Premier University Chittagong

Department of Computer Science and Engineering

Course Title : Compiler Construction Lab

Course Code: CSE 454

Report No : 08

Report Title : Write a Program to Compute the First of Nonterminals 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: 8

Experiment Name: Write a Program to Compute the First of Nonterminals for a given Context-Free Grammar.

Objectives:

To design and implement a program in C that takes a CFG as input and computes the FIRST sets of all nonterminals. The program also performs left recursion elimination and left factoring to ensure the grammar is suitable for FIRST computation.

Algorithm

1. Initialize FIRST sets of all nonterminals as empty.
2. Repeat the following steps until no further changes occur in any FIRST set:
3. For each nonterminal A:
4. For each production A → α:
5. If α begins with a terminal symbol, add that terminal to FIRST(A) and continue to the next production.
6. If α begins with ε, add ε to FIRST(A) and continue.
7. If α begins with a nonterminal B, add all symbols from FIRST(B) except ε to FIRST(A).
8. If FIRST(B) does not contain ε, stop processing the rest of α.
9. Otherwise, move to the next symbol in α and repeat steps 7–8.
10. If all symbols in α can derive ε, add ε to FIRST(A).
11. Continue the iteration until no FIRST set changes in any rule.

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];
int isTerminal(char c) {
return !isupper(c) && c != '\" && !isdigit(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) {
strcpy(set[*count], sym);
(*count)++;
}
}
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; j++) {
for (int k = j + 1; k < currentRule->prodCount; k++) {
if (strcmp(currentRule->prods[j], currentRule->prods[k]) > 0) {
char temp[SIZE];
```

```
strcpy(temp, currentRule->prods[j]);
strcpy(currentRule->prods[j], currentRule->prods[k]);
strcpy(currentRule->prods[k], temp);
}
}
}
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 k = 0;
int allHaveEpsilon = 1;
while (production[k] && allHaveEpsilon) {
char symbol[2] = {production[k], '\0'};
if (isTerminal(symbol[0])) {
addToSet(first, count, symbol);
allHaveEpsilon = 0;
break;
} else {
int symIndex = findRuleIndex(symbol);
if (symIndex == -1) {
addToSet(first, count, symbol);
allHaveEpsilon = 0;
break;
```

```
}
for (int m = 0; m < firstCount[symIndex]; m++) {
if (strcmp(firstSets[symIndex][m], "epsilon") != 0) {
addToSet(first, count, firstSets[symIndex][m]);
}
}
if (!inSet(firstSets[symIndex], firstCount[symIndex], "epsilon")) {
allHaveEpsilon = 0;
}
}
k++;
}
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++) {
Rule *rule = &rules[i];
int oldCount = firstCount[i];
for (int j = 0; j < rule->prodCount; j++) {
```

```
computeFirstForProduction(rule->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 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';
end--;
}
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();
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:

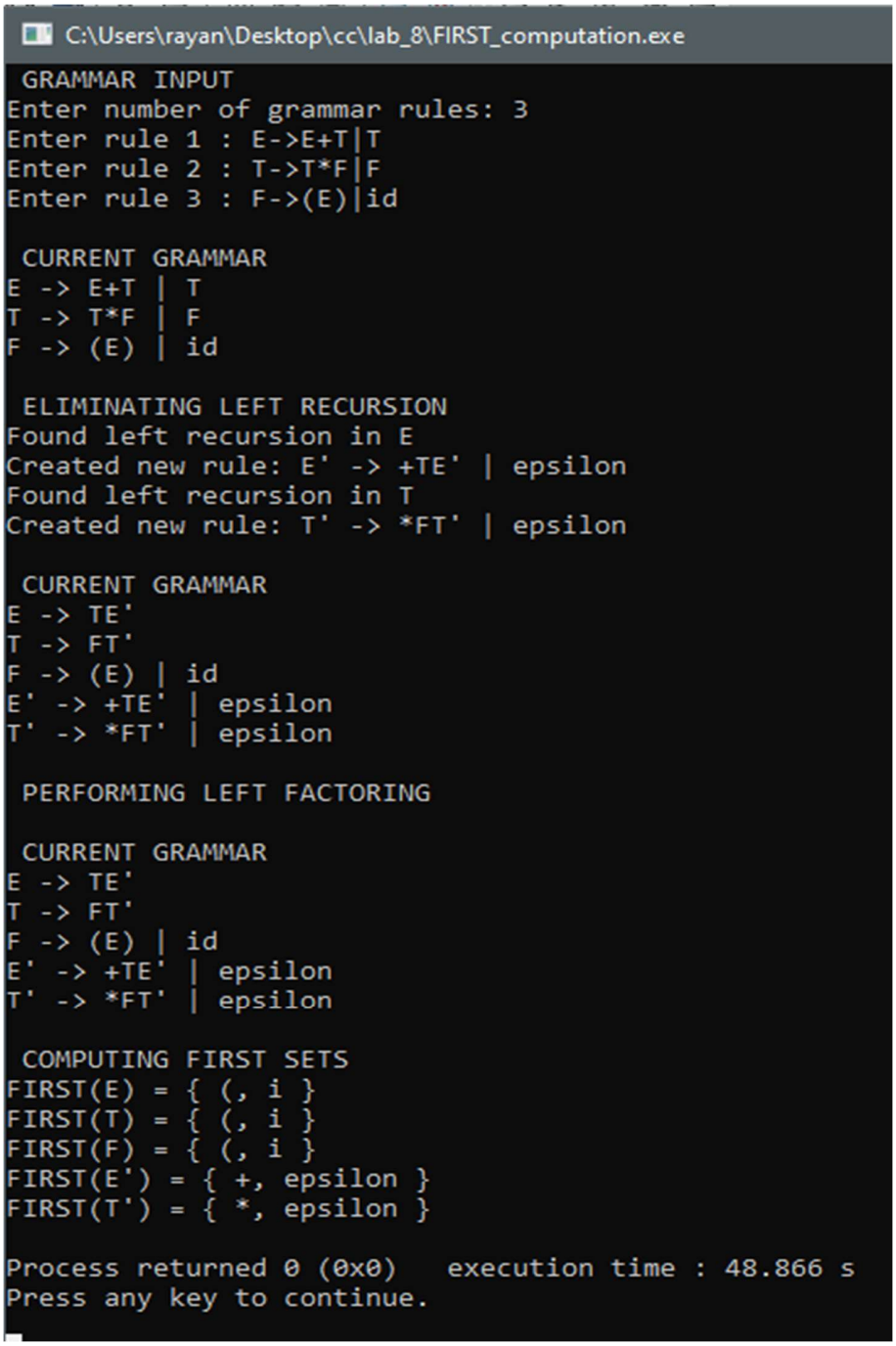

Figure 1: Output showing computed FIRST sets

Discussion:

While working on this experiment, I initially faced some confusion regarding how the entered grammar should be processed. The program does not directly compute the FIRST sets from the raw rules. Instead, it takes the grammar step by step: first it checks for and removes left recursion, then it performs left factoring, and only after these transformations the FIRST sets are computed. At first, this sequential processing made it harder for me to follow how each modification affected the grammar. Once I understood this workflow, the computation of FIRST sets proceeded correctly.