1 Write a program to implement symbol table

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define TABLE\_SIZE 100

// Structure for a symbol table entry

typedef struct Symbol {

char name[50];

char type[20];

int scope;

struct Symbol\* next;

} Symbol;

// Hash table for the symbol table

Symbol\* symbolTable[TABLE\_SIZE] = {NULL};

// Hash function to compute index

int hashFunction(const char\* name) {

int hash = 0;

for (int i = 0; name[i] != '\0'; i++) {

hash = (hash + name[i]) % TABLE\_SIZE;

}

return hash;

}

// Function to insert a symbol into the table

void insertSymbol(const char\* name, const char\* type, int scope) {

int index = hashFunction(name);

Symbol\* newSymbol = (Symbol\*)malloc(sizeof(Symbol));

strcpy(newSymbol->name, name);

strcpy(newSymbol->type, type);

newSymbol->scope = scope;

newSymbol->next = symbolTable[index];

symbolTable[index] = newSymbol;

printf("Symbol inserted: %s, Type: %s, Scope: %d\n", name, type, scope);

}

// Function to search for a symbol in the table

Symbol\* searchSymbol(const char\* name) {

int index = hashFunction(name);

Symbol\* current = symbolTable[index];

while (current != NULL) {

if (strcmp(current->name, name) == 0) {

return current;

}

current = current->next;

}

return NULL;

}

// Function to display the symbol table

void displaySymbolTable() {

printf("\nSymbol Table:\n");

printf("Index\tName\t\tType\t\tScope\n");

printf("---------------------------------------------------\n");

for (int i = 0; i < TABLE\_SIZE; i++) {

Symbol\* current = symbolTable[i];

while (current != NULL) {

printf("%d\t%s\t\t%s\t\t%d\n", i, current->name, current->type, current->scope);

current = current->next;

}

}

}

int main() {

int choice;

char name[50], type[20];

int scope;

while (1) {

printf("\nSymbol Table Operations:\n");

printf("1. Insert Symbol\n");

printf("2. Search Symbol\n");

printf("3. Display Symbol Table\n");

printf("4. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter Symbol Name: ");

scanf("%s", name);

printf("Enter Symbol Type: ");

scanf("%s", type);

printf("Enter Symbol Scope: ");

scanf("%d", &scope);

insertSymbol(name, type, scope);

break;

case 2:

printf("Enter Symbol Name to Search: ");

scanf("%s", name);

Symbol\* symbol = searchSymbol(name);

if (symbol) {

printf("Symbol found: Name: %s, Type: %s, Scope: %d\n", symbol->name, symbol->type, symbol->scope);

} else {

printf("Symbol not found.\n");

}

break;

case 3:

displaySymbolTable();

break;

case 4:

printf("Exiting...\n");

exit(0);

default:

printf("Invalid choice. Please try again.\n");

}

}

return 0;

}

Write a program to separate the tokens for given input program.

#include <iostream>

#include <string>

#include <cctype>

#include <vector>

#include <unordered\_set>

using namespace std;

// Function to check if a string is a keyword

bool isKeyword(const string& word) {

unordered\_set<string> keywords = {

"int", "float", "char", "double", "if", "else", "while", "for", "return", "void",

"break", "continue", "do", "switch", "case", "default", "struct", "class"

};

return keywords.find(word) != keywords.end();

}

// Function to tokenize the input program

void tokenize(const string& program) {

string token = "";

vector<string> tokens;

for (size\_t i = 0; i < program.length(); ++i) {

char ch = program[i];

// If the character is alphanumeric or '\_', add it to the token

if (isalnum(ch) || ch == '\_') {

token += ch;

}

else {

// If a token is formed, process it

if (!token.empty()) {

tokens.push\_back(token);

token.clear();

}

// Handle operators and special characters as individual tokens

if (ispunct(ch) || !isspace(ch)) {

string op(1, ch);

if ((ch == '+' || ch == '-' || ch == '=' || ch == '&' || ch == '|') && i + 1 < program.length() && program[i + 1] == ch) {

op += program[++i]; // Capture double operators like ++, --, ==, &&

}

tokens.push\_back(op);

}

}

}

// Process any remaining token

if (!token.empty()) {

tokens.push\_back(token);

}

// Categorize and display the tokens

for (const string& t : tokens) {

if (isKeyword(t)) {

cout << t << " -> Keyword" << endl;

}

else if (isdigit(t[0])) {

cout << t << " -> Number" << endl;

}

else if (ispunct(t[0])) {

cout << t << " -> Operator/Special Symbol" << endl;

}

else {

cout << t << " -> Identifier" << endl;

}

}

}

int main() {

string inputProgram;

cout << "Enter the program (use 'end' on a new line to finish):" << endl;

string line;

while (getline(cin, line)) {

if (line == "end") break;

inputProgram += line + "\n";

}

cout << "\nTokens from the given program:\n";

tokenize(inputProgram);

return 0;

}

Write a recursive descent parser for the CFG language and implement it.

#include <iostream>

#include <vector>

#include <cctype>

#include <string>

using namespace std;

vector<string> tokens;

int currentToken = 0;

// Forward declarations of the parsing functions

void E();

void E\_prime();

void T();

void T\_prime();

void F();

void match(string expected) {

if (tokens[currentToken] == expected) {

currentToken++;

} else {

cout << "Syntax error: expected " << expected << " but found " << tokens[currentToken] << endl;

exit(1);

}

}

// E -> T E'

void E() {

T();

E\_prime();

}

// E' -> + T E' | ε

void E\_prime() {

if (tokens[currentToken] == "+") {

match("+");

T();

E\_prime();

}

// else: ε (epsilon, i.e., do nothing)

}

// T -> F T'

void T() {

F();

T\_prime();

}

// T' -> \* F T' | ε

void T\_prime() {

if (tokens[currentToken] == "\*") {

match("\*");

F();

T\_prime();

}

// else: ε (epsilon, i.e., do nothing)

}

// F -> ( E ) | id

void F() {

if (tokens[currentToken] == "(") {

match("(");

E();

match(")");

} else if (tokens[currentToken] == "id") {

match("id");

} else {

cout << "Syntax error: unexpected token " << tokens[currentToken] << endl;

exit(1);

}

}

// Tokenizer to split input into tokens

void tokenize(string input) {

string token = "";

for (char ch : input) {

if (isspace(ch)) {

continue; // Ignore whitespace

} else if (ch == '+' || ch == '\*' || ch == '(' || ch == ')') {

if (!token.empty()) {

tokens.push\_back(token);

token = "";

}

tokens.push\_back(string(1, ch)); // Add single-character tokens

} else if (isalpha(ch)) {

token += ch; // Build identifier

}

}

if (!token.empty()) {

tokens.push\_back(token); // Add the last identifier

}

}

int main() {

string input;

cout << "Enter the expression (use 'id' for identifiers, e.g., id + id \* id): ";

getline(cin, input);

// Tokenize the input

tokenize(input);

// Append a special end token to denote the end of input

tokens.push\_back("$");

// Start parsing

E();

// If we have processed all tokens and the current token is the end token, the input is valid

if (tokens[currentToken] == "$") {

cout << "Parsing successful!" << endl;

} else {

cout << "Syntax error: unexpected extra tokens after parsing." << endl;

}

return 0;

}

Write a program to implement shift reduce parser.

#include <iostream>

#include <stack>

#include <string>

#include <unordered\_map>

using namespace std;

enum Action { SHIFT, REDUCE, ACCEPT, ERROR };

struct TableEntry {

Action action;

int nextState;

string reduceRule;

};

unordered\_map<int, unordered\_map<string, TableEntry>> parsingTable;

void initializeParsingTable() {

// Fill parsingTable with appropriate entries based on grammar

parsingTable[0]["id"] = { SHIFT, 1, "" };

parsingTable[1]["+"] = { SHIFT, 2, "" };

parsingTable[2]["id"] = { SHIFT, 3, "" };

parsingTable[3]["$"] = { REDUCE, 0, "T → id" };

parsingTable[4]["+"] = { REDUCE, 0, "E → T" };

parsingTable[5]["$"] = { REDUCE, 0, "E → E + T" };

parsingTable[6]["$"] = { ACCEPT, 0, "" };

}

void shiftReduceParse(string input) {

stack<string> stk;

input += "$"; // End of input

size\_t pos = 0;

string symbol;

while (true) {

if (pos < input.size()) {

symbol = string(1, input[pos]);

} else {

symbol = "$";

}

TableEntry entry = parsingTable[stk.size()][symbol];

if (entry.action == SHIFT) {

stk.push(symbol);

pos++;

} else if (entry.action == REDUCE) {

cout << "Reduce using " << entry.reduceRule << endl;

// Pop stack according to the rule

if (entry.reduceRule == "T → id") {

stk.pop();

stk.push("T");

} else if (entry.reduceRule == "E → T") {

stk.pop();

stk.push("E");

} else if (entry.reduceRule == "E → E + T") {

stk.pop();

stk.pop();

stk.pop();

stk.push("E");

}

} else if (entry.action == ACCEPT) {

cout << "Parsing successful!" << endl;

break;

} else if (entry.action == ERROR) {

cout << "Syntax error!" << endl;

break;

}

}

}

int main() {

initializeParsingTable();

string input;

cout << "Enter the input string (e.g., id + id): ";

getline(cin, input);

shiftReduceParse(input);

return 0;

}

Write a C / C++ program to accept a C program and perform error detection& correction ,

indicate the user for the following :

a) Check whether the multi-line comment statement is terminated correctly or not.

b) Check whether the single line comment statement is existing in your C program and report the

line numbers to the user

#include <iostream>

#include <fstream>

#include <string>

using namespace std;

int main() {

ifstream infile("test.cpp"); // Open the file

if (!infile.is\_open()) {

cout << "Error: Could not open the file!" << endl;

return 1;

}

string line;

int line\_number = 0; // Track the current line number

int line\_error = -1; // Store the line number where a multiline comment starts

bool multiline\_flag = false; // Track if inside a multi-line comment

bool singleline\_flag = false; // Track if a single-line comment is found

while (getline(infile, line)) {

line\_number++;

for (size\_t i = 0; i < line.length(); i++) {

// Check for single-line comments

if (!multiline\_flag && line[i] == '/' && i + 1 < line.length() && line[i + 1] == '/') {

singleline\_flag = true;

cout << "Single-line comment found at line " << line\_number << endl;

break; // No need to check further in this line

}

// Check for start of multi-line comment

if (!multiline\_flag && line[i] == '/' && i + 1 < line.length() && line[i + 1] == '\*') {

multiline\_flag = true;

line\_error = line\_number; // Record the line where the comment starts

i++; // Skip the '\*' character

}

// Check for end of multi-line comment

else if (multiline\_flag && line[i] == '\*' && i + 1 < line.length() && line[i + 1] == '/') {

multiline\_flag = false;

i++; // Skip the '/' character

}

}

}

infile.close(); // Close the file

// Report results

if (multiline\_flag) {

cout << "Multiline comment started at line " << line\_error << " but not closed" << endl;

} else {

cout << "No unclosed multiline comments" << endl;

}

if (!singleline\_flag) {

cout << "No single-line comments found" << endl;

}

return 0;

}

Write a C / C++ program to accept a C program and perform error detection& correction for the

following:

a) Check for un-terminated string constant and single character constant in the input C program.

i.e A string constant begins with double quotes and extends to more than one line.

b) Report the error line numbers and the corrective actions to user.

#include <iostream>

#include <fstream>

#include <string>

using namespace std;

int main() {

ifstream infile("file.txt");

if (!infile.is\_open()) {

cout << "Error: Could not open the file!" << endl;

return 1;

}

string line;

int line\_number = 0; // Track current line number

int line\_error = -1; // Track line where unclosed string starts

bool flag = false; // Tracks if a string is open

while (getline(infile, line)) {

line\_number++;

cout << "Processing line " << line\_number << ": " << line << endl; // Debugging output

for (size\_t i = 0; i < line.length(); i++) {

if (line[i] == '"') {

if (!flag) {

// Open string

flag = true;

line\_error = line\_number;

cout << "String opened at line " << line\_number << endl; // Debugging output

} else {

// Close string

flag = false;

cout << "String closed at line " << line\_number << endl; // Debugging output

}

}

}

}

infile.close();

if (!flag) {

cout << "No error" << endl;

} else {

cout << "String opened at line " << line\_error << " but not closed" << endl;

}

return 0;

}

Write a C program to generate three address code of any statement.

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

typedef struct {

char op[3];

char arg1[10];

char arg2[10];

char result[10];

} TAC;

TAC code[20];

int codeIndex = 0;

int tempVarCount = 0;

// Function to generate a temporary variable

char\* generateTemp() {

static char temp[10];

sprintf(temp, "t%d", tempVarCount++);

return temp;

}

// Function to add a TAC instruction

void addTAC(const char\* result, const char\* arg1, const char\* op, const char\* arg2) {

strcpy(code[codeIndex].result, result);

strcpy(code[codeIndex].arg1, arg1);

strcpy(code[codeIndex].op, op);

strcpy(code[codeIndex].arg2, arg2);

codeIndex++;

}

// Recursive function to generate TAC for expressions

char\* generateTAC(char\* expression) {

char arg1[10], arg2[10], op[3];

int len = strlen(expression);

// Handle expressions in parentheses

if (expression[0] == '(' && expression[len - 1] == ')') {

expression[len - 1] = '\0';

return generateTAC(expression + 1);

}

// Search for operators

for (int i = len - 1; i >= 0; i--) {

if (expression[i] == '+' || expression[i] == '-' || expression[i] == '\*' || expression[i] == '/') {

strncpy(arg1, expression, i);

arg1[i] = '\0';

strcpy(op, (char[2]){expression[i], '\0'});

strcpy(arg2, expression + i + 1);

// Recursively evaluate both sides if necessary

char\* left = generateTAC(arg1);

char\* right = generateTAC(arg2);

char\* temp = generateTemp();

// Add TAC instruction for current operation

addTAC(temp, left, op, right);

return temp;

}

}

// If no operators, return the variable or value directly

return expression;

}

// Function to display the generated TAC

void displayTAC() {

printf("Three Address Code:\n");

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

printf("%s = %s %s %s\n", code[i].result, code[i].arg1, code[i].op, code[i].arg2);

}

}

int main() {

char expression[30];

printf("Enter an expression (e.g., a = (b + c) \* d - e): ");

scanf(" %[^\n]", expression);

// Extract variable name and the expression

char result[10], expr[30];

sscanf(expression, "%s = %[^\n]", result, expr);

// Generate TAC

char\* finalResult = generateTAC(expr);

// Store the final assignment

addTAC(result, finalResult, "", "");

// Display TAC

displayTAC();

return 0;

}

Write a C program to implement simple code generator.

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// Structure to represent a simple code instruction

typedef struct {

char instruction[10]; // Instruction (e.g., LOAD, ADD, SUB, etc.)

char operand1[10]; // First operand

char operand2[10]; // Second operand (optional)

} Code;

Code code[20];

int codeIndex = 0;

int tempVarCount = 0;

// Function to add a code instruction

void addInstruction(const char\* instruction, const char\* operand1, const char\* operand2) {

strcpy(code[codeIndex].instruction, instruction);

strcpy(code[codeIndex].operand1, operand1);

if (operand2 != NULL) {

strcpy(code[codeIndex].operand2, operand2);

} else {

code[codeIndex].operand2[0] = '\0';

}

codeIndex++;

}

// Function to generate code for an expression of the form a = b op c

void generateCode(char\* expression) {

char result[10], arg1[10], op[3], arg2[10];

// Parse the expression into result, arg1, operator, and arg2

sscanf(expression, "%s = %s %s %s", result, arg1, op, arg2);

// Generate assembly-like instructions based on the operator

addInstruction("LOAD", arg1, NULL); // LOAD arg1 into accumulator

if (strcmp(op, "+") == 0) {

addInstruction("ADD", arg2, NULL); // ADD arg2 to accumulator

} else if (strcmp(op, "-") == 0) {

addInstruction("SUB", arg2, NULL); // SUB arg2 from accumulator

} else if (strcmp(op, "\*") == 0) {

addInstruction("MUL", arg2, NULL); // MUL arg2 with accumulator

} else if (strcmp(op, "/") == 0) {

addInstruction("DIV", arg2, NULL); // DIV accumulator by arg2

}

addInstruction("STORE", result, NULL); // STORE result from accumulator

}

// Function to display the generated code

void displayCode() {

printf("\nGenerated Code:\n");

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

if (strlen(code[i].operand2) > 0) {

printf("%s %s, %s\n", code[i].instruction, code[i].operand1, code[i].operand2);

} else {

printf("%s %s\n", code[i].instruction, code[i].operand1);

}

}

}

int main() {

int n;

printf("Enter the number of expressions: ");

scanf("%d", &n);

// Loop to take each expression as input and generate code

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

char expression[30];

printf("Enter expression %d (e.g., a = b + c): ", i + 1);

scanf(" %[^\n]", expression); // Read entire line including spaces

generateCode(expression);

}

// Display the generated code

displayCode();

return 0;

}