Lab Report

Course Code : CSE332
Course Title : Compiler Design

Lab Experiment Number: 01-11

Submitted To:
Muhammad Abu Rayan (MAR)
Lecturer, Department of CSE
Daffodil International University

Submitted By:
Name: Gargy Roy
ID: 221-15-4783
Section: 61_O2
Department of CSE
Daffodil International University

Submission Date: 24/04/2025

Lab 1:

```
#include <stdio.h>
#include <ctype.h>
int main() {
   FILE *in = fopen("input.c", "r");
   FILE *out = fopen("output.c", "w");

   if (in == NULL || out == NULL) {
      printf("Error opening file.\n");
      return 1;
   }

   char c, prev = '\0';
   int in_single_comment = 0, in_multi_comment = 0;

   while ((c = fgetc(in)) != EOF) {
      // Handle single-line comment
      if (!in_multi_comment && !in_single_comment && c == '/' && (prev = fgetc(in)) == '/') {
         in_single_comment = 1;
         continue;
      }

      // Handle multi-line comment
      if (!in_multi_comment && !in_single_comment && c == '/' && (prev = fgetc(in)) == '*') {
         in_multi_comment = 1;
         continue;
      }

      // End of single-line comment
```

```
    if (in_single_comment && c == '\n') {
        in_single_comment = 0;
        fputc('\n', out); // Optional: keep newline after comment
        continue;
    }

    // End of multi-line comment
    if (in_multi_comment && c == '*' && (prev = fgetc(in)) == '/') {
        in_multi_comment = 0;
        continue;
    }

    // If in a comment, skip the character
    if (in_single_comment || in_multi_comment) continue;

    // Remove redundant whitespace (spaces, tabs, newlines)
    if (isspace(c)) {
        if (!isspace(prev)) fputc(' ', out); // Only write one space
        prev = c;
        continue;
    }

    fputc(c, out);
    prev = c;
}

fclose(in);
fclose(out);
```

```
    printf("Processed output written to output.c\n");
    return 0;
}
```

ii)

Program output c:

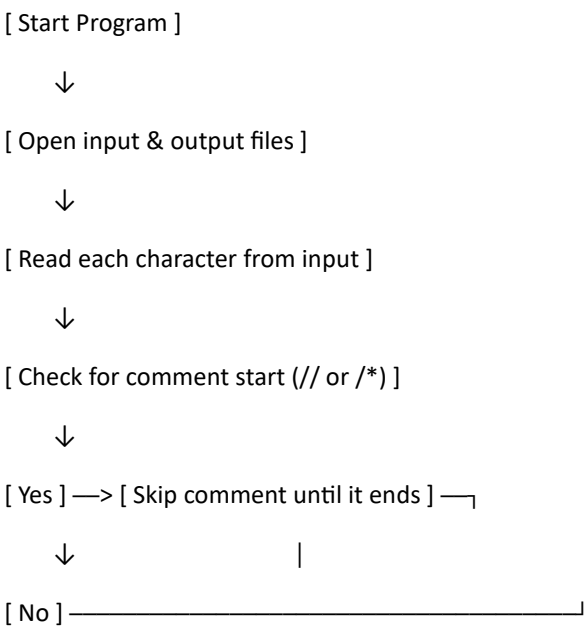
```
#include <stdio.h>
int main() {
    printf("Hello world");
    return 0;
}
```

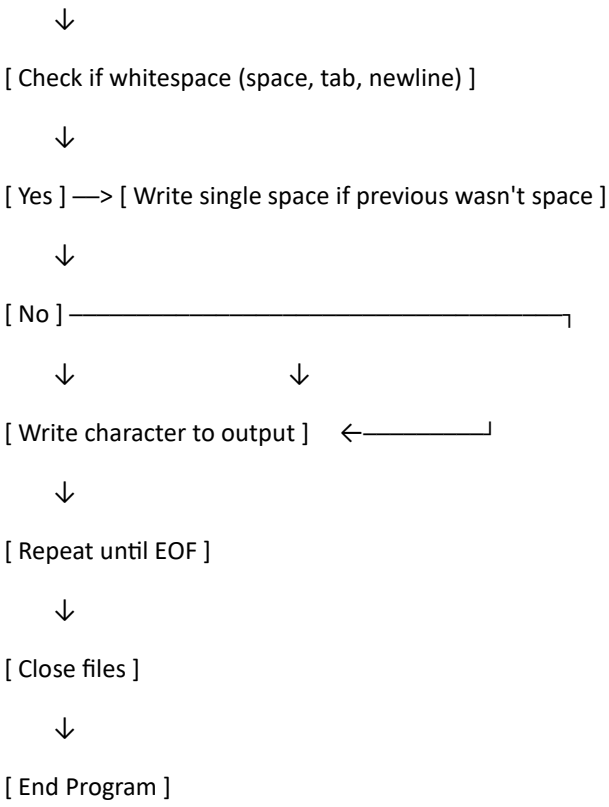
ii) Find Basic Block and Draw Flow Diagram for that C program.

Basic Block

- **B1** — #include <stdio.h>
- **B2** — int main() {
- **B3** — printf("Hello world");
- **B4** — return 0;
- **B5** — }

Flow Diagram:





Lab 2:

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

int main() {
    char line[1002];

    printf("Enter a line: ");
    fgets(line, sizeof(line), stdin);

    // Remove newline character if present
    size_t len = strlen(line);
    if (line[len - 1] == '\n') {
```

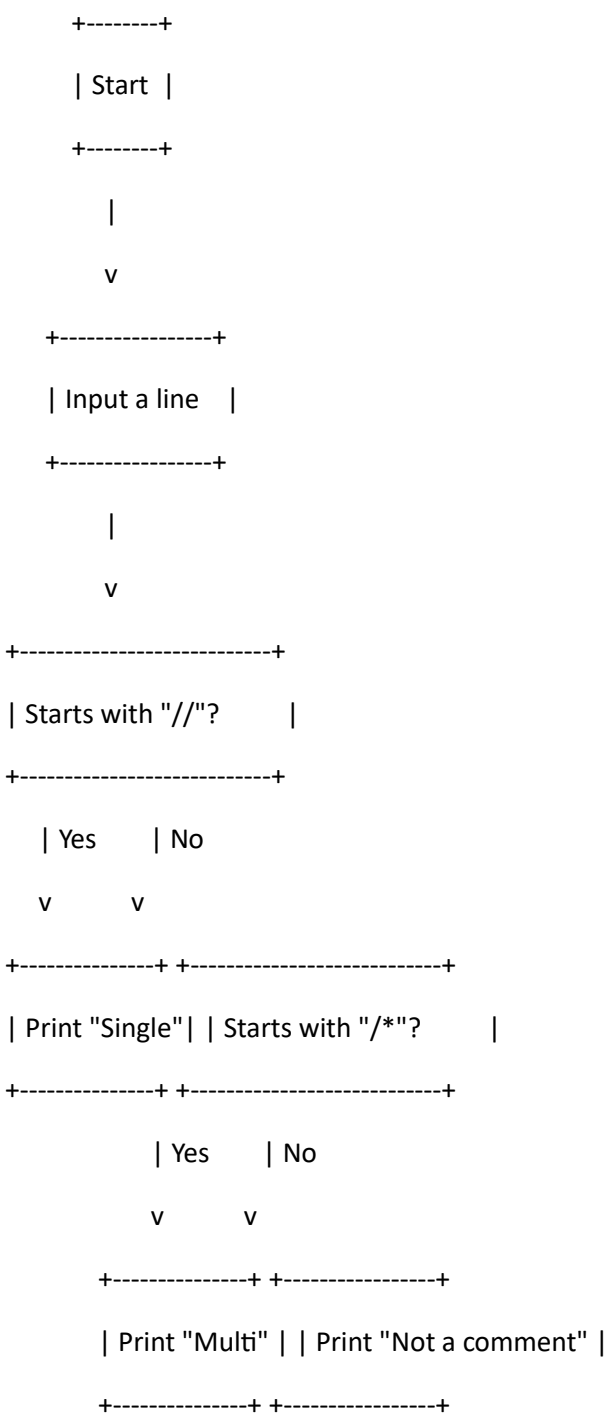
```
        line[len - 1] = '\0';
    }

    if (line[0] == '/' && line[1] == '/') {
        printf("Single-line comment\n");
    } else if (line[0] == '/' && line[1] == '*') {
        if (len >= 4 && line[len - 2] == '*' && line[len - 1] == '/') {
            printf("Multi-line comment\n");
        } else {
            printf("Incomplete multi-line comment\n");
        }
    } else {
        printf("Not a comment\n");
    }

    return 0;
}
```

ii) Find Basic Blocks and Draw Flow Diagram for that C program

Basic Blocks:

Block	Code
B1	Input a line
B2	Check if starts with //
B3	Check if starts with /*
B4	Print result

Flow Diagram (Textual Description):



Lab 3:

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

bool is_a_star(char str[]) {
    for (int i = 0; str[i]; i++) {
        if (str[i] != 'a')
            return false;
    }
    return true;
}

bool is_a_star_b_plus(char str[]) {
    int i = 0;

    // Consume all a's
    while (str[i] == 'a') i++;

    // At least one b should be present
    if (str[i] != 'b') return false;

    // Consume all b's
    while (str[i] == 'b') i++;

    // Nothing should remain
    return str[i] == '\0';
```

```
}

bool is_abb(char str[]) {
    return (strlen(str) == 3 && str[0] == 'a' && str[1] == 'b' && str[2] == 'b');
}

int main() {
    char str[100];
    printf("Enter a string: ");
    scanf("%s", str);

    if (is_abb(str)) {
        printf("The string is in the language: abb\n");
    } else if (is_a_star_b_plus(str)) {
        printf("The string is in the language: a*b+\n");
    } else if (is_a_star(str)) {
        printf("The string is in the language: a*\n");
    } else {
        printf("The string is not recognized by any pattern\n");
    }

    return 0;
}
```

Sample Outputs:

Input	Output
a	The string is in the language: a*
abb	The string is in the language: abb

Lab 4:

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

// Function to check if a word is a C keyword
int isKeyword(char *word) {
    char *keywords[] = {
        "int", "float", "return", "if", "else", "while", "for", "char", "double", "long"
    };
    int num_keywords = sizeof(keywords) / sizeof(keywords[0]);

    for (int i = 0; i < num_keywords; i++) {
        if (strcmp(word, keywords[i]) == 0)
            return 1;
    }
    return 0;
}

int main() {
    char str[1000], word[100];
    int i = 0, j = 0;

    printf("Enter a line: ");
    fgets(str, sizeof(str), stdin);

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

```
    if (isalnum(str[i]) || str[i] == '_') {
        word[j++] = str[i];
    } else {
        if (j != 0) {
            word[j] = '\0';
            if (isKeyword(word))
                printf("'%s' is a keyword\n", word);
            else
                printf("'%s' is an identifier\n", word);
            j = 0;
        }

        // Optionally print operators or symbols
        if (ispunct(str[i]) && str[i] != ' ') {
            printf("'%c' is a symbol/operator\n", str[i]);
        }
    }
    i++;
}

// Handle the last word if it exists
if (j != 0) {
    word[j] = '\0';
    if (isKeyword(word))
        printf("'%s' is a keyword\n", word);
    else
        printf("'%s' is an identifier\n", word);
```

```
    }


    return 0;
}
```

Input:

cpp

CopyEdit

int main = 5;

Output:

csharp

CopyEdit

'int' is a keyword

'main' is an identifier

'=' is a symbol/operator

'5' is an identifier

';' is a symbol/operator

Lab No: 5

Lexical Analyzer for Validating Operators

Introduction:A lexical analyzer (lexer) is the first phase of a compiler. It scans the source code and breaks it into meaningful units called tokens. One of its important functions is to recognize and validate operators.

Types of Operators to Recognize:

1. Arithmetic Operators:
   +, -, *, /
2. Relational Operators:
   ==, !=, <, >, <=, >=

3. Assignment Operators:
   =, +=, -=, *=, /=

Lexical Analysis Process for Operators:

The lexer uses regular expressions or finite automata (FA) to match operator patterns. The process involves:

---

1. Character Scanning:

- The lexer reads the source code character by character.
- When it encounters a symbol that may begin an operator (e.g., +, =, !), it starts checking for a valid operator pattern.

---

2. Pattern Matching:

- The lexer checks if the sequence of characters matches known operator patterns.
- For example:
    - = followed by another = becomes == (relational operator).
    - + followed by = becomes += (compound assignment operator).
    - ! followed by = becomes != (relational operator).

---

3. Token Generation:

- Once a valid operator is matched, it is converted into a token (e.g., T_PLUS, T_EQUAL, T_NOT_EQUAL).
- This token is passed to the next phase of compilation (syntax analysis).

Example Code (Lexical Analyzer for Operators):

```
#include <stdio.h>

#include <string.h>
```

```
int main() {
    char op[3];

    printf("Enter an operator: ");
    scanf("%s", op);

    if (strcmp(op, "+") == 0 || strcmp(op, "-") == 0 ||
        strcmp(op, "*") == 0 || strcmp(op, "/") == 0)
        printf("Arithmetic operator\n");

    else if (strcmp(op, "==") == 0 || strcmp(op, "!=") == 0 ||
             strcmp(op, "<") == 0 || strcmp(op, ">") == 0 ||
             strcmp(op, "<=") == 0 || strcmp(op, ">=") == 0)
        printf("Relational operator\n");

    else if (strcmp(op, "=") == 0 || strcmp(op, "+=") == 0 ||
             strcmp(op, "-=") == 0 || strcmp(op, "*=") == 0 ||
             strcmp(op, "/=") == 0)
        printf("Assignment operator\n");

    else
        printf("Invalid operator\n");

    return 0;
}
```

🧪 Sample Inputs & Outputs

Input	Output
+	Arithmetic operator
==	Relational operator
+=	Assignment operator
!=	Relational operator
>>	Invalid operator

Conclusion:

Lexical analyzers play a vital role in recognizing various operators in source code. They ensure that the syntax is valid by breaking input into tokens using character scanning and pattern matching techniques such as regular expressions or finite automata.

Lab 6:

What is a Symbol Table?

A **symbol table** is a **data structure** used by a compiler to store and manage information about the symbols in a program during **lexical**, **syntax**, and **semantic analysis**.

What Information Does It Store?

Each entry in the table usually contains:

- **Name** of the identifier
- **Type** (e.g., int, float, bool, char, class type, etc.)
- **Scope** (e.g., global, local, block-level)
- **Memory location** (address or offset)
- **Value** (optional, for constants or variables)
- **Line number(s)** where the symbol appears
- **Parameters** (for functions: types and number of parameters)

How to Implement a Symbol Table

You can implement a symbol table using different data structures depending on your needs:

Data Structure	Pros	Cons
Array	Simple and easy to code	Slow lookup, fixed size
Linked List	Dynamic size	Slow lookup
Binary Search Tree (BST)	Faster search than list	Slower than hash table
Hash Table	**Fast lookup & insert**	Hash collisions need handling

Hash tables are **most common** in real compilers due to their average-case constant-time performance.

Symbol Table Operations

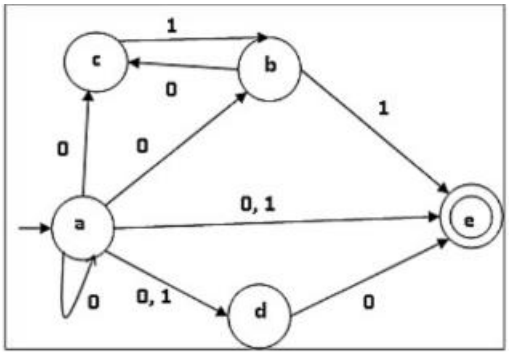
Typical operations include:

- **Insert** (add a new identifier)
- **Lookup** (search for an identifier)
- **Delete** (remove an identifier)
- **Update** (change symbol's data, e.g., after type inference)

Usage in Compiler Phases

- **Lexical Analysis**: Adds identifiers to the symbol table.
- **Syntax Analysis**: Checks scope and grammar-related rules.
- **Semantic Analysis**: Verifies types, detects redeclarations.
- **Code Generation**: Uses memory locations from symbol table.

Lab 7:



transition table:

State (q)	δ(q, 0)	δ(q, 1)
a	{a, b, c, d, e}	{d, e}
b	{c}	{e}
c	Ø	{b}
d	{e}	Ø
e	Ø	Ø

NFA State Transition Table:

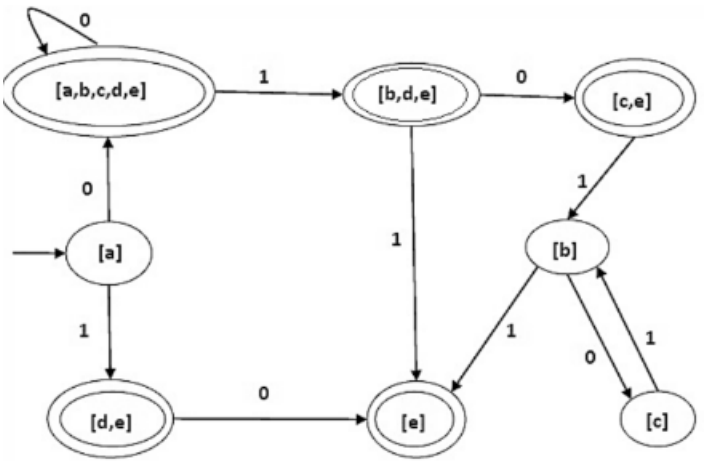
Each row shows how a state transitions on input 0 and 1.

q	δ(q,0)	δ(q,1)
a	{a,b,c,d,e}	{d,e}
b	{c}	{e}
c	∅	{b}
d	{e}	∅
e	∅	∅

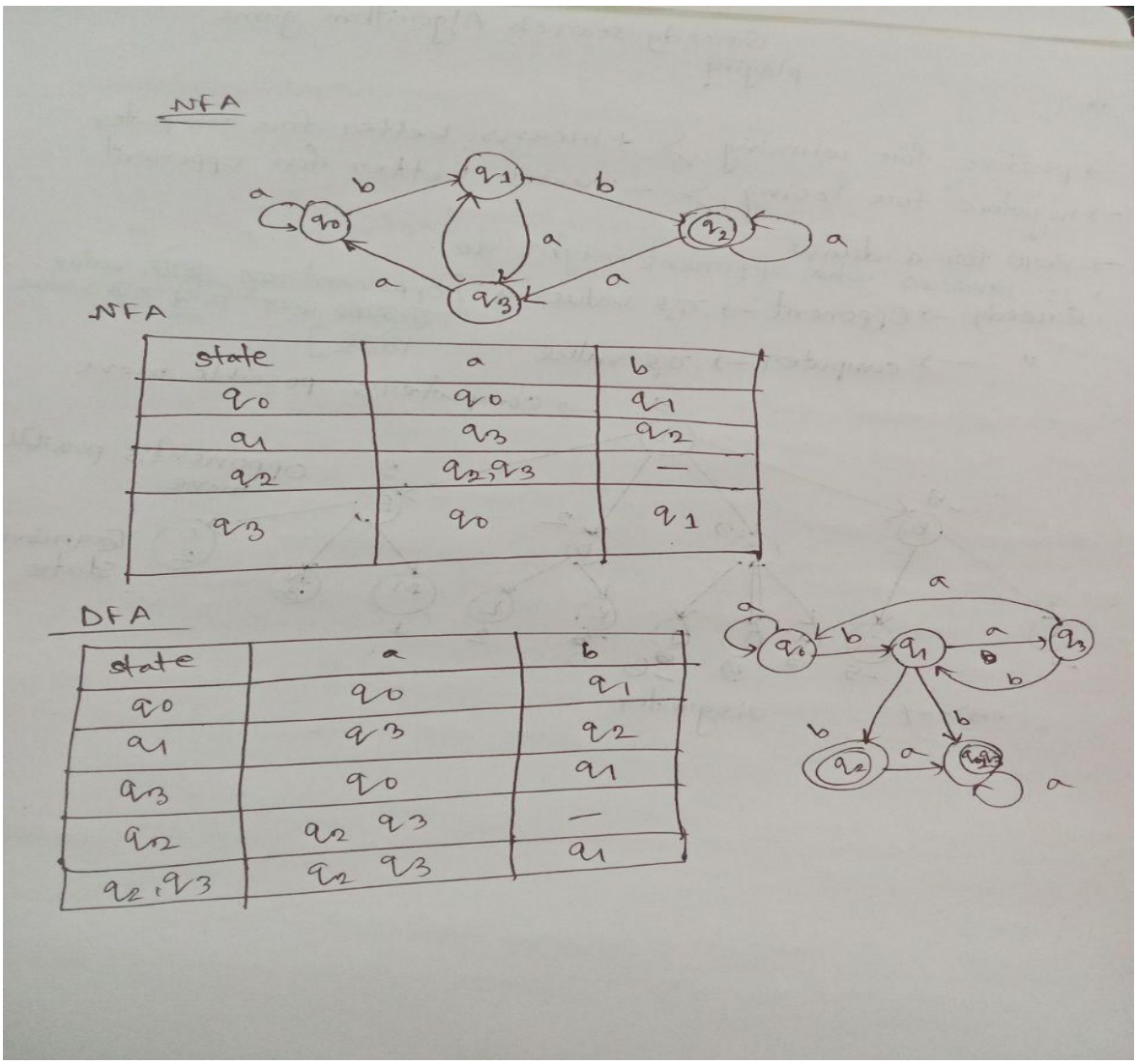
DFA State Transition Table

Each DFA state is a **set of NFA states**, created through subset construction. Transitions are deterministic now.

q	δ(q,0)	δ(q,1)
[a]	[a,b,c,d,e]	[d,e]
[a,b,c,d,e]	[a,b,c,d,e]	[b,d,e]
[d,e]	[e]	∅
[b,d,e]	[c,e]	[e]
[e]	∅	∅
[c,e]	∅	[b]
[b]	[c]	[e]
[c]	∅	[b]


0
[a,b,c,d,e]
1
[b,d,e]
0
[c,e]
0
1
[a]
1
[b]
1
1
0
1
[d,e]
0
[e]
[c]

LAB 8:


NFA
NFA
state
a
b
DFA
state
a
b

LAB 9:

A predictive parser is a top-down parser that uses a lookahead symbol to decide which production rule to apply. It does not use backtracking, making it efficient and faster for parsing certain types of grammars, particularly LL(1) grammars.

Key Functionalities of a Predictive Parser in a Mini Language:

1. Parses Without Backtracking
   It chooses the correct rule using one symbol of lookahead, eliminating the need for trial-and-error or backtracking.
2. Uses FIRST and FOLLOW Sets
   These sets help determine what terminals can appear at the beginning (FIRST) or follow (FOLLOW) a non-terminal in derivations.
3. Builds a Parse Table
   A predictive parsing table is created using the FIRST and FOLLOW sets. This table guides the parser in selecting the appropriate production rule.
4. Matches Input with Grammar Rules
   As it reads the input, it compares symbols with rules from the parse table and applies the appropriate derivation.
5. Reports Syntax Errors Clearly
   When the input does not match any expected production, the parser can detect and report the error precisely and early.

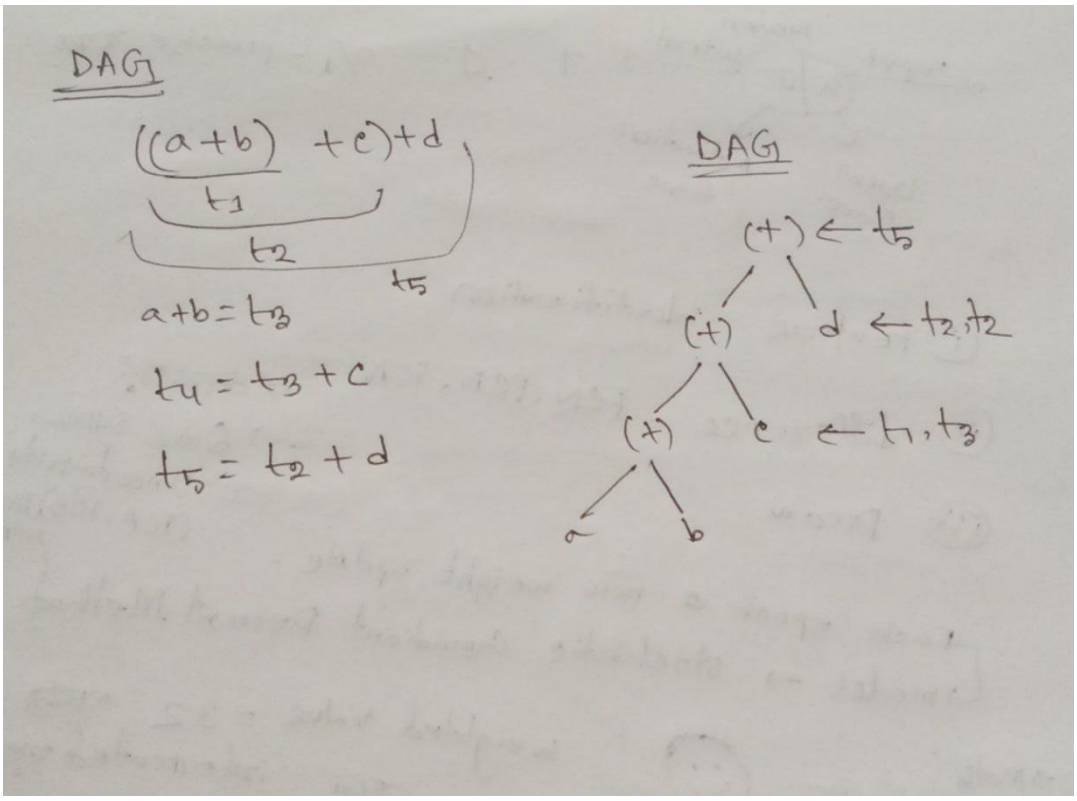
Example Grammar (Mini Language):

E → T E'

E' → + T E' | ε

T → id

LAB 10:



Lab 11:

A shift-reduce parser uses a **stack** to hold grammar symbols and an **input buffer** to process tokens. It performs **shift** (push input to stack) and **reduce** (replace handle on stack with a non-terminal) operations using predefined grammar rules.

Grammar Used:

E → E + E

E → id

C Implementation:

```
#include <stdio.h>

#include <string.h>


char stack[100], input[100];

int top = -1, i = 0;
```

```
void print() {
    printf("Stack: %s\tInput: %s\n", stack, input + i);
}

void shift() {
    stack[++top] = input[i++];
    stack[top + 1] = '\0';
    printf("Action: SHIFT\t");
    print();
}

int reduce() {
    if (top >= 2 && stack[top] == 'E' && stack[top - 1] == '+' && stack[top - 2] == 'E') {
        top -= 2;
        stack[top] = 'E';
        stack[top + 1] = '\0';
        printf("Action: REDUCE E → E + E\t");
        print();
        return 1;
    } else if (top >= 1 && stack[top - 1] == 'i' && stack[top] == 'd') {
        top -= 1;
        stack[top] = 'E';
        stack[top + 1] = '\0';
        printf("Action: REDUCE E → id\t");
        print();
        return 1;
    }
```

```
    return 0;
}

int main() {
    printf("Enter input (e.g. id+id+id): ");
    scanf("%s", input);

    printf("\nStarting Shift-Reduce Parsing:\n");
    while (i < strlen(input)) {
        shift();
        while (reduce());
    }
 while (reduce());
 if (strcmp(stack, "E") == 0)
        printf("\n Parsing Successful! Final Stack = %s\n", stack);
    else
        printf("\n Parsing Failed. Final Stack = %s\n", stack);

    return 0;
}
```