**Lab1:program to test whether the given string valid comment section**

#include <stdio.h>

#include <string.h>

#define MAX\_INPUT\_LENGTH 200

int isValidComment(const char \*input) {

int len = strlen(input);

if (len >= 2 && input[0] == '/' && input[1] == '/') {

return 1; // Single-line comment

}

if (len >= 4 && input[0] == '/' && input[1] == '\*' && input[len - 2] == '\*' && input[len - 1] == '/') {

return 1; // Multi-line comment

}

return 0; // Not a valid comment

}

int main() {

char input[MAX\_INPUT\_LENGTH];

printf("Enter a string to check if it's a valid comment: ");

fgets(input, MAX\_INPUT\_LENGTH, stdin);

input[strcspn(input, "\n")] = '\0'; // Remove newline character

if (isValidComment(input)) {

printf("Valid comment section.\n");

} else {

printf("Not a valid comment section.\n");

}

return 0;

}

**Lab 2 Valid Identifier**

#include <iostream>

#include <cctype>

using namespace std;

// List of C++ reserved keywords (using a plain array)

const string keywords[] = {

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

"switch", "case", "default", "break", "continue", "void", "do", "long",

"short", "signed", "unsigned", "struct", "class", "public", "private",

"protected", "namespace", "using", "new", "delete", "try", "catch", "throw",

"static", "const", "volatile", "true", "false", "virtual", "this"

};

// Function to check if a string is a keyword

bool isKeyword(const string& str) {

int keywordCount = sizeof(keywords) / sizeof(keywords[0]); // Get number of keywords

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

if (str == keywords[i])

return true;

}

return false;

}

// Function to check if a string is a valid identifier

bool isValidIdentifier(const string& str) {

if (str.empty()) return false; // Empty string is not a valid identifier

// First character must be a letter or underscore

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

return false;

}

// Remaining characters must be letters, digits, or underscores

for (size\_t i = 1; i < str.length(); i++) {

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

return false;

}

}

// Identifier should not be a keyword

if (isKeyword(str)) {

return false;

}

return true;

}

int main() {

string input;

cout << "Enter an identifier: ";

cin >> input;

if (isValidIdentifier(input)) {

cout << "\"" << input << "\" is a valid identifier.\n";

} else {

cout << "\"" << input << "\" is NOT a valid identifier.\n";

}

return 0;

}

**Lab 3: Lexical analysis**

#include <stdio.h>

#include <ctype.h>

#include <string.h>

#define MAX\_TOKEN\_LENGTH 100

void analyzeLexemes(const char \*input) {

int i = 0;

char token[MAX\_TOKEN\_LENGTH];

int tokenIndex = 0;

while (input[i] != '\0') {

if (isspace(input[i])) {

i++;

continue;

}

if (isalpha(input[i])) { // Identifier

tokenIndex = 0;

while (isalnum(input[i])) {

token[tokenIndex++] = input[i++];

}

token[tokenIndex] = '\0';

printf("Identifier: %s\n", token);

}

else if (isdigit(input[i])) { // Number

tokenIndex = 0;

while (isdigit(input[i])) {

token[tokenIndex++] = input[i++];

}

token[tokenIndex] = '\0';

printf("Number: %s\n", token);

}

else { // Operators and special characters

printf("Operator: %c\n", input[i]);

i++;

}

}

}

int main() {

char input[MAX\_TOKEN\_LENGTH];

printf("Enter the input string: ");

fgets(input, MAX\_TOKEN\_LENGTH, stdin);

input[strcspn(input, "\n")] = '\0'; // Remove newline character

printf("Lexical Analysis:\n");

analyzeLexemes(input);

return 0;

}

**Lab 4: First and follow**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

#define MAX\_SYMBOLS 10

#define MAX\_PRODUCTIONS 10

#define MAX\_LEN 20

char productions[MAX\_PRODUCTIONS][MAX\_LEN];

char first[MAX\_SYMBOLS][MAX\_LEN];

char follow[MAX\_SYMBOLS][MAX\_LEN];

int numProductions;

int isTerminal(char c) {

return !isupper(c);

}

void addToSet(char \*set, char c) {

if (strchr(set, c) == NULL) {

int len = strlen(set);

set[len] = c;

set[len + 1] = '\0';

}

}

void findFirst(char symbol, int index) {

if (isTerminal(symbol)) {

addToSet(first[index], symbol);

return;

}

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

if (productions[i][0] == symbol) {

int j = 2;

while (productions[i][j] != '\0') {

if (isTerminal(productions[i][j])) {

addToSet(first[index], productions[i][j]);

break;

} else {

int nextIndex = productions[i][j] - 'A';

findFirst(productions[i][j], nextIndex);

for (int k = 0; k < strlen(first[nextIndex]); k++) {

if (first[nextIndex][k] != 'ε') {

addToSet(first[index], first[nextIndex][k]);

}

}

if (strchr(first[nextIndex], 'ε') == NULL) break;

}

j++;

}

}

}

}

void findFollow(char symbol, int index) {

if (productions[0][0] == symbol) {

addToSet(follow[index], '$');

}

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

for (int j = 2; productions[i][j] != '\0'; j++) {

if (productions[i][j] == symbol) {

if (productions[i][j + 1] != '\0') {

int nextIndex = productions[i][j + 1] - 'A';

for (int k = 0; k < strlen(first[nextIndex]); k++) {

if (first[nextIndex][k] != 'ε') {

addToSet(follow[index], first[nextIndex][k]);

}

}

}

if (productions[i][j + 1] == '\0' || strchr(first[productions[i][j + 1] - 'A'], 'ε')) {

int lhsIndex = productions[i][0] - 'A';

for (int k = 0; k < strlen(follow[lhsIndex]); k++) {

addToSet(follow[index], follow[lhsIndex][k]);

}

}

}

}

}

}

int main() {

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

scanf("%d", &numProductions);

getchar();

printf("Enter the productions (e.g., S=AB):\n");

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

fgets(productions[i], MAX\_LEN, stdin);

productions[i][strcspn(productions[i], "\n")] = '\0';

}

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

findFirst(productions[i][0], productions[i][0] - 'A');

}

printf("\nFirst Sets:\n");

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

printf("First(%c) = {%s}\n", productions[i][0], first[productions[i][0] - 'A']);

}

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

findFollow(productions[i][0], productions[i][0] - 'A');

}

printf("\nFollow Sets:\n");

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

printf("Follow(%c) = {%s}\n", productions[i][0], follow[productions[i][0] - 'A']);

}

return 0;

}

**Lab 5: Predictive Parsing Table**

#include <stdio.h>

#include <string.h>

char prol[7][10] = { "S", "A", "A", "B", "B", "C", "C" };

char pror[7][10] = { "A", "Bb", "Cd", "aB", "@", "Cc", "@" };

char prod[7][10] = { "S->A", "A->Bb", "A->Cd", "B->aB", "B->@", "C->Cc", "C->@" };

char first[7][10] = { "abcd", "ab", "cd", "a@", "@", "c@", "@" };

char follow[7][10] = { "$", "$", "$", "a$", "b$", "c$", "d$" };

char table[5][6][10];

int numr(char c)

{

switch (c)

{

case 'S':

return 0;

case 'A':

return 1;

case 'B':

return 2;

case 'C':

return 3;

case 'a':

return 0;

case 'b':

return 1;

case 'c':

return 2;

case 'd':

return 3;

case '$':

return 4;

}

return (2);

}

int main()

{

int i, j, k;

for (i = 0; i < 5; i++)

for (j = 0; j < 6; j++)

strcpy(table[i][j], " ");

printf("The following grammar is used for Parsing Table:\n");

for (i = 0; i < 7; i++)

printf("%s\n", prod[i]);

printf("\nPredictive parsing table:\n");

fflush(stdin);

for (i = 0; i < 7; i++)

{

k = strlen(first[i]);

for (j = 0; j < 10; j++)

if (first[i][j] != '@')

strcpy(table[numr(prol[i][0]) + 1][numr(first[i][j]) + 1], prod[i]);

}

for (i = 0; i < 7; i++)

{

if (strlen(pror[i]) == 1)

{

if (pror[i][0] == '@')

{

k = strlen(follow[i]);

for (j = 0; j < k; j++)

strcpy(table[numr(prol[i][0]) + 1][numr(follow[i][j]) + 1], prod[i]);

}

}

}

strcpy(table[0][0], " ");

strcpy(table[0][1], "a");

strcpy(table[0][2], "b");

strcpy(table[0][3], "c");

strcpy(table[0][4], "d");

strcpy(table[0][5], "$");

strcpy(table[1][0], "S");

strcpy(table[2][0], "A");

strcpy(table[3][0], "B");

strcpy(table[4][0], "C");

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

for (i = 0; i < 5; i++)

for (j = 0; j < 6; j++)

{

printf("%-10s", table[i][j]);

if (j == 5)

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

}

}

**Lab 6: SLR Parsing**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// Global Variables

int z = 0, i = 0, j = 0, c = 0;

char a[100], ac[20], stk[100], act[10];

char grammar[10][10]; // Array to store grammar rules

int rule\_count;

// Function to take grammar input from user

void input\_grammar() {

printf("\nEnter number of grammar rules: ");

scanf("%d", &rule\_count);

printf("\nEnter Grammar Rules:\n");

printf("Format: LHS->RHS (e.g., S->AA)\n");

for (int k = 0; k < rule\_count; k++) {

printf("Rule %d: ", k + 1);

scanf("%s", grammar[k]);

}

}

// Function to print the grammar

void print\_grammar() {

printf("\nGRAMMAR:\n");

for (int k = 0; k < rule\_count; k++) {

printf("%s\n", grammar[k]);

}

}

// Function to check and reduce the stack

void check() {

strcpy(ac, "REDUCE TO");

for (int r = 0; r < rule\_count; r++) {

char lhs = grammar[r][0]; // Left-hand side of rule

char rhs[10];

strcpy(rhs, grammar[r] + 3); // Extract RHS from "A->XYZ"

int rhs\_len = strlen(rhs);

// Check for a match in the stack

for (z = 0; z <= c - rhs\_len; z++) {

if (strncmp(stk + z, rhs, rhs\_len) == 0) { // Rule matches

printf("\n%s %s", ac, grammar[r]);

// Reduce stack

stk[z] = lhs;

for (int k = z + 1; k < c; k++) {

stk[k] = '\0';

}

c -= (rhs\_len - 1);

printf("\n$%s\t%s$\t", stk, a);

return; // Stop after a successful reduction

}

}

}

}

int main() {

input\_grammar();

print\_grammar();

printf("\nEnter input string: ");

scanf("%s", a);

c = strlen(a);

strcpy(act, "SHIFT");

printf("\nstack \t input \t action");

printf("\n$\t%s$\t", a);

for (i = 0; j < c; i++, j++) {

printf("%s", act);

stk[i] = a[j];

stk[i + 1] = '\0';

a[j] = ' ';

printf("\n$%s\t%s$\t", stk, a);

check();

}

check(); // Final check after shifting all characters

// Accept if only start symbol remains

if (stk[0] == grammar[0][0] && stk[1] == '\0')

printf("Accept\n");

else

printf("Reject\n");

return 0;

}

**Lab 7: SLR Buttom up parser**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_STACK 100

#define MAX\_INPUT 100

typedef struct {

int state;

char symbol;

} StackElement;

typedef struct {

char nonTerminal;

char \*production;

int length;

} Production;

StackElement stack[MAX\_STACK];

int top = -1;

Production productions[] = {

{'S', "AA", 2}, // S -> AA (production 0)

{'A', "aA", 2}, // A -> aA (production 1)

{'A', "b", 1} // A -> b (production 2)

};

// Corrected ACTION table

const int ACTION[7][3] = {

// a b $

{2, 3, 0}, // State 0

{0, 0, 1}, // State 1 (Accept)

{2, 3, 0}, // State 2

{0, 0, -3}, // State 3 (Reduce A->b)

{0, 0, -1}, // State 4 (Reduce S->AA) - FIXED THIS STATE

{2, 3, 0}, // State 5

{0, 0, -2} // State 6 (Reduce A->aA)

};

// Corrected GOTO table

const int GOTO[7][2] = {

// A S

{4, 1}, // State 0

{0, 0}, // State 1

{6, 0}, // State 2

{0, 0}, // State 3

{0, 0}, // State 4

{0, 0}, // State 5

{0, 0} // State 6

};

void printDebug(char\* action, int state, char symbol) {

printf("DEBUG: %s - State: %d, Symbol: %c\n", action, state, symbol);

}

void push(int state, char symbol) {

if (top >= MAX\_STACK - 1) {

printf("Stack overflow\n");

exit(1);

}

top++;

stack[top].state = state;

stack[top].symbol = symbol;

printDebug("Push", state, symbol);

}

void pop(int count) {

printDebug("Pop", stack[top].state, stack[top].symbol);

top -= count;

if (top < -1) {

printf("Stack underflow\n");

exit(1);

}

}

int getColumnIndex(char input) {

switch(input) {

case 'a': return 0;

case 'b': return 1;

case '$': return 2;

case 'A': return 0;

case 'S': return 1;

default: return -1;

}

}

void printStack() {

printf("Stack: ");

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

printf("(%d,%c) ", stack[i].state, stack[i].symbol);

}

printf("\n");

}

int parse(char \*input) {

int currentPos = 0;

char currentInput;

push(0, '$');

while (1) {

currentInput = input[currentPos];

int currentState = stack[top].state;

int col = getColumnIndex(currentInput);

printf("\nCurrent input: %c\n", currentInput);

printStack();

if (col == -1) {

printf("Invalid input symbol: %c\n", currentInput);

return 0;

}

int action = ACTION[currentState][col];

printf("Current state: %d, Action: %d\n", currentState, action);

if (action == 1 && currentInput == '$') {

printf("Accept! Input string is valid.\n");

return 1;

}

else if (action > 0) {

printf("Shift to state %d\n", action);

push(action, currentInput);

currentPos++;

}

else if (action < 0) {

int productionIndex = -action - 1;

Production prod = productions[productionIndex];

printf("Reducing by production %d: %c -> %s\n",

productionIndex, prod.nonTerminal, prod.production);

pop(prod.length);

int newState = GOTO[stack[top].state][getColumnIndex(prod.nonTerminal)];

printf("After reduction, going to state %d\n", newState);

push(newState, prod.nonTerminal);

}

else {

printf("Syntax error at position %d\n", currentPos);

printf("Invalid action in state %d for input %c\n", currentState, currentInput);

return 0;

}

}

}

int main() {

char input[MAX\_INPUT];

printf("Enter input string (end with $): ");

scanf("%s", input);

if (parse(input)) {

printf("Parsing completed successfully\n");

} else {

printf("Parsing failed\n");

}

return 0;

}

**Lab 8: LR(1) Parser**

#include <iostream>

#include <stack>

#include <map>

#include <vector>

#include <string>

using namespace std;

struct Rule {

char lhs;

string rhs;

};

vector<Rule> rules = {

{'S', "AB"},

{'A', "aA"},

{'A', ""},

{'B', "bB"},

{'B', ""}

};

map<char, int> terminalMap = {

{'a', 0},

{'b', 1},

{'$', 2}

};

map<int, map<char, int>> actionTable = {

{0, {{'a', 1}, {'b', 2}, {'$', -1}}},

{1, {{'a', 1}, {'b', -2}, {'$', -2}}},

{2, {{'a', -3}, {'b', 2}, {'$', -3}}},

{3, {{'a', -1}, {'b', -1}, {'$', 0}}},

{4, {{'a', -4}, {'b', -4}, {'$', -4}}}

};

map<int, map<char, int>> gotoTable = {

{0, {{'A', 3}, {'B', 4}}},

{1, {{'A', 3}, {'B', 4}}},

{2, {{'A', 3}, {'B', 4}}},

{3, {{'S', 5}, {'A', 3}, {'B', 4}}},

{4, {{'A', 3}, {'B', 4}}}

};

class Parser {

public:

bool parse(string input) {

stack<int> stateStack;

stack<char> symbolStack;

stateStack.push(0);

symbolStack.push('$'); // Push initial symbol

int i = 0;

input += '$'; // Append '$' to input

while (true) {

int state = stateStack.top();

char currentInput = input[i];

cout << "Current state: " << state << ", Current token: " << currentInput << endl;

if (actionTable[state].find(currentInput) == actionTable[state].end()) {

cout << "Action found: REJECT" << endl;

return false;

}

int action = actionTable[state][currentInput];

if (action > 0) {

cout << "Action found: SHIFT to state " << action << endl;

stateStack.push(action);

symbolStack.push(currentInput);

i++;

} else if (action == 0) {

cout << "Action found: ACCEPT" << endl;

return true;

} else if (action < 0) {

int ruleIndex = -action - 1;

Rule rule = rules[ruleIndex];

cout << "Action found: REDUCE using production " << rule.lhs << " -> " << rule.rhs << endl;

// Pop symbols and states based on RHS length

for (int j = 0; j < rule.rhs.length(); j++) {

stateStack.pop();

symbolStack.pop();

}

// Get the state \*before\* the pop

int prevState = stateStack.top();

// Use prevState to get the next state from the GOTO table

int nextState = gotoTable[prevState][rule.lhs];

stateStack.push(nextState);

symbolStack.push(rule.lhs);

cout << "New state after reduction: " << nextState << endl;

}

}

}

};

int main() {

Parser parser;

string input;

cout << "Enter the input string to parse: ";

getline(cin, input);

bool result = parser.parse(input);

if (result) {

cout << "Accepted" << endl;

} else {

cout << "Rejected" << endl;

}

return 0;

}

**Lab 9:LALR bottom up parser**

#include <stdio.h>

#include <string.h>

#include <ctype.h>

#define MAX 100

char stack[MAX];

int top = -1;

char input[MAX];

void push(char c) {

stack[++top] = c;

}

void pop(int count) {

top -= count;

}

void printStackAndInput(int index) {

printf("\nStack: ");

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

printf("%c", stack[i]);

}

printf("\tInput: ");

for (int i = index; i < strlen(input); i++) {

printf("%c", input[i]);

}

printf("\tAction: ");

}

char peek(int pos) {

if (top - pos >= 0) {

return stack[top - pos];

}

return '\0';

}

int reduce() {

int reduced = 0;

do {

reduced = 0;

printf("\nChecking reductions for stack: ");

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

printf("%c", stack[i]);

}

// Rule 1: digit → F

if (isdigit(peek(0))) {

printf("\nReducing digit %c to F", stack[top]);

stack[top] = 'F';

reduced = 1;

continue;

}

// Rule 2: F → T (when not part of multiplication)

if (peek(0) == 'F' && peek(1) != '\*') {

printf("\nReducing F to T");

stack[top] = 'T';

reduced = 1;

continue;

}

// Rule 3: T \* F → T (handle multiplication)

if (peek(0) == 'F' && peek(1) == '\*' && peek(2) == 'T') {

printf("\nReducing T \* F to T");

pop(2);

stack[top] = 'T';

reduced = 1;

continue;

}

// Rule 4: E \* F → T (new rule to handle E\*F case)

if (peek(0) == 'F' && peek(1) == '\*' && peek(2) == 'E') {

printf("\nReducing E \* F to T");

pop(2);

stack[top] = 'T';

reduced = 1;

continue;

}

// Rule 5: T → E (when not part of addition)

if (peek(0) == 'T' && peek(1) != '+') {

printf("\nReducing T to E");

stack[top] = 'E';

reduced = 1;

continue;

}

// Rule 6: E + T → E (handle addition)

if (peek(0) == 'T' && peek(1) == '+' && peek(2) == 'E') {

printf("\nReducing E + T to E");

pop(2);

stack[top] = 'E';

reduced = 1;

continue;

}

} while (reduced);

return reduced;

}

void parseExpression() {

printf("\nParsing Expression: %s\n", input);

printf("\nTrace of parsing steps:");

int i = 0;

while (i < strlen(input)) {

printStackAndInput(i);

printf("Shift %c", input[i]);

push(input[i]);

i++;

reduce();

}

// Keep reducing until no more reductions are possible

printf("\n\nPerforming final reductions:");

while (1) {

int did\_reduce = reduce();

if (!did\_reduce) break;

}

printf("\n\nFinal stack: ");

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

printf("%c", stack[i]);

}

if (top == 0 && stack[top] == 'E') {

printf("\n✅ Accepted: Valid Expression!\n");

} else {

printf("\n❌ Rejected: Invalid Expression!\n");

printf("Expected single 'E' on stack, found: ");

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

printf("%c", stack[i]);

}

printf("\n");

}

}

int main() {

printf("Enter an arithmetic expression (use digits, +, \*): ");

scanf("%s", input);

// Input validation

for (int i = 0; i < strlen(input); i++) {

if (!isdigit(input[i]) && input[i] != '+' && input[i] != '\*') {

printf("\n❌ Invalid character in expression: %c\n", input[i]);

return 1;

}

}

parseExpression();

return 0;

}

**Lab 10 : Three address code**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

#define MAX 100

// Structure to represent a three-address code instruction

typedef struct {

char op[10]; // Operator

char arg1[10]; // First argument

char arg2[10]; // Second argument

char result[10]; // Result

} TAC;

// Function to generate three-address code

void generateTAC(char \*expression) {

TAC tac[MAX];

int tacIndex = 0;

char tempVar[10];

int tempCount = 1;

// Tokenize the expression

char \*token = strtok(expression, " ");

char lastOperand[10] = "";

char lastOperator[10] = "";

// Process the expression

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

if (isalnum(expression[i])) { // Check if the character is an operand (variable or number)

// Build the operand string

char operand[10] = {0};

int j = 0;

while (isalnum(expression[i]) && j < 9) {

operand[j++] = expression[i++];

}

operand[j] = '\0'; // Null-terminate the operand

if (strlen(lastOperand) == 0) {

// First operand

strcpy(lastOperand, operand);

} else {

// We have a last operand and now we have a new operand

sprintf(tempVar, "t%d", tempCount++);

strcpy(tac[tacIndex].result, tempVar);

strcpy(tac[tacIndex].arg1, lastOperand);

strcpy(tac[tacIndex].arg2, operand);

strcpy(tac[tacIndex].op, lastOperator);

tacIndex++;

strcpy(lastOperand, tempVar); // Update lastOperand to the new temp variable

}

i--; // Adjust index since the for loop will increment it

} else if (strchr("+-\*/", expression[i])) {

// If the character is an operator, store it for the next operation

lastOperator[0] = expression[i];

lastOperator[1] = '\0'; // Null-terminate the operator string

}

}

// Handle the last operation if there's a remaining lastOperand

if (strlen(lastOperand) > 0) {

strcpy(tac[tacIndex].result, lastOperand);

strcpy(tac[tacIndex].arg1, lastOperand);

strcpy(tac[tacIndex].arg2, "0"); // Assuming the last operation is with 0

strcpy(tac[tacIndex].op, ""); // No operation

tacIndex++;

}

// Print the generated three-address code

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

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

if (strlen(tac[i].op) > 0) {

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

}

}

}

int main() {

char expression[MAX];

printf("Enter a simple arithmetic expression (e.g., a + b \* c): ");

fgets(expression, sizeof(expression), stdin);

expression[strcspn(expression, "\n")] = 0; // Remove newline character

generateTAC(expression);

return 0;

}

**Lab 11: Code generation**

#include <stdio.h>

#include <string.h>

#define MAX 100

// Structure to represent a three-address statement

typedef struct {

char result[10];

char operand1[10];

char operand2[10];

char op[2]; // To accommodate both single and double character operators

} TAC;

// Function to generate target code from three-address code

void generateTargetCode(TAC tac[], int n) {

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

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

printf("LOAD R1, %s\n", tac[i].operand1); // Load operand1 into register R1

if (strcmp(tac[i].op, "=") != 0) { // Check if it is not a simple assignment

if (strcmp(tac[i].op, "+") == 0) {

printf("ADD R1, %s\n", tac[i].operand2); // Perform addition with operand2

} else if (strcmp(tac[i].op, "\*") == 0) {

printf("MUL R1, %s\n", tac[i].operand2); // Perform multiplication with operand2

} else if (strcmp(tac[i].op, "-") == 0) {

printf("SUB R1, %s\n", tac[i].operand2); // Perform subtraction with operand2

} else if (strcmp(tac[i].op, "/") == 0) {

printf("DIV R1, %s\n", tac[i].operand2); // Perform division with operand2

}

}

printf("STORE %s, R1\n", tac[i].result); // Store result back to memory

}

}

int main() {

int n;

TAC tac[MAX];

char input[100];

printf("Enter the number of three-address statements: ");

scanf("%d", &n);

getchar(); // Consume the newline character

printf("Enter the three-address statements in the format (result = operand1 op operand2) or (result = operand1):\n");

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

fgets(input, sizeof(input), stdin); // Read the entire line

sscanf(input, "%s = %s %s %s", tac[i].result, tac[i].operand1, tac[i].op, tac[i].operand2);

// Check if it's an assignment statement

if (strcmp(tac[i].op, "=") == 0 || (strcmp(tac[i].op, "") == 0 && strcmp(tac[i].operand2, "=") == 0)) {

strcpy(tac[i].op, "=");

strcpy(tac[i].operand2, ""); // Clear operand2 for assignment

}

// Remove newline if present

size\_t len = strlen(tac[i].result);

if (len > 0 && tac[i].result[len-1] == '\n') {

tac[i].result[len-1] = '\0';

}

}

generateTargetCode(tac, n);

return 0;

}

**Lab 12: Code optimization**

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <ctype.h>

#define MAX 100

// Structure to represent a three-address statement

typedef struct {

char result[10], operand1[10], operand2[10], op[2];

} TAC;

// Function for constant folding optimization

void constantFolding(TAC tac[], int \*n) {

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

if (isdigit(tac[i].operand1[0]) && isdigit(tac[i].operand2[0])) { // Check if both operands are constants

int op1 = atoi(tac[i].operand1);

int op2 = atoi(tac[i].operand2);

int result = 0;

// Perform the operation based on the operator

if (strcmp(tac[i].op, "+") == 0) result = op1 + op2;

else if (strcmp(tac[i].op, "-") == 0) result = op1 - op2;

else if (strcmp(tac[i].op, "\*") == 0) result = op1 \* op2;

else if (strcmp(tac[i].op, "/") == 0 && op2 != 0) result = op1 / op2;

// Update the TAC entry with the result

sprintf(tac[i].result, "%d", result);

strcpy(tac[i].op, "=");

strcpy(tac[i].operand1, ""); // Clear operand1

strcpy(tac[i].operand2, ""); // Clear operand2

}

}

}

// Function to generate target code from three-address code

void generateTargetCode(TAC tac[], int n) {

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

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

if (strcmp(tac[i].op, "=") == 0) {

printf("STORE %s, R1\n", tac[i].result); // Store result back to memory

} else {

printf("LOAD R1, %s\n", tac[i].operand1); // Load operand1 into register R1

if (strcmp(tac[i].op, "+") == 0) {

printf("ADD R1, %s\n", tac[i].operand2); // Perform addition with operand2

} else if (strcmp(tac[i].op, "\*") == 0) {

printf("MUL R1, %s\n", tac[i].operand2); // Perform multiplication with operand2

} else if (strcmp(tac[i].op, "-") == 0) {

printf("SUB R1, %s\n", tac[i].operand2); // Perform subtraction with operand2

} else if (strcmp(tac[i].op, "/") == 0) {

printf("DIV R1, %s\n", tac[i].operand2); // Perform division with operand2

}

printf("STORE %s, R1\n", tac[i].result); // Store result back to memory

}

}

}

int main() {

int n;

TAC tac[MAX];

char input[100];

printf("Enter the number of three-address statements: ");

scanf("%d", &n);

getchar(); // Consume newline

printf("Enter the three-address statements in the format (result = operand1 op operand2):\n");

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

fgets(input, sizeof(input), stdin); // Read the entire line

sscanf(input, "%s = %s %s %s", tac[i].result, tac[i].operand1, tac[i].op, tac[i].operand2);

// Check if it's an assignment statement without an operation

if (strcmp(tac[i].op, "=") == 0 || strcmp(tac[i].operand2, "") == 0) {

strcpy(tac[i].op, "=");

strcpy(tac[i].operand2, ""); // Clear operand2 for assignment

}

// Remove newline from result variable name

size\_t len = strlen(tac[i].result);

if (len > 0 && tac[i].result[len - 1] == '\n') {

tac[i].result[len - 1] = '\0';

}

}

// Perform constant folding optimization

constantFolding(tac, &n);

// Generate target code

generateTargetCode(tac, n);

return 0;

}