in the form

$A \to \alpha B$

By rule 3

Everything in follow (S) is in follow (A) —— (1)

$ is in follow (S) // since start symbol

(2) $S \to AB$ in the form

$A \to \alpha B$

By rule 3

Everything in follow (S) is in follow (B) —— (2)

$S \to AB$ is in the form

$A \to \alpha B \beta$

By rule 2

Everything in first(β) − ϵ is in follow (A) —— (3)

(3) $A \to cA'$ is in the form

$A \to \alpha B$

By rule 3

Everything in follow (A) is in follow (A') —— (4)

(4) $A' \to bA'$ is in form

$A \to \alpha B$

Everything in follow (A') is in follow (A') —— (5)

	first	follow
S	{a, c}	{$
A	{c}	{e, f, $}
A'	{b, ε}	{$, e, f}
B	{e, f}	{$}

Predictive Parsing Table

Non Terminals	Input symbols					
	a	b	c	e	f	$
S	$S \to aA$		$S \to AB$			
A			$A \to cA'$			
A'		$A' \to bA'$		$A' \to \varepsilon$	$A' \to \varepsilon$	$A' \to$
B				$B \to e$	$B \to f$	

(6) $S \to (+) \mid AB$

$A \to a \mid \varepsilon$

$B \to b \mid \varepsilon$

$First(S) = \{c\} \cup First(AB)$

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

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

$First(S) = \{c\} \cup \{a, b, \varepsilon\}$

$= \{a, b, c, \varepsilon\}$

Computing Follow

(1) Follow(S) = {$} ——— (1)

~~S → αAβ is in the form~~
~~A → αBβ~~

(2) S → AB is in the form
A → αB
By rule 3
Everything in Follow(S) is in Follow(B) —(2)

S → AB is in the form
A → αBβ
By rule 2
Everything in First(β) − {ε} is in Follow(A) —(3)

By rule 3
Everything in Follow(S) is in Follow(A) ——— (4)

	First	Follow
S	{c, a, b, ε}	{$}
A	{a, ε}	{b, $}
B	{b, ε}	{$}

Predictive Parsing table

Non Terminals	Input Symbols					
	(	+	)	a	b	$
S	S→(T)			S→AB	S→AB	S→AB
A				A→a	A→ε	A→ε
B					B→b	B→ε