Premier University Chittagong

Department of Computer Science and Engineering

Course Title : Compiler Construction Lab

Course Code: CSE 454

Report No : 10

Report Title : Write a Program to Construct a Predictive Parsing Table for a given Context-Free Grammar

Submission Date: 25.11.2025

Submitted By

Name : Rayanul Kader Chowdhury Abid
ID : 210401020 2162
Semester : 8th A Section

Submitted To

Ms. Tanni Dhoom
Assistant Professor
Department of Computer Science and Engineering

Experiment No: 10

Experiment Name: Write a Program to Construct a Predictive Parsing Table for a given Context-Free Grammar.

Objectives:

To take a user-defined context-free grammar, process it by eliminating left recursion and performing left factoring, compute the FIRST and FOLLOW sets for all non-terminals, and finally construct an LL(1) predictive parsing table using these computed sets. The goal is to understand how each of these steps contributes to building a valid parsing table and how predictive parsers use these tables in syntax analysis.

Algorithm

1. Read the number of grammar rules and store each rule along with its productions.
2. Display the grammar as entered by the user.
3. For each non-terminal, check if any production begins with the same non-terminal; if so, eliminate immediate left recursion by splitting the rule into two separate rules.
4. After removing left recursion, scan each rule for common prefixes among productions and apply left factoring to extract the shared prefix into a new non-terminal.
5. Initialize FIRST sets for all non-terminals.
6. For each production of each rule, compute FIRST by checking production symbols from left to right and applying FIRST rules until a terminal or a non-epsilon non-terminal is found.
7. Initialize FOLLOW sets, inserting the end marker symbol into the FOLLOW of the start symbol.
8. Scan each production and apply FOLLOW rules: add FIRST of the suffix (excluding epsilon) to FOLLOW of the symbol, and if epsilon is possible or the symbol is at the end, add FOLLOW of the left-hand non-terminal.
9. Keep repeating FOLLOW computation until no set changes.

10. Collect all terminal symbols and initialize an empty parsing table.

11. For each production, compute FIRST of the production; for all terminals in this FIRST set, insert the production in the corresponding table cell.

12. If FIRST contains epsilon, insert the production into all cells corresponding to terminals in the FOLLOW set.

13. Display the final LL(1) predictive parsing table.

Code:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#define SIZE 100
#define MAX_PROD 50
#define MAX_RULES 50
#define MAX_SYMBOLS 100
typedef struct Rule {
char nonTerminal[SIZE];
char prods[MAX_PROD][SIZE];
int prodCount;
} Rule;
Rule rules[MAX_RULES];
int ruleCount = 0;
char firstSets[MAX_RULES][MAX_SYMBOLS][SIZE];
int firstCount[MAX_RULES];
char followSets[MAX_RULES][MAX_SYMBOLS][SIZE];
int followCount[MAX_RULES];
char terminals[MAX_SYMBOLS][SIZE];
```

```
int termCount = 0;
char parsingTable[MAX_RULES][MAX_SYMBOLS][SIZE];
int isTerminalChar(char c) {
return !isupper((unsigned char)c) && c != '\'' && !isdigit((unsigned char)c);
}
int findRuleIndex(char *nt) {
for (int i = 0; i < ruleCount; i++) {
if (strcmp(rules[i].nonTerminal, nt) == 0)
return i;
}
return -1;
}
int inSet(char set[MAX_SYMBOLS][SIZE], int count, char *sym) {
for (int i = 0; i < count; i++)
if (strcmp(set[i], sym) == 0) return 1;
return 0;
}
void addToSet(char set[MAX_SYMBOLS][SIZE], int *count, char *sym) {
if (!inSet(set, *count, sym) && *count < MAX_SYMBOLS && strlen(sym) > 0) {
strcpy(set[*count], sym);
(*count)++;
}
}
void addAllToSetExceptEpsilon(char set1[MAX_SYMBOLS][SIZE], int *count1,
char set2[MAX_SYMBOLS][SIZE], int count2) {
for (int i = 0; i < count2; i++) {
if (strcmp(set2[i], "epsilon") != 0)
addToSet(set1, count1, set2[i]);
```

```
}
}
3
int extractSymbol(const char *str, int pos, char *out) {
int len = 0;
int slen = strlen(str);
if (pos >= slen) { out[0] = '\0'; return 0; }
if (isupper((unsigned char)str[pos])) {
out[len++] = str[pos];
int i = pos + 1;
while (i < slen && (str[i] == '\'' || isdigit((unsigned char)str[i]))) {
out[len++] = str[i];
i++;
}
out[len] = '\0';
return len;
} else {
out[0] = str[pos];
out[1] = '\0';
return 1;
}
}
void eliminateLeftRecursion() {
printf("\n ELIMINATING LEFT RECURSION \n");
int originalRuleCount = ruleCount;
for (int i = 0; i < originalRuleCount; i++) {
Rule *currentRule = &rules[i];
char alpha[MAX_PROD][SIZE];
```

```
char beta[MAX_PROD][SIZE];
int alphaCount = 0, betaCount = 0;
for (int j = 0; j < currentRule->prodCount; j++) {
if (currentRule->prods[j][0] == currentRule->nonTerminal[0]) {
if (strlen(currentRule->prods[j]) == 1)
strcpy(alpha[alphaCount++], "epsilon");
else
strcpy(alpha[alphaCount++], currentRule->prods[j] + 1);
} else {
strcpy(beta[betaCount++], currentRule->prods[j]);
}
}
if (alphaCount == 0) continue;
printf("Found left recursion in %s\n", currentRule->nonTerminal);
char newNT[SIZE];
snprintf(newNT, SIZE, "%s'", currentRule->nonTerminal);
currentRule->prodCount = 0;
for (int j = 0; j < betaCount; j++) {
if (strcmp(beta[j], "epsilon") == 0)
snprintf(currentRule->prods[currentRule->prodCount++], SIZE, "%s", newNT);
else
snprintf(currentRule->prods[currentRule->prodCount++], SIZE, "%s%s", beta[j], newNT);
}
Rule newRule;
strcpy(newRule.nonTerminal, newNT);
newRule.prodCount = 0;
for (int j = 0; j < alphaCount; j++) {
if (strcmp(alpha[j], "epsilon") == 0)
```

```
snprintf(newRule.prods[newRule.prodCount++], SIZE, "%s", newNT);
else
snprintf(newRule.prods[newRule.prodCount++], SIZE, "%s%s", alpha[j], newNT);
}
strcpy(newRule.prods[newRule.prodCount++], "epsilon");
rules[ruleCount++] = newRule;
printf("Created new rule: %s -> ", newNT);
for (int j = 0; j < newRule.prodCount; j++) {
printf("%s", newRule.prods[j]);
if (j < newRule.prodCount - 1) printf(" | ");
}
printf("\n");
}
}
int commonPrefixLength(char *s1, char *s2) {
int len = 0;
while (s1[len] && s2[len] && s1[len] == s2[len]) len++;
return len;
}
void leftFactorGrammar() {
printf("\n PERFORMING LEFT FACTORING \n");
int originalRuleCount = ruleCount;
int suffixCounter = 1;
for (int i = 0; i < originalRuleCount; i++) {
Rule *currentRule = &rules[i];
if (currentRule->prodCount <= 1) continue;
int changed;
do {
```

```
changed = 0;
for (int j = 0; j < currentRule->prodCount - 1 && !changed; j++) {
int groupSize = 1;
for (int k = j + 1; k < currentRule->prodCount; k++) {
if (currentRule->prods[j][0] == currentRule->prods[k][0]) groupSize++;
else break;
}
if (groupSize > 1) {
int prefixLen = strlen(currentRule->prods[j]);
for (int k = j + 1; k < j + groupSize; k++) {
int currentPrefix = commonPrefixLength(currentRule->prods[j], currentRule->prods[k]);
if (currentPrefix < prefixLen) prefixLen = currentPrefix;
}
if (prefixLen > 0) {
printf("Left factoring rule %s with prefix length %d\n",
currentRule->nonTerminal, prefixLen);
char newNT[SIZE];
snprintf(newNT, SIZE, "%s%d", currentRule->nonTerminal, suffixCounter++);
Rule newRule;
strcpy(newRule.nonTerminal, newNT);
newRule.prodCount = 0;
char commonPrefix[SIZE];
strncpy(commonPrefix, currentRule->prods[j], prefixLen);
commonPrefix[prefixLen] = '\0';
for (int k = j; k < j + groupSize; k++) {
if (strlen(currentRule->prods[k]) == prefixLen)
strcpy(newRule.prods[newRule.prodCount++], "epsilon");
else
```

```
strcpy(newRule.prods[newRule.prodCount++], currentRule->prods[k] + prefixLen);
}
strcpy(currentRule->prods[j], commonPrefix);
strcat(currentRule->prods[j], newNT);
for (int k = j + 1; k < currentRule->prodCount - groupSize + 1; k++)
strcpy(currentRule->prods[k], currentRule->prods[k + groupSize - 1]);
currentRule->prodCount -= (groupSize - 1);
rules[ruleCount++] = newRule;
printf("Created new rule: %s -> ", newNT);
for (int k = 0; k < newRule.prodCount; k++) {
printf("%s", newRule.prods[k]);
if (k < newRule.prodCount - 1) printf(" | ");
}
printf("\n");
changed = 1;
break;
}
}
}
} while (changed);
}
}
void computeFirstForProduction(char *production, char first[MAX_SYMBOLS][SIZE], int *count) {
if (strcmp(production, "epsilon") == 0) {
addToSet(first, count, "epsilon");
return;
}
int pos = 0, allHaveEpsilon = 1, len = strlen(production);
```

```
char symbol[SIZE];
while (pos < len && allHaveEpsilon) {
int slen = extractSymbol(production, pos, symbol);
if (slen == 0) break;
if (isTerminalChar(symbol[0]) && symbol[1] == '\0') {
addToSet(first, count, symbol);
allHaveEpsilon = 0;
} else {
int symIndex = findRuleIndex(symbol);
if (symIndex == -1) {
addToSet(first, count, symbol);
allHaveEpsilon = 0;
} else {
addAllToSetExceptEpsilon(first, count, firstSets[symIndex], firstCount[symIndex]);
if (!inSet(firstSets[symIndex], firstCount[symIndex], "epsilon"))
allHaveEpsilon = 0;
}
}
pos += slen;
}
if (allHaveEpsilon) addToSet(first, count, "epsilon");
}
void computeFirstSets() {
printf("\n COMPUTING FIRST SETS \n");
for (int i = 0; i < ruleCount; i++) firstCount[i] = 0;
int changed;
do {
changed = 0;
```

```
for (int i = 0; i < ruleCount; i++) {
int oldCount = firstCount[i];
for (int j = 0; j < rules[i].prodCount; j++)
computeFirstForProduction(rules[i].prods[j], firstSets[i], &firstCount[i]);
if (firstCount[i] != oldCount) changed = 1;
}
} while (changed);
for (int i = 0; i < ruleCount; i++) {
printf("FIRST(%s) = { ", rules[i].nonTerminal);
for (int j = 0; j < firstCount[i]; j++) {
printf("%s", firstSets[i][j]);
if (j < firstCount[i] - 1) printf(", ");
}
printf(" }\n");
}
}
void computeFirstOfString(char *str, char first[MAX_SYMBOLS][SIZE], int *count) {
*count = 0;
if (strlen(str) == 0) { addToSet(first, count, "epsilon"); return; }
int pos = 0, allHaveEpsilon = 1;
char symbol[SIZE];
int len = strlen(str);
while (pos < len && allHaveEpsilon) {
int slen = extractSymbol(str, pos, symbol);
if (slen == 0) break;
if (isTerminalChar(symbol[0]) && symbol[1] == '\0') {
addToSet(first, count, symbol);
allHaveEpsilon = 0;
```

```
} else {
int symIndex = findRuleIndex(symbol);
if (symIndex == -1) {
addToSet(first, count, symbol);
allHaveEpsilon = 0;
} else {
addAllToSetExceptEpsilon(first, count, firstSets[symIndex], firstCount[symIndex]);
if (!inSet(firstSets[symIndex], firstCount[symIndex], "epsilon"))
allHaveEpsilon = 0;
}
}
pos += slen;
}
if (allHaveEpsilon) addToSet(first, count, "epsilon");
}
void computeFollowSets() {
printf("\n COMPUTING FOLLOW SETS \n");
for (int i = 0; i < ruleCount; i++) followCount[i] = 0;
addToSet(followSets[0], &followCount[0], "$");
int changed, iterations = 0;
do {
changed = 0; iterations++;
for (int i = 0; i < ruleCount; i++) {
for (int j = 0; j < rules[i].prodCount; j++) {
char *prod = rules[i].prods[j];
int plen = strlen(prod), pos = 0;
while (pos < plen) {
char curSym[SIZE];
```

```
int slen = extractSymbol(prod, pos, curSym);
if (slen == 0) break;
int curIndex = findRuleIndex(curSym);
if (curIndex != -1) {
char beta[SIZE] = "";
if (pos + slen < plen) strcpy(beta, prod + pos + slen);
char firstBeta[MAX_SYMBOLS][SIZE];
int firstBetaCount = 0;
computeFirstOfString(beta, firstBeta, &firstBetaCount);
int oldCount = followCount[curIndex];
addAllToSetExceptEpsilon(followSets[curIndex], &followCount[curIndex],
firstBeta, firstBetaCount);
if (followCount[curIndex] != oldCount) changed = 1;
if (firstBetaCount == 0 || inSet(firstBeta, firstBetaCount, "epsilon") || strlen(beta) == 0) {
oldCount = followCount[curIndex];
for (int m = 0; m < followCount[i]; m++)
addToSet(followSets[curIndex], &followCount[curIndex], followSets[i][m]);
if (followCount[curIndex] != oldCount) changed = 1;
}
}
pos += slen;
}
}
}
if (iterations > 1000) break;
} while (changed);
printf("Follow sets computed in %d iterations\n", iterations);
for (int i = 0; i < ruleCount; i++) {
```

```
printf("FOLLOW(%s) = { ", rules[i].nonTerminal);
for (int j = 0; j < followCount[i]; j++) {
printf("%s", followSets[i][j]);
if (j < followCount[i] - 1) printf(", ");
}
printf(" }\n");
}
}
int isTerminalString(char *sym) {
return !isupper((unsigned char)sym[0]) || strcmp(sym, "epsilon") == 0;
}
void collectTerminals() {
termCount = 0;
for (int i = 0; i < ruleCount; i++) {
for (int j = 0; j < rules[i].prodCount; j++) {
char *prod = rules[i].prods[j];
int pos = 0;
char sym[SIZE];
while (pos < strlen(prod)) {
int slen = extractSymbol(prod, pos, sym);
if (slen == 0) break;
if (isTerminalString(sym) && !inSet(terminals, termCount, sym))
addToSet(terminals, &termCount, sym);
pos += slen;
}
}
}
addToSet(terminals, &termCount, "$");
```

```
}
void initializeParsingTable() {
for (int i = 0; i < MAX_RULES; i++)
for (int j = 0; j < MAX_SYMBOLS; j++)
parsingTable[i][j][0] = '\0';
}
int findTerminalIndex(char *sym) {
for (int i = 0; i < termCount; i++)
if (strcmp(terminals[i], sym) == 0) return i;
return -1;
}
void constructParsingTable() {
printf("\n CONSTRUCTING PARSING TABLE \n");
collectTerminals();
initializeParsingTable();
printf("Terminals: ");
for (int i = 0; i < termCount; i++) {
printf("%s ", terminals[i]);
}
printf("\n");
for (int i = 0; i < ruleCount; i++) {
Rule *rule = &rules[i];
for (int j = 0; j < rule->prodCount; j++) {
char *prod = rule->prods[j];
char firstProd[MAX_SYMBOLS][SIZE];
int firstProdCount = 0;
computeFirstOfString(prod, firstProd, &firstProdCount);
for (int k = 0; k < firstProdCount; k++) {
```

```
if (strcmp(firstProd[k], "epsilon") != 0) {
int col = findTerminalIndex(firstProd[k]);
if (col != -1) {
if (strlen(parsingTable[i][col]) > 0) {
printf("Conflict: Multiple entries for [%s][%s]\n",
rule->nonTerminal, firstProd[k]);
}
strcpy(parsingTable[i][col], prod);
}
} else {
for (int m = 0; m < followCount[i]; m++) {
int col = findTerminalIndex(followSets[i][m]);
if (col != -1) {
if (strlen(parsingTable[i][col]) > 0) {
printf("Conflict: Multiple entries for [%s][%s]\n",
rule->nonTerminal, followSets[i][m]);
}
strcpy(parsingTable[i][col], prod);
}
}
}
}
}
}
void displayParsingTable() {
printf("\n LL(1) PARSING TABLE \n\n");
int colWidths[MAX_SYMBOLS];
colWidths[0] = 15;
```

```
for (int i = 0; i < termCount; i++) {
colWidths[i+1] = strlen(terminals[i]) + 2;
if (colWidths[i+1] < 8) colWidths[i+1] = 8;
}
printf("+");
for (int i = 0; i <= termCount; i++) {
for (int j = 0; j < colWidths[i]; j++) printf("-");
if (i < termCount) printf("+");
else printf("+\n");
}
printf("|%-*s", colWidths[0], "Non-Terminal");
for (int i = 0; i < termCount; i++) {
printf("| %-*s", colWidths[i+1]-2, terminals[i]);
}
printf("|\n");
printf("+");
for (int i = 0; i <= termCount; i++) {
for (int j = 0; j < colWidths[i]; j++) printf("-");
if (i < termCount) printf("+");
else printf("+\n");
}
for (int i = 0; i < ruleCount; i++) {
printf("|%-*s", colWidths[0], rules[i].nonTerminal);
for (int j = 0; j < termCount; j++) {
printf("|");
if (strlen(parsingTable[i][j]) > 0) {
char display[SIZE];
if (strcmp(parsingTable[i][j], "epsilon") == 0) {
```

```
strcpy(display, "ε");
} else {
strncpy(display, parsingTable[i][j], colWidths[j+1]-3);
display[colWidths[j+1]-3] = '\0';
}
printf(" %-*s", colWidths[j+1]-2, display);
} else {
printf(" %-*s", colWidths[j+1]-2, "---");
}
}
printf("|\n");
printf("+");
for (int i = 0; i <= termCount; i++) {
for (int j = 0; j < colWidths[i]; j++) printf("-");
if (i < termCount) printf("+");
else printf("+\n");
}
}
}
void displayGrammar() {
printf("\n CURRENT GRAMMAR \n");
for (int i = 0; i < ruleCount; i++) {
printf("%s -> ", rules[i].nonTerminal);
for (int j = 0; j < rules[i].prodCount; j++) {
printf("%s", rules[i].prods[j]);
if (j < rules[i].prodCount - 1) printf(" | ");
}
printf("\n");
```

```
}
}
void getGrammarInput() {
printf(" GRAMMAR INPUT \n");
printf("Enter number of grammar rules: ");
int numRules;
scanf("%d", &numRules);
getchar();
for (int i = 0; i < numRules; i++) {
char input[SIZE];
printf("Enter rule %d: ", i + 1);
fgets(input, SIZE, stdin);
input[strcspn(input, "\n")] = 0;
char *arrow = strstr(input, "->");
if (!arrow) {
printf("Invalid format! Use '->' separator. Try again.\n");
i--;
continue;
}
Rule newRule;
newRule.prodCount = 0;
int ntLen = arrow - input;
strncpy(newRule.nonTerminal, input, ntLen);
newRule.nonTerminal[ntLen] = '\0';
char tempNT[SIZE];
int idx = 0;
for (int j = 0; j < strlen(newRule.nonTerminal); j++)
if (newRule.nonTerminal[j] != ' ')
```

```
tempNT[idx++] = newRule.nonTerminal[j];
tempNT[idx] = '\0';
strcpy(newRule.nonTerminal, tempNT);
char rhs[SIZE];
strcpy(rhs, arrow + 2);
char *token = strtok(rhs, "|");
while (token != NULL && newRule.prodCount < MAX_PROD) {
while (*token == ' ') token++;
char *end = token + strlen(token) - 1;
while (end > token && *end == ' ') *end-- = '\0';
if (strlen(token) > 0)
strcpy(newRule.prods[newRule.prodCount++], token);
token = strtok(NULL, "|");
}
rules[ruleCount++] = newRule;
}
}
int main() {
getGrammarInput();
displayGrammar();
eliminateLeftRecursion();
displayGrammar();
leftFactorGrammar();
displayGrammar();
computeFirstSets();
computeFollowSets();
constructParsingTable();
displayParsingTable();
```

return 0;

}

Input:

Enter the number of rules: 3

Enter grammar rule 1: E->E+T|T

Enter grammar rule 2: T->T*F|F

Enter grammar rule 3: F->(E)|id

Output:

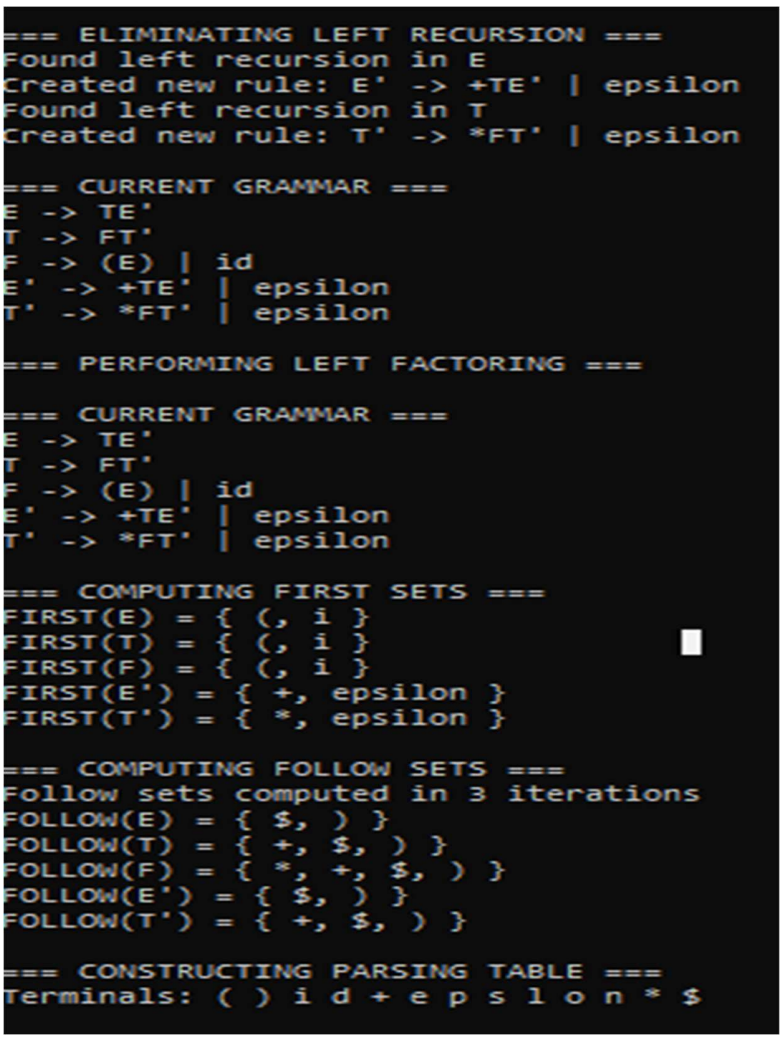
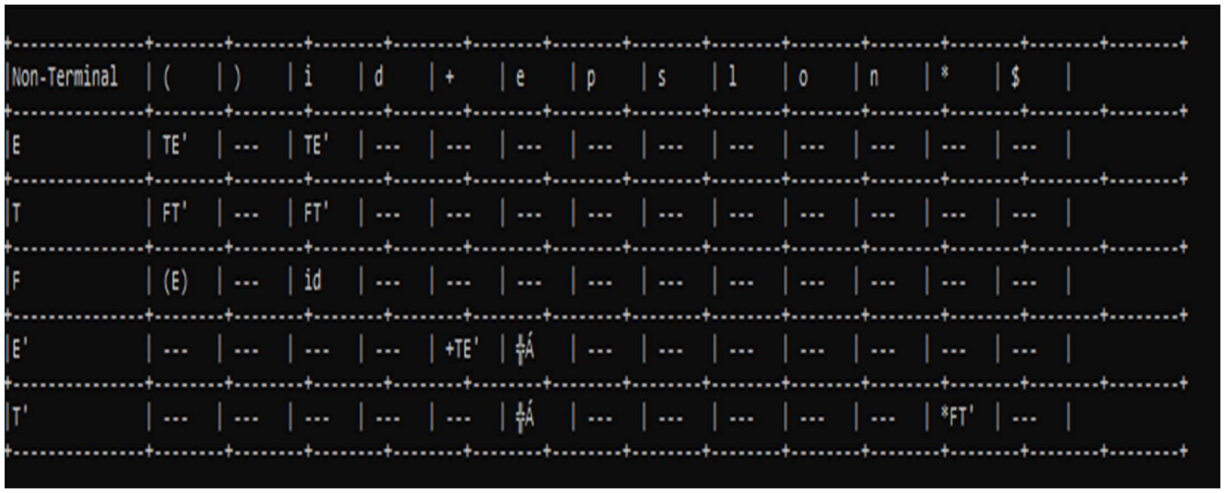


Figure 1: Output of the complete program



Non-Terminal	(	)	i	d	+	e	p	s	l	o	n	*	$
E	TE'	---	TE'	---	---	---	---	---	---	---	---	---	---
T	FT'	---	FT'	---	---	---	---	---	---	---	---	---	---
F	(E)	---	id	---	---	---	---	---	---	---	---	---	---
E'	---	---	---	---	+TE'	╬Á	---	---	---	---	---	---	---
T'	---	---	---	---	---	╬Á	---	---	---	---	---	*FT'	---

Figure 2: Output showing constructed parsing table

Discussion:

While implementing this experiment, one of the main difficulties I faced was getting the predictive parsing table to populate correctly without conflicts. Ensuring that FIRST and FOLLOW sets were computed accurately was crucial, because any mistake there led to incorrect table entries. Handling epsilon productions during FIRST and FOLLOW computation also required careful checking so that epsilon was neither wrongly propagated nor omitted. Overall, once the grammar processing order was properly handled and the conditions for FIRST and FOLLOW were carefully implemented, the parsing table generation worked correctly.