Lab 1: Implementing Lexical Analyzer

To implement a lexical analyzer in C that recognizes identifiers, keywords, constants, strings, comments, operators, and symbols, and displays their token type along with the lexeme.

Introduction

A lexical analyzer is the first phase of a compiler. It scans the source program character by character and groups them into meaningful sequences called **lexemes**.
Each lexeme is classified into a **token** such as keyword, identifier, operator, constant, string literal, or comment.

The output of the lexical analyzer is a stream of tokens, which simplifies the work of later compiler phases like syntax analysis.

Algorithm

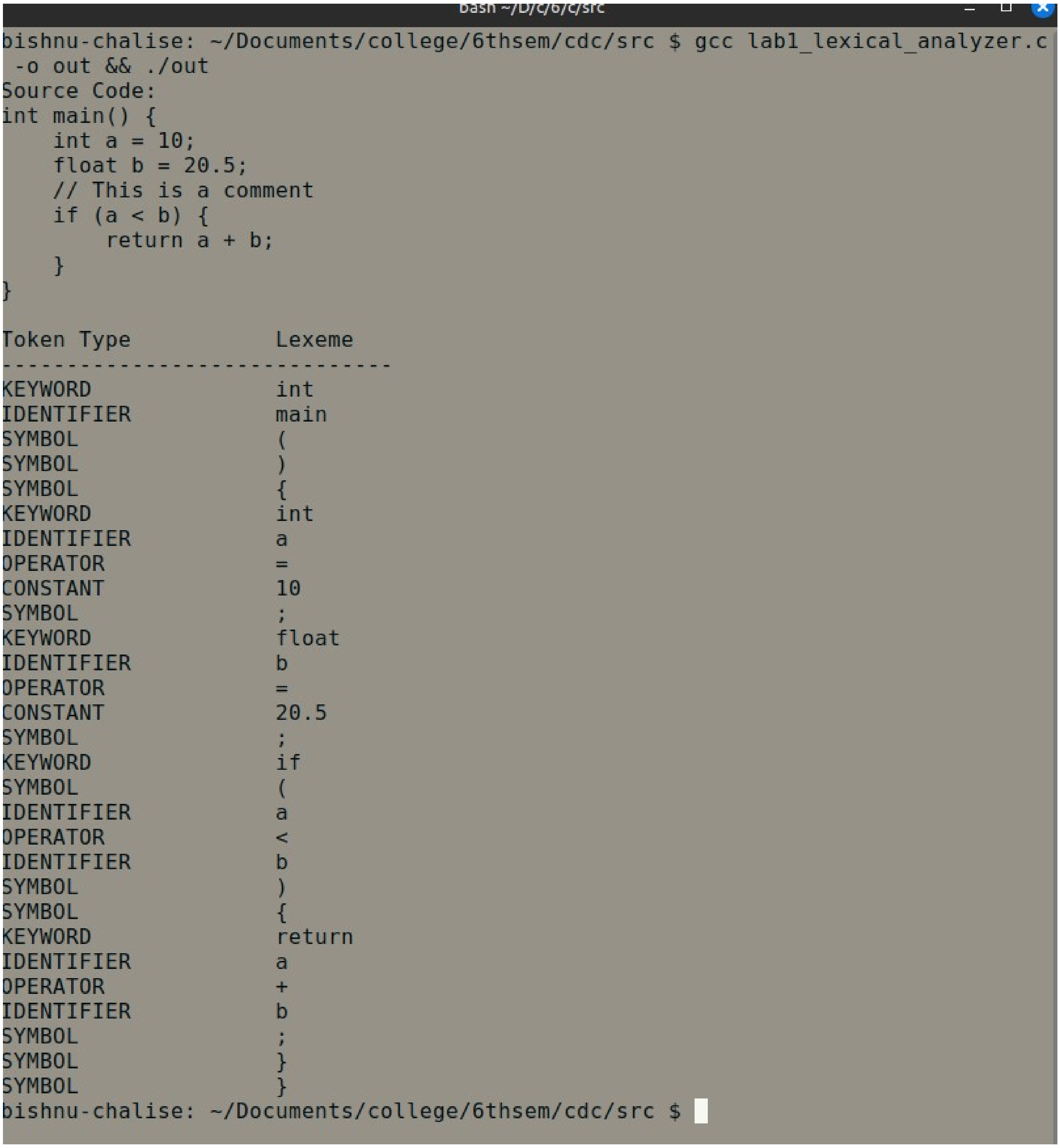
1. Read the source code from an input file.
2. Initialize a pointer to the first character.
3. Repeat until end of file:
    - Ignore whitespace and newline characters.
    - If the character starts with:
        - **Letter or underscore** → read full word → check keyword or identifier.
        - **Digit**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

Source Code

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

// Token definitions
typedef enum {
    KEYWORD, IDENTIFIER, CONSTANT, STRING, OPERATOR, SYMBOL, 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;
}

const char* getTokenTypeName(TokenType type) {
    switch (type) {
        case KEYWORD: return "KEYWORD";
        case IDENTIFIER: return "IDENTIFIER";
        case CONSTANT: return "CONSTANT";
        case STRING: return "STRING";
        case OPERATOR: return "OPERATOR";
        case SYMBOL: return "SYMBOL";
        default: return "UNKNOWN";
    }
}

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;
     }
```

```c
        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 (isKeyword(buffer)) {
                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", "SYMBOL", input[i]);
             i++;
             continue;
        }

        i++;
    }
}

int main() {
    const char* sourceCode =
        "int main() {\n"

        "    int a = 10;\n"
        "    float b = 20.5;\n"
        "    // This is a comment\n"
        "    if (a < b) {\n"
        "        return a + b;\n"
        "    }\n"
        "}";

    printf("Source Code:\n%s\n\n", sourceCode);
    analyze(sourceCode);
    return 0;
}
```

Output



```
bishnu-chalise: ~/Documents/college/6thsem/cdc/src $ gcc lab1_lexical_analyzer.c
 -o out && ./out
Source Code:
int main() {
    int a = 10;
    float b = 20.5;
    // This is a comment
    if (a < b) {
        return a + b;
    }
}

Token Type           Lexeme
------------------------------
KEYWORD              int
IDENTIFIER           main
SYMBOL               (
SYMBOL               )
SYMBOL               {
KEYWORD              int
IDENTIFIER           a
OPERATOR             =
CONSTANT             10
SYMBOL               ;
KEYWORD              float
IDENTIFIER           b
OPERATOR             =
CONSTANT             20.5
SYMBOL               ;
KEYWORD              if
SYMBOL               (
IDENTIFIER           a
OPERATOR             <
IDENTIFIER           b
SYMBOL               )
SYMBOL               {
KEYWORD              return
IDENTIFIER           a
OPERATOR             +
IDENTIFIER           b
SYMBOL               ;
SYMBOL               }
SYMBOL               }
bishnu-chalise: ~/Documents/college/6thsem/cdc/src $
```

Conclusion

The lexical analyzer successfully identifies keywords, identifiers, constants, strings, comments, operators, and symbols from the source code. This experiment demonstrates the first phase of c ompiler design, where raw source code is converted into a structured sequence of tokens, makin g further analysis easier and more efficient.

Lab 2: Implementing Symbol Table Operations

To implement a symbol table in C that supports **insert**, **lookup**, and **display** operations while maintaining attributes such as identifier name, data type, and scope.

Introduction

A **symbol table** is an important data structure used by a compiler to store information about identifiers appearing in a program. These identifiers may represent variables, functions, constants, or objects.

Each symbol table entry generally contains:

- Identifier name
- Data type
- Scope (global or local)

The symbol table helps the compiler detect errors such as multiple declarations, undeclared variables, and scope violations during compilation.

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

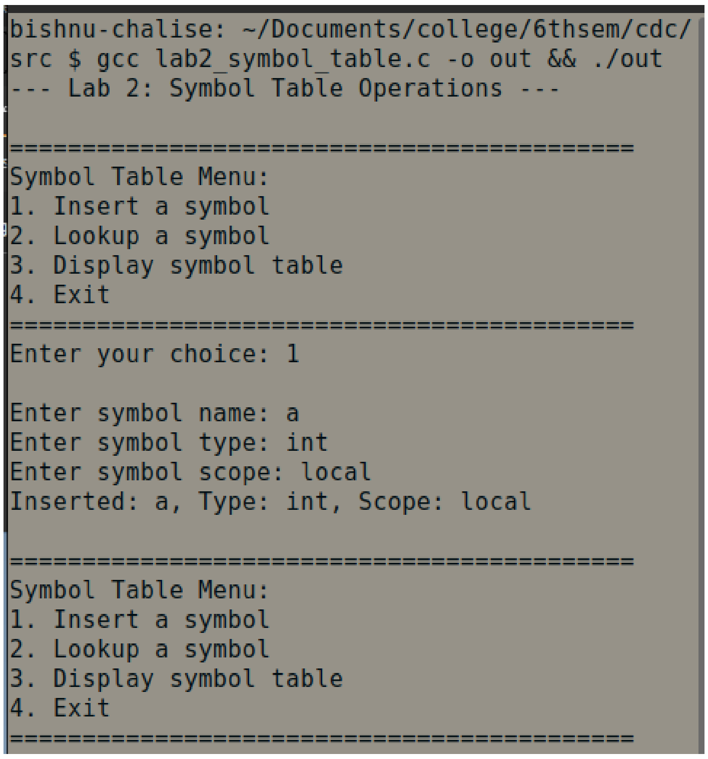
Display Operation

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

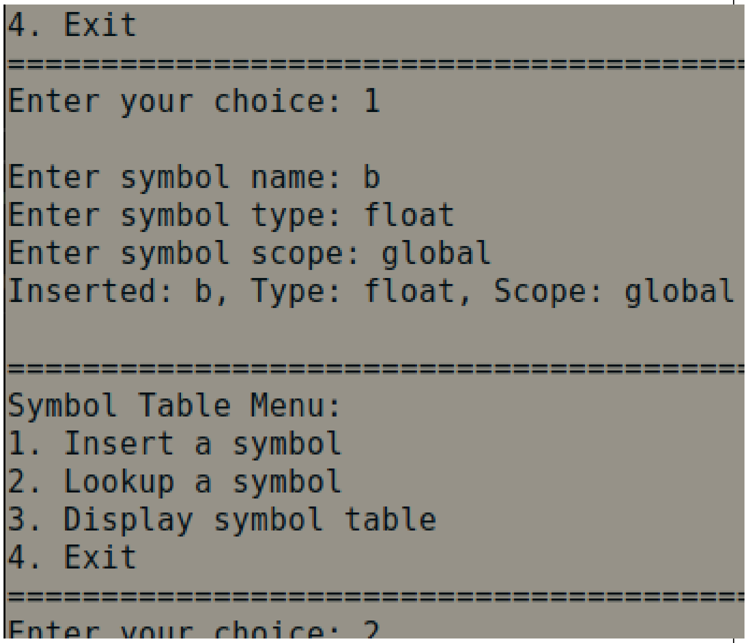
Source Code

```
#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("\n===========================================\n");
        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;
}
```

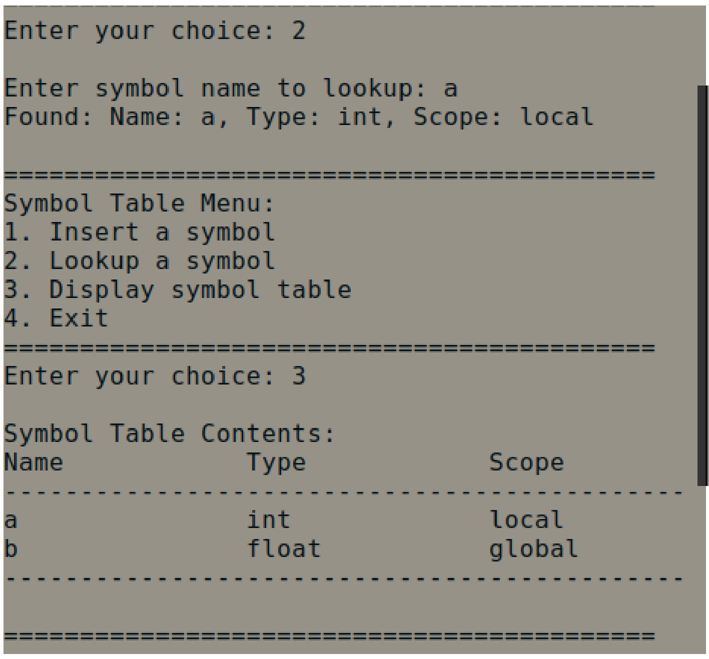
Output



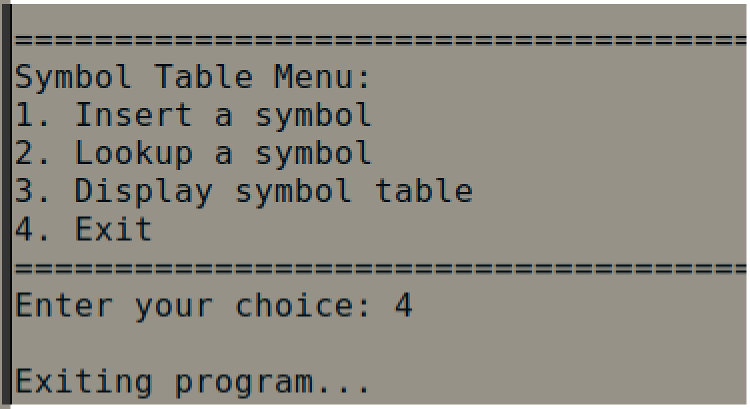
```
bishnu-chalise: ~/Documents/college/6thsem/cdc/
src $ gcc lab2_symbol_table.c -o out && ./out
--- Lab 2: Symbol Table Operations ---

============================================
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

============================================
Symbol Table Menu:
1. Insert a symbol
2. Lookup a symbol
3. Display symbol table
4. Exit
============================================
```



```
4. Exit
========================================
Enter your choice: 1

Enter symbol name: b
Enter symbol type: float
Enter symbol scope: global
Inserted: b, Type: float, Scope: global

========================================
Symbol Table Menu:
1. Insert a symbol
2. Lookup a symbol
3. Display symbol table
4. Exit
========================================
Enter your choice: 2
```



```
Enter your choice: 2

Enter symbol name to lookup: a
Found: Name: a, Type: int, Scope: local

============================================
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
b                   float               global
----------------------------------------------

============================================
```



```
========================================
Symbol Table Menu:
1. Insert a symbol
2. Lookup a symbol
3. Display symbol table
4. Exit
========================================
Enter your choice: 4

Exiting program...
```

Conclusion

The symbol table implementation successfully demonstrates **insert**, **lookup**, and **display** operations. This experiment shows how a compiler stores and manages identifier information and helps detect semantic errors such as duplicate declarations and undeclared variables.
The symbol table plays a crucial role in ensuring correctness during compilation.

Lab 3: Recursive Descent Parser

To implement a **recursive descent parser** in C for the given grammar and check whether an inp ut string is accepted or rejected.

Grammar:

```
S → a A b
A → a | ε
```

Introduction

Parsing is the phase of a compiler that checks whether the sequence of tokens follows the gram matical structure of a programming language. **Recursive Descent Parsing** is a top-down parsin g technique where each non-terminal in the grammar is implemented as a recursive function.

The parser starts from the start symbol and attempts to derive the input string by recursively app lying grammar rules. If the entire input is consumed successfully, the string is accepted; otherwi se, it is rejected.

Algorithm

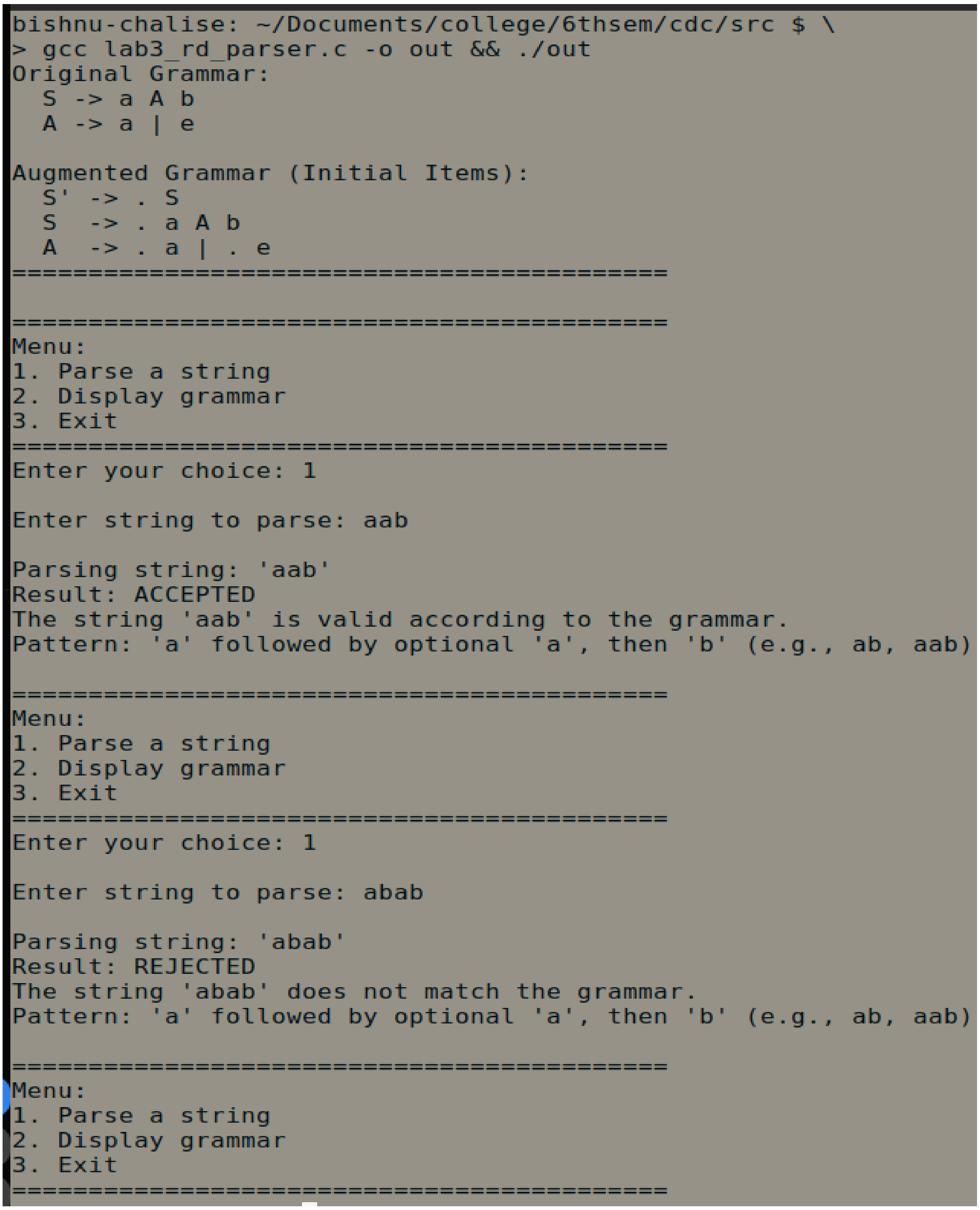
1. Define recursive functions for each non-terminal (`S` and `A`).
2. Start parsing from the start symbol `S`.
3. For production `S → a A b`:
   - Match character `a`
   - Call function `A`
   - Match character `b`
4. For production `A → a | ε`:
   - If the next input symbol is `a`, match it
   - Otherwise, 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 <stdlib.h>
#include <string.h>
#include <ctype.h>

char input[100];
int pos = 0;
int error_flag = 0;


void S();
void A();
void displayGrammar();
int parse(const char* str);

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");
    printf("===========================================\n");
}

void match(char expected) {
    if (error_flag) return;

    if (input[pos] == expected) {
        pos++;
    } else {
        error_flag = 1;
    }
}

void A() {
    if (error_flag) return;


    if (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 = 0;
    error_flag = 0;


    S();
```

```c
    if (!error_flag && pos == strlen(input)) {
        return 1;
    } else {
        return 0;
    }
}

int main() {
    char buffer[100];
    int choice;

    displayGrammar();

    while (1) {
        printf("\n==========================================\n");
        printf("Menu:\n");
        printf("1. Parse a string\n");
        printf("2. Display grammar\n");
        printf("3. Exit\n");
        printf("==========================================\n");
        printf("Enter your choice: ");

        if (scanf("%d", &choice) != 1) {

            while (getchar() != '\n');
            printf("\nInvalid input! Please enter a number.\n");
            continue;
        }
        getchar();

        switch (choice) {
            case 1:
                printf("\nEnter string to parse: ");
                if (fgets(buffer, sizeof(buffer), stdin) != NULL) {

                    buffer[strcspn(buffer, "\n")] = 0;

                    printf("\nParsing string: '%s'\n", buffer);

                    if (parse(buffer)) {
                        printf("Result: ACCEPTED\n");
                        printf("The string '%s' is valid according to the
                              grammar.\n", buffer);
                    } else {
                        printf("Result: REJECTED\n");
                        printf("The string '%s' does not match the grammar.\n",
                              buffer);
                    }

                    printf("Pattern: 'a' followed by optional 'a', then 'b'
                        (e.g., ab, aab)\n");
                }
                break;

            case 2:
                displayGrammar();
                break;

            case 3:
                printf("\nExiting program...\n");
                return 0;

            default:
                printf("\nInvalid choice! Please try again.\n");
        }
    }

    return 0;
  }
```

Output



```
bishnu-chalise: ~/Documents/college/6thsem/cdc/src $ \
> gcc lab3_rd_parser.c -o out && ./out
Original Grammar:
  S -> a A b
  A -> a | e

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

===========================================
Menu:
1. Parse a string
2. Display grammar
3. Exit
===========================================
Enter your choice: 1

Enter string to parse: aab

Parsing string: 'aab'
Result: ACCEPTED
The string 'aab' is valid according to the grammar.
Pattern: 'a' followed by optional 'a', then 'b' (e.g., ab, aab)

===========================================
Menu:
1. Parse a string
2. Display grammar
3. Exit
===========================================
Enter your choice: 1

Enter string to parse: abab

Parsing string: 'abab'
Result: REJECTED
The string 'abab' does not match the grammar.
Pattern: 'a' followed by optional 'a', then 'b' (e.g., ab, aab)

===========================================
Menu:
1. Parse a string
2. Display grammar
3. Exit
===========================================
```

Conclusion

The recursive descent parser correctly analyzes the input string based on the given grammar.
This experiment demonstrates the working of a **top-down parsing technique**, where grammar

Lab 4: Implementation of Shift-Reduce Parsing

To implement a **Shift-Reduce Parser** in C for the given grammar and check whether the input s tring is successfully parsed.

Grammar:

```
E → E + E
E → E * E
E → ( E )
E → a
```

Input String: `a+a*a`

Introduction

Shift-Reduce parsing is a **bottom-up parsing technique** used in compiler design.
The parser works by shifting input symbols onto a stack and then reducing sequences of symbol s (handles) on the stack to non-terminals according to grammar rules.

This method continues until the entire input string is reduced to the start symbol of the grammar. Shift-Reduce parsing forms the basis for LR parsers such as **SLR, LALR, and LR(1)**.

Algorithm

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

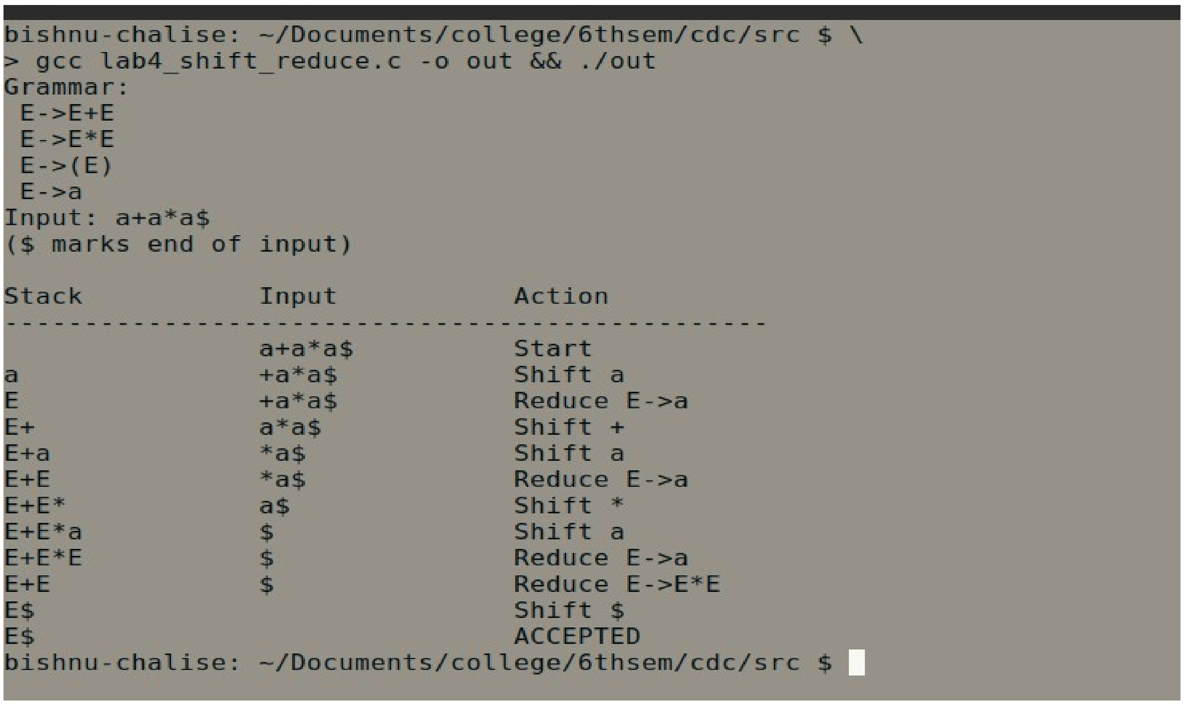
}

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("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 {

                    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



```
bishnu-chalise: ~/Documents/college/6thsem/cdc/src $ \
> gcc lab4_shift_reduce.c -o out && ./out
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
bishnu-chalise: ~/Documents/college/6thsem/cdc/src $
```

Conclusion

The shift-reduce parser successfully parses the given input string a+a*a using bottom-up parsin g. This experiment demonstrates how shifting and reducing operations work together to reduce the input to the start symbol. Shift-Reduce parsing is an important foundation for understanding LR parsing techniques used in modern compilers.

Lab 5: Generation of Closure Set of LR(0) Items

To write a C program that generates the **closure set of LR(0) items** for a given grammar.

Grammar:

```
S → A B
A → a
B → b
```

Introduction

LR parsing is a powerful bottom-up parsing technique used in compiler design.
An **LR(0) item** is a production with a dot (.) indicating how much of the production has been re cognized.

The **closure operation** is used to expand a given set of LR(0) items by adding all possible produ ctions that may occur next during parsing. Closure is a fundamental step in constructing **canoni cal LR(0) item sets**, which are later used to build parsing tables.

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:
   - 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.

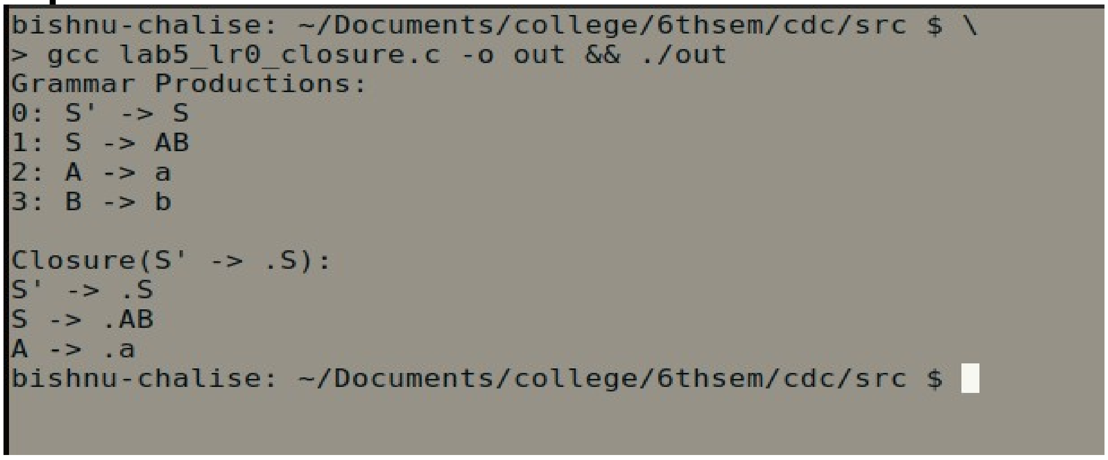
Source Code

```
#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("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



```
bishnu-chalise: ~/Documents/college/6thsem/cdc/src $ \
> gcc lab5_lr0_closure.c -o out && ./out
Grammar Productions:
0: S' -> S
1: S -> AB
2: A -> a
3: B -> b

Closure(S' -> .S):
S' -> .S
S -> .AB
A -> .a
bishnu-chalise: ~/Documents/college/6thsem/cdc/src $
```

Conclusion

The closure set of LR(0) items was successfully generated for the given grammar.
This experiment demonstrates how new items are added when a non-terminal appears immediat ely after the dot. Closure computation is an essential step in building LR parsing tables used in S LR, LALR, and LR(1) parsers.

Lab 6: Intermediate Code Generation

To implement a C program that generates **Three-Address Code (TAC)** for a given arithmetic a ssignment expression.

Introduction

Intermediate Code Generation is an important phase of a compiler that comes after syntax analy sis.
Instead of directly producing machine code, the compiler first generates an **intermediate repre sentation** that is easy to analyze and optimize.

Three-Address Code (TAC) is a common intermediate form where each instruction contains a t most three operands. It simplifies complex expressions by using temporary variables and helps in optimization and target code generation.

Algorithm

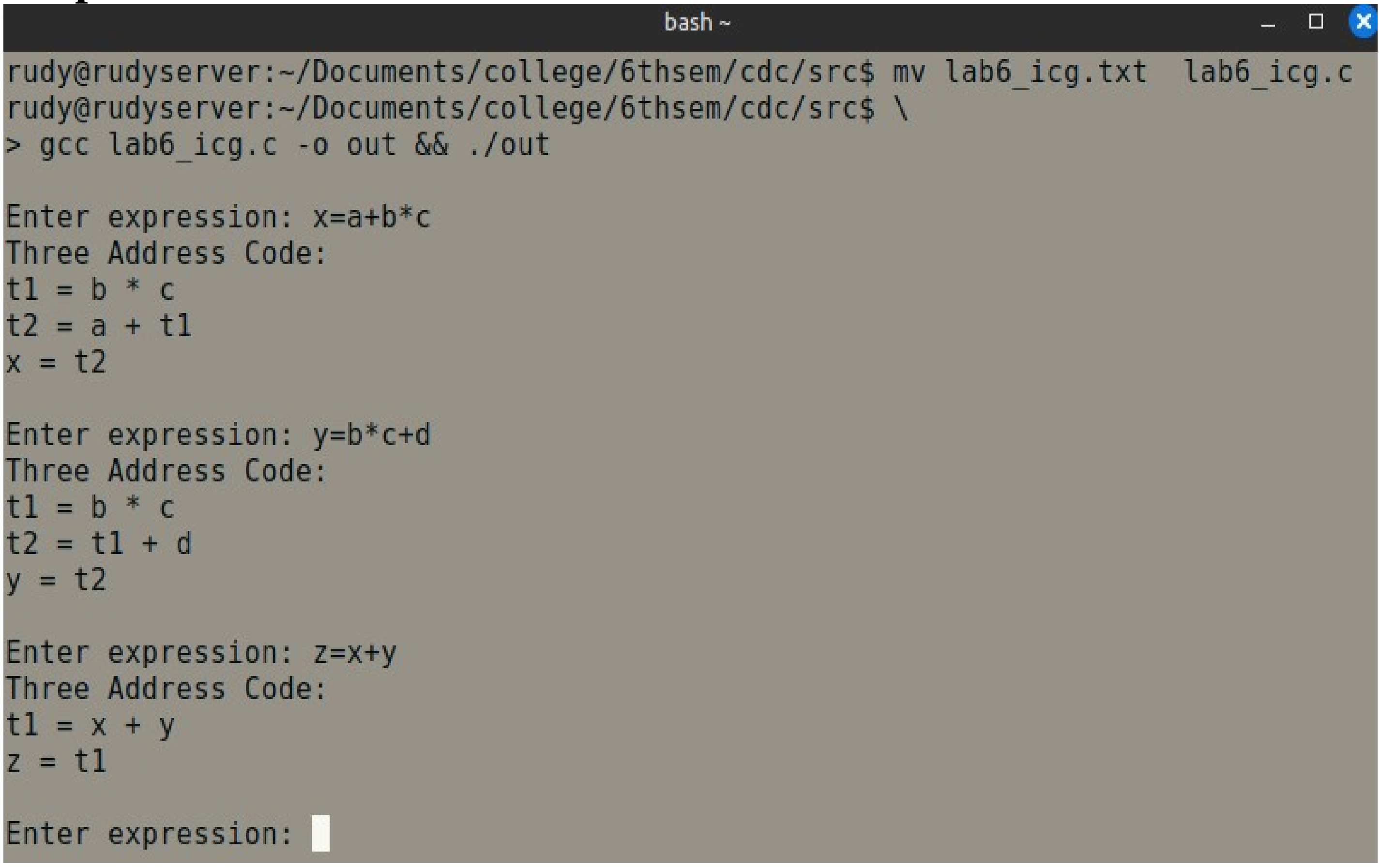
1. Read an arithmetic assignment expression from the user.
2. Identify operators based on precedence (`*` and `/` before `+` and `-`).
3. Generate temporary variables (`t1`, `t2`, …) for intermediate results.
4. Convert the expression step-by-step into three-address statements.
5. Display the generated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

Source Code

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

/*
 Simple Intermediate Code Generator
 Input: x = a + b * c
 Output:
 t1 = b * c
 t2 = a + t1
 x = t2

 We will parse a simple expression with +, *, =.
 Assumption: Input is valid and simple (Single digit or char variables).
 Priority: * over +
*/

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--]);
}
```

```c
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[100];

    while (1) {
        printf("\nEnter expression: ");
        if (fgets(input, sizeof(input), stdin) == NULL) break;

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

        if (strlen(input) == 0) continue;

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

        top = -1;
        tempCount = 1;

        char opStack[100];
        int opTop = -1;

        char lhs[10];
        int i = 0;


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


        if (input[i] == '=') {
            i++;
        } else {

            if (i == strlen(input)) {
                printf("Error: format should be 'variable = expression'\n");
                continue;
            }
        }
        for (; input[i] != '\0'; i++) {
            if (isalnum(input[i])) {
                char operand[2] = {input[i], '\0'};
                push(operand);
            } else if (input[i] == '(') {
                opStack[++opTop] = input[i];
            } else if (input[i] == ')') {
                while (opTop >= 0 && opStack[opTop] != '(') {
                    process(opStack[opTop--]);
                }
                if (opTop >= 0 && opStack[opTop] == '(') {
                    opTop--;
                }
            } else if (strchr("+-*/", input[i])) {
                while (opTop >= 0 && getPrecedence(opStack[opTop]) >=
getPrecedence(input[i])) {
                    process(opStack[opTop--]);
                }
                opStack[++opTop] = input[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@rudyserver:~/Documents/college/6thsem/cdc/src$ mv lab6_icg.txt  lab6_icg.c
rudy@rudyserver:~/Documents/college/6thsem/cdc/src$ \
> gcc lab6_icg.c -o out && ./out

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

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

Enter expression: z=x+y
Three Address Code:
t1 = x + y
z = t1

Enter expression: 
```

Conclusion

The program successfully generates **Three-Address Code** for an arithmetic assignment statem
ent.
This experiment demonstrates how complex expressions are broken down into simpler instructi
ons using temporary variables. Intermediate code generation forms a bridge between syntax ana
lysis and target code generation, making compiler optimization easier.

Lab 7: Target Code Generation

To write a C program that generates **target code** for a simple **register-based machine** from an arithmetic assignment expression.

Introduction

Target code generation is the **final phase of a compiler**, where intermediate code is converted i nto machine-level instructions. The generated target code is usually close to assembly language and is optimized for execution on a specific machine architecture.

In this experiment, a simple **register-based machine model** is assumed. Operands are loaded i nto registers, arithmetic operations are performed using registers, and the final result is stored b ack into memory.

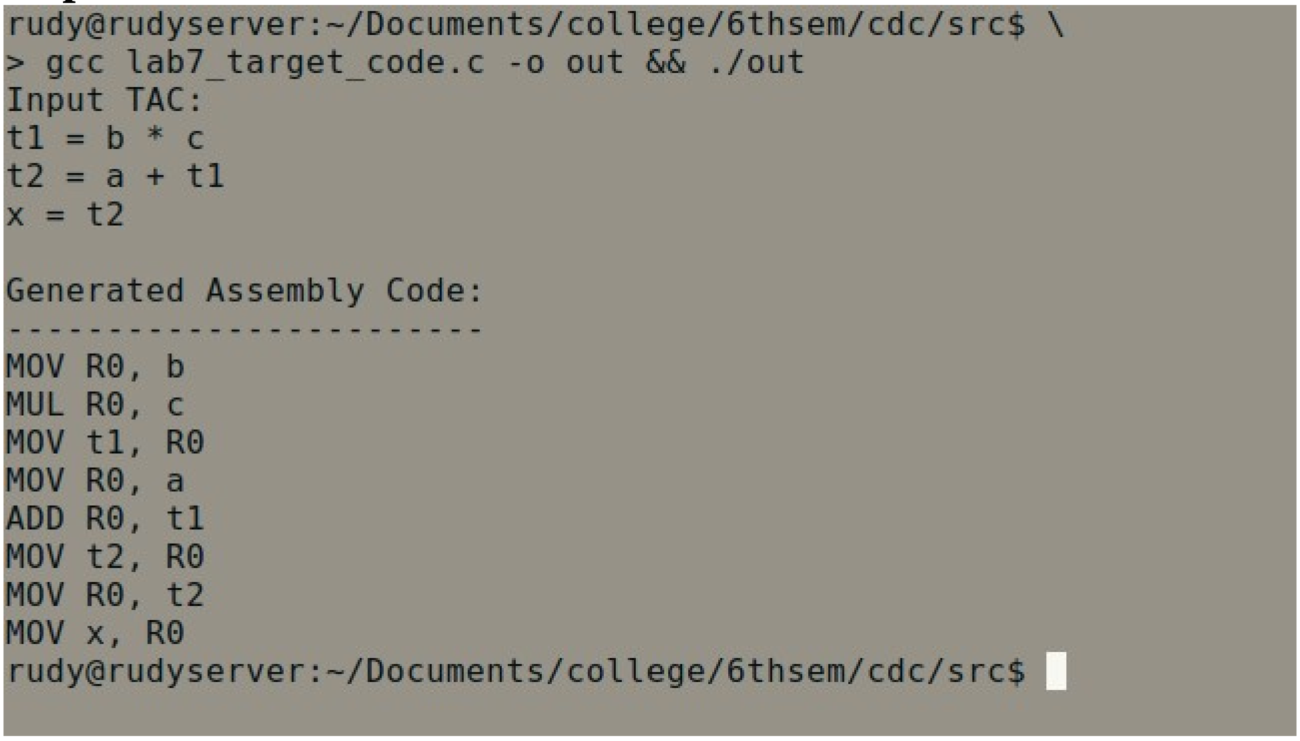
Algorithm

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

```c
#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() {


    Quadruple q[] = {
        {"t1", "b", "*", "c"},
        {"t2", "a", "+", "t1"},
        {"x", "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@rudyserver:~/Documents/college/6thsem/cdc/src$ \
> gcc lab7_target_code.c -o out && ./out
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
rudy@rudyserver:~/Documents/college/6thsem/cdc/src$ 
```

Conclusion

The program successfully generates target code for a simple register-based machine.
This experiment demonstrates how high-level arithmetic expressions are translated into low-lev
el machine instructions. Target code generation bridges the gap between intermediate represent
ation and actual execution on hardware.