Premier University Chittagong

Department of Computer Science and Engineering

Course Title : Compiler Construction Lab

Course Code: CSE 454

Report No : 09

Report Title : Write a Program to Compute the Follow 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: 9

Experiment Name: Write a Program to Compute the Follow 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 Follow sets of all nonterminals. The program also performs left recursion elimination and left factoring, computes First Sets to ensure the grammar is suitable for Follow computation.

Algorithm

1. Initialize FOLLOW sets of all nonterminals as empty.
2. Add $ to FOLLOW of the start symbol.
3. Repeat the following steps until no FOLLOW set changes:
4. For each production A → αBβ, compute FIRST(β) and add all symbols of FIRST(β) except epsilon to FOLLOW(B).
5. If β is empty or FIRST(β) contains epsilon, add all symbols of FOLLOW(A) to FOLLOW(B).
6. Continue iterating until FOLLOW sets remain unchanged across a pass.

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[MX_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];
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]);
}}}
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 {
// terminal -> single char
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 new Rule;
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 pos = 0;
int allHaveEpsilon = 1;
int 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++) {
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 computeFirstOfString(char *str, char first[MAX_SYMBOLS][SIZE], int *count) {
*count = 0;
if (strlen(str) == 0) {
addToSet(first, count, "epsilon");
return;
}
int pos = 0;
int allHaveEpsilon = 1;
int len = strlen(str);
char symbol[SIZE];
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;
int 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);
int pos = 0;
while (pos < plen) {
char curSym[SIZE];
int slen = extractSymbol(prod, pos, curSym);
if (slen == 0) break;
if (!isTerminalChar(curSym[0]) || (isupper((unsigned char)curSym[0]) && curSym[0])) {
// curSym is a non-terminal (extracted as multi-char if present)
int curIndex = findRuleIndex(curSym);
if (curIndex != -1) {
// compute FIRST(beta) where beta is prod after current symbol
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) {
printf("Warning: FOLLOW set computation exceeded 1000 iterations. Breaking loop.\n");
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");
}}
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();
computeFollowSets();
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:

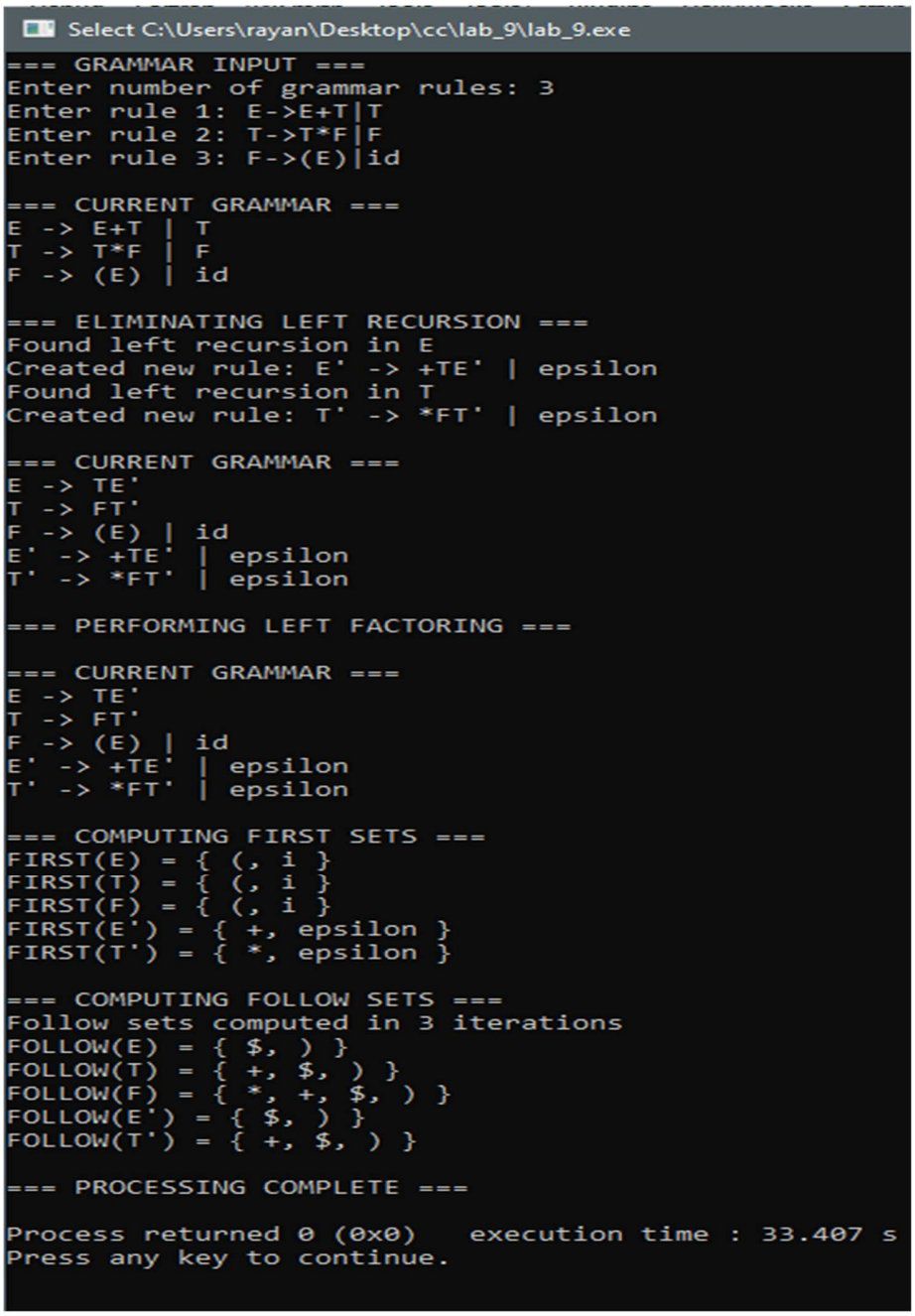

Figure 1: Output showing computed Follow sets.

Discussion:

In this experiment, I initially faced difficulty because the grammar is not used directly for FOLLOW computation. The program first removes left recursion and then performs left

factoring, so the grammar gets transformed before FIRST and FOLLOW sets are calculated. It took some time to understand how the transformed productions affected the final FOLLOW sets. Once I understood the sequence of operations and how the algorithm propagates symbols across rules, the computation worked as expected.