# Program 7

Write a C program to find FIRST( ) - predictive parser for the given grammar

#include <stdio.h>

#include <string.h>

#include <ctype.h>

#define NT 5 // Number of non-terminals: E, X, T, Y, F

#define PROD 8 // Total number of productions

#define MAX\_RHS 20 // Maximum length for right-hand side

// Production structure: lhs -> rhs

typedef struct {

char lhs;

char rhs[MAX\_RHS];

} Production;

// Our grammar productions:

Production productions[PROD] = {

{'E', "TX"}, // E -> T X

{'X', "+TX"}, // X -> + T X

{'X', "#"}, // X -> epsilon (represented by '#' )

{'T', "FY"}, // T -> F Y

{'Y', "\*FY"}, // Y -> \* F Y

{'Y', "#"}, // Y -> epsilon

{'F', "(E)"}, // F -> ( E )

{'F', "i"} // F -> i

};

// List of non-terminals.

char nonTerminals[NT] = {'E', 'X', 'T', 'Y', 'F'};

// first[nt][c] is 1 if terminal c is in FIRST(nonTerminal nt)

int first[NT][128] = {0};

// computed[nt] is 1 if FIRST(nonTerminal nt) has already been computed.

int computed[NT] = {0};

// Utility function: returns 1 if c is a non-terminal.

int isNonTerminal(char c) {

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

if (nonTerminals[i] == c)

return 1;

}

return 0;

}

// Returns the index of non-terminal c in nonTerminals array.

int getIndex(char c) {

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

if (nonTerminals[i] == c)

return i;

}

return -1;

}

// Recursively compute FIRST set for the non-terminal at index nt.

void computeFirst(int nt) {

// If already computed, no need to redo it.

if (computed[nt])

return;

computed[nt] = 1; // Mark as computed (assumes no left recursion)

char A = nonTerminals[nt]; // Current non-terminal

// Loop over all productions for A

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

if (productions[i].lhs == A) {

int epsilon\_in\_all = 1; // Assume epsilon will be in FIRST(alpha)

// Process the right-hand side of the production

for (int j = 0; j < strlen(productions[i].rhs); j++) {

char symbol = productions[i].rhs[j];

// If symbol is terminal, add it and stop processing this production.

if (!isNonTerminal(symbol)) {

first[nt][(int)symbol] = 1;

epsilon\_in\_all = 0;

break;

} else {

// symbol is non-terminal; get its FIRST set.

int idx = getIndex(symbol);

computeFirst(idx); // Ensure FIRST(idx) is computed.

// Add everything from FIRST(symbol) except epsilon.

for (int k = 0; k < 128; k++) {

if (first[idx][k] && k != '#') { // '#' represents epsilon

first[nt][k] = 1;

}

}

// If FIRST(symbol) does not contain epsilon, stop.

if (!first[idx]['#']) {

epsilon\_in\_all = 0;

break;

}

// Otherwise, continue to the next symbol.

}

}

// If all symbols in the production can derive epsilon, add epsilon.

if (epsilon\_in\_all)

first[nt]['#'] = 1;

}

}

}

// List of C keywords

const char\* keywords[] = {

"auto", "break", "case", "char", "const", "continue",

"default", "do", "double", "else", "enum", "extern",

"float", "for", "goto", "if", "int", "long", "register",

"return", "short", "signed", "sizeof", "static", "struct",

"switch", "typedef", "union", "unsigned", "void", "volatile", "while"

};

// Function to check if the given string is a keyword

int isKeyword(char \*str) {

int i;

for (i = 0; i < sizeof(keywords) / sizeof(keywords[0]); i++) {

if (strcmp(str, keywords[i]) == 0)

return 1;

}

return 0;

}

// Function to check if the given identifier is valid

int isValidIdentifier(char \*str) {

int i, len = strlen(str);

// Check if the first character is valid

if (!(isalpha(str[0]) || str[0] == '\_'))

return 0;

// Check the remaining characters

for (i = 1; i < len; i++) {

if (!(isalnum(str[i]) || str[i] == '\_'))

return 0;

}

// Check if it's a keyword

if (isKeyword(str))

return 0;

return 1;

}

int main() {

// Compute FIRST sets for all non-terminals.

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

computeFirst(i);

}

// Print the FIRST sets.

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

printf("FIRST(%c) = { ", nonTerminals[i]);

for (int c = 0; c < 128; c++) {

if (first[i][c]) {

// Print "epsilon" instead of '#' for clarity.

if (c == '#')

printf("epsilon ");

else

printf("%c ", c);

}

}

printf("}\n");

}

// Check if an identifier is valid

char identifier[100];

printf("Enter an identifier: ");

scanf("%99s", identifier); // Limit input to 99 characters to avoid buffer overflow

if (isValidIdentifier(identifier))

printf("Valid Identifier\n");

else

printf("Invalid Identifier\n");

return 0;

}

