SHARDA UNIVERSITY GREATER NOIDA

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

COMPILER DESIGN LAB

CSP 353

2024-2025 (VI Semester)

Name : Rishabh Kushwaha

Class : B.Tech CSE section -C

Rollno : 2201010575

SystemID: 2022448095

Submitted To:

Mrs. Ravikant Kumar Nirala

Asst. Professor

INDEX

S.No	TOPIC	DATE	SIGNATURE
1	Write a C program to identify whether a given line is a comment or not?	10/01/25	
2	write a C program to recognize strings under "a", "a*b+", "abb".	15/01/25	
3	Lex and Yacc compiler and Implement Lexical analyzer using Lex compiler.	22/01/25	
4	Write a Program for constructing of LL(1) Parsing for any given Language	29/01/25	
5	Write a Program for constructing Recursive Descent Parsing for any given Language	19/02/25	
6	Write a C program to implement semantic rules to calculate expression that takes an expression with digits, +, and *, and computes the value.	05/03/25	
7	Write a C program to generate intermediate code (3 address code) from an expression	12/03/25	
8	Write a C program to Implement Symbol Table	19/03/25	
9	Write a C program to Implement Directed Acyclic Graph (DAG)	26/03/25	

Experiment 1: Write a C program to identify whether a given line is a comment or not?

```
#include <stdio.h>
 #include <string.h>
 int main() {
 char line[256];
 printf("Enter a line: ");
 if (fgets(line, sizeof(line), stdin) != NULL) {
 // Check if the line starts with // or /*
 if (line[0] == '/' && line[1] == '/') {
 printf("The line is a single-line comment.\n");
 } else if (line[0] == '/' && line[1] == '*') {
 printf("The line is a multi-line comment.\n");
 } else {
 printf("The line is NOT a comment.\n");
 }
 }
 return 0;
 }
```

Output:

Enter a line: // Hi how are you?
The line is a single-line comment.
Enter a line: /*Hi how are you?*/
The line is a multi-line comment.
Enter a line: Hi how are you?
The line is NOT a comment

Experiment 2: write a C program to recognize strings under "a", "a*b+", "abb".

```
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
// Function to check if a string matches any of the three patterns
const char* recognizePattern(const char *str) {
int i = 0;
// Check for "a"
if (strcmp(str, "a") == 0) {
return "a";
}
// Check for "abb"
if (strcmp(str, "abb") == 0) {
return "abb";
}
// Check for "a*b+"
while (str[i] == 'a') i++; // Skip leading 'a's
if (str[i] == 'b') { // At least one 'b' is required
while (str[i] == 'b') i++; // Skip remaining 'b's
if (str[i] == '\0') { // Ensure no other characters remain
return "a*b+";
}
}
// No pattern matches
return NULL;
}
int main() {
char input[100];
printf("Enter a string: ");
scanf("%s", input);
const char *pattern = recognizePattern(input);
if (pattern) {
printf("The string matches the pattern \"%s\".\n", pattern);
} else {
printf("The string does not match any pattern.\n");
}
return 0;
}
```

Sample I/O

Enter a string: a
The string matches the pattern "a".
Enter a string: b
The string matches the pattern "a*b+".
Enter a string: ab
The string matches the pattern "a*b+".
Enter a string: abb
The string matches the pattern "abb".

Experiment 3:

PartA: Lex and Yacc compiler installation

Installer Required in same directory (Program Files (x86))

1. Flex (https://gnuwin32.sourceforge.net/packages/flex.htm)
2. Bison (https://gnuwin32.sourceforge.net/packages/bison.htm)
3. Dev++ (https://www.bloodshed.net/)

Set Path

C:\Program Files (x86)\GnuWin32\bin;C:\Program Files (x86)\Dev-Cpp\MinGW64\bin

Command to Run

Step 1: flex file.l

Step 2: gcc lex.yy.c

Step 3: .\a.exe

PartB: Implement Lexical analyzer using Lex compiler.

```
%{
#include <stdio.h>
%}
%%
[a-zA-Z_][a-zA-Z0-9_]* { printf("IDENTIFIER: %s\n", yytext); }
[0-9]+ { printf("NUMBER: %s\n", yytext); }
[+\-*/=] { printf("OPERATOR: %s\n", yytext); }
[ \t\n] { /* Ignore whitespace */ }
. { printf("UNKNOWN TOKEN: %s\n", yytext); }
%%
int main() {
printf("Enter input (Ctrl+D to end):\n");
yylex();
return 0;
}
int yywrap() {
return 1; }
```

OUTPUT

```
Enter input (Ctrl+D to end):
a = 5+b
IDENTIFIER: a
OPERATOR: =
NUMBER: 5
OPERATOR: +
IDENTIFIER: b
```

Experiment 4: Write a Program for constructing of LL(1) Parsing for any given Language

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
// Function prototypes
void E();
void X();
void T();
// Global variables for input string and current position
const char *input;
int pos = 0;
// Function to match a terminal symbol
void match(char expected) {
if (input[pos] == expected) {
pos++;
} else {
printf("Error: Expected '%c' but found '%c'\n", expected, input[pos]);
exit(1);
}
}
// Rule for E -> TX
void E() {
printf("Applying E -> TX\n");
T();
X();
}
// Rule for T -> i
void T() {
if (input[pos] == 'i') {
printf("Applying T -> i\n");
match('i');
} else {
printf("Error: Expected 'i' but found '%c'\n", input[pos]);
exit(1);
}
}
// Rule for X -> +TX | ε
void X() {
```

```
if (input[pos] == '+') {
printf("Applying X -> +TX\n");
match('+');
T();
X();
} else {
printf("Applying X -> ε\n");
}
}
// Main function to drive the parser
int main() {
char buffer[100];
printf("Enter input string (end with $): ");
scanf("%s", buffer);
input = buffer;
// Start parsing
E();
return 0;
}
// Check if the input is fully consumed and ends with $
if (input[pos] == '$') {
printf("Input string successfully parsed!\n");
} else {
printf("Error: Input string not fully consumed!\n");
}
```

OUTPUT

```
Enter input string (end with $): i+i$
Applying E -> TX
Applying T -> i
Applying X -> +TX
Applying T -> i
Applying X -> ε
Input string successfully parsed!
```

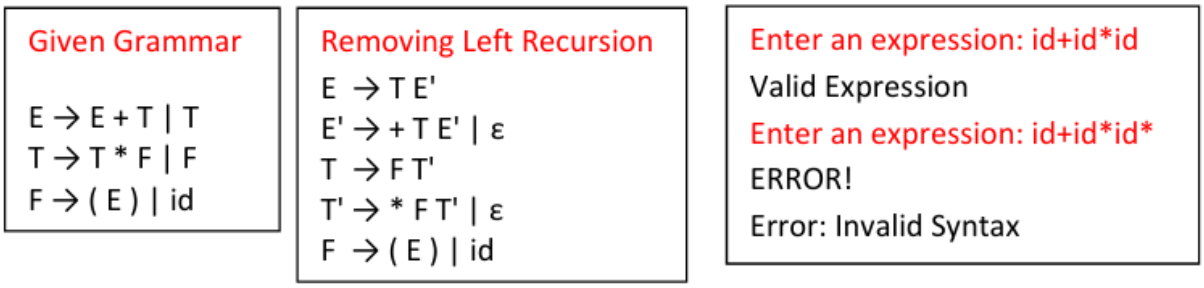
Experiment 5: Write a Program for constructing Recursive Descent Parsing for any given Language

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_LEN 100
char input[MAX_LEN];
int pos = 0;
// Function prototypes
void E();
void EPrime();
void T();
void TPrime();
void F();

void error() {
printf("Error: Invalid Syntax\n");
exit(1);
}

// Match function to check expected character
void match(char expected) {
if (input[pos] == expected) {
pos++;
} else {
error();
}
}

// E -> T E'
void E() {
T();
EPrime();
}

// E' -> + T E' | ε
void EPrime() {
if (input[pos] == '+') {
match('+');
```

```
T();
EPrime();
}
// ε (do nothing)
}

// T -> F T'
void T() {
F();
TPrime();
}

// T' -> * F T' | ε
void TPrime() {
if (input[pos] == '*') {
match('*');
F();
TPrime();
}
// ε (do nothing)
}

// F -> ( E ) | id
void F() {
if (input[pos] == '(') {
match('(');
E();
match(')');
} else if (input[pos] == 'i' && input[pos+1] == 'd') {
match('i');
match('d');
} else {
error();
}
}

int main() {
printf("Enter an expression: ");
scanf("%s", input);
E();  // Start parsing from E
```

```
// If we have consumed the whole input, it's valid
if (input[pos] == '\0') {
printf("Valid Expression\n");
} else {
printf("Error: Unexpected characters\n");
}
return 0;

}
```

OUTPUT



Experiment 6: Write a C program to implement semantic rules to calculate expression that takes an expression with digits, +, and *, and computes the value.

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

char *input; // Pointer to input expression

// Function to read the next character and advance the input pointer
char get_next() {
    return *input++;
}

// Function to parse a number
int parse_number() {
    int num = 0;
    while (isdigit(*input)) {
        num = num * 10 + (*input - '0');
        input++;
    }
    return num;
}

// Forward declaration of parsing functions
int parse_expr();
int parse_term();
int parse_factor();

// Parse expressions: Handles addition
int parse_expr() {
    int result = parse_term();
    char op;
    while ((op = *input) == '+' || op == '-') {
        get_next(); // Consume the operator
        int value = parse_term();

        if (op == '+')
            result += value;
```

```
 else
            result -= value;
    }
    return result;
}

// Parse terms: Handles multiplication
int parse_term() {
    int result = parse_factor();
    char op;
    while ((op = *input) == '*') {
        get_next(); // Consume the operator
        result *= parse_factor();
    }
    return result;
}

// Parse factors: Handles numbers
int parse_factor() {
    if (isdigit(*input)) {
        return parse_number();
    } else {
        printf("Error: Invalid character '%c'\n", *input);
        exit(1);
    }
}

int main() {
    char expr[100];
    printf("Enter an expression: ");
    fgets(expr, 100, stdin);
    input = expr;
    int result = parse_expr();
    printf("Result: %d\n", result);
    return 0;
}
```

Output:

```
Enter an expression: 2+3*4
Result: 14
```

Experiment 7: Write a C program to generate intermediate code (3 address code) from an expression

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#define MAX 100
char stack[MAX];
int top = -1;
int tempCount = 1;
void push(char c) {
    if (top == MAX - 1) {
        printf("Stack overflow\n");
        return;
    }
    stack[++top] = c;
}
char pop() {
    if (top == -1) {
        printf("Stack underflow\n");
        return -1;
    }
    return stack[top--];
}
int precedence(char op) {
    if (op == '+' || op == '-') return 1;
    if (op == '*' || op == '/') return 2;
    return 0;
}
void infixToPostfix(char *infix, char *postfix) {
    int i, j = 0;
    for (i = 0; infix[i] != '\0'; i++) {
        if (isalnum(infix[i])) {
            postfix[j++] = infix[i];
        } else if (infix[i] == '(') {
            push(infix[i]);
        } else if (infix[i] == ')') {
            while (top != -1 && stack[top] != '(') {
                postfix[j++] = pop();
```

```
            }
        pop(); // Remove '('
        } else {
            while (top != -1 && precedence(stack[top]) >= precedence(infix[i])) {
                postfix[j++] = pop();
            }
            push(infix[i]);
        }
    }
    while (top != -1) {
        postfix[j++] = pop();
    }
    postfix[j] = '\0';
}
void generateTAC(char *postfix) {
    char operands[MAX][MAX];
    int opTop = -1;
    char temp[5];
    for (int i = 0; postfix[i] != '\0'; i++) {
        if (isalnum(postfix[i])) {
            char operand[2] = {postfix[i], '\0'};
            strcpy(operands[++opTop], operand);
        } else {
            char op2[MAX], op1[MAX];
            strcpy(op2, operands[opTop--]);
            strcpy(op1, operands[opTop--]);
            sprintf(temp, "t%d", tempCount++);
            printf("%s = %s %c %s\n", temp, op1, postfix[i], op2);
            strcpy(operands[++opTop], temp);
        }
    }
}

int main() {
    char infix[MAX], postfix[MAX];
    printf("Enter an infix expression: ");
    scanf("%s", infix);
    infixToPostfix(infix, postfix);
    printf("Postfix Expression: %s\n", postfix);
    printf("Three Address Code:\n");
```

```
    generateTAC(postfix);
    return 0;
}
```

OUTPUT

```
I/P:  Enter an infix expression: a+b*c
Postfix Expression: abc*+
Three Address Code:
t1 = b * c
t2 = a + t1
```

Experiment 9: Write a C program to Implement Directed Acyclic Graph (DAG)

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define MAX 100
int adj[MAX][MAX];    // Adjacency matrix
bool visited[MAX];    // Visited nodes
bool recStack[MAX];   // Stack to detect cycles
int n;                // Number of nodes
// Function to perform DFS and remove cycles
bool dfs(int u) {
  visited[u] = true;
  recStack[u] = true;
  for (int v = 0; v < n; v++) {
    if (adj[u][v]) {
      if (!visited[v]) {
        if (dfs(v))
          return true;
      } else if (recStack[v]) {
        // Cycle detected: remove the edge u -> v
        printf("Removing edge %d -> %d to break cycle.\n", u, v);
        adj[u][v] = 0;
      }  }  }
  recStack[u] = false;
  return false;
}
// Convert graph to DAG
void convertToDAG() {
  for (int i = 0; i < n; i++) {
    if (!visited[i])
      dfs(i);
```

```
    }
}
// Display the DAG
void printDAG() {
    printf("\n--- DAG (Adjacency Matrix) ---\n");
     for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            printf("%d ", adj[i][j]);
        }
        printf("\n");    }   }
int main() {
    int edges, u, v;
    printf("Enter number of nodes: ");
    scanf("%d", &n);
    printf("Enter number of edges: ");
    scanf("%d", &edges);
    // Initialize adjacency matrix
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            adj[i][j] = 0;
    printf("Enter edges (from to):\n");
    for (int i = 0; i < edges; i++) {
        scanf("%d %d", &u, &v);
        adj[u][v] = 1;
    }
    convertToDAG();
    printDAG();
    return 0;
}
```

```
Enter number of nodes: 4
Enter number of edges: 5
Enter edges (from to):
0 1
1 2
2 0
2 3
3 1
Removing edge 2 -> 0 to break cycle.
Removing edge 3 -> 1 to break cycle.

--- DAG (Adjacency Matrix) ---
0 1 0 0
0 0 1 0
0 0 0 1
0 0 0 0
```