LAB 1: IMPLEMENTING LEXICAL ANALYZER

Implement a lexical analyzer to recognize identifiers, keywords, comments, strings, operators, and constants. Display token type and lexeme.

Theory:

The lexical analyzer, often referred to as a scanner, represents the initial phase of the compiler design process. Its primary function is to read the raw stream of characters from the source code and group them into meaningful distinct sequences known as lexemes. These lexemes are then classified into specific tokens, such as keywords, identifiers, or operators. By filtering out non-essential elements like whitespace and comments, the lexical analyzer prepares a structured stream of tokens, which serves as the standardized input for the subsequent syntax analysis phase.

Algorithm:

1. Read the source code from an input file.
2. Initialize a pointer to the first character.
3. Repeat until end of file: o Ignore whitespace and newline characters. o If the character starts with:
    a. Letter or underscore → read full word → check keyword or identifier.
    b. Digit → read full number → classify as constant.
    c. Double quote (") → read until closing quote → classify as string.
    d. // or /* */ → classify as comment.
    e. Special symbols (+ - * / = < > etc.) → classify as operator.
4. Display token type and lexeme.
5. Stop when EOF is reached.

Source Code:

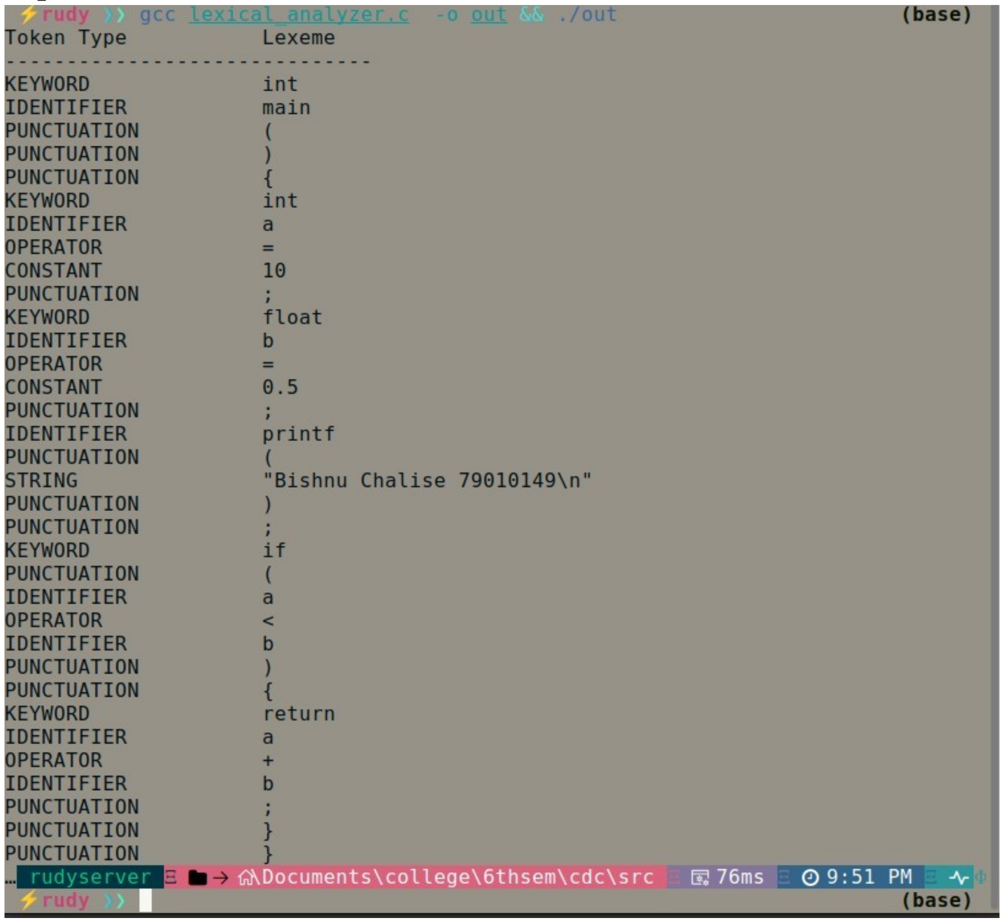
```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <stdbool.h>

typedef enum {
KEYWORD, IDENTIFIER, CONSTANT, STRING, OPERATOR, PUNCTUATION,
UNKNOWN
} TokenType;
typedef struct {
```

```
char lexeme[100];
TokenType type;
} Token;
bool isKeyword(const char* str) {
const char* keywords[] = {"if", "else", "while", "return", "int", "float", "void", "for", "do",
"break", "continue"};
int numKeywords = 11;
for (int i = 0; i < numKeywords; i++) {
if (strcmp(str, keywords[i]) == 0) return true;
}
return false;
}

void analyze(const char* input) {
int i = 0;
int len = strlen(input);
printf("%-20s %s\n", "Token Type", "Lexeme");
printf("------------------------------\n");

while (i < len) {
if (isspace(input[i])) { i++; continue; }

if (input[i] == '/' && input[i+1] == '/') {
i += 2;
while (i < len && input[i] != '\n') i++;
continue;
}
if (input[i] == '/' && input[i+1] == '*') {
i += 2;
while (i < len - 1 && !(input[i] == '*' && input[i+1] == '/')) i++;
i += 2;
continue;
}

if (input[i] == '"') {
int start = i++;
while (i < len && input[i] != '"') i++;
if (i < len) i++;
char buffer[100];
```

```
int tokenLen = i - start;
strncpy(buffer, input + start, tokenLen);
buffer[tokenLen] = '\0';
printf("%-20s %s\n", "STRING", buffer);
continue;
}

if (isalpha(input[i]) || input[i] == '_') {
int start = i;
while (i < len && (isalnum(input[i]) || input[i] == '_')) i++;
char buffer[100];
int tokenLen = i - start;
strncpy(buffer, input + start, tokenLen);
buffer[tokenLen] = '\0';
if (strcmp(buffer, "int") == 0 || strcmp(buffer, "float") == 0 || strcmp(buffer, "if") == 0 ||
strcmp(buffer, "return") == 0)
printf("%-20s %s\n", "KEYWORD", buffer);
else
printf("%-20s %s\n", "IDENTIFIER", buffer);
continue;
}

if (isdigit(input[i])) {
int start = i;
while (i < len && (isdigit(input[i]) || input[i] == '.')) i++;
char buffer[100];
int tokenLen = i - start;
strncpy(buffer, input + start, tokenLen);
buffer[tokenLen] = '\0';
printf("%-20s %s\n", "CONSTANT", buffer);
continue;
}

if (strchr("+-*/%=<>!&|", input[i])) {
char buffer[3] = {input[i], '\0', '\0'};
if (i + 1 < len && strchr("=+-&|", input[i+1])) {
buffer[1] = input[i+1];
i++;
}
```

```
i++;
printf("%-20s %s\n", "OPERATOR", buffer);
continue;
}

if (strchr(";,(){}[]", input[i])) {
printf("%-20s %c\n", "PUNCTUATION", input[i]);
i++;
continue;
}

i++;
}
}
int main() {
const char* sourceCode =
"// Sample program by Bishnu Chalise\n"
"int main() {\n"
" int a = 10;\n"
" float b = 0.5;\n"
" // This is a comment\n"
" printf(\"Bishnu Chalise 79010174\\n\");\n"
" if (a < b) {\n"
" return a + b;\n"
" }\n"
"}";

analyze(sourceCode);
return 0;
}
```

Output:



```
⚡rudy ›› gcc lexical_analyzer.c  -o out && ./out                    (base)
Token Type                 Lexeme
------------------------------
KEYWORD                    int
IDENTIFIER                 main
PUNCTUATION                (
PUNCTUATION                )
PUNCTUATION                {
KEYWORD                    int
IDENTIFIER                 a
OPERATOR                   =
CONSTANT                   10
PUNCTUATION                ;
KEYWORD                    float
IDENTIFIER                 b
OPERATOR                   =
CONSTANT                   0.5
PUNCTUATION                ;
IDENTIFIER                 printf
PUNCTUATION                (
STRING                     "Bishnu Chalise 79010149\n"
PUNCTUATION                )
PUNCTUATION                ;
KEYWORD                    if
PUNCTUATION                (
IDENTIFIER                 a
OPERATOR                   <
IDENTIFIER                 b
PUNCTUATION                )
PUNCTUATION                {
KEYWORD                    return
IDENTIFIER                 a
OPERATOR                   +
IDENTIFIER                 b
PUNCTUATION                ;
PUNCTUATION                }
PUNCTUATION                }
… rudyserver  → ⌂Documents\college\6thsem\cdc\src  76ms  9:51 PM
⚡rudy ››                                                             (base)
```

Conclusion:

In this experiment, we successfully designed a lexical analyzer capable of tokenizing raw source code. The program effectively distinguished between various language constructs—including keywords, identifiers, and literals—while correctly ignoring comments and whitespace. This confirms the fundamental role of lexical analysis in structuring raw input for the parsing phase of compilation.

LAB 2: IMPLEMENTING SYMBOL TABLE OPERATIONS

Implement a symbol table to demonstrate the operations: insert, lookup and display. Maintain attributes such as identifier name, type, and scope.

Theory:

A symbol table is a vital data structure maintained by compilers to track the semantics of variables and functions. It acts as a database where every identifier in the source code is associated with its specific attributes, such as data type, scope (local or global), and memory location. This structure is essential for semantic analysis, allowing the compiler to verify that variables are declared before use and to enforce scope rules, thereby preventing naming conflicts and type errors.

Algorithm:

Insert Operation:

1. Accept identifier name, type, and scope.
2. Check whether the identifier already exists in the same scope.
3. If it exists, display an error message.
4. Otherwise, insert the identifier into the symbol table.

Lookup Operation:

1. Accept identifier name.
2. Search the symbol table.
3. If found, display its details.
4. If not found, report that the identifier is undeclared.

Display Operation:

1. Print all entries of the symbol table in tabular form.
2. Show identifier name, type, and scope.

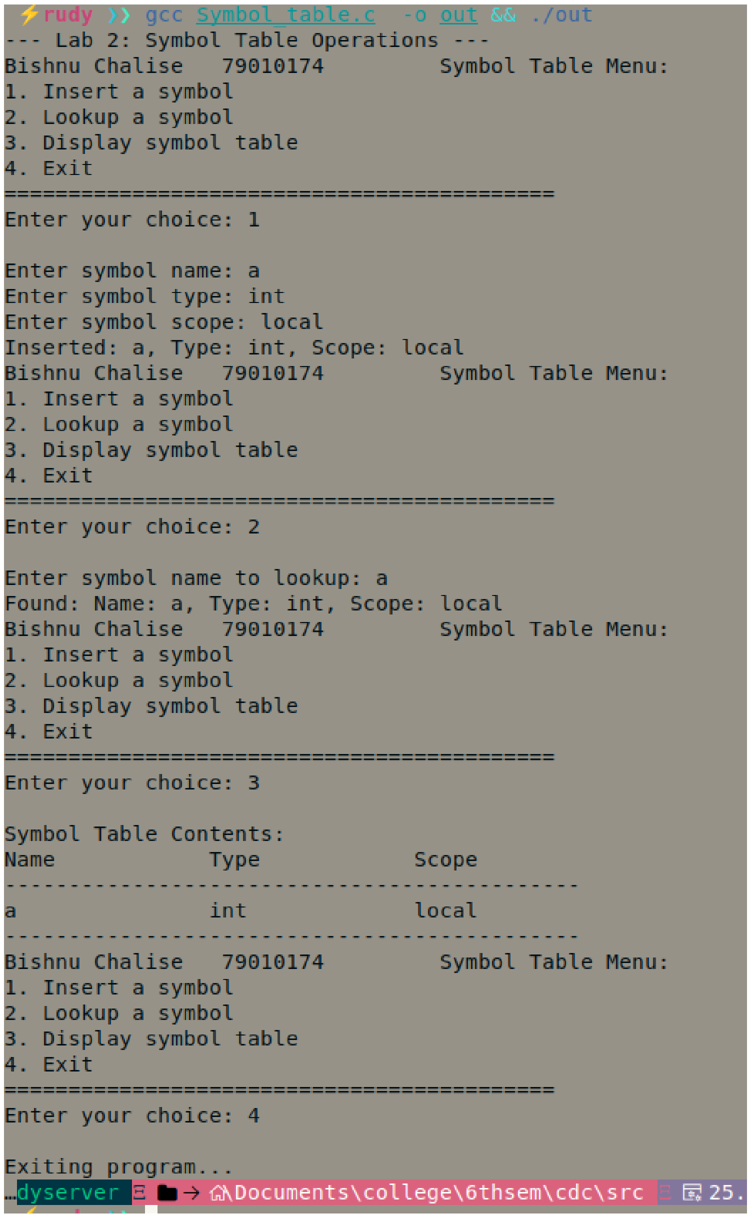
Source Code:

```
 #include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define MAX_SYMBOLS 100
```

```
typedef struct {
char name[50];
char type[50];
char scope[50];
} Symbol;

Symbol symbolTable[MAX_SYMBOLS];
int count = 0;

void insert(const char* name, const char* type, const char* scope) {
for (int i = 0; i < count; i++) {
if (strcmp(symbolTable[i].name, name) == 0 && strcmp(symbolTable[i].scope, scope) == 0) {
printf("Error: Symbol '%s' already exists in scope '%s'.\n", name, scope);
return;
}
}
if (count < MAX_SYMBOLS) {
strcpy(symbolTable[count].name, name);
strcpy(symbolTable[count].type, type);
strcpy(symbolTable[count].scope, scope);
count++;
printf("Inserted: %s, Type: %s, Scope: %s\n", name, type, scope);
} else {
printf("Error: Symbol table full.\n");
}
}
void lookup(const char* name) {
int found = 0;
for (int i = 0; i < count; i++) {
if (strcmp(symbolTable[i].name, name) == 0) {
printf("Found: Name: %s, Type: %s, Scope: %s\n", symbolTable[i].name, symbolTable[i].type,
symbolTable[i].scope);
found = 1;
}
}
if (!found) {
printf("Symbol '%s' not found.\n", name);
}
}
```

```
void display() {
printf("\nSymbol Table Contents:\n");
printf("%-15s %-15s %-15s\n", "Name", "Type", "Scope");
printf("----------------------------------------\n");
for (int i = 0; i < count; i++) {
printf("%-15s %-15s %-15s\n", symbolTable[i].name, symbolTable[i].type,
symbolTable[i].scope);
}
printf("----------------------------------------\n");
}
int main() {
int choice;
char name[50], type[50], scope[50];
printf("--- Lab 2: Symbol Table Operations ---\n");
while (1) {
printf("Bishnu Chalise\t 79010174 \t ");
printf("Symbol Table Menu:\n");
printf("1. Insert a symbol\n");
printf("2. Lookup a symbol\n");
printf("3. Display symbol table\n");
printf("4. Exit\n");
printf("========================================\n");
printf("Enter your choice: ");
scanf("%d", &choice);
switch (choice) {
case 1:
printf("\nEnter symbol name: ");
scanf("%s", name);
printf("Enter symbol type: ");
scanf("%s", type);
printf("Enter symbol scope: ");
scanf("%s", scope);
insert(name, type, scope);
break;
case 2:
printf("\nEnter symbol name to lookup: ");
scanf("%s", name);
lookup(name);
break;
case 3:
display();
break;
case 4:
printf("\nExiting program...\n");
return 0;
default:
printf("\nInvalid choice! Please try again.\n");
}
}
return 0;
}
```

Output:



```
 ⚡rudy 〉〉 gcc Symbol_table.c  -o out && ./out
--- Lab 2: Symbol Table Operations ---
Bishnu Chalise   79010174         Symbol Table Menu:
1. Insert a symbol
2. Lookup a symbol
3. Display symbol table
4. Exit
===========================================
Enter your choice: 1

Enter symbol name: a
Enter symbol type: int
Enter symbol scope: local
Inserted: a, Type: int, Scope: local
Bishnu Chalise   79010174         Symbol Table Menu:
1. Insert a symbol
2. Lookup a symbol
3. Display symbol table
4. Exit
===========================================
Enter your choice: 2

Enter symbol name to lookup: a
Found: Name: a, Type: int, Scope: local
Bishnu Chalise   79010174         Symbol Table Menu:
1. Insert a symbol
2. Lookup a symbol
3. Display symbol table
4. Exit
===========================================
Enter your choice: 3

Symbol Table Contents:
Name            Type            Scope
---------------------------------------------
a               int             local
---------------------------------------------
Bishnu Chalise   79010174         Symbol Table Menu:
1. Insert a symbol
2. Lookup a symbol
3. Display symbol table
4. Exit
===========================================
Enter your choice: 4

Exiting program...
…dyserver   → ⌂\Documents\college\6thsem\cdc\src   25.
```

Conclusion:

In this lab , the symbol table was successfully implemented with functional insertion, lookup, and display capabilities. This experiment demonstrated how compilers manage identifier metadata to ensure semantic consistency. By handling scope and type information, the symbol table served as a critical component in error detection and memory management during the compilation process.

LAB 3: RECURSIVE DESCENT PARSER

Implement a recursive descent parser for the grammar

S → aAb

A → a | ε

Theory:

Syntax analysis checks if the stream of tokens generated by the parser conforms to the grammatical rules of the programming language. Recursive Descent Parsing is a top-down parsing strategy where every non-terminal symbol in the grammar corresponds to a specific recursive function in the code. The parser attempts to construct a parse tree starting from the root (start symbol) and working down to the leaves, matching input characters against the expected production rules.

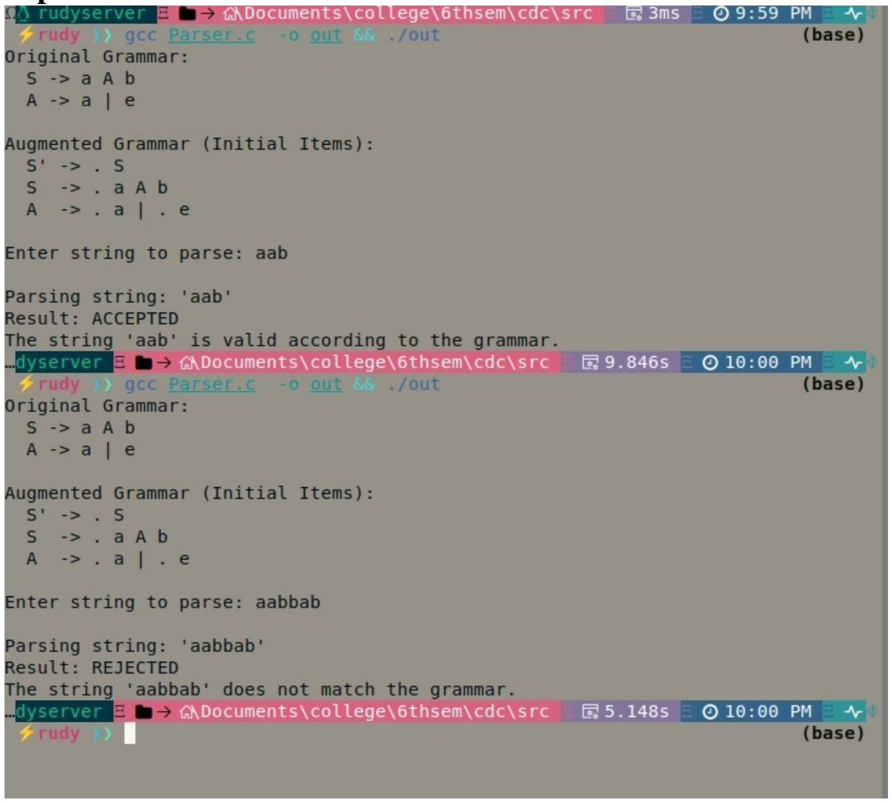
Algorithm:

1. Define recursive functions for each non-terminal (S and A).
2. Start parsing from the start symbol S.
3. For production S → a A b:
    a. Match character 'a'
    b. Call function A
    c. Match character 'b'
4. For production A → a | ε:
    a. If the next input symbol is 'a', match it.
    b. Otherwise, apply ε (empty production).
5. If input is fully parsed and no symbols remain, accept the string.
6. Otherwise, reject the string.

Source Code:

```
#include <stdio.h>
#include <string.h>

char input[100];
int pos, error_flag;

void S();
void A();

void displayGrammar() {
printf("Original Grammar:\n");
printf(" S -> a A b\n");
printf(" A -> a | e\n");
printf("\nAugmented Grammar (Initial Items):\n");
printf(" S' -> . S\n");
printf(" S -> . a A b\n");
printf(" A -> . a | . e\n");
}
void match(char c) {
if (!error_flag && input[pos] == c) pos++;
else error_flag = 1;
}

void A() {
if (!error_flag && input[pos] == 'a') match('a');
}
void S() {
if (error_flag) return;
match('a');
A();
match('b');
}
int parse(const char *str) {
strcpy(input, str);
pos = error_flag = 0;
S();
return (!error_flag && pos == strlen(input));
}
int main() {
char buffer[100];
displayGrammar();
printf("\nEnter string to parse: ");
if (fgets(buffer, sizeof(buffer), stdin)) {
buffer[strcspn(buffer, "\n")] = 0;
printf("\nParsing string: '%s'\n", buffer);
if (parse(buffer))
printf("Result: ACCEPTED\nThe string '%s' is valid according to the grammar.\n", buffer);
else
printf("Result: REJECTED\nThe string '%s' does not match the grammar.\n", buffer);
}
return 0;
}
```

Output:



```
rudyserver  Documents\college\6thsem\cdc\src  3ms  9:59 PM
⚡rudy » gcc Parser.c  -o out && ./out                              (base)
Original Grammar:
  S -> a A b
  A -> a | e

Augmented Grammar (Initial Items):
  S' -> . S
  S  -> . a A b
  A  -> . a | . e

Enter string to parse: aab

Parsing string: 'aab'
Result: ACCEPTED
The string 'aab' is valid according to the grammar.
…dyserver  Documents\college\6thsem\cdc\src  9.846s  10:00 PM
⚡rudy » gcc Parser.c  -o out && ./out                              (base)
Original Grammar:
  S -> a A b
  A -> a | e

Augmented Grammar (Initial Items):
  S' -> . S
  S  -> . a A b
  A  -> . a | . e

Enter string to parse: aabbab

Parsing string: 'aabbab'
Result: REJECTED
The string 'aabbab' does not match the grammar.
…dyserver  Documents\college\6thsem\cdc\src  5.148s  10:00 PM
⚡rudy »                                                            (base)
```

Conclusion:

In this lab, we successfully implemented a recursive descent parser that validates strings against a specific grammar. The experiment highlighted the direct relationship between grammar productions and recursive function calls. The program correctly accepted valid strings (like "ab" and "aab") and rejected invalid ones, demonstrating the efficacy of top-down parsing for standard context-free grammars.

LAB 4: IMPLEMENTATION OF SHIFT-REDUCE PARSING

Implement Shift-Reduce parsing for the following grammar and input string: a +a*a.

E → E + E

E → E * E

E → (E)

E → a

Input String: a+a*a

Theory:

Shift-Reduce parsing is a robust bottom-up technique widely used in modern compilers. Unlike top-down methods, this approach begins with the input string (the leaves of the parse tree) and attempts to reduce it back to the start symbol (the root). It utilizes a stack to hold symbols; the parser either "shifts" the next input symbol onto the stack or "reduces" a sequence of symbols on top of the stack to a non-terminal based on the grammar rules.

Algorithm:

1. Initialize an empty stack and append $ to the input string.
2. Repeat until input is exhausted: o Shift: Move the next input symbol onto the stack. o Reduce: If the top of the stack matches the RHS of any production, replace it with the LHS.
3. Continue shifting and reducing until: o Stack contains only the start symbol E o Input symbol is $
4. If both conditions are satisfied, accept the string.
5. Otherwise, reject the string.

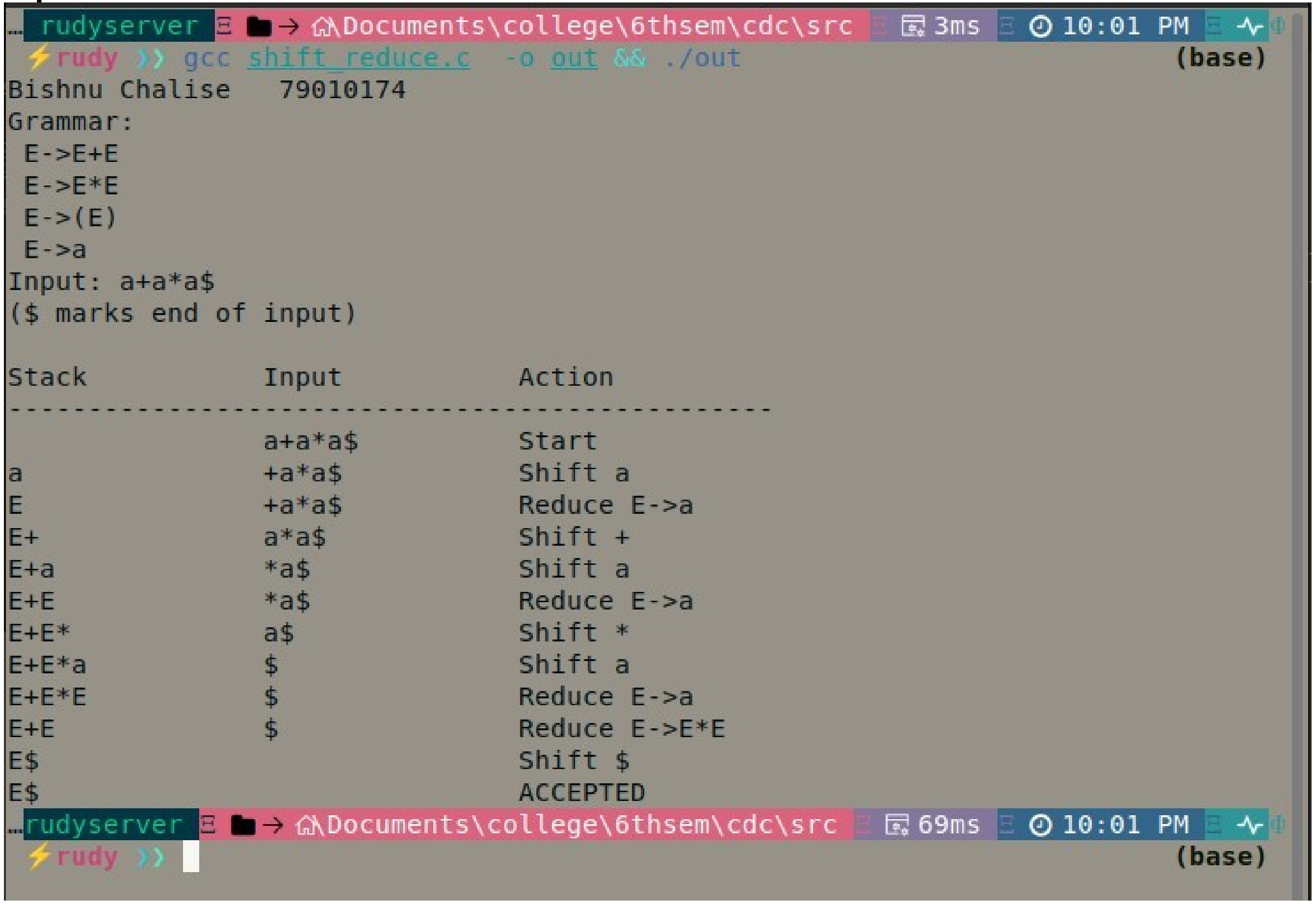
Source Code:

```
#include <stdio.h>
#include <string.h>

char stack[100];
int top = -1;
char input[100];
int pos = 0;
```

```
void push(char c) {
stack[++top] = c;
stack[top+1] = '\0';
}

void pop() {
if (top >= 0) {
stack[top] = '\0';
top--;
}
}
void printState(const char* action) {
printf("%-15s %-15s %s\n", stack, input + pos, action);
}
int main() {
printf("Bishnu Chalise\t 79010174\n");
printf("Grammar:\n E->E+E\n E->E*E\n E->(E)\n E->a\n");
printf("Input: a+a*a$\n");
printf("($ marks end of input)\n\n");
strcpy(input, "a+a*a$");
int len = strlen(input);
printf("%-15s %-15s %s\n", "Stack", "Input", "Action");
printf("-----------------------------------------------\n");

printState("Start");
while (1) {
int reduced = 0;
if (top >= 0 && stack[top] == 'a') {
stack[top] = 'E';
printState("Reduce E->a");
reduced = 1;
}
if (!reduced && top >= 2 && stack[top] == 'E' && stack[top-1] == '*' && stack[top-2] == 'E') {
pop(); pop();
top -= 2;
printState("Reduce E->E*E");
reduced = 1;
}
if (!reduced && top >= 2 && stack[top] == 'E' && stack[top-1] == '+' && stack[top-2] == 'E') {
```

```
if (pos < len && input[pos] == '*') {
} else {
top -= 2;
printState("Reduce E->E+E");
reduced = 1;
}
}
if (!reduced && top >= 2 && stack[top] == ')' && stack[top-1] == 'E' && stack[top-2] == '(') {
stack[top-2] = 'E';
top -= 2;
printState("Reduce E->(E)");
reduced = 1;
}
if (!reduced) {

if (pos < len) {
push(input[pos]);
pos++;
char action[20];
sprintf(action, "Shift %c", stack[top]);
printState(action);
} else {
// End of input - check if we have E$ on stack
if (top == 1 && stack[0] == 'E' && stack[1] == '$') {
printf("%-15s %-15s %s\n", stack, "", "ACCEPTED");
break;
} else {
printf("%-15s %-15s %s\n", stack, "", "REJECTED");
break;
}
}
}
}

return 0;
}
```

Output:



```
… rudyserver  → ⌂\Documents\college\6thsem\cdc\src   3ms   10:01 PM
 ⚡rudy ›› gcc shift_reduce.c  -o out && ./out                          (base)
Bishnu Chalise   79010174
Grammar:
 E->E+E
 E->E*E
 E->(E)
 E->a
Input: a+a*a$
($ marks end of input)

Stack           Input           Action
------------------------------------------------
                a+a*a$          Start
a               +a*a$           Shift a
E               +a*a$           Reduce E->a
E+              a*a$            Shift +
E+a             *a$             Shift a
E+E             *a$             Reduce E->a
E+E*            a$              Shift *
E+E*a           $               Shift a
E+E*E           $               Reduce E->a
E+E             $               Reduce E->E*E
E$                              Shift $
E$                              ACCEPTED
…rudyserver  → ⌂\Documents\college\6thsem\cdc\src   69ms   10:01 PM
 ⚡rudy ››                                                               (base)
```

Conclusion:

The shift-reduce parser was successfully implemented to analyze the arithmetic expression "a+a*a". The step-by-step trace of the stack operations clearly demonstrated the bottom-up parsing logic. This experiment provided a practical understanding of how handles are identified and reduced, serving as a foundation for understanding more complex LR parsing algorithms.

LAB 5: Write a program to generate closure set on LR(0) items for the grammar:

S → A B

A → a, B → b

Theory:

In LR parsing, the parser's state is determined by sets of items. An LR(0) item is simply a grammar production with a distinct marker (a dot) that indicates how much of that production has been processed. The "Closure" operation is a fundamental algorithm used to expand an initial set of items. If the dot precedes a non-terminal, the closure adds all production rules for that non-terminal to the set, effectively predicting all possible structures that might appear next in the input.

Algorithm (Closure of LR(0) Items):

1. Start with an initial set of LR(0) items.
2. For each item of the form A → α . B β, where B is a non-terminal:
   a. Add all productions of B in the form B → . γ to the item set.
3. Repeat step 2 until no new items can be added.
4. The final set of items is called the closure of the initial item set.

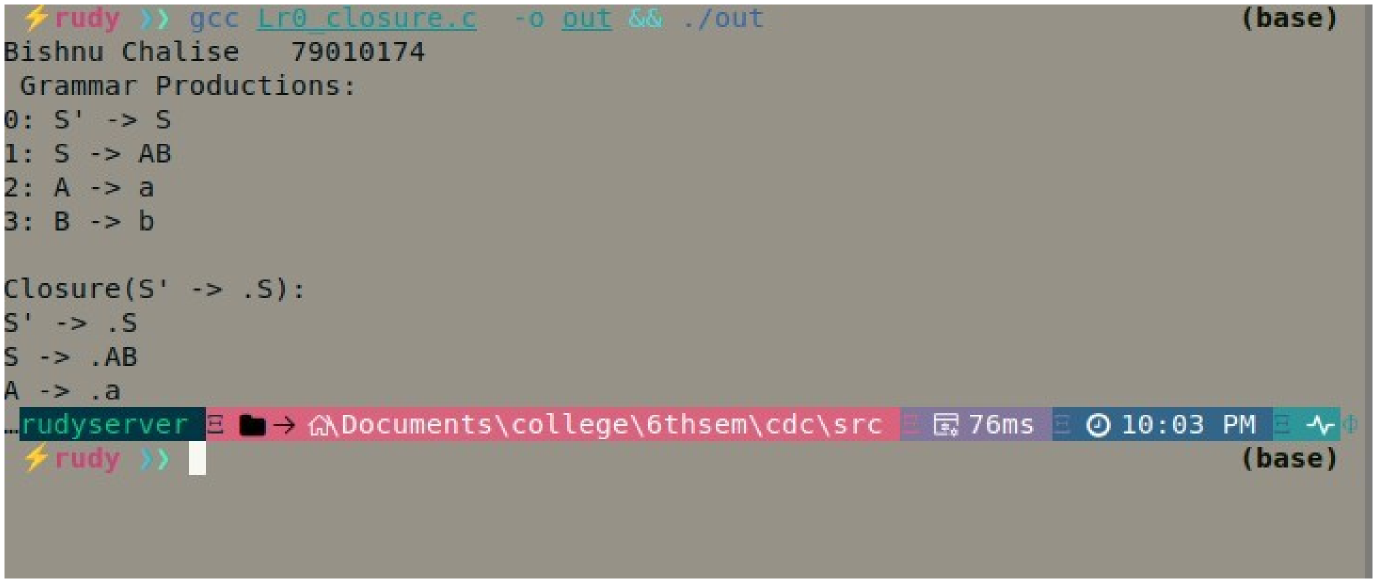
Source Code:

```
#include <stdio.h>
#include <string.h>
#include <stdbool.h>

typedef struct {
char lhs;
char rhs[10];
} Production;

Production rules[] = {
{'Z', "S"},
{'S', "AB"},
{'A', "a"},
{'B', "b"}
};
```

```
int numRules = 4;

typedef struct {
int ruleIndex;
int dotPosition;
} Item;

Item closure[20];
int closureSize = 0;

void additem(int ruleIndex, int dotPos) {
for (int i = 0; i < closureSize; i++) {
if (closure[i].ruleIndex == ruleIndex && closure[i].dotPosition == dotPos)
return;
}
closure[closureSize].ruleIndex = ruleIndex;
closure[closureSize].dotPosition = dotPos;
closureSize++;
}

void computeClosure() {
int changed = 1;
while (changed) {
changed = 0;
int currentSize = closureSize;

for (int i = 0; i < currentSize; i++) {
Item it = closure[i];
char* rhs = rules[it.ruleIndex].rhs;

if (it.dotPosition < strlen(rhs)) {
char nextSymbol = rhs[it.dotPosition];

for (int j = 0; j < numRules; j++) {
if (rules[j].lhs == nextSymbol) {
int prevSize = closureSize;
additem(j, 0);
if (closureSize > prevSize)
changed = 1;
}}}}}}

void printItem(Item it) {
char* lhsChar;
if (rules[it.ruleIndex].lhs == 'Z') {
lhsChar = "S'";
} else {
static char buf[2];
buf[0] = rules[it.ruleIndex].lhs;
buf[1] = '\0';
lhsChar = buf;
}

printf("%s -> ", lhsChar);
char* rhs = rules[it.ruleIndex].rhs;
for (int i = 0; i < strlen(rhs); i++) {
if (i == it.dotPosition)
printf(".");
printf("%c", rhs[i]);
}
if (it.dotPosition == strlen(rhs))
printf(".");
printf("\n");
}

int main() {

printf("Bishnu Chalise\t 79010174 \n Grammar Productions:\n");
for (int i = 0; i < numRules; i++) {
char* lhsChar;
if (rules[i].lhs == 'Z') {
lhsChar = "S'";
} else {
static char buf[2];
buf[0] = rules[i].lhs;
buf[1] = '\0';
lhsChar = buf;
}
printf("%d: %s -> %s\n", i, lhsChar, rules[i].rhs);
```

```
}
printf("\n");
additem(0, 0);
computeClosure();
printf("Closure(S' -> .S):\n");
for (int i = 0; i < closureSize; i++) {
printItem(closure[i]);
}
return 0;
}
```

Output:



```
⚡rudy ❯❯ gcc Lr0_closure.c  -o out && ./out                               (base)
Bishnu Chalise   79010174
 Grammar Productions:
0: S' -> S
1: S -> AB
2: A -> a
3: B -> b

Closure(S' -> .S):
S' -> .S
S -> .AB
A -> .a
…rudyserver  → ⌂\Documents\college\6thsem\cdc\src  76ms  10:03 PM
⚡rudy ❯❯                                                                   (base)
```

Conclusion:

The program successfully computed the closure set of LR(0) items for the provided grammar. This experiment illustrated how the parser anticipates future input by expanding non-terminals that appear after the current parsing position (the dot). Mastering closure computation is essential for constructing the canonical collection of LR items used in SLR, LALR, and CLR parsers.

LAB 6: INTERMEDIATE CODE GENERATION

Write a program to generate three-address code for arithmetic assignment statement.

Theory:

Intermediate Code Generation (ICG) serves as a bridge between the high-level syntax of source code and the low-level machine code. By converting the parse tree into an intermediate representation like Three-Address Code (TAC), the compiler can perform optimizations that are independent of the specific target machine. In TAC, complex expressions are broken down into a sequence of instructions where each instruction comprises at most one operator and three operands (two inputs and one result).

Algorithm:

1. Read an arithmetic assignment expression from the user.
2. Identify operators based on precedence (* and / before + and -).
3. Generate temporary variables (t1, t2, …) for intermediate results.
4. Convert the expression step-by-step into three-address statements.
5. Display the generated TAC.

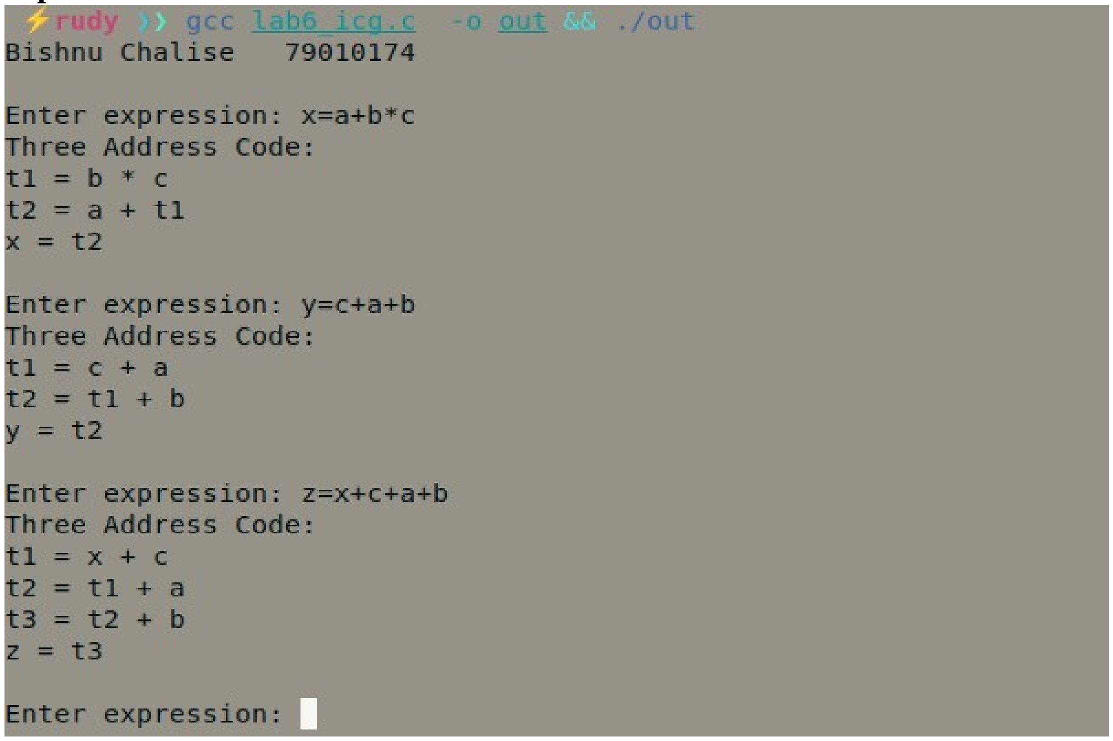
Source Code:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
char stack[100][10];
int top = -1;
int tempCount = 1;

void push(char* str) {
strcpy(stack[++top], str);
}
void pop(char* dest) {
strcpy(dest, stack[top--]);
}
void newTemp(char* dest) {
sprintf(dest, "t%d", tempCount++);
}
int getPrecedence(char op) {
```

```
if (op == '*' || op == '/') return 2;
if (op == '+' || op == '-') return 1;
return 0;
}

void process(char op) {
char right[10], left[10], temp[10];
pop(right);
pop(left);
newTemp(temp);
printf("%s = %s %c %s\n", temp, left, op, right);
push(temp);
}

int main() {
char input[200];
printf("Bishnu Chalise\t 79010174\n");
while (1) {
printf("\nEnter expression (or empty to exit): ");
if (fgets(input, sizeof(input), stdin) == NULL) break;

input[strcspn(input, "\n")] = 0;
if (strlen(input) == 0) break;

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

top = -1;
tempCount = 1;

char opStack[100];
int opTop = -1;
char lhs[50];
int i = 0;

while (isspace(input[i])) i++;

int k = 0;
while (input[i] != '=' && input[i] != '\0') {
lhs[k++] = input[i++];
```

```
}
lhs[k] = '\0';

if (input[i] == '=') i++;
else {
printf("Error: format should be 'variable = expression'\n");
continue;
}

while (input[i] != '\0') {
if (isspace(input[i])) { i++; continue; }

if (isalnum(input[i])) {
char operand[50];
int m = 0;
while (isalnum(input[i])) {
operand[m++] = input[i++];
}
operand[m] = '\0';
push(operand);
continue;
}

if (input[i] == '(') {
opStack[++opTop] = input[i++];
continue;
}
if (input[i] == ')') {
while (opTop >= 0 && opStack[opTop] != '(') {
process(opStack[opTop--]);
}
if (opTop >= 0 && opStack[opTop] == '(') opTop--;
i++;
continue;
}

if (strchr("+-*/", input[i])) {
while (opTop >= 0 && getPrecedence(opStack[opTop]) >= getPrecedence(input[i])) {
process(opStack[opTop--]);
}
opStack[++opTop] = input[i++];
continue;
}

i++;
}

while (opTop >= 0) {
process(opStack[opTop--]);
}

char finalRes[10];
if (top >= 0) {
pop(finalRes);
printf("%s = %s\n", lhs, finalRes);
} else {
printf("Error parsing expression.\n");
}
}

return 0;
}
```

Output:



```
⚡rudy ❯❯ gcc lab6_icg.c  -o out && ./out
Bishnu Chalise   79010174

Enter expression: x=a+b*c
Three Address Code:
t1 = b * c
t2 = a + t1
x = t2

Enter expression: y=c+a+b
Three Address Code:
t1 = c + a
t2 = t1 + b
y = t2

Enter expression: z=x+c+a+b
Three Address Code:
t1 = x + c
t2 = t1 + a
t3 = t2 + b
z = t3

Enter expression: 
```

Conclusion:

We successfully implemented a generator for Three-Address Code. The program correctly decomposed complex arithmetic expressions into simplified, sequential instructions using temporary variables. This experiment demonstrated the utility of intermediate representations in simplifying the translation process from high-level logic to executable machine instructions.

LAB 7: TARGET CODE GENERATION

Write a program to generate target code for a simple register-based machine.

Theory:

Target Code Generation is the final stage of the compilation pipeline. In this phase, the optimized intermediate representation is mapped to the specific instruction set architecture (ISA) of the target hardware. This experiment simulates a simple register-based machine. The goal is to translate abstract operations into concrete assembly instructions, handling the loading of data into registers, performing arithmetic, and storing results back to memory.

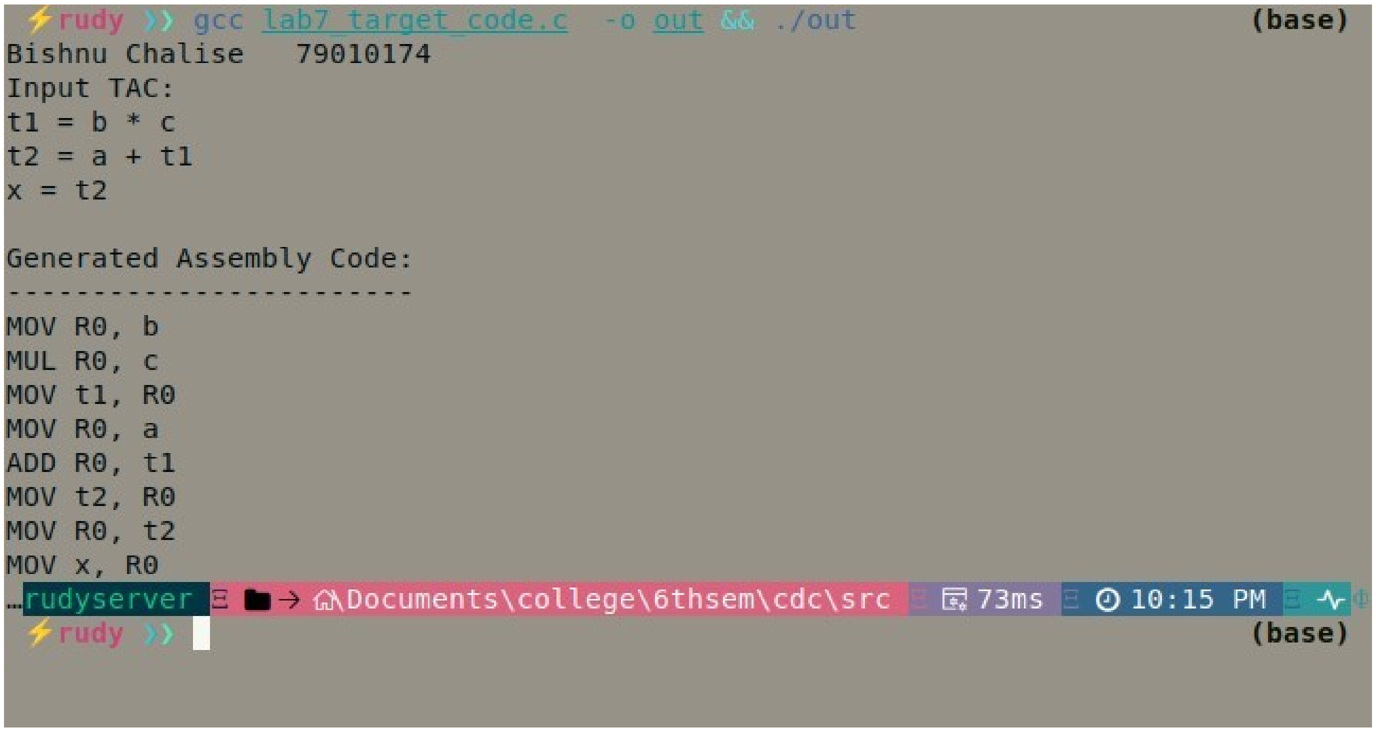
Algorithm:

1. Read an arithmetic expression from the user.
2. Identify operands and operators from the expression.
3. Load operands into registers using MOV instructions.
4. Perform arithmetic operations using instructions like ADD, SUB, MUL, or DIV.
5. Store the final result in the destination variable.
6. Display the generated target code.

Source Code:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

typedef struct {
char result[10];
char arg1[10];
char op[5];
char arg2[10];
} Quadruple;

int main() {
printf("Bishnu Chalise\t 79010174\n");
Quadruple q[] = {
{"t1", "p", "+", "q"},
{"t2", "t1", "*", "r"},
{"y", "t2", "=", ""}
};
int n = 3;
printf("Input TAC:\n");
for(int i=0; i<n; i++) {
if (strcmp(q[i].op, "=") == 0) {
printf("%s = %s\n", q[i].result, q[i].arg1);
} else {
printf("%s = %s %s %s\n", q[i].result, q[i].arg1, q[i].op, q[i].arg2);
}
}
printf("\nGenerated Assembly Code:\n");
printf("------------------------\n");
for(int i=0; i<n; i++) {
if (strcmp(q[i].op, "*") == 0) {
printf("MOV R0, %s\n", q[i].arg1);
printf("MUL R0, %s\n", q[i].arg2);
printf("MOV %s, R0\n", q[i].result);
} else if (strcmp(q[i].op, "+") == 0) {
printf("MOV R0, %s\n", q[i].arg1);
printf("ADD R0, %s\n", q[i].arg2);
printf("MOV %s, R0\n", q[i].result);
} else if (strcmp(q[i].op, "-") == 0) {
printf("MOV R0, %s\n", q[i].arg1);
printf("SUB R0, %s\n", q[i].arg2);
printf("MOV %s, R0\n", q[i].result);
} else if (strcmp(q[i].op, "/") == 0) {
printf("MOV R0, %s\n", q[i].arg1);
printf("DIV R0, %s\n", q[i].arg2);
printf("MOV %s, R0\n", q[i].result);
} else if (strcmp(q[i].op, "=") == 0) {
printf("MOV R0, %s\n", q[i].arg1);
printf("MOV %s, R0\n", q[i].result);
}
}
return 0;
}
```

Output:



```
 ⚡rudy ❯❯ gcc lab7_target_code.c  -o out && ./out                              (base)
Bishnu Chalise   79010174
Input TAC:
t1 = b * c
t2 = a + t1
x = t2

Generated Assembly Code:
------------------------
MOV R0, b
MUL R0, c
MOV t1, R0
MOV R0, a
ADD R0, t1
MOV t2, R0
MOV R0, t2
MOV x, R0
…rudyserver  → ⌂\Documents\college\6thsem\cdc\src  73ms  10:15 PM
 ⚡rudy ❯❯                                                                       (base)
```

Conclusion:

The program successfully generated assembly-level target code from the input expression. By simulating a register-based architecture, we demonstrated the final translation step where logical operations are converted into physical machine instructions. This validates the process of bridging the gap between software logic and hardware execution.