**EXPERIMENT – 7**

**CLASS: TE CMPN B**

**DATE: 15/02/ ‘22**

**NAME: Keegan Vaz**

**ROLL NO.: 28**

**AIM: TO DESIGN AND IMPLEMENT A PROGRAM FOR LL(1) PARSER**

**THEORY:**

**-** Role of a Syntax analyzer

**Ans:** A syntax analyzer or parser takes the input from a lexical analyzer in the form of token streams. The parser analyzes the source code (token stream) against the production rules to detect any errors in the code. The output of this phase is a parse tree.

**-** Compare Top down Parsing and Bottom up Parsing.

**Ans:**

|  |  |  |
| --- | --- | --- |
| Sr. No. | Top Down Parsing | Bottom Up Parsing |
|  | It is a parsing strategy that first looks at the highest level of the parse tree and works down the parse tree by using the rules of grammar. | It is a parsing strategy that first looks at the lowest level of the parse tree and works up the parse tree by using the rules of grammar. |
|  | Top-down parsing attempts to find the leftmost derivations for an input string. | Bottom-up parsing can be defined as an attempt to reduce the input string to the start symbol of a grammar. |
|  | In this parsing technique we start parsing from top (start symbol of parse tree) to down (the leaf node of parse tree) in a top-down manner. | In this parsing technique we start parsing from bottom (leaf node of parse tree) to up (the start symbol of parse tree) in a bottom-up manner. |
|  | This parsing technique uses Left Most Derivation. | This parsing technique uses Right Most Derivation. |
|  | It’s main decision is to select what production rule to use in order to construct the string. | It’s main decision is to select when to use a production rule to reduce the string to get the starting symbol. |

**-** Design of LL(k) parser

**Ans:** An LL Parser accepts LL grammar. LL grammar is a subset of context-free grammar but with some restrictions to get the simplified version, in order to achieve easy implementation.

LL grammar can be implemented by means of both algorithms namely, recursive-descent or table-driven.

LL parser is denoted as LL(k). The first L in LL(k) is parsing the input from left to right, the second L in LL(k) stands for left-most derivation and k itself represents the number of look aheads. Generally k = 1, so LL(k) may also be written as LL(1).

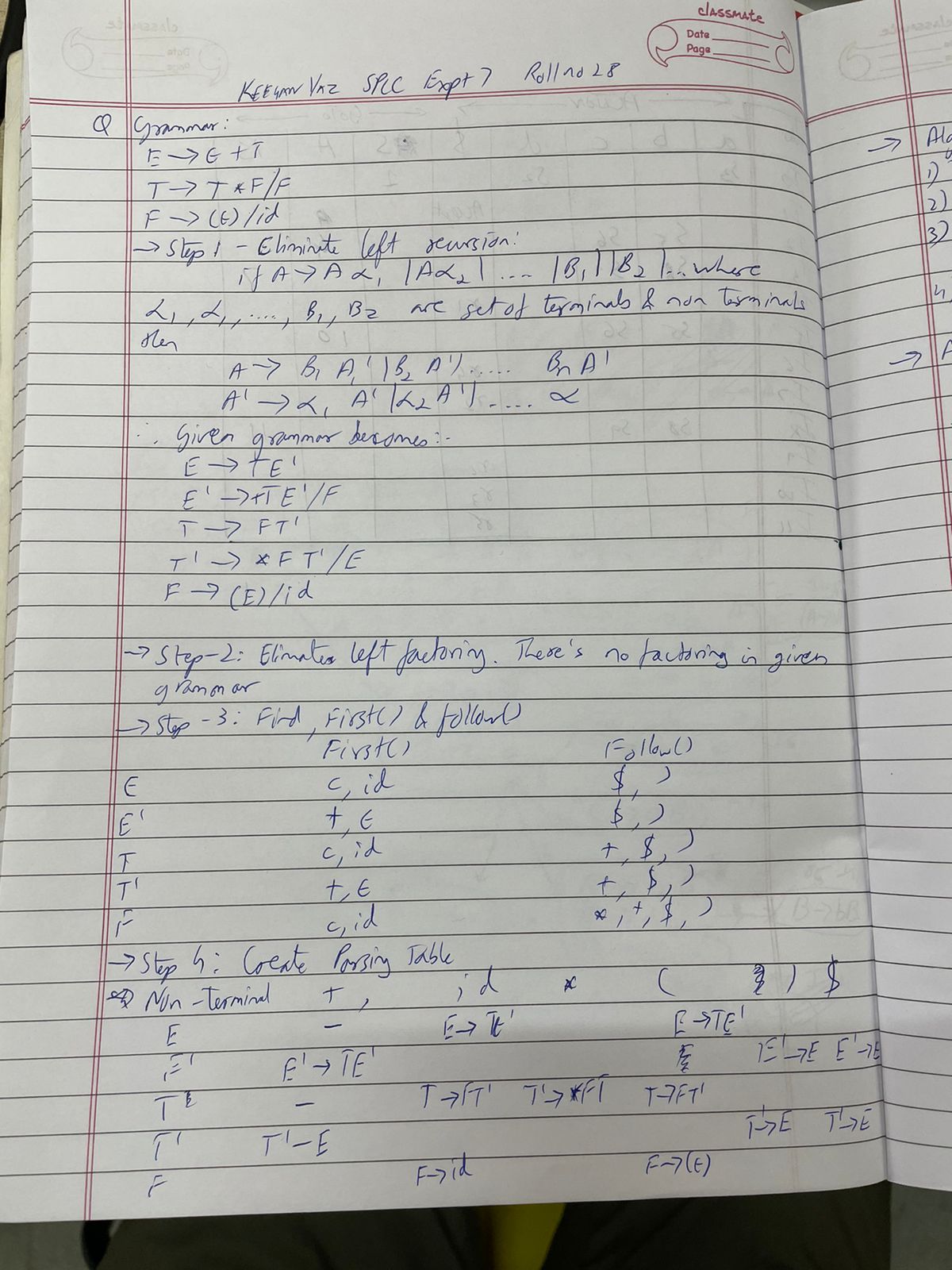
We may stick to deterministic LL(1) for parser explanation, as the size of the table grows exponentially with the value of k. Secondly, if a given grammar is not LL(1), then usually, it is not LL(k), for any given k.

A grammar G is LL(1) if A → α | β are two distinct productions of G:

* for no terminal, both α and β derive strings beginning with a.
* at most one of α and β can derive an empty string.
* if β → t, then α does not derive any string beginning with a terminal in FOLLOW(A).

**-** Construct LL(1) parser for a given grammar (hand written)

**Ans:**



**-** Pseudo code (Algorithm)

**Ans:**

**Input-**

1. stack = S //stack initially contains only S.
2. Input string = w$

where S is the start symbol of grammar, w is given string, and $ is used for the end of string.

1. PT is a parsing table of given grammar in the form of a matrix or 2D array.

**Output-**

Determines that given string can be produced by given grammar(parsing table) or not, if not then it produces an error.

**Steps -**

1. while(stack is not empty) {

// initially it is S

2. A = top symbol of stack;

//initially it is first symbol in string, it can be $ also

3. r = next input symbol of given string;

4. if (A∈T or A==$) {

5. if(A==r){

6. pop A from stack;

7. remove r from input;

8. }

9. Else

10. ERROR();

11. }

12. else if (A∈V) {

13. if(PT[A,r]= A⇢B1B2....Bk) {

14. pop A from stack;

// B1 on top of stack at final of this step

15. push Bk,Bk-1......B1 on stack

16. }

17. else if (PT[A,r] = error())

18. error();

19. }

20. }

// if the parser terminates without error()then the given string can be generated by the given parsing table.

**Given Grammar G: (Check whether it is LL(1)or not HAND WRITTEN) E E+T / T**

**T T\*F/F**

**F (E) / id**

**IMPLEMENTATION:**

**CODE** (C / Java)

import java.util.\*;

public class ll1 {

public static void main(String[] args) {

HashMap<String, String[]> Grammar = new LinkedHashMap<String, String[]>();

// Taking input of grammar

System.out.print("\nENTER THE NUMBER OF PRODUCTIONS: ");

int n;

try (Scanner sc = new Scanner(System.in)) {

n = sc.nextInt();

sc.nextLine();

for (int i = 0; i < n; i++) {

System.out.println("\nEnter the non-terminal: ");

String NT = sc.nextLine();

System.out.println("Enter the productions: ");

String[] productions = sc.nextLine().split("/");

Grammar.put(NT, productions);

}

}

System.out.println("\nGIVEN GRAMMAR IS: ");

printGrammar(Grammar);

// Calling function for eliminating Left Recursion

HashMap<String, String[]> freeofLR = eliminateLR(Grammar);

System.out.println("\nAFTER ELIMINATING LEFT RECURSION:");

printGrammar(freeofLR);

System.out.println("\nCALCULATION OF FIRST():");

String[] NT = new String[freeofLR.size()];

int i = 0;

for (String key : freeofLR.keySet()) {

NT[i] = key;

i++;

}

// Calling function for calculating FIRST()

String[] first = calculateFirst(freeofLR, NT);

System.out.println("\nCALCULATION OF FOLLOW():");

// Calling function for calculating FOLLOW()

String[] follow = calculateFollow(freeofLR, NT, first);

System.out.println("\nPARSING TABLE:");

// Finding the terminals

String t = findTerminals(first, follow, 0);

// Parsing table creation

String[][] pt = parsingTable(freeofLR, NT, first, follow, t);

// Check if grammar is LL1 or not

checkIfLL1(pt, first, t);

}

// Function for printing grammar

public static void printGrammar(HashMap<String, String[]> grammar) {

for (String key : grammar.keySet()) {

System.out.print(key + " ->");

for (int j = 0; j < grammar.get(key).length; j++) {

if (j == grammar.get(key).length - 1) {

System.out.print(grammar.get(key)[j]);

} else {

System.out.print(grammar.get(key)[j] + " / ");

}

}

System.out.println();

}

}

// Function for eliminating Left Recursion

public static HashMap<String, String[]> eliminateLR(HashMap<String, String[]> grammar) {

HashMap<String, String[]> freeOfLR = new LinkedHashMap<String, String[]>();

for (String key : grammar.keySet()) {

// Check if left recursion exists

if (key.equals(String.valueOf((grammar.get(key)[0]).charAt(0)))) {

// System.out.println("LR");

String NT = key;

String[] beta = { grammar.get(key)[1] + NT + "'" };

freeOfLR.put(NT, beta);

String[] temp = { grammar.get(key)[0].substring(1, grammar.get(key)[0].length()) + NT + "'/\u03A3" };

freeOfLR.put(NT + "'", temp);

} else {

String NT = key;

String[] beta = { grammar.get(key)[0] + "/" + grammar.get(key)[1] };

freeOfLR.put(NT, beta);

}

}

return freeOfLR;

}

// Function for calculating FIRST()

public static String[] calculateFirst(HashMap<String, String[]> grammar, String[] NT) {

String[] first = new String[grammar.size()];

int i = 0;

for (int j = NT.length - 1; j >= 0; j--) {

for (String key : grammar.keySet()) {

if (key.equals(NT[j])) {

String temp = grammar.get(key)[0].toString();

String temps[] = temp.split("/");

String value = "";

for (int k = 0; k < temps.length; k++) {

// FIRST() if alpha is epsilon

if (temps[k].charAt(0) == '\u03A3') {

value = value + "\u03A3";

}

// FIRST() if alpha is Non Terminal

else if (Character.isUpperCase(temps[k].charAt(0))) {

for (i = 0; i < NT.length; i++) {

if (temps[k].charAt(0) == NT[i].charAt(0)) {

value = value + first[i];

break;

}

}

}

// FIRST() if alpha is Terminal

else {

if (temps[k].equals("id")) {

value = value + temps[k];

} else {

value = value + temps[k].charAt(0);

}

}

if (k != temps.length - 1) {

value = value + ",";

}

}

first[j] = value;

}

}

}

// Printing FIRST() of all Non terminals

for (i = 0; i < first.length; i++) {

System.out.println("First(" + NT[i] + ") = " + first[i]);

}

return first;

}

// Function for calculating the FOLLOW() of non-terminals

private static String[] calculateFollow(HashMap<String, String[]> grammar, String[] NT, String[] first) {

String[] follow = new String[grammar.size()];

// Default value for FOLLOW() of first non-terminal

follow[0] = "$";

for (int i = 0; i < follow.length; i++) {

for (String key : grammar.keySet()) {

// Finding the position of element in the production whose FOLLOW() is to be

// calculated

int ind = grammar.get(key)[0].indexOf(NT[i]);

if (ind >= 0) {

if ((String.valueOf(grammar.get(key)[0].charAt(ind + 1)).equals("'"))

&& (NT[i].indexOf("'") >= 0)) {

ind++;

} else if ((String.valueOf(grammar.get(key)[0].charAt(ind + 1)).equals("'"))

&& (NT[i].indexOf("'") < 0)) {

ind = -1;

}

if (ind >= 0) {

// FOLLOW() if Beta is epsilon

if (ind == grammar.get(key)[0].length() - 1) {

int m;

for (m = 0; m < NT.length; m++) {

if (key.equals(NT[m])) {

break;

}

}

if (follow[i] != null) {

follow[i] = follow[i] + follow[m];

} else {

follow[i] = follow[m];

}

}

// FOLLOW() if Beta is Non-terminal

else if (Character.isUpperCase(grammar.get(key)[0].charAt(ind + 1))) {

int m;

String temp;

if (ind + 2 != grammar.get(key)[0].length()

&& String.valueOf(grammar.get(key)[0].charAt(ind + 2)).equals("'")) {

temp = String.valueOf(grammar.get(key)[0].charAt(ind + 1))

+ String.valueOf(grammar.get(key)[0].charAt(ind + 2));

} else {

temp = String.valueOf(grammar.get(key)[0].charAt(ind + 1));

}

for (m = 0; m < NT.length; m++) {

if (temp.equals(NT[m])) {

break;

}

}

// FOLLOW() if the first of non-terminal is epsilon

if (first[m].indexOf("\u03A3") > 0) {

int l;

for (l = 0; l < NT.length; l++) {

if (key.equals(NT[l])) {

break;

}

}

if (follow[i] != null) {

follow[i] = follow[i] + follow[l];

} else {

follow[i] = follow[l];

}

}

if (follow[0] != null) {

follow[i] = follow[i] + first[m];

} else {

follow[i] = first[m];

}

}

// FOLLOW() if Beta is terminal

else {

if ((String.valueOf(grammar.get(key)[0].charAt(ind + 1)).equals("/"))) {

break;

} else {

if (follow[i] != null) {

follow[i] = follow[i] + String.valueOf(grammar.get(key)[0].charAt(ind + 1));

} else {

follow[i] = String.valueOf(grammar.get(key)[0].charAt(ind + 1));

}

}

}

}

}

}

// Sorting and removal of duplicates

char temp[] = follow[i].toCharArray();

Arrays.sort(temp);

follow[i] = new String(temp);

int res\_ind = 1, ip\_ind = 1;

char arr[] = follow[i].toCharArray();

while (ip\_ind != arr.length) {

if (arr[ip\_ind] != arr[ip\_ind - 1]) {

arr[res\_ind] = arr[ip\_ind];

res\_ind++;

}

ip\_ind++;

}

follow[i] = new String(arr);

String str = follow[i];

str = str.substring(0, res\_ind);

follow[i] = str;

follow[i] = follow[i].replace(",", "");

follow[i] = follow[i].replace("\u03A3", "");

follow[i] = follow[i].replace("", ",");

str = follow[i];

follow[i] = str.substring(1, str.length() - 1);

}

// Printing the FOLLOW() for the non-terminals

for (int i = 0; i < follow.length; i++) {

System.out.println("Follow(" + NT[i] + ") = " + follow[i]);

}

return follow;

}

// Function for enlisting the non-terminals

private static String findTerminals(String[] first, String[] follow, int i) {

String t = Arrays.toString(first) + Arrays.toString(follow);

t = t.replace("[", "");

t = t.replace("]", "");

t = t.replace("\u03A3", "");

t = t.replace(",", "");

// Sorting and removal of duplicates

char temp[] = t.toCharArray();

Arrays.sort(temp);

t = new String(temp);

int res\_ind = 1, ip\_ind = 1;

char arr[] = t.toCharArray();

while (ip\_ind != arr.length) {

if (arr[ip\_ind] != arr[ip\_ind - 1]) {

arr[res\_ind] = arr[ip\_ind];

res\_ind++;

}

ip\_ind++;

}

t = new String(arr);

String str = t;

str = str.substring(0, res\_ind);

t = str;

t = t.replace("di", "id");

t = t.replace(" ", "");

// Printing the non-terminals

System.out.println("Non-Terminals Terminals");

for (i = 0; i < t.length() - 1; i++) {

if (i == 0) {

System.out.print(" ");

}

if (i != t.length() - 2) {

System.out.print(" " + t.charAt(i) + " ");

} else {

System.out.print(" " + t.charAt(i) + t.charAt(i + 1));

}

}

return t;

}

// Function for creating Parsing Table

private static String[][] parsingTable(HashMap<String, String[]> freeofLR, String[] NT, String[] first,

String[] follow,

String t) {

String[][] pt = new String[100][t.length() - 1];

int i = 0;

int n = 0;

String[] temps = new String[2];

for (String key : freeofLR.keySet()) {

// Splitting the productions for allocating separating positions in the parsing

// table

int in = freeofLR.get(key)[0].indexOf("/");

String str = Arrays.toString(freeofLR.get(key));

str = str.replace("[", "");

str = str.replace("]", "");

if (in > 0) {

temps = str.split("/");

} else {

temps[0] = str;

temps[1] = null;

}

for (int c = 0; c < temps.length; c++) {

if (temps[c] == null) {

break;

}

char temp = temps[c].charAt(0);

// Finding FOLLOW() of variable producing epsilon for the parsing table

if (temp == '\u03A3') {

for (i = 0; i < NT.length; i++) {

if (key == NT[i]) {

break;

}

}

String[] f = follow[i].split(",");

for (int k = 0; k < f.length; k++) {

int ind = t.indexOf(f[k]);

if (pt[n][ind] != null) {

pt[n][ind] = pt[n][ind] + "," + key + "->" + temps[c];

} else {

pt[n][ind] = key + "->" + temps[c];

}

}

}

// Finding FIRST() of the non-terminal and segregation in parsing table based

// on FIRST() values

else if (Character.isUpperCase(temp)) {

for (i = 0; i < NT.length; i++) {

if (temp == NT[i].charAt(0)) {

break;

}

}

String[] f = first[i].split(",");

for (int k = 0; k < f.length; k++) {

int ind = t.indexOf(f[k]);

if (pt[n][ind] != null) {

pt[n][ind] = pt[n][ind] + "," + key + "->" + temps[c];

} else {

pt[n][ind] = key + "->" + temps[c];

}

}

}

// Allocation of location based on terminal

else {

int ind = t.indexOf(temp);

if (pt[n][ind] != null) {

pt[n][ind] = pt[n][ind] + "," + key + "->" + temps[c];

} else {

pt[n][ind] = key + "->" + temps[c];

}

}

}

n++;

}

// Printing Parsing table

System.out.println();

for (int l = 0; l < n; l++) {

System.out.print(NT[l] + ": ");

for (int h = 0; h < t.length() - 1; h++) {

if (pt[l][h] == null) {

System.out.print("---- ");

} else {

System.out.print(pt[l][h] + " ");

}

}

System.out.println();

}

return pt;

}

// Function to check if Grammar is LL1 or not

private static void checkIfLL1(String[][] pt, String[] first, String t) {

int m = 0;

for (int i = 0; i < first.length; i++) {

for (int j = 0; j < t.length() - 1; j++) {

String s = pt[i][j];

// Check if more than one entry is present in a single cell

if (s != null && s.indexOf(",") != -1) {

System.out.println("\nGRAMMAR IS NOT LL(1)");

m = 1;

break;

}

}

}

// If single entry, then Grammar is LL1

if (m != 1) {

System.out.println("\nGRAMMAR IS LL(1)");

}

}

}

**OUTPUT:**

Input: Take a grammar from the user (scanf)

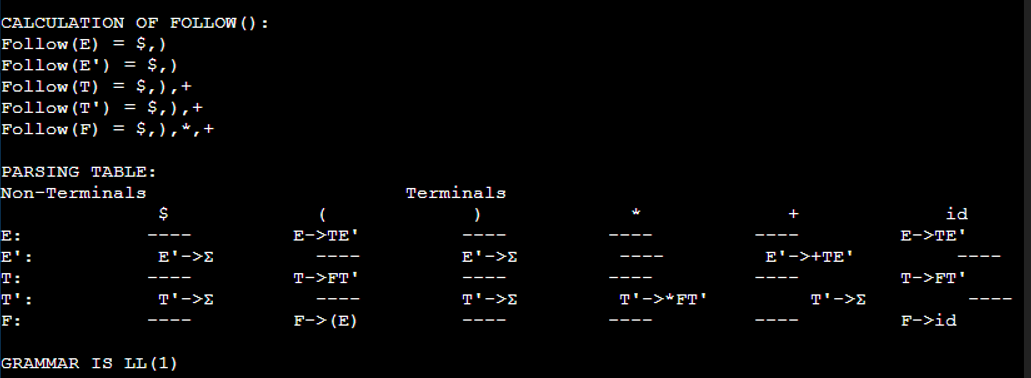
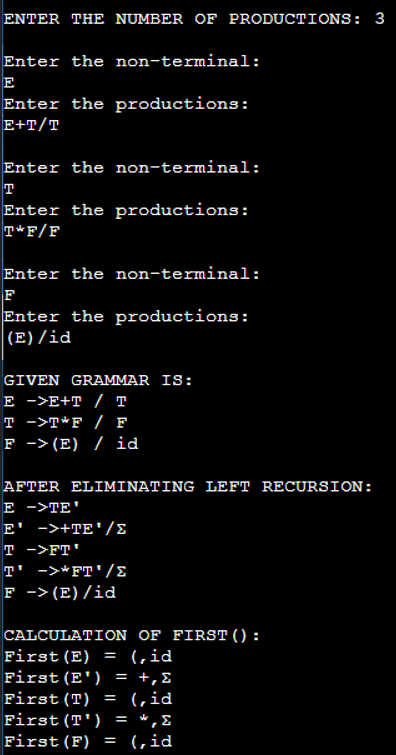
Output: Display

**-** Left Recursion free grammar

**-** First and Follow sets

**-** The LL(1) Parsing Table

**-** Check the given grammar is LL(1)



**CONCLUSION:**

LL(1) Parser is designed to create Parse Tree using top down approach.