**Experiment No - 5**

**Aim**: Implement a program for constructing

1. LL(1) Parser
2. Predictive Parser

**Date:**

**Competency and Practical Skills:**

* Understanding Parsers and its role in compiler construction
* Ability to write first and follow for given grammar
* Ability to develop LL(1) and predictive parser using top down parsing approach

**Relevant CO:** CO2

**Objectives:**

By the end of this experiment, the students should be able to:

* + Understand the concept parsers and its significance in compiler construction
  + Write first and follow set for given grammar
  + Implement a LL(1) and predictive grammar using top down parser

**Software/Equipment:** C compiler

**Theory:**

* + **LL(1) Parsing:** Here the 1st **L** represents that the scanning of the Input will be done from the Left to Right manner and the second **L** shows that in this parsing technique, we are going to use the Left most Derivation Tree. And finally, the **1** represents the number of look-ahead, which means how many symbols are you going to see when you want to make a decision.

**Essential conditions to check first are as follows:**

* 1. The grammar is free from left recursion.
  2. The grammar should not be ambiguous.
  3. The grammar has to be left factored in so that the grammar is deterministic grammar. These conditions are necessary but not sufficient for proving a LL(1) parser.

**Algorithm to construct LL(1) Parsing Table:**

**Step 1:** First check all the essential conditions mentioned above and go to step 2.

**Step 2:** Calculate First() and Follow() for all non-terminals.

* 1. [First**(**](https://www.geeksforgeeks.org/first-set-in-syntax-analysis/)**):** If there is a variable, and from that variable, if we try to drive all the strings then the beginning Terminal Symbol is called the First.
  2. [Follow(](https://www.geeksforgeeks.org/follow-set-in-syntax-analysis/)): What is the Terminal Symbol which follows a variable in the process of derivation.

**Step 3:** For each production A –> α. (A tends to alpha)

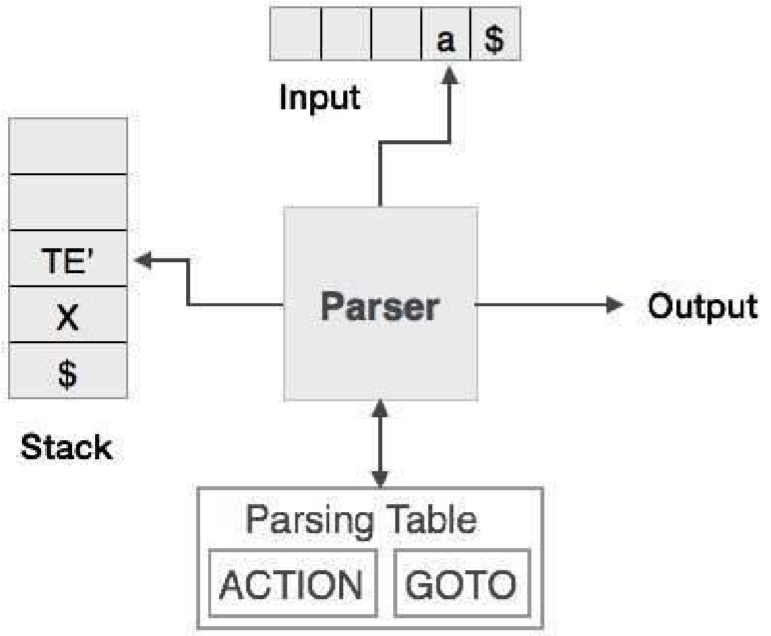
* 1. Find First(α) and for each terminal in First(α), make entry A –> α in the table.
  2. If First(α) contains ε (epsilon) as terminal, then find the Follow(A) and for each terminal in Follow(A), make entry A –> ε in the table.
  3. If the First(α) contains ε and Follow(A) contains $ as terminal, then make entry A –> ε in the table for the $.

To construct the parsing table, we have two functions:

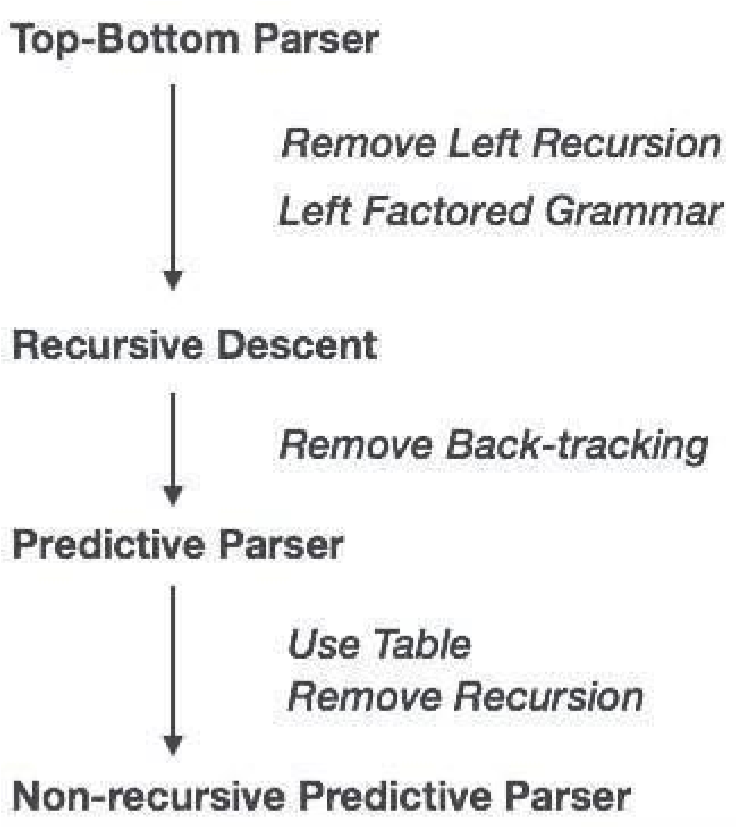
In the table, rows will contain the Non-Terminals and the column will contain the Terminal Symbols. All the **Null Productions** of the Grammars will go under the Follow elements and the remaining productions will lie under the elements of the First set.

* + **Predictive Parser**

Predictive parser is a recursive descent parser, which has the capability to predict which production is to be used to replace the input string. The predictive parser does not suffer from backtracking. To accomplish its tasks, the predictive parser uses a look-ahead pointer, which points to the next input symbols. To make the parser back-tracking free, the predictive parser puts some constraints on the grammar and accepts only a class of grammar known as LL(k) grammar.



Predictive parsing uses a stack and a parsing table to parse the input and generate a parse tree. Both the stack and the input contains an end symbol **$** to denote that the stack is empty and the input is consumed. The parser refers to the parsing table to take any decision on the input and stack element combination.



In recursive descent parsing, the parser may have more than one production to choose from for a single instance of input, whereas in predictive parser, each step has at most one production to choose. There might be instances where there is no production matching the input string, making the parsing procedure to fail.

***Program-1*:**

#include<stdio.h>

#include<string.h>

#define TSIZE 128

// table[i][j] stores

// the index of production that must be applied on

// ith varible if the input is // jth nonterminal int table[100][TSIZE];

// stores all list of terminals

// the ASCII value if use to index terminals

// terminal[i] = 1 means the character with

// ASCII value is a terminal char terminal[TSIZE];

// stores all list of terminals

// only Upper case letters from 'A' to 'Z'

// can be nonterminals

// nonterminal[i] means ith alphabet is present as

// nonterminal is the grammar char nonterminal[26];

// structure to hold each production

// str[] stores the production // len is the length of production struct product { char str[100]; int len;

}pro[20];

// no of productions in form A->ß int no\_pro; char first[26][TSIZE]; char follow[26][TSIZE];

// stores first of each production in form A->ß char first\_rhs[100][TSIZE]; // check if the symbol is nonterminal int isNT(char c) { return c >= 'A' && c <= 'Z';

}

// reading data from the file void readFromFile() { FILE\* fptr;

fptr = fopen("text.txt", "r"); char buffer[255];

int i; int j;

while (fgets(buffer, sizeof(buffer), fptr)) { printf("%s", buffer); j = 0;

nonterminal[buffer[0] - 'A'] = 1; for (i = 0; i < strlen(buffer) - 1; ++i) { if (buffer[i] == '|') {

++no\_pro;

pro[no\_pro - 1].str[j] = '\0'; pro[no\_pro - 1].len = j;

pro[no\_pro].str[0] = pro[no\_pro - 1].str[0]; pro[no\_pro].str[1] = pro[no\_pro - 1].str[1]; pro[no\_pro].str[2] = pro[no\_pro - 1].str[2]; j = 3;

} else { pro[no\_pro].str[j] = buffer[i];

++j; if (!isNT(buffer[i]) && buffer[i] != '-' && buffer[i] != '>') { terminal[buffer[i]] = 1;

}

}

}

pro[no\_pro].len = j;

++no\_pro;

}

}

void add\_FIRST\_A\_to\_FOLLOW\_B(char A, char B)

{

int i;

for (i = 0; i < TSIZE; ++i)

{ if (i != '^'){

follow[B - 'A'][i] = follow[B - 'A'][i] || first[A - 'A'][i];

}

}

void add\_FOLLOW\_A\_to\_FOLLOW\_B(char A, char B)

{

int i;

for (i = 0; i < TSIZE; ++i)

{ if (i != '^')

follow[B - 'A'][i] = follow[B - 'A'][i] || follow[A - 'A'][i];

}

}

void FOLLOW()

{ int t = 0;

int i, j, k, x;

while (t++ < no\_pro)

{ for (k = 0; k < 26; ++k) { if (!nonterminal[k]) continue; char nt = k + 'A'; for (i = 0; i < no\_pro; ++i) { for (j = 3; j < pro[i].len; ++j) {

if (nt == pro[i].str[j]) {

for (x = j + 1; x < pro[i].len; ++x) { char sc = pro[i].str[x]; if (isNT(sc)) {

add\_FIRST\_A\_to\_FOLLOW\_B(sc, nt); if (first[sc - 'A']['^']) continue;

} else { follow[nt - 'A'][sc] = 1;

} break; }

if (x == pro[i].len)

add\_FOLLOW\_A\_to\_FOLLOW\_B(pro[i].str[0], nt);

}

}

}

}

}

}

void add\_FIRST\_A\_to\_FIRST\_B(char A, char B) { int i;

for (i = 0; i < TSIZE; ++i) { if (i != '^') {

first[B - 'A'][i] = first[A - 'A'][i] || first[B - 'A'][i];

}

}

} void FIRST() {

int i, j;

int t = 0; while (t < no\_pro) { for (i = 0; i < no\_pro; ++i) { for (j = 3; j < pro[i].len; ++j) { char sc = pro[i].str[j];

if (isNT(sc)) {

add\_FIRST\_A\_to\_FIRST\_B(sc, pro[i].str[0]);

if (first[sc - 'A']['^']) continue;

} else {

first[pro[i].str[0] - 'A'][sc] = 1;

} break; }

if (j == pro[i].len) first[pro[i].str[0] - 'A']['^'] = 1;

}

++t;

}

}

void add\_FIRST\_A\_to\_FIRST\_RHS\_\_B(char A, int B) { int i;

for (i = 0; i < TSIZE; ++i) { if (i != '^') first\_rhs[B][i] = first[A - 'A'][i] || first\_rhs[B][i]; }

}

// Calculates FIRST(ß) for each A->ß

void FIRST\_RHS() { int i, j;

int t = 0; while (t < no\_pro) { for (i = 0; i < no\_pro; ++i) { for (j = 3; j < pro[i].len; ++j) { char sc = pro[i].str[j];

if (isNT(sc)) {

add\_FIRST\_A\_to\_FIRST\_RHS\_\_B(sc, i); if (first[sc - 'A']['^'])

continue; } else { first\_rhs[i][sc] = 1;

}

break;

} if (j == pro[i].len) first\_rhs[i]['^'] = 1;

}

++t;

}

}

int main() { readFromFile();

follow[pro[0].str[0] - 'A']['$'] = 1;

FIRST();

FOLLOW(); FIRST\_RHS();

int i, j, k;

// display first of each variable printf("\n"); for (i = 0; i < no\_pro; ++i) { if (i == 0 || (pro[i - 1].str[0] != pro[i].str[0])) { char c = pro[i].str[0]; printf("FIRST OF %c: ", c); for (j = 0; j < TSIZE; ++j) { if (first[c - 'A'][j]) { printf("%c ", j);

}

} printf("\n");

}

}

// display follow of each variable printf("\n"); for (i = 0; i < no\_pro; ++i) { if (i == 0 || (pro[i - 1].str[0] != pro[i].str[0])) {

char c = pro[i].str[0]; printf("FOLLOW OF %c: ", c); for (j = 0; j < TSIZE; ++j) { if (follow[c - 'A'][j]) { printf("%c ", j);

} } printf("\n");

}

}

// display first of each variable ß

// in form A->ß printf("\n"); for (i = 0; i < no\_pro; ++i) { printf("FIRST OF %s: ", pro[i].str);

for (j = 0; j < TSIZE; ++j) { if (first\_rhs[i][j]) { printf("%c ", j);

}

} printf("\n");

}

// the parse table contains '$'

// set terminal['$'] = 1

// to include '$' in the parse table terminal['$'] = 1;

// the parse table do not read '^'

// as input

// so we set terminal['^'] = 0 // to remove '^' from terminals terminal['^'] = 0; // printing parse table printf("\n");

printf("\n\t\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* LL(1) PARSING TABLE \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf("\t--------------------------------------------------------\n"); printf("%-10s", ""); for (i = 0; i < TSIZE; ++i) { if (terminal[i]) printf("%-10c", i);

} printf("\n"); int p = 0;

for (i = 0; i < no\_pro; ++i) { if (i != 0 && (pro[i].str[0] != pro[i - 1].str[0])) p = p + 1;

for (j = 0; j < TSIZE; ++j) { if (first\_rhs[i][j] && j != '^') { table[p][j] = i + 1;

}

else if (first\_rhs[i]['^']) { for (k = 0; k < TSIZE; ++k) { if (follow[pro[i].str[0] - 'A'][k]) { table[p][k] = i + 1; }

}

}

}

} k = 0;

for (i = 0; i < no\_pro; ++i) { if (i == 0 || (pro[i - 1].str[0] != pro[i].str[0])) {

printf("%-10c", pro[i].str[0]);

for (j = 0; j < TSIZE; ++j) { if (table[k][j]) {

printf("%-10s", pro[table[k][j] - 1].str);

}

else if (terminal[j]) { printf("%-10s", "");

}

}

++k; printf("\n");

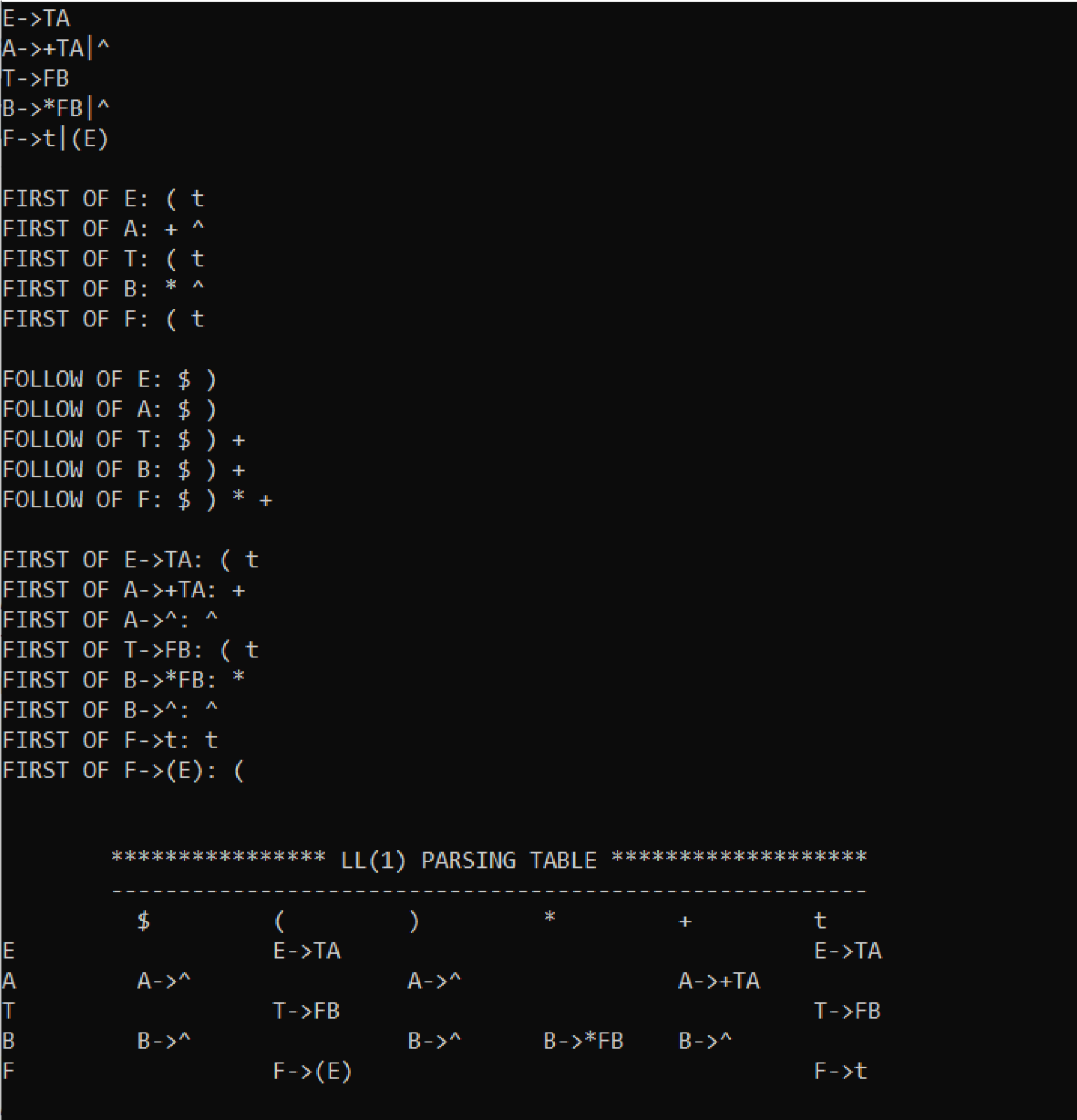
}

}

}

**Observations and Conclusion:**

***Program -1:***



In the above example, the grammar is given as input and first set and follow set of nonterminals are identified.Further the LL1 parsing table is constructed .

**Quiz:**

1. What is a parser and state the Role of it?
2. Types of parsers? Examples to each.
3. What are the Tools available for implementation?
4. How do you calculate FIRST(),FOLLOW() sets used in Parsing Table construction?
5. Name the most powerful parser.

**Suggested Reference:**

1. Introduction to Automata Theory, Languages and Computation by John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman.
2. Geeks for geeks: <https://www.geeksforgeeks.org/construction-of-ll1-parsing-table/>
3. http://www.cs.ecu.edu/karl/5220/spr16/Notes/Top-down/LL1.html

**References used by the students:**

**Rubric wise marks obtained:**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Rubrics** | **Knowledge of Parsing**  **techniques**  **(2)** | | **Problem implementati**  **on (2)** | | **Code**  **Quality (2)** | | **Completeness and accuracy**  **(2)** | | **Presentation**  **(2)** | | **Total** |
| **Good**  **(2)** | **Avg.**  **(1)** | **Good**  **(2)** | **Avg.**  **(1)** | **Good**  **(2)** | **Avg.**  **(1)** | **Good**  **(2)** | **Avg.**  **(1)** | **Good**  **(2)** | **Avg.**  **(1)** |
| **Marks** |  |  |  |  |  |  |  |  |  |  |  |