Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)
CHENNAI

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

BCSE307P – Compiler Design Laboratory

Register Number : 22BCE1462
Name of the Student : Guha Pranav Yelchuru

TABLE OF CONTENTS

BCSE307P Compiler Design Lab
ASSESSMENT
22BCE1462 Guha Pranav Yelchuru

Experiment 1

Aim: To implement a Deterministic Finite Automaton (DFA) from a given regular grammar using C, and to display both the transition table and the sequence of transitions to determine whether a given input string is accepted by the DFA.

Algorithm:

Input DFA Components:

- Set of states Q
- Input alphabets T
- Set of final states F
- Transition function in table form

Build Transition Table:

- For each state and input symbol, store the corresponding transition.

Display Transition Table:

- Print the table with states as rows and input symbols as columns.

Input Terminal String w.

DFA Simulation:

- Start from the initial state (first in the list).
- For each character in w, check if the symbol is in T.
- Use the transition table to move to the next state.
- If, after processing the string, the DFA is in a final state, accept it; otherwise, reject.

Source Code

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

int getIndex(char alphabets[], int n, char c){
    for (int i = 0; i < n; i++) {
        if (alphabets[i] == c)
            return i;
    }
    return -1;
}

int getStateIndex(char states[], int m, char c){
    for (int i = 0; i < m; i++) {
        if (states[i] == c)
            return i;
    }
    return -1;
}

bool isFinal(char final[], int f, char c){
    for (int i = 0; i < f; i++) {
        if (final[i] == c)
            return true;
    }
    return false;
}

int main() {
    int m,n,f;
    printf("Enter number of states: ");
    scanf("%d", &m);
    char states[m];
    for(int i=0; i<m; i++){
        printf("Enter state %d: ", (i+1));
        scanf(" %c", &states[i]);
    }

    printf("Enter number of alphabets: ");
    scanf("%d", &n);
    char alphabets[n];
    for(int i=0; i<n; i++){
        printf("Enter alphabet %d: ", (i+1));
        scanf(" %c", &alphabets[i]);
    }

    printf("Enter number of final states: ");
    scanf("%d", &f);
    char final[f];
    for(int i=0; i<f; i++){
        printf("Enter final state %d: ", (i+1));
        scanf(" %c", &final[i]);
```

```
}

char transition[m][n];

for(int i=0; i<m; i++){
    for(int j=0; j<n; j++){
        printf("Enter transition from %c on %c: ", states[i], alphabets[j]);
        scanf(" %c", &transition[i][j]);
    }
}

printf("\nTransition Table:\n");
printf("State\t");
for(int j=0; j<n; j++){
    printf("%c\t", alphabets[j]);
}
printf("\n");

for(int i=0; i<m; i++){
    printf("%c\t", states[i]);
    for(int j=0; j<n; j++){
        printf("%c\t", transition[i][j]);
    }
    printf("\n");
}

char inp[30];
printf("\nEnter string: ");
scanf("%s", inp);

int currentState = 0;
printf("\nTransitions:\n");
printf("Start at state %c\n", states[currentState]);

for (int i = 0; i<strlen(inp); i++) {
    int index = getIndex(alphabets, n, inp[i]);
    if (index == -1) {
        printf("Error: Input symbol %c not in alphabet\n", inp[i]);
        return 1;
    }

    int nextState = getStateIndex(states, m, transition[currentState][index]);
    if (nextState == -1) {
        printf("No transition from state %c on input %c. String rejected.\n", states[currentState], inp[i]);
        return 0;
    }

    printf("On input %c, move from state %c to state %c\n", inp[i], states[currentState], states[nextState]);
    currentState = nextState;
}

if (isFinal(final, f, states[currentState])) {
    printf("String accepted. Reached final state %c.\n", states[currentState]);
} else {
    printf("String rejected. Ended at state %c.\n", states[currentState]);
}
```

```
    return 0;
}
```

OUTPUT:

```
Enter number of states: 4
Enter state 1: A
Enter state 2: B
Enter state 3: C
Enter state 4: D
Enter number of alphabets: 2
Enter alphabet 1: a
Enter alphabet 2: b
Enter number of final states: 1
Enter final state 1: C
Enter transition from A on a: B
Enter transition from A on b: D
Enter transition from B on a: B
Enter transition from B on b: C
Enter transition from C on a: D
Enter transition from C on b: C
Enter transition from D on a: D
Enter transition from D on b: D

Transition Table:
State   a   b
A   B   D
B   B   C
C   D   C
D   D   D
Enter string: aaabbb

Transitions:
Start at state A
On input a, move from state A to state B
On input a, move from state B to state B
On input a, move from state B to state B
On input b, move from state B to state C
On input b, move from state C to state C
On input b, move from state C to state C
String accepted. Reached final state C.
```

Experiment 2

Aim: To implement a C program to derive the regular grammar from a given Deterministic Finite Automaton (DFA).

Algorithm:

Input DFA Components:

- States Q (Non-terminals)
- Input alphabet T (Terminals)
- Final states F
- Transition function δ as a table

Define Grammar G = (N, T, P, S):

- **N** = set of states (non-terminals)
- **T** = input alphabet (terminals)
- **S** = start symbol (initial state)
- **P** = productions:

For each transition δ(q, a) = p, add q → a p to P

If p is a final state, also add q → a to P

If q itself is final, optionally add q → ε

Display Grammar

Source Code:

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <stdbool.h>
```

```
bool isFinal(char final[], int f, char c) {
    for (int i = 0; i < f; i++) {
        if (final[i] == c)
            return true;
    }
    return false;
}
int getIndex(char arr[], int len, char ch) {
    for (int i = 0; i < len; i++) {
        if (arr[i] == ch)
            return i;
    }
    return -1;
}
int main() {
    int m, n, f;
    printf("Enter number of states: ");
    scanf("%d", &m);
    char states[m];
    for (int i = 0; i < m; i++) {
        printf("Enter state %d: ", i + 1);
        scanf(" %c", &states[i]);
    }

    printf("Enter number of alphabets: ");
```

```
scanf("%d", &n);
char alphabets[n];
for (int i = 0; i < n; i++) {
    printf("Enter alphabet %d: ", i + 1);
    scanf(" %c", &alphabets[i]);
}
printf("Enter number of final states: ");
scanf("%d", &f);
char final[f];
for (int i = 0; i < f; i++) {
    printf("Enter final state %d: ", i + 1);
    scanf(" %c", &final[i]);
}
char transition[m][n];
for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
        printf("Enter transition from %c on %c: ", states[i], alphabets[j]);
        scanf(" %c", &transition[i][j]);
    }
}

printf("\nGrammar G = (N, T, P, S)\n");
printf("N = { ");
for (int i = 0; i < m; i++) printf("%c ", states[i]);
printf("}\n");
```

```
printf("T = { ");
for (int i = 0; i < n; i++) printf("%c ", alphabets[i]);
printf("}\n");

printf("S = %c\n", states[0]);

printf("P = {\n");

for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
        char from = states[i];
        char symbol = alphabets[j];
        char to = transition[i][j];

        printf("  %c → %c%c\n", from, symbol, to);

        if (isFinal(final, f, to)) {
            printf("  %c → %c\n", from, symbol);
        }
    }
    if (isFinal(final, f, states[i])) {
        printf("  %c → ε\n", states[i]);
    }
}
```

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

    return 0;

}
```

Output

```
Enter number of states: 4
Enter state 1: A
Enter state 2: B
Enter state 3: C
Enter state 4: D
Enter number of alphabets: 2
Enter alphabet 1: a
Enter alphabet 2: b
Enter number of final states: 1
Enter final state 1: C
Enter transition from A on a: B
Enter transition from A on b: D
Enter transition from B on a: B
Enter transition from B on b: C
Enter transition from C on a: D
Enter transition from C on b: C
Enter transition from D on a: D
Enter transition from D on b: D
```

```
Grammar G = (N, T, P, S)
N = { A B C D }
T = { a b }
S = A
P = {
  A → aB
  A → bD
  B → aB
  B → bC
  B → b
  C → aD
  C → bC
  C → b
  C → ε
  D → aD
  D → bD
}
```

Experiment 3

Aim: To implement a Deterministic Finite Automaton (DFA) from a given Non-deterministic Finite Automaton (NFA) without ε-transitions, using C language.

Algorithm:

Input NFA: States, alphabets, final states, and transition table (can have multiple transitions per symbol).

Use Subset Construction to create DFA:

- Each DFA state represents a set of NFA states.
- Start from the ε-closure of the NFA's start state.
- For each DFA state and symbol, calculate the union of NFA transitions from all states in that set.

Track visited DFA states, store transitions.

Identify DFA final states (any DFA state that includes an NFA final state).

Print DFA transition table and states.

Source Code:

```
#include <stdio.h>

#include <string.h>

#include <stdbool.h>

#include <stdlib.h>



#define MAX 100



int nfaStates, alphabets;

char states[MAX], alphabet[MAX];

char nfa[MAX][MAX][MAX];

int dfaTrans[MAX][MAX];
```

```
char dfaStates[MAX][MAX];

bool visited[MAX];

int dfaCount = 0;

char dfaStateNames[MAX];


bool isPresent(char dfaStates[][MAX], int count, char *state) {

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

        if (strcmp(dfaStates[i], state) == 0)

            return true;

    }

    return false;

}


int getStateIndex(char dfaStates[][MAX], int count, char *state) {

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

        if (strcmp(dfaStates[i], state) == 0)

            return i;

    }

    return -1;

}


void sortString(char *str) {

    for (int i = 0; i < strlen(str) - 1; i++) {

        for (int j = i + 1; j < strlen(str); j++) {

            if (str[i] > str[j]) {
```

```
            char tmp = str[i];
            str[i] = str[j];
            str[j] = tmp;
        }
    }
  }
}

void constructDFA() {
  dfaCount = 1;
  dfaStates[0][0] = states[0];
  dfaStates[0][1] = '\0';

  int i = 0;
  while (i < dfaCount) {
    visited[i] = true;
    for (int a = 0; a < alphabets; a++) {
        char newState[MAX] = "";
        for (int s = 0; s < strlen(dfaStates[i]); s++) {
            char from = dfaStates[i][s];
            char *ptr = strchr(states, from);
            if (!ptr) continue;
            int fromIndex = ptr - states;
            strcat(newState, nfa[fromIndex][a]);
        }
```

```
        sortString(newState);

        if (strlen(newState) == 0) {
            dfaTrans[i][a] = -1;
        } else {
            if (!isPresent(dfaStates, dfaCount, newState)) {
                strcpy(dfaStates[dfaCount++], newState);
            }
            dfaTrans[i][a] = getStateIndex(dfaStates, dfaCount, newState);
        }
    }
    i++;
}

bool needDeadState = false;
for (int i = 0; i < dfaCount; i++) {
    for (int a = 0; a < alphabets; a++) {
        if (dfaTrans[i][a] == -1) {
            needDeadState = true;
        }
    }
}

if (needDeadState) {
    strcpy(dfaStates[dfaCount], "#");
```

```
        int deadIndex = dfaCount++;
        for (int a = 0; a < alphabets; a++) {
            dfaTrans[deadIndex][a] = deadIndex;
        }

        for (int i = 0; i < dfaCount; i++) {
            for (int a = 0; a < alphabets; a++) {
                if (dfaTrans[i][a] == -1) {
                    dfaTrans[i][a] = deadIndex;
                }
            }
        }
    }
}

void assignStateNames() {
    char name = 'P';
    for (int i = 0; i < dfaCount; i++) {
        dfaStateNames[i] = name++;
    }
}

bool isFinalDFAState(char *dfaState, char *finalStates, int fCount) {
    for (int i = 0; i < fCount; i++) {
        if (strchr(dfaState, finalStates[i])) {
```

```
            return true;
        }
    }
    return false;
}

int main() {
    int f;
    char finalStates[MAX];

    printf("Enter number of NFA states: ");
    scanf("%d", &nfaStates);
    for (int i = 0; i < nfaStates; i++) {
        printf("Enter state %d: ", i + 1);
        scanf(" %c", &states[i]);
    }

    printf("Enter number of input alphabets: ");
    scanf("%d", &alphabets);
    for (int i = 0; i < alphabets; i++) {
        printf("Enter alphabet %d: ", i + 1);
        scanf(" %c", &alphabet[i]);
    }

    printf("Enter number of final states in NFA: ");
```

```
scanf("%d", &f);
for (int i = 0; i < f; i++) {
    printf("Enter final state %d: ", i + 1);
    scanf(" %c", &finalStates[i]);
}

for (int i = 0; i < nfaStates; i++) {
    for (int j = 0; j < alphabets; j++) {
        printf("Enter transitions from %c on %c (e.g., AB or -): ", states[i], alphabet[j]);
        scanf("%s", nfa[i][j]);
        if (strcmp(nfa[i][j], "-") == 0)
            strcpy(nfa[i][j], "");
    }
}

constructDFA();
assignStateNames();

printf("\nIntermediate DFA States:\n");
for (int i = 0; i < dfaCount; i++) {
    printf("%c: {%s}\n", dfaStateNames[i], dfaStates[i]);
}

printf("\n Final DFA Table \n");
printf("States: ");
```

```
for (int i = 0; i < dfaCount; i++) {
    printf("%c ", dfaStateNames[i]);
}

printf("\nAlphabets: ");
for (int i = 0; i < alphabets; i++) {
    printf("%c ", alphabet[i]);
}

printf("\nFinal States: ");
for (int i = 0; i < dfaCount; i++) {
    if (isFinalDFAState(dfaStates[i], finalStates, f)) {
        printf("%c ", dfaStateNames[i]);
    }
}

printf("\n\nDFA Transition Table:\n");
printf("State\t");
for (int i = 0; i < alphabets; i++) {
    printf("%c\t", alphabet[i]);
}
printf("\n");

for (int i = 0; i < dfaCount; i++) {
    printf("%c\t", dfaStateNames[i]);
```

```
        for (int j = 0; j < alphabets; j++) {
            printf("%c\t", dfaStateNames[dfaTrans[i][j]]);
        }
        printf("\n");
    }

    return 0;
}
```

Output:

```
Enter number of NFA states: 3
Enter state 1: A
Enter state 2: B
Enter state 3: C
Enter number of input alphabets: 2
Enter alphabet 1: a
Enter alphabet 2: b
Enter number of final states in NFA: 1
Enter final state 1: C
Enter transitions from A on a (e.g., AB or -): AB
Enter transitions from A on b (e.g., AB or -): -
Enter transitions from B on a (e.g., AB or -): -
Enter transitions from B on b (e.g., AB or -): BC
Enter transitions from C on a (e.g., AB or -): -
Enter transitions from C on b (e.g., AB or -): -
```

```
Intermediate DFA States:
P: {A}
Q: {AB}
R: {BC}
S: {#}

 Final DFA Table
States: P Q R S
Alphabets: a b
Final States: R

DFA Transition Table:
State   a   b
P   Q   S
Q   Q   R
R   S   R
S   S   S
```

Experiment 4

Aim:
a) Implement a DFA in LEX code which accepts Odd number of a's and even number of b's.

Algorithm:
Initialize counters:

- a_count = 0
- b_count = 0

Start reading input characters one by one using yylex().

For each character:

- If character is 'a', increment a_count.
- If character is 'b', increment b_count.
- If character is newline \n, do the following:
    - **Check condition**:
        - If a_count % 2 == 1 and b_count % 2 == 0 → print **"Accepted"**.
        - Else → print **"Rejected"**.
    - **Reset both counts to 0** to process the next line.

Any other character (like space, digit, etc.) is ignored.

The process continues until the end of input.

Source Code:

```
%{

#include <stdio.h>

int a_count = 0;

int b_count = 0;

%}



%%

a   { a_count++; }
```

```
b   { b_count++; }

\n  {

        if (a_count % 2 == 1 && b_count % 2 == 0)

            printf("Accepted: Odd number of a's and Even number of b's\n");

        else

            printf("Rejected\n");



        a_count = 0;

        b_count = 0;

    }

.   ;

%%

int main() {

    printf("Enter strings (Ctrl+D to end):\n");

    yylex();

    return 0;

}
```

Output:

```
Asus@LAPTOP-N4CPSGA2 /cygdrive/c/Users/Asus/Desktop/GuhaPranavYelchuru/VITC/sem7
/CompilerLab
$ lex dfa_odd_even.l

Asus@LAPTOP-N4CPSGA2 /cygdrive/c/Users/Asus/Desktop/GuhaPranavYelchuru/VITC/sem7
/CompilerLab
$ gcc lex.yy.c /usr/lib/libfl.a -o dfa1

Asus@LAPTOP-N4CPSGA2 /cygdrive/c/Users/Asus/Desktop/GuhaPranavYelchuru/VITC/sem7
/CompilerLab
$ ./dfa1
Enter strings (Ctrl+D to end):
aabb
Rejected
aab
Rejected
abb
Accepted: Odd number of a's and Even number of b's
```

Aim:

b) Implement a DFA in LEX code which accepts strings over {a, b, c} having bca as substring.

Algorithm:

Display a prompt to enter strings from the alphabet {a, b, c}.
Read one line at a time from standard input until EOF (Ctrl+D) is reached.

Pattern Matching:

- Check if the string contains 'bca' as a substring using the pattern: .*bca.*
- Print result
- If not matched in, check if the string contains only characters a, b, c, or newline, using the pattern: [a-c\n]+ to say rejected
- If the line contains characters outside {a, b, c}, use '.', to ignore other characters.

Once user inputs EOF, end the program.

Source Code:

```
%{

#include <stdio.h>

%}



%%

.*bca.*    { printf("Accepted: Contains 'bca' as a substring\n"); }

[a-c\n]+   { printf("Rejected: Does not contain 'bca'\n"); }

.        ;

%%

int main() {

   printf("Enter strings over {a, b, c} (Ctrl+D to end):\n");

   yylex();

   return 0;

}
```

Output:

```
Asus@LAPTOP-N4CPSGA2 /cygdrive/c/Users/Asus/Desktop/GuhaPranavYelchuru/VITC/sem7
/CompilerLab
$ lex dfa_bca.l

Asus@LAPTOP-N4CPSGA2 /cygdrive/c/Users/Asus/Desktop/GuhaPranavYelchuru/VITC/sem7
Asus@LAPTOP-N4CPSGA2 /cygdrive/c/Users/Asus/Desktop/GuhaPranavYelchuru/VITC/sem7
/CompilerLab.c /usr/lib/libfl.a -o dfa2
$ ./dfa2
Enter strings over {a, b, c} (one per line):
abc
Rejected, doesn't contain 'bca'
bca
Accepted, contains 'bca'
bcca
Rejected, doesn't contain 'bca'
aaccbcaaa
Accepted, contains 'bca'
```

BCSE307P Compiler Design Lab
ASSESSMENT 2 (Part 1+ Part2 -> EXP 5 & 6)
22BCE1462 Guha Pranav Yelchuru

EXPERIMENT 5

Q 1

Aim: Identify the tokens from simple statement as input stored in a linear array

Algorithm:

Initialize an empty token list and a temporary string temp.

1. For each character in the input string:
2. If it's alphanumeric or underscore, append it to temp.
3. Otherwise:
    a. If temp is not empty, save it as a token and clear temp.
    b. If the character is a quote (" or '), read until the matching quote to form a literal token.
    c. Else, treat the character as a single-character token.
4. After processing, classify each token as:
5. **Keyword** if it matches a keyword list.
6. **Literal** if it's a number, string literal, or char literal.
7. **Identifier** if it starts with a letter or underscore.
8. **Operator/Symbol** otherwise.

Source Code

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

#define MAX_TOKENS 100
#define MAX_LEN 50

char tokens[MAX_TOKENS][MAX_LEN];
int token_count = 0;

void store_token(const char *tok) {
   if (token_count < MAX_TOKENS) {
       strcpy(tokens[token_count++], tok);
   }
}

int isKeyword(const char *word) {
   const char *keywords[] = {
       "auto", "break", "case", "char", "const", "continue", "default",
       "do", "double", "else", "enum", "extern", "float", "for",
       "goto", "if", "int", "long", "register", "return", "short",
       "signed", "sizeof", "static", "struct", "switch", "typedef", "union",
       "unsigned", "void", "volatile", "while", "include"
   };
```

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

int main() {
    char input[] = "int variable = 46 'A' \"Hello\"";

    char temp[MAX_LEN] = "";
    int len = strlen(input);

    for (int i = 0; i <= len; i++) {
        if (isalnum(input[i]) || input[i] == '_') {
            int l = strlen(temp);
            temp[l] = input[i];
            temp[l+1] = '\0';
        }
        else {
            if (strlen(temp) > 0) {
                store_token(temp);
                temp[0] = '\0';
            }
            if (!isspace(input[i]) && input[i] != '\0') {
                if (input[i] == '"') {
                    char str[MAX_LEN] = {0};
                    int j = 0;
                    str[j++] = input[i++];
                    while (i < len && input[i] != '"') {
                        str[j++] = input[i++];
                    }
                    str[j++] = '"';
                    str[j] = '\0';
                    store_token(str);
                }
                else if (input[i] == '\'') {
                    char chlit[MAX_LEN] = {0};
                    int j = 0;
                    chlit[j++] = input[i++];
                    while (i < len && input[i] != '\'') {
                        chlit[j++] = input[i++];
                    }
                    chlit[j++] = '\'';
                    chlit[j] = '\0';
                    store_token(chlit);
                }
                else {
                    char op[2] = {input[i], '\0'};
                    store_token(op);
                }
```

```
            }
        }
    }

    printf("Statement: %s\n\n", input);
    printf("Tokens:\n");
    for (int i = 0; i < token_count; i++) {
        if (isKeyword(tokens[i])) {
            printf("%s -> Keyword\n", tokens[i]);
        }
        else if (isdigit(tokens[i][0]) ||
                (tokens[i][0] == '"' && tokens[i][strlen(tokens[i])-1] == '"') ||
                (tokens[i][0] == '\'' && tokens[i][strlen(tokens[i])-1] == '\'')) {
            printf("%s -> Literal\n", tokens[i]);
        }
        else if (isalpha(tokens[i][0]) || tokens[i][0] == '_') {
            printf("%s -> Identifier\n", tokens[i]);
        }
        else {
            printf("%s -> Operator/Symbol\n", tokens[i]);
        }
    }
    return 0;
}
```

OUTPUT:

```
Statement: int variable = 46 'A' "Hello"

Tokens:
int -> Keyword
variable -> Identifier
= -> Operator/Symbol
46 -> Literal
'A' -> Literal
"Hello" -> Literal
```

Q 2

1. **Aim**: Identify the tokens from small program (not exceeding 5 lines) as input stored in a text file

Algorithm:

1. Read the entire file into a string input.
2. Initialize an empty token temp.
3. For each character in input:
    - If alphanumeric or underscore, add to temp.
    - Else:
        - If temp not empty, save as token and clear temp.

- If char is " or ', read full literal including quotes and save as token.
- Otherwise, save single character as token.

4. Classify each token as Keyword, Literal, Identifier, or Operator/Symbol based on rules.

Source Code:

```
#include <stdio.h>

#include <string.h>

#include <ctype.h>


#define MAX_TOKENS 1000

#define MAX_LEN 100


char tokens[MAX_TOKENS][MAX_LEN];

int token_count = 0;

int i;


void store_token(const char *tok) {

   if (token_count < MAX_TOKENS) {

      strcpy(tokens[token_count++], tok);

   }

}


int isKeyword(const char *word) {

   const char *keywords[] = {

      "auto", "break", "case", "char", "const", "continue", "default",

      "do", "double", "else", "enum", "extern", "float", "for",
```

```
        "goto", "if", "int", "long", "register", "return", "short",
        "signed", "sizeof", "static", "struct", "switch", "typedef", "union",
        "unsigned", "void", "volatile", "while", "include"
    };
    int n = sizeof(keywords) / sizeof(keywords[0]);
    for ( i = 0; i < n; i++) {
        if (strcmp(word, keywords[i]) == 0) return 1;
    }
    return 0;
}

int main() {
    FILE *fp = fopen("input.c", "r");
    if (!fp) {
        printf("Error: Could not open input.c\n");
        return 1;
    }

    char input[5000] = "";
    char line[256];
    while (fgets(line, sizeof(line), fp)) {
        strcat(input, line);
    }
    fclose(fp);
```

```
printf("Program from file:\n%s\n", input);

char temp[MAX_LEN] = "";
int len = strlen(input);
int i = 0;

while (i <= len) {
    if (isalnum(input[i]) || input[i] == '_') {
        int l = strlen(temp);
        temp[l] = input[i];
        temp[l+1] = '\0';
        i++;
    }
    else {
        if (strlen(temp) > 0) {
            store_token(temp);
            temp[0] = '\0';
        }

        if (!isspace(input[i]) && input[i] != '\0') {
            // String literal
            if (input[i] == '"') {
                char str[MAX_LEN] = {0};
                int j = 0;
                str[j++] = input[i++];
```

```
        while (i < len && input[i] != '"') {
            str[j++] = input[i++];
        }
        if (i < len) str[j++] = input[i++];
        str[j] = '\0';
        store_token(str);
    }
    // Character literal
    else if (input[i] == '\'') {
        char chlit[MAX_LEN] = {0};
        int j = 0;
        chlit[j++] = input[i++];
        while (i < len && input[i] != '\'') {
            chlit[j++] = input[i++];
        }
        if (i < len) chlit[j++] = input[i++];
        chlit[j] = '\0';
        store_token(chlit);
    }
    else {
        char op[2] = {input[i], '\0'};
        store_token(op);
        i++;
    }
}
```

```
            else {
                i++;
            }
        }
    }

    printf("\nTokens:\n");
    for (i = 0; i < token_count; i++) {
        if (isKeyword(tokens[i])) {
            printf("%s -> Keyword\n", tokens[i]);
        }
        else if (isdigit(tokens[i][0]) ||
                 (tokens[i][0] == '"' && tokens[i][strlen(tokens[i])-1] == '"') ||
                 (tokens[i][0] == '\'' && tokens[i][strlen(tokens[i])-1] == '\'')) {
            printf("%s -> Literal\n", tokens[i]);
        }
        else if (isalpha(tokens[i][0]) || tokens[i][0] == '_') {
            printf("%s -> Identifier\n", tokens[i]);
        }
        else {
            printf("%s -> Operator/Symbol\n", tokens[i]);
        }
    }
    return 0;
}
```

Output

```
Program from file:
#include <stdio.h>

int main() {
    int num;

    printf("Enter an integer: ");
    scanf("%d", &num);

    if (num % 2 == 0)
        printf("%d is Even.\n", num);
    else
        printf("%d is Odd.\n", num);

    return 0;
}
```

```
Tokens:
# -> Operator/Symbol
include -> Keyword
< -> Operator/Symbol
stdio -> Identifier
. -> Operator/Symbol
h -> Identifier
> -> Operator/Symbol
int -> Keyword
main -> Identifier
( -> Operator/Symbol
) -> Operator/Symbol
{ -> Operator/Symbol
int -> Keyword
num -> Identifier
; -> Operator/Symbol
printf -> Identifier
( -> Operator/Symbol
"Enter an integer: " -> Literal
) -> Operator/Symbol
; -> Operator/Symbol
scanf -> Identifier
( -> Operator/Symbol
"%d" -> Literal
```

```
, -> Operator/Symbol
& -> Operator/Symbol
num -> Identifier
) -> Operator/Symbol
; -> Operator/Symbol
if -> Keyword
( -> Operator/Symbol
num -> Identifier
% -> Operator/Symbol
2 -> Literal
= -> Operator/Symbol
= -> Operator/Symbol
0 -> Literal
) -> Operator/Symbol
printf -> Identifier
( -> Operator/Symbol
"%d is Even.\n" -> Literal
, -> Operator/Symbol
num -> Identifier
) -> Operator/Symbol
; -> Operator/Symbol
else -> Keyword
printf -> Identifier
( -> Operator/Symbol
"%d is Odd.\n" -> Literal
, -> Operator/Symbol
num -> Identifier
) -> Operator/Symbol
; -> Operator/Symbol
return -> Keyword
0 -> Literal
; -> Operator/Symbol
} -> Operator/Symbol
```

Q 3

1. **Aim:** Identify the tokens from small program (not exceeding 5 lines) as input get it from the user and store it in a text file

Algorithm:

1. Open output file output.txt for writing.
2. Repeatedly read lines from user input until the line equals "END".
3. For each input line:

   - Tokenize it by:
     - Building tokens from sequences of letters, digits, or underscore.
     - Extracting string literals ("...") and char literals ('...') as single tokens.
     - Treating any other non-space characters as separate tokens.
   - Store tokens in an array.

4. After all input lines are processed, for each token:

   - If token matches a keyword → classify as **Keyword**.
   - Else if token is number or string/char literal → classify as **Literal**.
   - Else if token starts with letter or underscore → classify as **Identifier**.
   - Else → classify as **Operator/Symbol**.

5. Print classifications to console and write the same output to output.txt.
6. Close the file and end.

Source Code:

```
#include <stdio.h>

#include <string.h>

#include <ctype.h>



#define MAX_TOKENS 100

#define MAX_LEN 50

#define MAX_LINE_LEN 200



char tokens[MAX_TOKENS][MAX_LEN];
```

```
int token_count = 0;

void store_token(const char *tok) {
    if (token_count < MAX_TOKENS && strlen(tok) > 0) {
        strcpy(tokens[token_count++], tok);
    }
}

int isKeyword(const char *word) {
    const char *keywords[] = {
        "auto", "break", "case", "char", "const", "continue", "default",
        "do", "double", "else", "enum", "extern", "float", "for",
        "goto", "if", "int", "long", "register", "return", "short",
        "signed", "sizeof", "static", "struct", "switch", "typedef", "union",
        "unsigned", "void", "volatile", "while"
    };
    int n = sizeof(keywords) / sizeof(keywords[0]);
    for (int i = 0; i < n; i++) {
        if (strcmp(word, keywords[i]) == 0) return 1;
    }
    return 0;
}

void tokenize_line(const char *input) {
    char temp[MAX_LEN] = "";
```

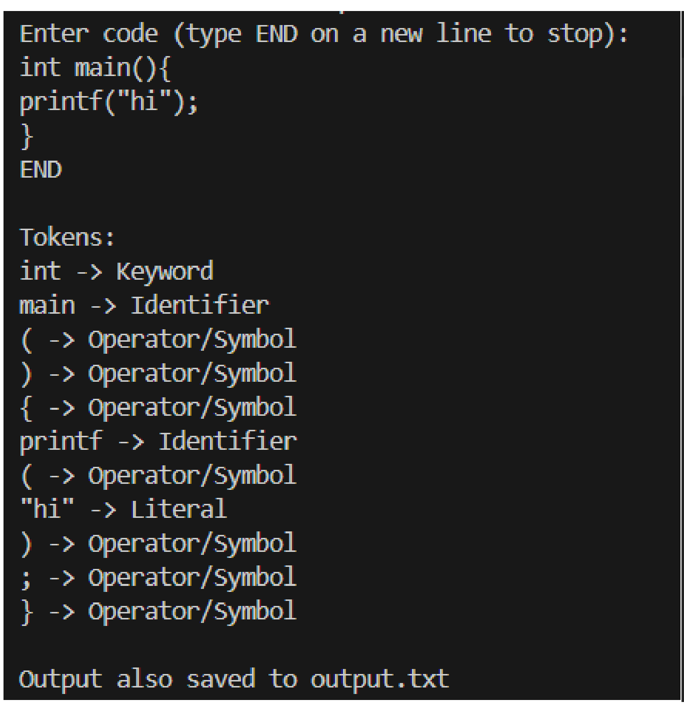
```
int len = strlen(input);


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

    if (isalnum((unsigned char)input[i]) || input[i] == '_') {

        int l = strlen(temp);

        temp[l] = input[i];

        temp[l+1] = '\0';

    } else {

        if (strlen(temp) > 0) {

            store_token(temp);

            temp[0] = '\0';

        }

        if (!isspace((unsigned char)input[i]) && input[i] != '\0') {

            if (input[i] == '"') {

                char str[MAX_LEN] = {0};

                int j = 0;

                str[j++] = input[i++];

                while (i < len && input[i] != '"') {

                    str[j++] = input[i++];

                }

                str[j++] = '"';

                str[j] = '\0';

                store_token(str);

            }

            else if (input[i] == '\'') {
```

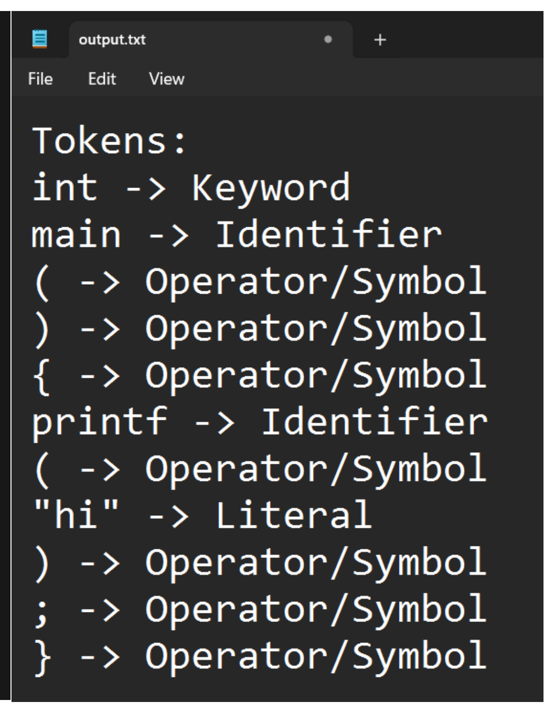
```
                char chlit[MAX_LEN] = {0};

                int j = 0;

                chlit[j++] = input[i++];

                while (i < len && input[i] != '\'') {

                    chlit[j++] = input[i++];

                }

                chlit[j++] = '\'';

                chlit[j] = '\0';

                store_token(chlit);

            }

            else {

                char op[2] = {input[i], '\0'};

                store_token(op);

            }

        }

    }

}


int main() {

    char line[MAX_LINE_LEN];

    FILE *fp = fopen("output.txt", "w");

    if (!fp) {

        printf("Error opening output.txt for writing.\n");

        return 1;
```

```
}

printf("Enter code (type END on a new line to stop):\n");

while (1) {
    if (!fgets(line, sizeof(line), stdin)) break;
    line[strcspn(line, "\r\n")] = '\0'; // remove CR+LF

    if (strcmp(line, "END") == 0) break;

    tokenize_line(line);
}
printf("\nTokens:\n");
fprintf(fp, "Tokens:\n");
for (int i = 0; i < token_count; i++) {
    if (isKeyword(tokens[i])) {
        printf("%s -> Keyword\n", tokens[i]);
        fprintf(fp, "%s -> Keyword\n", tokens[i]);
    }
    else if (isdigit((unsigned char)tokens[i][0]) ||
            (tokens[i][0] == '"' && tokens[i][strlen(tokens[i])-1] == '"') ||
            (tokens[i][0] == '\'' && tokens[i][strlen(tokens[i])-1] == '\'')) {
        printf("%s -> Literal\n", tokens[i]);
        fprintf(fp, "%s -> Literal\n", tokens[i]);
    }
```

```
        else if (isalpha((unsigned char)tokens[i][0]) || tokens[i][0] == '_') {
            printf("%s -> Identifier\n", tokens[i]);
            fprintf(fp, "%s -> Identifier\n", tokens[i]);
        }
        else {
            printf("%s -> Operator/Symbol\n", tokens[i]);
            fprintf(fp, "%s -> Operator/Symbol\n", tokens[i]);
        }
    }
    fclose(fp);
    printf("\nOutput also saved to output.txt\n");
    return 0;
}
```

Output:





EXPERIMENT 6

Q1

Aim: Implement LEX code to count the frequency of the given word in a file

Algorithm:

1. Read the target word and input filename from command-line arguments.
2. Open the input file for reading.
3. For each word token ([a-zA-Z]+) in the file:
   a. If the token matches the target word exactly, increment a counter.
4. After scanning the file, print the total count of occurrences.
5. Close the input file.

Source Code:

```
%{

#include <stdio.h>

#include <string.h>

char target[100];

int count = 0;

%}


%%

[a-zA-Z]+ {

   if (strcmp(yytext, target) == 0) {

       count++;

   }

}

.   ;
```

```
%%

int main(int argc, char *argv[]) {
    if (argc != 3) {
        printf("Usage: %s <word> <filename>\n", argv[0]);
        return 1;
    }

    strcpy(target, argv[1]);
    yyin = fopen(argv[2], "r");
    if (!yyin) {
        printf("Error: could not open file %s\n", argv[2]);
        return 1;
    }

    yylex();
    printf("The word '%s' occurred %d times in file %s\n", target, count, argv[2]);

    fclose(yyin);
    return 0;
}
```

Output:

```
Asus@LAPTOP-N4CPSGA2 /cygdrive/c/Users/Asus/Desktop/GuhaPranavYelchuru/VITC/sem7
/CompilerLab/ass5
$ lex Lex1.l

Asus@LAPTOP-N4CPSGA2 /cygdrive/c/Users/Asus/Desktop/GuhaPranavYelchuru/VITC/sem7
/CompilerLab/ass5
$ gcc lex.yy.c /usr/lib/libfl.a -o Lex1

Asus@LAPTOP-N4CPSGA2 /cygdrive/c/Users/Asus/Desktop/GuhaPranavYelchuru/VITC/sem7
/CompilerLab/ass5
$ ./Lex1 Operator output.txt









The word 'Operator' occurred 7 times in file output.txt
```

Q2

Aim: Implement LEX code to replace a word with another taking input from file.

Algorithm:

1. Read old_word, new_word, and filename from command-line arguments.
2. Open the input file for reading.
3. For each word token ([a-zA-Z]+) in the file:
    a. If the token equals old_word, print new_word.
    b. Else, print the token as-is.
4. For whitespace and other characters, print them unchanged.
5. Close the input file.

Source Code:

```
%{

#include <stdio.h>

#include <string.h>
```

```
char oldWord[100];
char newWord[100];
%}

%%
[a-zA-Z]+ {
    if (strcmp(yytext, oldWord) == 0) {
        printf("%s", newWord);
    } else {
        printf("%s", yytext);
    }
}
[ \t\n]   { ECHO; }
.        { ECHO; }
%%

int main(int argc, char *argv[]) {
    if (argc != 4) {
        printf("Usage: %s <old_word> <new_word> <filename>\n", argv[0]);
        return 1;
    }
    strcpy(oldWord, argv[1]);
    strcpy(newWord, argv[2]);

    yyin = fopen(argv[3], "r");
```

```
    if (!yyin) {
        perror("Error opening file");
        return 1;
    }
    yylex();
    fclose(yyin);
    return 0;
}
```

Output:

```
Asus@LAPTOP-N4CPSGA2 /cygdrive/c/Users/Asus/Desktop/GuhaPranavYelchuru/VITC/sem7
/CompilerLab/ass5
$ lex Lex2.l

Asus@LAPTOP-N4CPSGA2 /cygdrive/c/Users/Asus/Desktop/GuhaPranavYelchuru/VITC/sem7
/CompilerLab/ass5
$ gcc lex.yy.c /usr/lib/libfl.a -o Lex2

Asus@LAPTOP-N4CPSGA2 /cygdrive/c/Users/Asus/Desktop/GuhaPranavYelchuru/VITC/sem7
/CompilerLab/ass5
$ ./Lex2 Identifier Magic_Word output.txt > updated_output.txt
```

```
output.txt
File Edit View

Tokens:
int -> Keyword
main -> Identifier
( -> Operator/Symbol
) -> Operator/Symbol
{ -> Operator/Symbol
printf -> Identifier
( -> Operator/Symbol
"hi" -> Literal
) -> Operator/Symbol
; -> Operator/Symbol
} -> Operator/Symbol
```

```
updated_output.txt
File Edit View

Tokens:
int -> Keyword
main -> Magic_Word
( -> Operator/Symbol
) -> Operator/Symbol
{ -> Operator/Symbol
printf -> Magic_Word
( -> Operator/Symbol
"hi" -> Literal
) -> Operator/Symbol
; -> Operator/Symbol
} -> Operator/Symbol
```

Q3

Aim: Implement LEX code to find the length of the longest word.

Algorithm:

1. Read the input filename from command-line arguments.
2. Open the file for reading.
3. For each word token ([a-zA-Z]+) in the file:
    a. Check its length.
    b. If longer than current maximum, update maxLength and save the word.
4. Ignore non-word characters.
5. After scanning the entire file, print the longest word and its length.
6. Close the file.

Source Code:

```
%{

#include <stdio.h>

#include <string.h>



int maxLength = 0;
```

```
char longestWord[256];
%}

%%
[a-zA-Z]+ {
    int len = strlen(yytext);
    if (len > maxLength) {
        maxLength = len;
        strcpy(longestWord, yytext);
    }
}
.   ;   // Ignore non-word characters
%%

int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: %s <filename>\n", argv[0]);
        return 1;
    }

    yyin = fopen(argv[1], "r");
    if (!yyin) {
        perror("Error opening file");
        return 1;
```

```
    }


    yylex();

    fclose(yyin);


    printf("Longest word: %s\n", longestWord);

    printf("Length: %d\n", maxLength);


    return 0;

}
```

Output

Q4

Aim: Construct a lexical analyser using LEX tool.

Algorithm:

1. Start the program.
2. Read input character stream (from file or stdin).
3. Match tokens using regular expressions defined in LEX:
    a. If the input matches "if", "else", "while", "for" → print Keyword.
    b. If the input matches the pattern {id} → print Identifier.
    c. If the input matches {number} → print Number.
    d. If the input matches "+", "-", "*", "/", "=" → print Operator.
    e. If the input matches {whitespace} → ignore.
    f. Otherwise → print Unknown token.
4. Repeat until end of file is reached.
5. Stop the program.

Source Code:

```
%{

#include <stdio.h>

#include <stdlib.h>



int yywrap();

%}



digit      [0-9]

letter     [a-zA-Z]

id         {letter}({letter}|{digit})*

number     {digit}+

whitespace  [ \t\n]+



%%



"if"          { printf("Keyword: IF\n"); }
```

```
"else"        { printf("Keyword: ELSE\n"); }
"while"       { printf("Keyword: WHILE\n"); }
"for"         { printf("Keyword: FOR\n"); }

{id}          { printf("Identifier: %s\n", yytext); }
{number}      { printf("Number: %s\n", yytext); }
"+"           { printf("Operator: PLUS\n"); }
"-"           { printf("Operator: MINUS\n"); }
"*"           { printf("Operator: MULTIPLY\n"); }
"/"           { printf("Operator: DIVIDE\n"); }
"="           { printf("Operator: ASSIGN\n"); }

{whitespace}

.             { printf("Unknown token: %s\n", yytext); }

%%

int yywrap() {
   return 1;
}

int main() {
   yylex();
```

```
    return 0;
}
```

Output:

```
input.txt
if count1 = 100
    while count1 = count1 - 1
        sum = sum + count1
else
    result = sum / 2
for index = 0
    index = index + 1
```

```
Asus@LAPTOP-N4CPSGA2 /cygdrive/c/Users/Asus/Desktop/GuhaPranavYelchuru/VITC/sem7
/CompilerLab/ass5
$ lex lexanalyzer.l

Asus@LAPTOP-N4CPSGA2 /cygdrive/c/Users/Asus/Desktop/GuhaPranavYelchuru/VITC/sem7
/CompilerLab/ass5
$ gcc lex.yy.c /usr/lib/libfl.a -o lexanalyzer
```

```
/CompilerLab/ass5
$ ./lexanalyzer


Asus@LAPTOP-N4CPSGA2 /cygdrive/c/Users/Asus/Desktop/GuhaPranavYelchuru/VITC/sem7
/CompilerLab/ass5
$ ./lexanalyzer < input.txt
Keyword: IF
Identifier: count1
Operator: ASSIGN
Number: 100
Unknown token:
Keyword: WHILE
Identifier: count1
Operator: ASSIGN
Identifier: count1
Operator: MINUS
Number: 1
Unknown token:
Identifier: sum
Operator: ASSIGN
Identifier: sum
Operator: PLUS
Identifier: count1
Unknown token:
Keyword: ELSE
Unknown token:
Identifier: result
Operator: ASSIGN
Identifier: sum
Operator: DIVIDE
Number: 2
Unknown token:
Keyword: FOR
Identifier: index
Operator: ASSIGN
Number: 0
Unknown token:
Identifier: index
Operator: ASSIGN
Identifier: index
Operator: PLUS
Number: 1
Unknown token:
```

BCSE307P Compiler Design Lab
ASSESSMENT 3 (Part 1)
22BCE1462 Guha Pranav Yelchuru

EXPERIMENT 7

(a)

Aim: Construct Predictive parse table using C language.
Hint: Consider the input grammar without left recursion, find FIRST and FOLLOW for each non-terminal and then construct the parse table.

Algorithm:

- Input: Read grammar productions and parse LHS→RHS
- Classify: Separate terminals and non-terminals
- FIRST Sets: For each non-terminal, compute all possible first terminals
- FOLLOW Sets: For each non-terminal, compute terminals that can follow it
- Parse Table: Fill table[non-terminal] [terminal] with production rules using FIRST/FOLLOW
- Output: Display FIRST/FOLLOW sets and parse table

Source Code:

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>



#define MAX_RULES 100

#define MAX_SYMBOLS 100

#define MAX_LEN 50



typedef char Token[MAX_LEN];



typedef struct {

   Token left;
```

```
    int rightCount;
    Token right[MAX_SYMBOLS];
} Rule;

Rule grammar[MAX_RULES];
int grammarCount = 0;

Token nonTerminals[MAX_SYMBOLS];
int nonTermCount = 0;

Token terminals[MAX_SYMBOLS];
int termCount = 0;

Token startSym;

typedef struct {
    Token elements[MAX_SYMBOLS];
    int size;
} TokenSet;

TokenSet firstMap[MAX_SYMBOLS];
TokenSet followMap[MAX_SYMBOLS];
int parseTable[MAX_SYMBOLS][MAX_SYMBOLS];
Token termSymbols[MAX_SYMBOLS];
```

```
int termIndexCount = 0;

int contains(Token arr[], int size, const char *tok) {
    for (int i=0; i<size; i++)
        if (strcmp(arr[i], tok)==0) return 1;
    return 0;
}

int addSymbol(Token arr[], int *size, const char *tok) {
    if (!contains(arr,*size,tok)) {
        strcpy(arr[*size], tok);
        (*size)++;
        return 1;
    }
    return 0;
}

void initSet(TokenSet *s){ s->size=0; }
int addToSet(TokenSet *s, const char *tok){
    if (!contains(s->elements, s->size, tok)) {
        strcpy(s->elements[s->size++], tok);
        return 1;
    }
    return 0;
```

```
}
int findNonTermIndex(const char *nt){
   for (int i=0; i<nonTermCount; i++)
      if(strcmp(nonTerminals[i],nt)==0) return i;
   return -1;
}
int findTermIndex(const char *t){
   for (int i=0; i<termIndexCount; i++)
      if(strcmp(termSymbols[i],t)==0) return i;
   return -1;
}

int splitWithBrackets(const char *line, Token tokens[]){
   int count=0,i=0;
   while(line[i]){
      if (isspace(line[i])) { i++; continue; }
      if (line[i]=='('||line[i]==')'||line[i]=='{'||line[i]=='}'||line[i]=='['||line[i]==']'){
         tokens[count][0]=line[i]; tokens[count][1]='\0'; count++; i++;
      } else {
         int j=0;
         while(line[i] && !isspace(line[i]) &&
             line[i]!='('&&line[i]!=')'&&line[i]!='{'&&line[i]!='}'&&line[i]!='['&&line[i]!=']'){
            tokens[count][j++]=line[i++];
         }
```

```
            tokens[count][j]='\0'; count++;
        }
    }
    return count;
}

void computeFIRST(const char *sym, TokenSet *result){
    if (!isupper(sym[0]) || strcmp(sym,"epsilon")==0){
        addToSet(result, sym); return;
    }
    int idx=findNonTermIndex(sym);
    for (int r=0;r<grammarCount;r++){
        if(strcmp(grammar[r].left,sym)==0){
            int allEps=1;
            for (int i=0;i<grammar[r].rightCount;i++){
                TokenSet temp; initSet(&temp);
                computeFIRST(grammar[r].right[i], &temp);
                for (int j=0;j<temp.size;j++)
                    if (strcmp(temp.elements[j],"epsilon")!=0)
                        addToSet(result,temp.elements[j]);
                if(!contains(temp.elements,temp.size,"epsilon")){
                    allEps=0; break;
                }
            }
```

```
            if(allEps) addToSet(result,"epsilon");
        }
    }
}

void computeFOLLOW(const char *sym, TokenSet *result){
    if(strcmp(sym,startSym)==0)
        addToSet(result,"$");
    for (int r=0;r<grammarCount;r++){
        for (int i=0;i<grammar[r].rightCount;i++){
            if(strcmp(grammar[r].right[i],sym)==0){
                int allEps=1;
                for (int j=i+1;j<grammar[r].rightCount;j++){
                    TokenSet temp; initSet(&temp);
                    computeFIRST(grammar[r].right[j],&temp);
                    for (int k=0;k<temp.size;k++)
                        if(strcmp(temp.elements[k],"epsilon")!=0)
                            addToSet(result,temp.elements[k]);
                    if(!contains(temp.elements,temp.size,"epsilon")){ allEps=0; break; }
                }
                if(i+1==grammar[r].rightCount || allEps){
                    if(strcmp(grammar[r].left,sym)!=0){
                        int lidx=findNonTermIndex(grammar[r].left);
                        computeFOLLOW(grammar[r].left,&followMap[lidx]);
```

```
                for (int k=0;k<followMap[lidx].size;k++)
                    addToSet(result,followMap[lidx].elements[k]);
              }
            }
          }
        }
    }
}

void createParseTable(){
    for(int i=0;i<nonTermCount;i++)
        for(int j=0;j<termIndexCount;j++) parseTable[i][j]=-1;

    for (int r=0;r<grammarCount;r++){
        int allEps=1;
        for (int i=0;i<grammar[r].rightCount;i++){
            TokenSet temp; initSet(&temp);
            computeFIRST(grammar[r].right[i],&temp);
            for (int k=0;k<temp.size;k++){
                if(strcmp(temp.elements[k],"epsilon")!=0){
                    int nt=findNonTermIndex(grammar[r].left);
                    int t=findTermIndex(temp.elements[k]);
                    parseTable[nt][t]=r;
                }
```

```
        }
        if (!contains(temp.elements,temp.size,"epsilon")){ allEps=0; break; }
      }
      if(allEps){
        int nt=findNonTermIndex(grammar[r].left);
        TokenSet f; initSet(&f); computeFOLLOW(grammar[r].left,&f);
        for(int k=0;k<f.size;k++){
          int t=findTermIndex(f.elements[k]);
          parseTable[nt][t]=r;
        }
      }
    }
}

void showFirstFollow(){
    printf("\nFIRST and FOLLOW Sets:\n");
    for(int i=0;i<nonTermCount;i++){
      initSet(&firstMap[i]);
      computeFIRST(nonTerminals[i],&firstMap[i]);
      printf("FIRST(%s)={",nonTerminals[i]);
      for(int j=0;j<firstMap[i].size;j++) printf("%s ",firstMap[i].elements[j]);
      printf("}  ");

      initSet(&followMap[i]);
```

```
        computeFOLLOW(nonTerminals[i],&followMap[i]);
        printf("FOLLOW(%s)={",nonTerminals[i]);
        for(int j=0;j<followMap[i].size;j++) printf("%s ",followMap[i].elements[j]);
        printf("}\n");
    }
}

void showParseTable(){
    printf("\nParse Table:\n        ");
    for(int j=0;j<termIndexCount;j++) printf("%8s",termSymbols[j]);
    printf("\n");
    for(int i=0;i<nonTermCount;i++){
        printf("%-8s",nonTerminals[i]);
        for(int j=0;j<termIndexCount;j++){
            if(parseTable[i][j]!=-1){
                printf("%8s->",grammar[parseTable[i][j]].left);
                for(int k=0;k<grammar[parseTable[i][j]].rightCount;k++)
                    printf("%s",grammar[parseTable[i][j]].right[k]);
            } else printf("%8s","-");
        }
        printf("\n");
    }
}
```

```
int main(){
    int prodCount;
    printf("Enter number of productions: ");
    scanf("%d",&prodCount); getchar();
    printf("Enter productions (e.g., E->T E', E'->+ T E', E'->epsilon, T->( E )):\n");
    for(int i=0;i<prodCount;i++){
        char line[200];
        fgets(line,sizeof(line),stdin);
        line[strcspn(line,"\n")]=0;

        char *arrow=strstr(line,"->");
        *arrow='\0';
        char *lhs=line; char *rhs=arrow+2;

        strcpy(grammar[i].left,lhs);
        grammar[i].rightCount=splitWithBrackets(rhs,grammar[i].right);
        grammarCount++;

        addSymbol(nonTerminals,&nonTermCount,lhs);
        for(int j=0;j<grammar[i].rightCount;j++){
            if(!isupper(grammar[i].right[j][0]) && strcmp(grammar[i].right[j],"epsilon")!=0)
                addSymbol(terminals,&termCount,grammar[i].right[j]);
        }
    }
```

```
    strcpy(startSym,grammar[0].left);

    for(int j=0;j<termCount;j++) strcpy(termSymbols[termIndexCount++],terminals[j]);
    strcpy(termSymbols[termIndexCount++],"$");

    showFirstFollow();
    createParseTable();
    showParseTable();
    return 0;
}
```

OUTPUT:

```
Enter number of productions: 8
Enter productions (e.g., E->T E', E'->+ T E', E'->epsilon, T->( E )):
E->T E'
E'->+ T E'
E'->epsilon
T->F T'
T'->* F T'
T'->epsilon
F->( E )
F->id

FIRST and FOLLOW Sets:
FIRST(E)={( id }  FOLLOW(E)={$ ) }
FIRST(E')={+ epsilon }  FOLLOW(E')={$ ) }
FIRST(T)={( id }  FOLLOW(T)={+ $ ) }
FIRST(T')={* epsilon }  FOLLOW(T')={+ $ ) }
FIRST(F)={( id }  FOLLOW(F)={* + $ ) }
Parse Table:
                +       *       (       )       id       $
E             -       -       E->TE'       -       E->TE'       -
E'           E'->+TE'       -       -       E'->epsilon       -       E'->epsilon
T             -       -       T->FT'       -       T->FT'       -
T'           T'->epsilon       T'->*FT'       -       T'->epsilon       -       T'
    ->epsilon
F             -       -       F->(E)       -       F->id       -
```

(b)

Aim: Implement the Predictive parsing algorithm, get parse table and input string are inputs. Use C language for implementation.

Algorithm:

- Input Setup: Read non-terminals, terminals, and parse table entries manually
- Parse Table: Store production rules indexed by [non-terminal][terminal]
- Stack Initialization: Push '$' then start symbol onto stack
- Parsing Loop:
    - If stack top matches input: pop stack, advance input
    - If stack top is non-terminal: lookup rule in parse table, pop stack, push RHS in reverse
    - If epsilon production: just pop stac
    - If mismatch or no rule: ERROR
- Accept Condition: Stack contains only '$' and input is fully consumed
- Output: Show parsing steps and accept/reject result

Source Code:

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>



#define MAX_SYMBOLS 50

#define MAX_RULES 100

#define MAX_TOKENS 100

#define MAX_LEN 50



typedef char Token[MAX_LEN];



typedef struct {
```

```
    Token left;

    int rightCount;

    Token right[MAX_SYMBOLS];

} Rule;


Rule parseTable[MAX_SYMBOLS][MAX_SYMBOLS];

int used[MAX_SYMBOLS][MAX_SYMBOLS];


Token nonTerminals[MAX_SYMBOLS]; int nonTermCount=0;

Token terminals[MAX_SYMBOLS]; int termCount=0;

Token startSym;


int findNonTermIndex(const char *nt){

    for(int i=0;i<nonTermCount;i++)

        if(strcmp(nonTerminals[i],nt)==0) return i;

    return -1;

}

int findTermIndex(const char *t){

    for(int i=0;i<termCount;i++)

        if(strcmp(terminals[i],t)==0) return i;

    return -1;

}


typedef struct{
```

```c
    Token arr[MAX_TOKENS];
    int top;
} Stack;

void push(Stack* s,const char *tok){ strcpy(s->arr[++s->top],tok); }
void pop(Stack* s){ if(s->top>=0) s->top--; }
char* peek(Stack* s){ return s->arr[s->top]; }

int split(const char *line, Token tokens[]){
    int count=0,i=0;
    while(line[i]){
        if(isspace(line[i])){ i++; continue; }
        int j=0;
        while(line[i] && !isspace(line[i]))
            tokens[count][j++]=line[i++];
        tokens[count][j]='\0';
        count++;
    }
    return count;
}

int runParser(Token input[], int n){
    Stack stk; stk.top=-1;
    push(&stk,"$"); push(&stk,startSym);
```

```
int idx=0;

printf("\nParsing steps:\n");

printf("%-20s %-20s %s\n", "Stack", "Input", "Action");


while(stk.top>=0){

   char *top=peek(&stk);

   char *cur=input[idx];


   {


      printf("[");

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

         printf("%s", stk.arr[i]);

         if (i!=stk.top) printf(" ");

      }

      printf("]");


      printf("     [");

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

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

         if (i!=n-1) printf(" ");

      }

      printf("]  ");

   }
```

```
if(strcmp(top,cur)==0){
    if(strcmp(top,"$")==0){
        printf("ACCEPT\n");
        return 1;
    }
    pop(&stk);
    idx++;
    printf("Match %s\n", cur);
}
else if(isupper(top[0])){
    int nt=findNonTermIndex(top);
    int t=findTermIndex(cur);
    if(nt<0||t<0||!used[nt][t]){
        printf("\nERROR: No rule for (%s,%s)\n",top,cur);
        return 0;
    }
    Rule r=parseTable[nt][t];
    pop(&stk);
    if(!(r.rightCount==1 && strcmp(r.right[0],"epsilon")==0)){
        for(int k=r.rightCount-1;k>=0;k--)
            push(&stk,r.right[k]);
    }
    printf("%s -> ",r.left);
```

```
        for(int k=0;k<r.rightCount;k++) printf("%s ",r.right[k]);
        printf("\n");
      } else {
        printf("\nERROR: Terminal mismatch (%s vs %s)\n",top,cur);
        return 0;
      }
    }
    return 0;
}

int main(){
    printf("Enter number of non-terminals: ");
    scanf("%d",&nonTermCount); getchar();
    printf("Enter non-terminals (one per line):\n");
    for(int i=0;i<nonTermCount;i++){
      fgets(nonTerminals[i],MAX_LEN,stdin);
      nonTerminals[i][strcspn(nonTerminals[i],"\n")]=0; // remove newline
    }

    printf("Enter number of terminals: ");
    scanf("%d",&termCount); getchar();
    printf("Enter terminals (one per line, include $ as end marker):\n");
    for(int i=0;i<termCount;i++){
      fgets(terminals[i],MAX_LEN,stdin);
```

```
    terminals[i][strcspn(terminals[i],"\n")]=0;
}

printf("\nEnter parse table entries (enter # for empty entry):\n");
for(int i=0;i<nonTermCount;i++){
    for(int j=0;j<termCount;j++){
        char line[200];
        printf("Entry for [%s, %s]: ", nonTerminals[i], terminals[j]);
        fgets(line, sizeof(line), stdin);
        line[strcspn(line,"\n")]=0;
        if(strcmp(line, "#")==0){
            used[i][j] = 0;
            continue;
        }

        used[i][j] = 1;

        char *arrow = strstr(line, "->");
        if(!arrow){
            printf("Invalid format. Use A->... format.\n");
            return 1;
        }
        *arrow = 0;
        strcpy(parseTable[i][j].left, line);
```

```
        char *rhs = arrow + 2;


        parseTable[i][j].rightCount = split(rhs, parseTable[i][j].right);
    }
}


int startIndex = -1;
for (int i = 0; i < nonTermCount; i++) {
    for (int j = 0; j < termCount; j++) {
        if (used[i][j]) {
            startIndex = i;
            break;
        }
    }
    if (startIndex != -1) break;
}
if (startIndex == -1) {
    printf("Error: No valid parse table entries found.\n");
    return 1;
}
strcpy(startSym, nonTerminals[startIndex]);
printf("\nStart symbol automatically set to: %s\n", startSym);


while(1){
```

```
        char inputLine[200];
        printf("\nEnter input string to parse (tokens separated by space, 0 to exit): ");
        fgets(inputLine,sizeof(inputLine),stdin);
        inputLine[strcspn(inputLine,"\n")]=0;
        if(strcmp(inputLine,"0")==0) break;

        Token tokens[MAX_TOKENS];
        int count = split(inputLine, tokens);
        strcpy(tokens[count++], "$");

        int ok = runParser(tokens, count);
        printf("\nResult: The string %s accepted by the grammar.\n", ok ? "IS" : "is NOT");
    }

    printf("\nParser terminated. Goodbye!\n");
    return 0;
}
```

Output

```
Enter number of non-terminals: 5
Enter non-terminals (one per line):
E
E'
T
T'
F
Enter number of terminals: 6
Enter terminals (one per line, include $ as end marker):
+
*
(
)
id
$
```

```
Enter parse table entries (enter # for empty entry):
Entry for [E, +]: #
Entry for [E, *]: #
Entry for [E, (]: E->T E'
Entry for [E, )]: #
Entry for [E, id]: E->T E'
Entry for [E, $]: #
Entry for [E', +]: E'->+ T E'
Entry for [E', *]: #
Entry for [E', (]: #
Entry for [E', )]: E'->epsilon
Entry for [E', id]: #
Entry for [E', $]: E'->epsilon
Entry for [T, +]: #
Entry for [T, *]: #
Entry for [T, (]: T->F T'
Entry for [T, )]: #
Entry for [T, id]: T->F T'
Entry for [T, $]: #
Entry for [T', +]: T'->epsilon
Entry for [T', *]: T'->* F T'
Entry for [T', (]: #
Entry for [T', )]: T'->epsilon
Entry for [T', id]: #
Entry for [T', $]: T'->epsilon
Entry for [F, +]: #
Entry for [F, *]: #
Entry for [F, (]: F->( E )
Entry for [F, )]: #
Entry for [F, id]: F->id
Entry for [F, $]: #

Start symbol automatically set to: E
```

```
Enter input string to parse (tokens separated by space, 0 to exit): id + id * id

Parsing steps:
Stack                    Input                   Action
[$ E]      [id + id * id $]  E -> T E'
[$ E' T]      [id + id * id $]  T -> F T'
[$ E' T' F]      [id + id * id $]  F -> id
[$ E' T' id]      [id + id * id $]  Match id
[$ E' T']      [+ id * id $]  T' -> epsilon
[$ E']      [+ id * id $]  E' -> + T E'
[$ E' T +]      [+ id * id $]  Match +
[$ E' T]      [id * id $]  T -> F T'
[$ E' T' F]      [id * id $]  F -> id
[$ E' T' id]      [id * id $]  Match id
[$ E' T']      [* id $]  T' -> * F T'
[$ E' T' F *]      [* id $]  Match *
[$ E' T' F]      [id $]  F -> id
[$ E' T' id]      [id $]  Match id
[$ E' T']      [$]  T' -> epsilon
[$ E']      [$]  E' -> epsilon
[$]      [$]  ACCEPT

Result: The string IS accepted by the grammar.

Enter input string to parse (tokens separated by space, 0 to exit):
```

EXPERIMENT 8

(a)

Aim: Construct precedence table for the given operator grammar.

Algorithm:

- Input: Read grammar productions (e.g., E->E+T, T->T*F)
- Collect Symbols: Extract non-terminals (uppercase) and terminals (lowercase/operators)
- Compute Leading Sets: For each non-terminal, find all terminals that can appear first in its derivations
- Compute Trailing Sets: For each non-terminal, find all terminals that can appear last in its derivations
- Build Precedence Table: Apply rules:
    - Terminal adjacent to terminal → a = b
    - Terminal before non-terminal → a < Leading(B)
    - Non-terminal before terminal → Trailing(A) > b
    - Terminal-NonTerminal-Terminal → a = c
- Set Boundaries: Add '$' with $ < all and all > $
- Output: Display Leading/Trailing sets and precedence table with <, =, > relations

Source Code:

```
#include <stdio.h>

#include <string.h>

#include <ctype.h>



#define MAX 20

#define SIZE 100



char productions[MAX][SIZE];

int n;

char nonTerminals[MAX];

int ntCount = 0;
```

```
char terminals[MAX];

int termCount = 0;

int leading[MAX][MAX];

int trailing[MAX][MAX];

int opPrecedenceTable[MAX][MAX];


int findNonTerminalIndex(char c) {

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

      if (nonTerminals[i] == c)

         return i;

   }

   return -1;

}


int findTerminalIndex(char c) {

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

      if (terminals[i] == c)

         return i;

   }

   return -1;

}


int isNonTerminal(char c) {

   return (c >= 'A' && c <= 'Z');
```

```
}

void addTerminal(char c) {
    if (c == '#' || c == '\0')
        return;
    if (!isNonTerminal(c) && strchr(terminals, c) == NULL) {
        terminals[termCount++] = c;
    }
}

void collectNonTerminals() {
    for (int i = 0; i < n; i++) {
        if (!strchr(nonTerminals, productions[i][0])) {
            nonTerminals[ntCount++] = productions[i][0];
        }
    }
}

void collectTerminals() {
    for (int i = 0; i < n; i++) {
        char *rhs = productions[i] + 3;
        for (int j = 0; rhs[j] != '\0'; j++) {
            addTerminal(rhs[j]);
        }
```

```
    }
}

void computeLeading() {
    for (int i = 0; i < ntCount; i++)
        for (int j = 0; j < termCount; j++)
            leading[i][j] = 0;

    int changed;
    do {
        changed = 0;
        for (int i = 0; i < n; i++) {
            int A = findNonTerminalIndex(productions[i][0]);
            char *rhs = productions[i] + 3;

            if (rhs[0] == '\0')
                continue;

            if (!isNonTerminal(rhs[0])) {
                int t = findTerminalIndex(rhs[0]);
                if (t != -1 && leading[A][t] == 0) {
                    leading[A][t] = 1;
                    changed = 1;
                }
```

```
        }
        else if (rhs[1] != '\0' && !isNonTerminal(rhs[1])) {
          int t = findTerminalIndex(rhs[1]);
          if (t != -1 && leading[A][t] == 0) {
            leading[A][t] = 1;
            changed = 1;
          }
        }

        if (isNonTerminal(rhs[0])) {
          int B = findNonTerminalIndex(rhs[0]);
          if (B != -1) {
            for (int k = 0; k < termCount; k++) {
              if (leading[B][k] == 1 && leading[A][k] == 0) {
                leading[A][k] = 1;
                changed = 1;
              }
            }
          }
        }
      }
    } while (changed);
}
```

```
void computeTrailing() {
    for (int i = 0; i < ntCount; i++)
        for (int j = 0; j < termCount; j++)
            trailing[i][j] = 0;

    int changed;
    do {
        changed = 0;
        for (int i = 0; i < n; i++) {
            int A = findNonTerminalIndex(productions[i][0]);
            char *rhs = productions[i] + 3;
            int len = strlen(rhs);

            if (len == 0)
                continue;

            if (!isNonTerminal(rhs[len - 1])) {
                int t = findTerminalIndex(rhs[len - 1]);
                if (t != -1 && trailing[A][t] == 0) {
                    trailing[A][t] = 1;
                    changed = 1;
                }
            }
            else if (len > 1 && !isNonTerminal(rhs[len - 2])) {
```

```
            int t = findTerminalIndex(rhs[len - 2]);
            if (t != -1 && trailing[A][t] == 0) {
                trailing[A][t] = 1;
                changed = 1;
            }
        }

        if (isNonTerminal(rhs[len - 1])) {
            int B = findNonTerminalIndex(rhs[len - 1]);
            if (B != -1) {
                for (int k = 0; k < termCount; k++) {
                    if (trailing[B][k] == 1 && trailing[A][k] == 0) {
                        trailing[A][k] = 1;
                        changed = 1;
                    }
                }
            }
        }
    }
  } while (changed);
}

void buildOperatorPrecedenceTable() {
  for (int i = 0; i < termCount; i++)
```

```
    for (int j = 0; j < termCount; j++)
        opPrecedenceTable[i][j] = 0;

for (int i = 0; i < n; i++) {
    char *rhs = productions[i] + 3;
    int len = strlen(rhs);

    for (int k = 0; k < len - 1; k++) {
        char a = rhs[k];
        char b = rhs[k + 1];

        if (!isNonTerminal(a) && !isNonTerminal(b)) {
            int row = findTerminalIndex(a);
            int col = findTerminalIndex(b);
            if (row != -1 && col != -1)
                opPrecedenceTable[row][col] = 2; // =
        }

        if (!isNonTerminal(a) && isNonTerminal(b)) {
            int row = findTerminalIndex(a);
            int B = findNonTerminalIndex(b);
            if (row != -1 && B != -1) {
                for (int t = 0; t < termCount; t++)
                    if (leading[B][t])
```

```
                opPrecedenceTable[row][t] = 1; // <
        }
    }

    if (isNonTerminal(a) && !isNonTerminal(b)) {
        int A = findNonTerminalIndex(a);
        int col = findTerminalIndex(b);
        if (A != -1 && col != -1) {
            for (int t = 0; t < termCount; t++)
                if (trailing[A][t])
                    opPrecedenceTable[t][col] = 3; // >
        }
    }

    if (k < len - 2 && !isNonTerminal(rhs[k]) &&
        isNonTerminal(rhs[k + 1]) && !isNonTerminal(rhs[k + 2])) {
        int row = findTerminalIndex(rhs[k]);     // left terminal
        int col = findTerminalIndex(rhs[k + 2]); // right terminal
        if (row != -1 && col != -1)
            opPrecedenceTable[row][col] = 2; // =
    }
  }
}
```

```
    if (!strchr(terminals, '$'))
        terminals[termCount++] = '$';

    int dollarIndex = findTerminalIndex('$');
    for (int i = 0; i < termCount; i++) {
        if (i == dollarIndex)
            continue;
        opPrecedenceTable[dollarIndex][i] = 1; // $ <
        opPrecedenceTable[i][dollarIndex] = 3; // > $
    }
}

void printSet(const char *setName, int sets[MAX][MAX], int idx) {
    printf("%s(%c) = { ", setName, nonTerminals[idx]);
    for (int i = 0; i < termCount; i++) {
        if (sets[idx][i])
            printf("%c ", terminals[i]);
    }
    printf("}\n");
}

void printOperatorPrecedenceTable() {
    printf("\nOperator Precedence Table:\n ");
    for (int i = 0; i < termCount; i++) {
```

```
        printf("%2c ", terminals[i]);
    }
    printf("\n");

    for (int i = 0; i < termCount; i++) {
        printf("%2c ", terminals[i]);
        for (int j = 0; j < termCount; j++) {
            char *rel = " ";
            switch (opPrecedenceTable[i][j]) {
                case 1: rel = "<"; break;
                case 2: rel = "="; break;
                case 3: rel = ">"; break;
            }
            printf(" %s ", rel);
        }
        printf("\n");
    }
}

int main() {
    printf("Enter number of productions: ");
    scanf("%d", &n);
    getchar();
```

```
printf("Enter productions (like E->E+T or E->T):\n");
for (int i = 0; i < n; i++) {
    fgets(productions[i], SIZE, stdin);
    productions[i][strcspn(productions[i], "\n")] = '\0';
}

collectNonTerminals();
collectTerminals();

printf("\nNonTerminals: ");
for (int i = 0; i < ntCount; i++)
    printf("%c ", nonTerminals[i]);

printf("\nTerminals: ");
for (int i = 0; i < termCount; i++)
    printf("%c ", terminals[i]);
printf("\n");

computeLeading();
computeTrailing();

printf("\nLeading sets:\n");
for (int i = 0; i < ntCount; i++)
    printSet("Leading", leading, i);
```

```
    printf("\nTrailing sets:\n");
    for (int i = 0; i < ntCount; i++)
        printSet("Trailing", trailing, i);
    buildOperatorPrecedenceTable();
    printOperatorPrecedenceTable();

    return 0;
}
```

Output:

```
Enter number of productions: 6
Enter productions (like E->E+T or E->T):
E->E+T
E->T
T->T*F
T->F
F->(E)
F->i

NonTerminals: E T F
Terminals: + * ( ) i

Leading sets:
Leading(E) = { + * ( i }
Leading(T) = { * ( i }
Leading(F) = { ( i }

Trailing sets:
Trailing(E) = { + * ) i }
Trailing(T) = { * ) i }
Trailing(F) = { ) i }
```

```
Operator Precedence Table:
  +  *  (  )  i  $
+ >  <  <  >  <  >
* >  >  <  >  <  >
( <  <  <  =  <  >
) >  >     >     >
i >  >     >     >
$ <  <  <  <  <
```

(b)

Aim: Use the Operator-precedence table in Experiment 8(a), perform the parsing for the given string.

Algorithm:

- Input Setup: Read 6x6 precedence table for symbols [+ * i ( ) $] with relations <, >, =, e, a
- Initialize: Set stack with '$', input pointer to start
- Parse Loop:
  - Compare stack top with current input symbol using precedence table
  - < or = → Shift: Push input symbol to stack, advance input
  - > → Reduce: Pop from stack
  - a → Accept: String accepted
  - e or invalid → Error: String rejected
- Display: Show stack, remaining input, and action at each step
- Result: Accept if 'a' relation found, otherwise reject

Source Code:

```
#include <stdio.h>

#include <string.h>

#include <stdlib.h>



#define SIZE 50

#define N 6
```

```
char symbols[] = { '+', '*', 'i', '(', ')', '$' };
char table[N][N];

int getIndex(char c) {
    for (int i = 0; i < N; i++) {
        if (symbols[i] == c)
            return i;
    }
    return -1;
}

int main() {
    char input[SIZE], stack[SIZE];
    int top = 0, i = 0;
    int accepted = 0;

    printf("Enter the operator precedence table (6x6) for symbols [+ * i ( ) $]:\n");
    printf("Enter one row at a time (use <, >, =, e, a):\n\n");

    for (int r = 0; r < N; r++) {
        for (int c = 0; c < N; c++) {
            scanf(" %c", &table[r][c]);
        }
    }
```

```
printf("\nEntered Operator Precedence Table:\n   ");
for (int c = 0; c < N; c++)
    printf("%c ", symbols[c]);
printf("\n");

for (int r = 0; r < N; r++) {
    printf("%c: ", symbols[r]);
    for (int c = 0; c < N; c++) {
        printf("%c ", table[r][c]);
    }
    printf("\n");
}

printf("\nEnter the input expression ending with $: ");
scanf("%s", input);

stack[top] = '$';
stack[top + 1] = '\0';

printf("\n%-20s %-20s %-10s\n", "Stack", "Input", "Action");
printf("-----------------------------------------------------\n");

while (1) {
```

```
int sIndex = getIndex(stack[top]);
int iIndex = getIndex(input[i]);

if (sIndex == -1 || iIndex == -1) {
   printf("Invalid symbol encountered!\n");
   accepted = 0;
   break;
}

char relation = table[sIndex][iIndex];

printf("%-20s %-20s ", stack, input + i);

if (relation == '<' || relation == '=') {
   stack[++top] = input[i++];
   stack[top + 1] = '\0';
   printf("Shift\n");
}
else if (relation == '>') {
   printf("Reduce\n");
   stack[top] = '\0';
   top--;
}
else if (relation == 'a') {
```

```
            printf("Accept\n");
            accepted = 1;
            break;
        }
        else {
            printf("Error\n");
            accepted = 0;
            break;
        }
    }

    if (accepted)
        printf("\nResult: The string is ACCEPTED\n");
    else
        printf("\nResult: The string is REJECTED\n");

    return 0;
}
```

Output:

```
Enter the operator precedence table (6x6) for symbols [+ * i ( ) $]:
Enter one row at a time (use <, >, =, e, a):

> < < < > >
> > < < > >
> > e e > >
< < < < = e
> > e e > >
< < < < e a

Entered Operator Precedence Table:
   + * i ( ) $
+: > < < < > >
*: > > < < > >
i: > > e e > >
(: < < < < = e
): > > e e > >
$: < < < < e a

Enter the input expression ending with $: i+i*i$
```

```
Stack                    Input                    Action
-------------------------------------------------------------
$                        i+i*i$                   Shift
$i                       +i*i$                    Reduce
$                        +i*i$                    Shift
$+                       i*i$                     Shift
$+i                      *i$                      Reduce
$+                       *i$                      Shift
$+*                      i$                       Shift
$+*i                     $                        Reduce
$+*                      $                        Reduce
$+                       $                        Reduce
$                        $                        Accept

Result: The string is ACCEPTED
```

EXPERIMENT 9

(a)

Aim: Construct Simple LR (SLR) table using C language.

Algorithm:

- Input Grammar from user
- Augment Grammar
- Compute FIRST and FOLLOW
- Compute canonical collection of LR(0) items (states)
- Show states + transitions
- Build and display SLR parsing table

Source Code:

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>



#define MAX_RULES 20

#define MAX_LEN 30

#define MAX_STATES 50

#define MAX_ITEMS 100

#define MAX_SYMBOLS 20



typedef struct {

    char lhs;

    char rhs[MAX_LEN];
```

```
} Production;

typedef struct {
    int prodIndex;
    int dotPos;
} Item;

typedef struct {
    Item items[MAX_ITEMS];
    int count;
} ItemSet;

Production prods[MAX_RULES];
int nProds;
char nonTerminals[MAX_SYMBOLS];
char terminals[MAX_SYMBOLS];
int nNonTerms=0, nTerms=0;

ItemSet states[MAX_STATES];
int nStates=0;

int transitions[MAX_STATES][MAX_SYMBOLS];

char firstSets[MAX_SYMBOLS][MAX_SYMBOLS];
```

```
char followSets[MAX_SYMBOLS][MAX_SYMBOLS];


int isNonTerminal(char c){
    return isupper(c);
}
int symbolIndex(char c){
    for(int i=0;i<nNonTerms+nTerms;i++){
        if(i<nNonTerms && nonTerminals[i]==c) return i;
        if(i>=nNonTerms && terminals[i-nNonTerms]==c) return i;
    }
    return -1;
}
void addSymbol(char *set, char c){
    for(int i=0;set[i];i++) if(set[i]==c) return;
    int len=strlen(set);
    set[len]=c; set[len+1]='\0';
}


void computeFirst(){
    for(int i=0;i<nNonTerms;i++) firstSets[i][0]='\0';
    for(int i=0;i<nNonTerms;i++){
        for(int j=0;j<nProds;j++){
            if(prods[j].lhs==nonTerminals[i]){
                char c=prods[j].rhs[0];
```

```
            if(isNonTerminal(c)){
                addSymbol(firstSets[i], firstSets[symbolIndex(c)][0]);
            } else {
                addSymbol(firstSets[i], c);
            }
        }
    }
  }
}

void computeFollow(){
  for(int i=0;i<nNonTerms;i++) followSets[i][0]='\0';
  addSymbol(followSets[0], '$');
  for(int j=0;j<nProds;j++){
    char *rhs=prods[j].rhs;
    int len=strlen(rhs);
    for(int k=0;k<len;k++){
      if(isNonTerminal(rhs[k])){
        if(k+1<len){
          if(isNonTerminal(rhs[k+1])){
            int idx=symbolIndex(rhs[k]);
            int idx2=symbolIndex(rhs[k+1]);
            for(int x=0;followSets[idx2][x];x++)
              addSymbol(followSets[idx], followSets[idx2][x]);
```

```
                } else {
                    addSymbol(followSets[symbolIndex(rhs[k])], rhs[k+1]);
                }
            } else {
                int idx=symbolIndex(rhs[k]);
                for(int x=0;followSets[symbolIndex(prods[j].lhs)][x];x++)
                    addSymbol(followSets[idx], followSets[symbolIndex(prods[j].lhs)][x]);
            }
        }
    }
  }
}

ItemSet closure(ItemSet I){
    ItemSet J=I;
    int changed=1;
    while(changed){
        changed=0;
        for(int i=0;i<J.count;i++){
            Item it=J.items[i];
            char *rhs=prods[it.prodIndex].rhs;
            if(it.dotPos<strlen(rhs)){
                char B=rhs[it.dotPos];
                if(isNonTerminal(B)){
```

```
                for(int p=0;p<nProds;p++){
                    if(prods[p].lhs==B){
                        int exists=0;
                        for(int q=0;q<J.count;q++){
                            if(J.items[q].prodIndex==p && J.items[q].dotPos==0){ exists=1; break; }
                        }
                        if(!exists){
                            J.items[J.count].prodIndex=p;
                            J.items[J.count].dotPos=0;
                            J.count++;
                            changed=1;
                        }
                    }
                }
            }
        }
    }
    return J;
}

ItemSet goTo(ItemSet I,char X){
    ItemSet J; J.count=0;
    for(int i=0;i<I.count;i++){
```

```
        Item it=I.items[i];
        char *rhs=prods[it.prodIndex].rhs;
        if(it.dotPos<strlen(rhs) && rhs[it.dotPos]==X){
            J.items[J.count].prodIndex=it.prodIndex;
            J.items[J.count].dotPos=it.dotPos+1;
            J.count++;
        }
    }
    return closure(J);
}

int equalSets(ItemSet A, ItemSet B){
    if(A.count!=B.count) return 0;
    for(int i=0;i<A.count;i++){
        int found=0;
        for(int j=0;j<B.count;j++){
            if(A.items[i].prodIndex==B.items[j].prodIndex && A.items[i].dotPos==B.items[j].dotPos){
                found=1; break;
            }
        }
        if(!found) return 0;
    }
    return 1;
}
```

```
void buildStates(){
   ItemSet I0;
   I0.count=1;
   I0.items[0].prodIndex=0;
   I0.items[0].dotPos=0;
   I0=closure(I0);
   states[0]=I0; nStates=1;

   for(int i=0;i<nStates;i++){
      for(int s=0;s<nNonTerms+nTerms;s++){
         char X=(s<nNonTerms)?nonTerminals[s]:terminals[s-nNonTerms];
         ItemSet J=goTo(states[i],X);
         if(J.count==0) continue;
         int exists=-1;
         for(int k=0;k<nStates;k++){
            if(equalSets(states[k],J)){exists=k; break;}
         }
         if(exists==-1){
            states[nStates]=J;
            transitions[i][s]=nStates;
            nStates++;
         } else {
            transitions[i][s]=exists;
```

```
        }
      }
    }
  }

void printStates(){
    for(int i=0;i<nStates;i++){
        printf("\nI%d:\n",i);
        for(int j=0;j<states[i].count;j++){
            Item it=states[i].items[j];
            char *rhs=prods[it.prodIndex].rhs;
            printf("  %c -> ", prods[it.prodIndex].lhs);
            for(int k=0;k<strlen(rhs);k++){
                if(k==it.dotPos) printf(".");
                printf("%c",rhs[k]);
            }
            if(it.dotPos==strlen(rhs)) printf(".");
            printf("\n");
        }
    }
}


void buildParsingTable(){
```

```
printf("\nSLR Parsing Table:\n");

printf("S\t");

for(int i=0;i<nTerms;i++) printf("%c\t",terminals[i]);

printf("$\t");

for(int i=0;i<nNonTerms;i++) printf("%c\t",nonTerminals[i]);

printf("\n");


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

  printf("%d\t",i);

  for(int t=0;t<nTerms+1;t++){

    char a=(t<nTerms)?terminals[t]:'$';

    char entry[10]="-";

    for(int j=0;j<states[i].count;j++){

      Item it=states[i].items[j];

      char *rhs=prods[it.prodIndex].rhs;

      if(it.dotPos<strlen(rhs) && rhs[it.dotPos]==a){

        sprintf(entry,"s%d", transitions[i][symbolIndex(a)]);

      }

      else if(it.dotPos==strlen(rhs)){

        if(prods[it.prodIndex].lhs=='S' && it.prodIndex==0 &&
a == '$'){

          strcpy(entry,"A");

        } else {

          int lhsIdx=symbolIndex(prods[it.prodIndex].lhs);
```

```
                    for(int f=0;followSets[lhsIdx][f];f++){
                        if(followSets[lhsIdx][f]==a)
                            sprintf(entry,"r%d", it.prodIndex);
                    }
                }
            }
        }
        printf("%s\t",entry);
    }
    for(int nt=0;nt<nNonTerms;nt++){
        if(transitions[i][nt]!=0) printf("%d\t",transitions[i][nt]);
        else printf("-\t");
    }
    printf("\n");
  }
}
int main(){
    printf("Enter number of productions: ");
    scanf("%d",&nProds);
    printf("Enter productions (e.g., E->E+T):\n");
    for(int i=0;i<nProds;i++){
        char input[50];
        scanf("%s",input);
        prods[i].lhs=input[0];
```

```
        strcpy(prods[i].rhs,input+3);
        if(strchr(nonTerminals,prods[i].lhs)==NULL){
            nonTerminals[nNonTerms++]=prods[i].lhs;
        }
        for(int j=0;j<strlen(prods[i].rhs);j++){
            if(!isNonTerminal(prods[i].rhs[j]) && strchr(terminals,prods[i].rhs[j])==NULL){
                terminals[nTerms++]=prods[i].rhs[j];
            }
        }
    }
    for(int i=nProds;i>0;i--) prods[i]=prods[i-1];
    prods[0].lhs='S'; sprintf(prods[0].rhs,"%c",prods[1].lhs);
    nProds++;


    computeFirst();
    computeFollow();
    buildStates();
    printStates();
    buildParsingTable();
    return 0;
}
```

OUTPUT:

```
Enter number of productions: 6
Enter productions (e.g., E->E+T):
E->E+T
E->T
T->T*F
T->F
F->(E)
F->i

I0:
  S -> .E
  E -> .E+T
  E -> .T
  T -> .T*F
  T -> .F
  F -> .(E)
  F -> .i

I1:
  S -> E.
  E -> E.+T

I2:
  E -> T.
  T -> T.*F

I3:
  T -> F.
```

```
I4:
  F -> (.E)
  E -> .E+T
  E -> .T
  T -> .T*F
  T -> .F
  F -> .(E)
  F -> .i

I5:
  F -> i.

I6:
  E -> E+.T
  T -> .T*F
  T -> .F
  F -> .(E)
  F -> .i

I7:
  T -> T*.F
  F -> .(E)
  F -> .i

I8:
  F -> (E.)
  E -> E.+T
```

```
I9:
  E -> E+T.
  T -> T.*F

I10:
  T -> T*F.

I11:
  F -> (E).
```

```
SLR Parsing Table:
S   +   *   (   )   i   $   E   T   F
0   -   -   s4  -   s5  -   1   2   3
1   s6  -   -   -   -   A   -   -   -
2   r2  s7  -   r2  -   r2  -   -   -
3   r4  r4  -   -   -   r4  -   -   -
4   -   -   s4  -   s5  -   8   2   3
5   r6  r6  -   -   -   r6  -   -   -
6   -   -   s4  -   s5  -   -   9   3
7   -   -   s4  -   s5  -   -   -   10
8   s6  -   -   s11 -   -   -   -   -
9   r1  s7  -   r1  -   r1  -   -   -
10  r3  r3  -   -   -   r3  -   -   -
11  r5  r5  -   -   -   r5  -   -   -
```

(b)

Aim: Implement the SLR parsing algorithm, get parse table and input string are inputs. Use C language for implementation.

Algorithm:

- Input Grammar and SLR Table
- Initialize Parsing Stack
- Parsing Loop (Shift/Reduce/Accept/Error)
- Print Parsing Trace
- End of Parsing

Source Code:

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>



#define MAX_PRODS 50

#define MAX_LEN 100

#define MAX_STATES 50

#define MAX_SYMBOLS 50



typedef struct {

   char lhs;

   char rhs[MAX_LEN];

} Production;



Production prods[MAX_PRODS];

int nProds;
```

```c
int nStates, nTerm, nNonTerm;

char actionType[MAX_STATES][MAX_SYMBOLS];
int  actionVal[MAX_STATES][MAX_SYMBOLS];

int goToTable[MAX_STATES][MAX_SYMBOLS];

char terminals[MAX_SYMBOLS];
char nonTerminals[MAX_SYMBOLS];

int termIndex(char c){
   for(int i=0;i<nTerm;i++) if(terminals[i]==c) return i;
   if(c=='$') return nTerm;
   return -1;
}

int nonTermIndex(char c){
   for(int i=0;i<nNonTerm;i++) if(nonTerminals[i]==c) return i;
   return -1;
}

void parseString(char *input){
   int stack[100]; int top=0;
```

```
stack[top]=0;
int ip=0;

printf("\n%-20s %-20s %-20s\n","Stack","Input","Action");

while(1){
    char stkStr[100] = "";
    for(int i=0;i<=top;i++){
        char tmp[5];
        sprintf(tmp,"%d ",stack[i]);
        strcat(stkStr,tmp);
    }

    char *inpStr = input+ip;

    char a = input[ip];
    int col = termIndex(a);
    if(col==-1){ printf("Invalid symbol %c\n",a); return; }
    int state = stack[top];
    char act = actionType[state][col];
    int val = actionVal[state][col];
    char actionDesc[50];

    if(act=='s') sprintf(actionDesc,"Shift %d",val);
```

```
        else if(act=='r') sprintf(actionDesc,"Reduce by %c->%s",prods[val-1].lhs,prods[val-1].rhs);

        else if(act=='A') sprintf(actionDesc,"ACCEPT");

        else sprintf(actionDesc,"ERROR");


        printf("%-20s %-20s %-20s\n",stkStr,inpStr,actionDesc);


        if(act=='s'){

            top++; stack[top]=val; ip++;

        }

        else if(act=='r'){

            Production p = prods[val-1];

            int len = strlen(p.rhs);

            if(p.rhs[0]=='#') len=0;

            top -= len;

            if(top<0){ printf("Stack underflow\n"); return; }

            int st = stack[top];

            int A = nonTermIndex(p.lhs);

            if(A==-1 || goToTable[st][A]==-1){ printf("Invalid GOTO\n"); return; }

            top++; stack[top]=goToTable[st][A];

        }

        else if(act=='A') break;

        else break;

    }

}
```

```
int main(){
  printf("Enter number of productions: ");
  scanf("%d",&nProds);
  for(int i=0;i<nProds;i++){
    char buf[MAX_LEN];
    scanf("%s",buf);
    prods[i].lhs = buf[0];
    strcpy(prods[i].rhs, buf+3);
    int len = strlen(prods[i].rhs);
    if(prods[i].rhs[len-1]=='\r' || prods[i].rhs[len-1]=='\n') prods[i].rhs[len-1]='\0';
  }

  printf("Enter number of terminals (without $): ");
  scanf("%d",&nTerm);
  for(int i=0;i<nTerm;i++){
    char tmp;
    scanf(" %c",&tmp);
    terminals[i] = tmp;
  }
  terminals[nTerm]='$';


  printf("Enter number of nonterminals: ");
```

```
scanf("%d",&nNonTerm);
for(int i=0;i<nNonTerm;i++){
   char tmp;
   scanf(" %c",&tmp);
   nonTerminals[i] = tmp;
}

printf("Enter number of states: ");
scanf("%d",&nStates);

printf("Enter ACTION table (rows=states, cols=terminals+$):\n");
for(int i=0;i<nStates;i++){
   for(int j=0;j<=nTerm;j++){
      char buf[10]; scanf("%s",buf);
      if(buf[0]=='s'){ actionType[i][j]='s'; actionVal[i][j]=atoi(buf+1); }
      else if(buf[0]=='r'){ actionType[i][j]='r'; actionVal[i][j]=atoi(buf+1); }
      else if(buf[0]=='A'){ actionType[i][j]='A'; }
      else { actionType[i][j]='-'; }
   }
}

printf("Enter GOTO table (rows=states, cols=nonterminals), use '-' for empty:\n");
for(int i=0;i<nStates;i++){
   for(int j=0;j<nNonTerm;j++){
```

```
            char buf[10]; scanf("%s", buf);
            if(buf[0]=='-') goToTable[i][j] = -1;
            else goToTable[i][j] = atoi(buf);
        }
    }

    char inp[MAX_LEN];
    printf("Enter input string (end with $): ");
    scanf("%s",inp);

    parseString(inp);

    return 0;
}
```

Output

```
Enter number of productions: 6
E->E+T
E->T
T->T*F
T->F
F->(E)
F->i
Enter number of terminals (without $): 5
+ * ( ) i
Enter number of nonterminals: 3
E T F
Enter number of states: 12
Enter ACTION table (rows=states, cols=terminals+$):
- - s4 - s5 -
s6 - - - - A
r2 s7 - r2 - r2
r4 r4 - - - r4
- - s4 - s5 -
r6 r6 - - - r6
- - s4 - s5 -
- - s4 - s5 -
s6 - - s11 - -
r1 s7 - r1 - r1
r3 r3 - - - r3
r5 r5 - - - r5
```

```
Enter GOTO table (rows=states, cols=nonterminals), use '-' for empty:
1 2 3
- - -
- - -
- - -
8 2 3
- - -
- 9 3
- - 10
- - -
- - -
- - -
- - -
Enter input string (end with $): i+i*i$

Stack               Input               Action
0                   i+i*i$              Shift 5
0 5                 +i*i$               Reduce by F->i
0 3                 +i*i$               Reduce by T->F
0 2                 +i*i$               Reduce by E->T
0 1                 +i*i$               Shift 6
0 1 6               i*i$                Shift 5
0 1 6 5             *i$                 Reduce by F->i
0 1 6 3             *i$                 Reduce by T->F
0 1 6 9             *i$                 Shift 7
0 1 6 9 7           i$                  Shift 5
0 1 6 9 7 5         $                   Reduce by F->i
0 1 6 9 7 10        $                   Reduce by T->T*F
0 1 6 9             $                   Reduce by E->E+T
0 1                 $                   ACCEPT
```

EXPERIMENT 10

(a)

Aim: Construct Canonical LR (CLR) table using C language.

Algorithm:

- Input Grammar
- Compute First Sets
- Initialize LR(1) States
- Closure Operation
- Goto Operation
- Construct Canonical Collection of LR(1) States
- Print LR(1) Item Sets
- Build CLR(1) Parsing Table
- Output CLR(1) Parsing Table

Source Code:

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>



#define MAX_PRODS 20

#define MAX_LEN 30

#define MAX_STATES 50

#define MAX_ITEMS 200

#define MAX_SYMBOLS 20



typedef struct {

    char lhs;
```

```
    char rhs[MAX_LEN];
} Production;

typedef struct {
    int prodIndex;
    int dotPos;
    char lookahead[MAX_SYMBOLS];
} ItemLR1;

typedef struct {
    ItemLR1 items[MAX_ITEMS];
    int count;
} ItemSetLR1;

Production prods[MAX_PRODS];
int nProds;

char terminals[MAX_SYMBOLS];
int nTerms=0;
char nonTerminals[MAX_SYMBOLS];
int nNonTerms=0;

ItemSetLR1 states[MAX_STATES];
int nStates=0;
```

```
int transitions[MAX_STATES][MAX_SYMBOLS];

int isNonTerminal(char c){ return isupper(c); }

int contains(char *set, char c){
   for(int i=0;i<strlen(set);i++) if(set[i]==c) return 1;
   return 0;
}

void addSymbol(char *set, char c){
   if(!contains(set,c)){
      int len=strlen(set);
      set[len]=c;
      set[len+1]='\0';
   }
}

int symbolIndex(char c){
   for(int i=0;i<nNonTerms+nTerms;i++){
      if(i<nNonTerms && nonTerminals[i]==c) return i;
      if(i>=nNonTerms && terminals[i-nNonTerms]==c) return i;
   }
   return -1;
```

```
}

char firstSets[MAX_SYMBOLS][MAX_SYMBOLS];
int firstCount[MAX_SYMBOLS];

int idxNonTerm(char c){
  for(int i=0;i<nNonTerms;i++) if(nonTerminals[i]==c) return i;
  return -1;
}

void computeFirstSets(){
  for(int i=0;i<nNonTerms;i++){
    firstSets[i][0]='\0';
    firstCount[i]=0;
  }

  int changed=1;
  while(changed){
    changed=0;
    for(int i=0;i<nProds;i++){
      char A = prods[i].lhs;
      char *rhs = prods[i].rhs;
      int idxA = idxNonTerm(A);
      char old[MAX_SYMBOLS]; strcpy(old, firstSets[idxA]);
```

```
            if(isNonTerminal(rhs[0])){
                int idxB = idxNonTerm(rhs[0]);
                for(int k=0;k<strlen(firstSets[idxB]);k++)
                    addSymbol(firstSets[idxA], firstSets[idxB][k]);
            } else addSymbol(firstSets[idxA], rhs[0]);
            if(strcmp(old, firstSets[idxA])!=0) changed=1;
        }
    }

    printf("\n--- First Sets ---\n");
    for(int i=0;i<nNonTerms;i++){
        printf("First(%c) = { ", nonTerminals[i]);
        for(int j=0;j<strlen(firstSets[i]);j++) printf("%c ", firstSets[i][j]);
        printf("}\n");
    }
}

ItemSetLR1 closure(ItemSetLR1 I){
    ItemSetLR1 J=I;
    int changed=1;
    while(changed){
        changed=0;
        for(int i=0;i<J.count;i++){
            ItemLR1 it=J.items[i];
```

```
char *rhs=prods[it.prodIndex].rhs;
if(it.dotPos<strlen(rhs)){
   char B=rhs[it.dotPos];
   if(isNonTerminal(B)){
      for(int p=0;p<nProds;p++){
         if(prods[p].lhs==B){
            char newLA[MAX_SYMBOLS]="";

            if(it.dotPos+1 < strlen(rhs)){
               char beta = rhs[it.dotPos+1];
               if(isNonTerminal(beta)){
                  int idxB = idxNonTerm(beta);
                  for(int k=0;k<strlen(firstSets[idxB]);k++)
                     addSymbol(newLA, firstSets[idxB][k]);
               } else addSymbol(newLA, beta);
            } else strcpy(newLA, it.lookahead);

            int exists=0;
            for(int q=0;q<J.count;q++){
               if(J.items[q].prodIndex==p && J.items[q].dotPos==0){
                  for(int k=0;k<strlen(newLA);k++)
                     addSymbol(J.items[q].lookahead,newLA[k]);
                  exists=1;
               }
```

```
                    }
                    if(!exists){
                        J.items[J.count].prodIndex=p;
                        J.items[J.count].dotPos=0;
                        strcpy(J.items[J.count].lookahead,newLA);
                        J.count++;
                        changed=1;
                    }
                }
            }
        }
    }
    }
    }
    return J;
}

ItemSetLR1 goTo(ItemSetLR1 I,char X){
    ItemSetLR1 J; J.count=0;
    for(int i=0;i<I.count;i++){
        ItemLR1 it=I.items[i];
        char *rhs=prods[it.prodIndex].rhs;
        if(it.dotPos<strlen(rhs) && rhs[it.dotPos]==X){
            J.items[J.count].prodIndex=it.prodIndex;
```

```
            J.items[J.count].dotPos=it.dotPos+1;
            strcpy(J.items[J.count].lookahead,it.lookahead);
            J.count++;
        }
    }
    return closure(J);
}

int equalSets(ItemSetLR1 A, ItemSetLR1 B){
    if(A.count!=B.count) return 0;
    for(int i=0;i<A.count;i++){
        int found=0;
        for(int j=0;j<B.count;j++){
            if(A.items[i].prodIndex==B.items[j].prodIndex && A.items[i].dotPos==B.items[j].dotPos){
                int k, match=1;
                for(k=0;k<strlen(A.items[i].lookahead);k++)
                    if(!contains(B.items[j].lookahead,A.items[i].lookahead[k])) match=0;
                if(match){found=1; break;}
            }
        }
        if(!found) return 0;
    }
    return 1;
}
```

```
void printItemSet(ItemSetLR1 I, int stateNum){
    printf("\nState I%d:\n", stateNum);
    for(int i=0;i<I.count;i++){
        ItemLR1 it = I.items[i];
        char *rhs = prods[it.prodIndex].rhs;
        printf("%c -> ", prods[it.prodIndex].lhs);
        for(int j=0;j<strlen(rhs);j++){
            if(j==it.dotPos) printf(".");
            printf("%c", rhs[j]);
        }
        if(it.dotPos == strlen(rhs)) printf(".");
        printf(", lookahead: {");
        for(int k=0;k<strlen(it.lookahead);k++) printf("%c", it.lookahead[k]);
        printf("}\n");
    }
}

void buildStates(){
    ItemSetLR1 I0;
    I0.count=1;
    I0.items[0].prodIndex=0;
    I0.items[0].dotPos=0;
    I0.items[0].lookahead[0]='$'; I0.items[0].lookahead[1]='\0';
```

```
    states[0]=closure(I0);
    nStates=1;
    printItemSet(states[0], 0);

    for(int i=0;i<nStates;i++){
        for(int s=0;s<nNonTerms+nTerms;s++){
            char X=(s<nNonTerms)?nonTerminals[s]:terminals[s-nNonTerms];
            ItemSetLR1 J=goTo(states[i],X);
            if(J.count==0) continue;
            int exists=-1;
            for(int k=0;k<nStates;k++)
                if(equalSets(states[k],J)){ exists=k; break; }
            if(exists==-1){
                states[nStates]=J;
                transitions[i][s]=nStates;
                printItemSet(J, nStates);
                nStates++;
            } else transitions[i][s]=exists;
        }
    }
}

void printCLRTable(){
    printf("\nCLR Parsing Table:\n");
```

```
printf("S\t");
for(int i=0;i<nTerms;i++) printf("%c\t",terminals[i]);
printf("$\t");
for(int i=0;i<nNonTerms;i++) printf("%c\t",nonTerminals[i]);
printf("\n");

for(int i=0;i<nStates;i++){
  printf("%d\t",i);
  for(int t=0;t<nTerms;t++){
    char a=terminals[t];
    char entry[10]="-";
    if(transitions[i][symbolIndex(a)] != -1){
      sprintf(entry,"s%d",transitions[i][symbolIndex(a)]);
    } else {
      for(int j=0;j<states[i].count;j++){
        ItemLR1 it=states[i].items[j];
        char *rhs=prods[it.prodIndex].rhs;
        if(it.dotPos==strlen(rhs) && contains(it.lookahead,a)){
          if(it.prodIndex==0 && a=='$') strcpy(entry,"A");
          else sprintf(entry,"r%d",it.prodIndex);
        }
      }
    }
    printf("%s\t",entry);
```

```
        }

        char entry[10]="-";
        for(int j=0;j<states[i].count;j++){
            ItemLR1 it=states[i].items[j];
            char *rhs=prods[it.prodIndex].rhs;
            if(it.dotPos==strlen(rhs) && contains(it.lookahead,'$')){
                if(it.prodIndex==0) strcpy(entry,"A");
                else sprintf(entry,"r%d",it.prodIndex);
            }
        }
        printf("%s\t",entry);

        for(int nt=0;nt<nNonTerms;nt++){
            int idx=transitions[i][nt];
            if(idx!=-1) printf("%d\t",idx);
            else printf("-\t");
        }
        printf("\n");
    }
}

int main(){
    printf("Enter number of productions: ");
```

```
scanf("%d",&nProds);
printf("Enter productions (e.g., A->CC):\n");
for(int i=0;i<nProds;i++){
    char buf[50]; scanf("%s",buf);
    prods[i].lhs=buf[0];
    strncpy(prods[i].rhs,buf+3,MAX_LEN-1);
    prods[i].rhs[MAX_LEN-1]='\0';
    if(!contains(nonTerminals,prods[i].lhs)) nonTerminals[nNonTerms++]=prods[i].lhs;
    for(int j=0;j<strlen(prods[i].rhs);j++)
        if(!isNonTerminal(prods[i].rhs[j]) && !contains(terminals,prods[i].rhs[j]))
            terminals[nTerms++]=prods[i].rhs[j];
}

for(int i=nProds;i>0;i--) prods[i]=prods[i-1];
prods[0].lhs='S'; sprintf(prods[0].rhs,"%c",prods[1].lhs);
nProds++;
for(int i=0;i<MAX_STATES;i++)
    for(int j=0;j<MAX_SYMBOLS;j++)
        transitions[i][j]=-1;

computeFirstSets();
buildStates();
printCLRTable();
```

```
    return 0;

}
```

Output:

```
Enter number of productions: 3
Enter productions (e.g., A->CC):
A->CC
C->cC
C->d

--- First Sets ---
First(A) = { c d }
First(C) = { c d }

State I0:
S -> .A, lookahead: {$}
A -> .CC, lookahead: {$}
C -> .cC, lookahead: {cd}
C -> .d, lookahead: {cd}

State I1:
S -> A., lookahead: {$}

State I2:
A -> C.C, lookahead: {$}
C -> .cC, lookahead: {$}
C -> .d, lookahead: {$}
```

```
State I3:
C -> c.C, lookahead: {cd}
C -> .cC, lookahead: {cd}
C -> .d, lookahead: {cd}

State I4:
C -> d., lookahead: {cd}

State I5:
A -> CC., lookahead: {$}

State I6:
C -> c.C, lookahead: {$}
C -> .cC, lookahead: {$}
C -> .d, lookahead: {$}

State I7:
C -> d., lookahead: {$}

State I8:
C -> cC., lookahead: {cd}

State I9:
C -> cC., lookahead: {$}
```

```
CLR Parsing Table:
S   c   d   $   A   C
0   s3  s4  -   1   2
1   -   -   A   -   -
2   s6  s7  -   -   5
3   s3  s4  -   -   8
4   r3  r3  -   -   -
5   -   -   r1  -   -
6   s6  s7  -   -   9
7   -   -   r3  -   -
8   r2  r2  -   -   -
9   -   -   r2  -   -
```

(b)

Aim: Implement the CLR parsing algorithm, get parse table and input string are inputs. Use C language for implementation.

Algorithm:

- Input Grammar and Productions
- Input Terminals
- Input Non-terminals
- Augment Grammar (Add S' → S)
- Input Number of States
- Input Action Table
- Input GOTO Table
- Input String to Parse
- Initialize Stack with State 0
- Set Input Pointer to First Symbol
- Loop Until Accept or Error:
- Print Stack, Input, and Action at Each Step

Source Code:

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>



#define MAX_PRODS 20

#define MAX_TERMS 20

#define MAX_NT 20

#define MAX_STATES 50

#define MAX_INPUT 50

#define MAX_ACTION_LEN 10



typedef struct {

    char lhs;
```

```
    char rhs[50];

} Production;


Production prods[MAX_PRODS];

int nProds;


char terminals[MAX_TERMS];

int nTerms;

char nonTerminals[MAX_NT];

int nNonTerms;


char action[MAX_STATES][MAX_TERMS+1][MAX_ACTION_LEN];

int gotoTable[MAX_STATES][MAX_NT];


int termIndex(char c) {

    if(c=='$') return nTerms;

    for(int i=0;i<nTerms;i++) if(terminals[i]==c) return i;

    return -1;

}


int nonTermIndex(char c) {

    for(int i=0;i<nNonTerms;i++) if(nonTerminals[i]==c) return i;

    return -1;

}
```

```
void parseInput(char *input) {
    int stack[MAX_INPUT], top=0;
    char symStack[MAX_INPUT];
    stack[0]=0;
    symStack[0]='$';
    int ip=0;
    int lenInput = strlen(input);

    printf("\n%-15s%-15s%-25s\n", "Stack", "Input", "Action");
    printf("-------------------------------------------------------\n");

    while(1){
        int state = stack[top];
        char curr = (ip<lenInput)?input[ip]:'$';
        int tIndex = termIndex(curr);

        char stackStr[100]="";
        for(int i=0;i<=top;i++){
            char buf[5];
            sprintf(buf, "%d ", stack[i]);
            strcat(stackStr, buf);
        }
```

```
printf("%-15s%-15s", stackStr, &input[ip]);

if(strcmp(action[state][tIndex],"-")==0){
    printf("%-25s\n", "Error!");
    return;
}

if(action[state][tIndex][0]=='s'){
    int s = atoi(&action[state][tIndex][1]);
    printf("%-25s\n", action[state][tIndex]);
    top++;
    stack[top]=s;
    symStack[top]=curr;
    ip++;
}
else if(action[state][tIndex][0]=='r'){
    int r = atoi(&action[state][tIndex][1]);
    int len = strlen(prods[r].rhs);
    char actStr[25];
    sprintf(actStr, "Reduce by %c->%s", prods[r].lhs, prods[r].rhs);
    printf("%-25s\n", actStr);
    top -= len;
    char lhs = prods[r].lhs;
    int g = gotoTable[stack[top]][nonTermIndex(lhs)];
```

```
        if(g==-1){
            printf("%-25s\n", "Error! No GOTO");
            return;
        }
        top++;
        stack[top]=g;
        symStack[top]=lhs;
    }
    else if(action[state][tIndex][0]=='A'){
        printf("%-25s\n", "Accept");
        return;
    }
  }
}

int main(){
  printf("Enter number of productions: ");
  scanf("%d",&nProds);
  for(int i=0;i<nProds;i++){
    char buf[50];
    scanf("%s",buf);
    prods[i].lhs = buf[0];
    strcpy(prods[i].rhs, buf+3);
  }
```

```
printf("Enter number of terminals: ");

scanf("%d",&nTerms);

for(int i=0;i<nTerms;i++) scanf(" %c",&terminals[i]);


printf("Enter number of non-terminals: ");

scanf("%d",&nNonTerms);

for(int i=0;i<nNonTerms;i++) scanf(" %c",&nonTerminals[i]);


for(int i=nProds;i>0;i--) prods[i]=prods[i-1];

prods[0].lhs='S';

sprintf(prods[0].rhs,"%c",prods[1].lhs);

nProds++;


int nStates;

printf("Enter number of states: ");

scanf("%d",&nStates);


printf("Enter Action Table (rows = states, columns = terminals + $):\n");

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

   for(int j=0;j<=nTerms;j++) scanf("%s", action[i][j]);

}


printf("Enter GOTO Table (rows = states, columns = non-terminals, use - for no transition):\n");
```

```
    for(int i=0;i<nStates;i++){
        for(int j=0;j<nNonTerms;j++){
            char buf[10];
            scanf("%s",buf);
            if(buf[0]=='-') gotoTable[i][j]=-1;
            else gotoTable[i][j]=atoi(buf);
        }
    }

    char input[50];
    printf("Enter input string: ");
    scanf("%s", input);

    parseInput(input);

    return 0;
}
```

Output:

```
Enter number of productions: 3
A->CC
C->cC
C->d
Enter number of terminals: 2
c d
Enter number of non-terminals: 2
A C
Enter number of states: 10
Enter Action Table (rows = states, columns = terminals + $):
s3 s4 -
- - A
s6 s7 -
s3 s4 -
r3 r3 -
- - r1
s6 s7 -
- - r3
r2 r2 -
- - r2
Enter GOTO Table (rows = states, columns = non-terminals, use - for no transition):
1 2
- -
- 5
- 8
- -
- -
- 9
- -
- -
- -
```

```
Enter input string: cdcd

Stack          Input          Action
-------------------------------------------------------
0              cdcd           s3
0 3            dcd            s4
0 3 4          cd             Reduce by C->d
0 3 8          cd             Reduce by C->cC
0 2            cd             s6
0 2 6          d              s7
0 2 6 7                       Reduce by C->d
0 2 6 9                       Reduce by C->cC
0 2 5                         Reduce by A->CC
0 1                           Accept
```

EXPERIMENT 11

(a)

Aim: Construct Look-Ahead LR (LALR) table using C language.

Algorithm:

- Augment Grammar
- Compute FIRST Sets
- Build LR(1) Items
- Closure Operation
- GOTO Function
- Construct LR(1) States
- Build LR(1) Transitions
- Extract Cores
- Identify Same Cores
- Merge Lookaheads
- Create LALR States
- Update LALR Transitions
- Build ACTION Table
- Build GOTO Table
- Handle Conflicts
- Display LALR States
- Show Merge Information
- Generate Parsing Table

Source Code:

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>



#define MAX_PRODS 20

#define MAX_LEN 30

#define MAX_STATES 100

#define MAX_ITEMS 200
```

```
#define MAX_SYMBOLS 20

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

typedef struct {
    int prodIndex;
    int dotPos;
    char lookahead[MAX_SYMBOLS];
} ItemLR1;

typedef struct {
    ItemLR1 items[MAX_ITEMS];
    int count;
} ItemSetLR1;

typedef struct {
    int prodIndex;
    int dotPos;
} ItemLR0;

typedef struct {
```

```
    ItemLR0 items[MAX_ITEMS];
    int count;
} CoreSet;

typedef struct {
    int lalrState;
    int originalStates[MAX_STATES];
    int numOriginal;
} MergeInfo;

Production prods[MAX_PRODS];
int nProds;
char terminals[MAX_SYMBOLS];
int nTerms = 0;
char nonTerminals[MAX_SYMBOLS];
int nNonTerms = 0;

ItemSetLR1 lr1States[MAX_STATES];
int nLR1States = 0;
int lr1Transitions[MAX_STATES][MAX_SYMBOLS];

ItemSetLR1 lalrStates[MAX_STATES];
int nLALRStates = 0;
int lalrTransitions[MAX_STATES][MAX_SYMBOLS];
```

```
MergeInfo mergeTable[MAX_STATES];
int nMergeEntries = 0;

int isNonTerminal(char c) { return isupper(c); }

int contains(char *set, char c) {
    for(int i = 0; i < strlen(set); i++)
        if(set[i] == c) return 1;
    return 0;
}

void addSymbol(char *set, char c) {
    if(!contains(set, c)) {
        int len = strlen(set);
        set[len] = c;
        set[len+1] = '\0';
    }
}

int symbolIndex(char c) {
    for(int i = 0; i < nNonTerms + nTerms; i++) {
        if(i < nNonTerms && nonTerminals[i] == c) return i;
        if(i >= nNonTerms && terminals[i-nNonTerms] == c) return i;
```

```
    }
    return -1;
}

char firstSets[MAX_SYMBOLS][MAX_SYMBOLS];

int idxNonTerm(char c) {
    for(int i = 0; i < nNonTerms; i++)
        if(nonTerminals[i] == c) return i;
    return -1;
}

void computeFirstSets() {
    for(int i = 0; i < nNonTerms; i++) {
        firstSets[i][0] = '\0';
    }

    int changed = 1;
    while(changed) {
        changed = 0;
        for(int i = 0; i < nProds; i++) {
            char A = prods[i].lhs;
            char *rhs = prods[i].rhs;
            int idxA = idxNonTerm(A);
```

```
        char old[MAX_SYMBOLS];
        strcpy(old, firstSets[idxA]);

        if(isNonTerminal(rhs[0])) {
          int idxB = idxNonTerm(rhs[0]);
          for(int k = 0; k < strlen(firstSets[idxB]); k++)
            addSymbol(firstSets[idxA], firstSets[idxB][k]);
        } else {
          addSymbol(firstSets[idxA], rhs[0]);
        }

        if(strcmp(old, firstSets[idxA]) != 0) changed = 1;
      }
    }
  }

CoreSet extractCore(ItemSetLR1 I) {
  CoreSet core;
  core.count = 0;

  for(int i = 0; i < I.count; i++) {
    int exists = 0;
    for(int j = 0; j < core.count; j++) {
      if(core.items[j].prodIndex == I.items[i].prodIndex &&
```

```
            core.items[j].dotPos == I.items[i].dotPos) {
             exists = 1;
             break;
          }
       }

       if(!exists) {
          core.items[core.count].prodIndex = I.items[i].prodIndex;
          core.items[core.count].dotPos = I.items[i].dotPos;
          core.count++;
       }
    }
    return core;
}

int sameCores(CoreSet A, CoreSet B) {
    if(A.count != B.count) return 0;

    for(int i = 0; i < A.count; i++) {
       int found = 0;
       for(int j = 0; j < B.count; j++) {
          if(A.items[i].prodIndex == B.items[j].prodIndex &&
            A.items[i].dotPos == B.items[j].dotPos) {
             found = 1;
```

```
            break;
          }
        }
        if(!found) return 0;
    }
    return 1;
}

int equalItemSets(ItemSetLR1 A, ItemSetLR1 B) {
    if(A.count != B.count) return 0;

    for(int i = 0; i < A.count; i++) {
        int found = 0;
        for(int j = 0; j < B.count; j++) {
            if(A.items[i].prodIndex == B.items[j].prodIndex &&
               A.items[i].dotPos == B.items[j].dotPos &&
               strcmp(A.items[i].lookahead, B.items[j].lookahead) == 0) {
                found = 1;
                break;
            }
        }
        if(!found) return 0;
    }
    return 1;
```

```
}

ItemSetLR1 mergeLookaheads(ItemSetLR1 A, ItemSetLR1 B) {
    ItemSetLR1 merged = A;

    for(int i = 0; i < B.count; i++) {
        int found = 0;
        for(int j = 0; j < merged.count; j++) {
            if(merged.items[j].prodIndex == B.items[i].prodIndex &&
               merged.items[j].dotPos == B.items[i].dotPos) {
                for(int k = 0; k < strlen(B.items[i].lookahead); k++) {
                    addSymbol(merged.items[j].lookahead, B.items[i].lookahead[k]);
                }
                found = 1;
                break;
            }
        }

        if(!found) {
            merged.items[merged.count] = B.items[i];
            merged.count++;
        }
    }
    return merged;
```

```
}

ItemSetLR1 closure(ItemSetLR1 I) {
    ItemSetLR1 J = I;
    int changed = 1;

    while(changed) {
        changed = 0;
        for(int i = 0; i < J.count; i++) {
            ItemLR1 it = J.items[i];
            char *rhs = prods[it.prodIndex].rhs;

            if(it.dotPos < strlen(rhs)) {
                char B = rhs[it.dotPos];
                if(isNonTerminal(B)) {
                    for(int p = 0; p < nProds; p++) {
                        if(prods[p].lhs == B) {
                            char newLA[MAX_SYMBOLS] = "";

                            if(it.dotPos + 1 < strlen(rhs)) {
                                char beta = rhs[it.dotPos + 1];
                                if(isNonTerminal(beta)) {
                                    int idxB = idxNonTerm(beta);
                                    for(int k = 0; k < strlen(firstSets[idxB]); k++)
```

```
            addSymbol(newLA, firstSets[idxB][k]);
        } else {
            addSymbol(newLA, beta);
        }
    } else {
        strcpy(newLA, it.lookahead);
    }

    int exists = 0;
    for(int q = 0; q < J.count; q++) {
        if(J.items[q].prodIndex == p && J.items[q].dotPos == 0) {
            for(int k = 0; k < strlen(newLA); k++)
                addSymbol(J.items[q].lookahead, newLA[k]);
            exists = 1;
        }
    }

    if(!exists) {
        J.items[J.count].prodIndex = p;
        J.items[J.count].dotPos = 0;
        strcpy(J.items[J.count].lookahead, newLA);
        J.count++;
        changed = 1;
    }
```

```
                }
              }
            }
          }
        }
      }
    return J;
}

ItemSetLR1 goTo(ItemSetLR1 I, char X) {
    ItemSetLR1 J;
    J.count = 0;

    for(int i = 0; i < I.count; i++) {
        ItemLR1 it = I.items[i];
        char *rhs = prods[it.prodIndex].rhs;

        if(it.dotPos < strlen(rhs) && rhs[it.dotPos] == X) {
            J.items[J.count].prodIndex = it.prodIndex;
            J.items[J.count].dotPos = it.dotPos + 1;
            strcpy(J.items[J.count].lookahead, it.lookahead);
            J.count++;
        }
    }
```

```
    return closure(J);
}

void buildLR1States() {
    for(int i = 0; i < MAX_STATES; i++) {
        for(int j = 0; j < MAX_SYMBOLS; j++) {
            lr1Transitions[i][j] = -1;
        }
    }

    ItemSetLR1 I0;
    I0.count = 1;
    I0.items[0].prodIndex = 0;
    I0.items[0].dotPos = 0;
    I0.items[0].lookahead[0] = '$';
    I0.items[0].lookahead[1] = '\0';

    lr1States[0] = closure(I0);
    nLR1States = 1;

    for(int i = 0; i < nLR1States; i++) {
        for(int s = 0; s < nNonTerms + nTerms; s++) {
            char X = (s < nNonTerms) ? nonTerminals[s] : terminals[s - nNonTerms];
            ItemSetLR1 J = goTo(lr1States[i], X);
```

```
            if(J.count == 0) continue;

            int exists = -1;
            for(int k = 0; k < nLR1States; k++) {
                if(equalItemSets(lr1States[k], J)) {
                    exists = k;
                    break;
                }
            }

            if(exists == -1) {
                lr1States[nLR1States] = J;
                lr1Transitions[i][s] = nLR1States;
                nLR1States++;
            } else {
                lr1Transitions[i][s] = exists;
            }
        }
    }
}

void buildLALRStates() {
    for(int i = 0; i < MAX_STATES; i++) {
```

```
    for(int j = 0; j < MAX_SYMBOLS; j++) {
        lalrTransitions[i][j] = -1;
    }
}

nLALRStates = 0;
nMergeEntries = 0;

for(int i = 0; i < nLR1States; i++) {
    CoreSet currentCore = extractCore(lr1States[i]);
    int merged = -1;

    for(int j = 0; j < nLALRStates; j++) {
        CoreSet existingCore = extractCore(lalrStates[j]);
        if(sameCores(currentCore, existingCore)) {
            merged = j;
            break;
        }
    }

    if(merged == -1) {
        lalrStates[nLALRStates] = lr1States[i];

        mergeTable[nMergeEntries].lalrState = nLALRStates;
```

```
      mergeTable[nMergeEntries].originalStates[0] = i;
      mergeTable[nMergeEntries].numOriginal = 1;
      nMergeEntries++;

      nLALRStates++;
    } else {
      lalrStates[merged] = mergeLookaheads(lalrStates[merged], lr1States[i]);

      for(int k = 0; k < nMergeEntries; k++) {
        if(mergeTable[k].lalrState == merged) {
          mergeTable[k].originalStates[mergeTable[k].numOriginal] = i;
          mergeTable[k].numOriginal++;
          break;
        }
      }
    }
  }

  for(int i = 0; i < nLALRStates; i++) {
    for(int s = 0; s < nNonTerms + nTerms; s++) {
      char X = (s < nNonTerms) ? nonTerminals[s] : terminals[s - nNonTerms];
      ItemSetLR1 J = goTo(lalrStates[i], X);

      if(J.count == 0) continue;
```

```
        CoreSet gotoCore = extractCore(J);
        for(int k = 0; k < nLALRStates; k++) {
          CoreSet targetCore = extractCore(lalrStates[k]);
          if(sameCores(gotoCore, targetCore)) {
            lalrTransitions[i][s] = k;
            break;
          }
        }
      }
    }
}

void printLALRStates() {
  printf("\n=== LALR STATES (After Merging) ===\n");

  for(int i = 0; i < nLALRStates; i++) {
    printf("\nLALR State I%d:\n", i);
    for(int j = 0; j < lalrStates[i].count; j++) {
      ItemLR1 it = lalrStates[i].items[j];
      char *rhs = prods[it.prodIndex].rhs;
      printf("%c -> ", prods[it.prodIndex].lhs);

      for(int k = 0; k < strlen(rhs); k++) {
```

```
            if(k == it.dotPos) printf(".");
            printf("%c", rhs[k]);
        }
        if(it.dotPos == strlen(rhs)) printf(".");

        printf(", lookahead: {");
        for(int k = 0; k < strlen(it.lookahead); k++) {
            printf("%c", it.lookahead[k]);
            if(k < strlen(it.lookahead) - 1) printf(",");
        }
        printf("}\n");
      }
   }
}

void printMergeTable() {
   printf("\n=== LALR STATE MERGING INFORMATION ===\n");
   printf("LALR State\tMerged from LR(1) States\n");
   printf("----------------------------------------\n");

   for(int i = 0; i < nMergeEntries; i++) {
      printf("I%d\t\t", mergeTable[i].lalrState);
      for(int j = 0; j < mergeTable[i].numOriginal; j++) {
         printf("I%d", mergeTable[i].originalStates[j]);
```

```
            if(j < mergeTable[i].numOriginal - 1) printf(", ");
        }
        printf("\n");
    }

    printf("\nTotal LR(1) states: %d\n", nLR1States);
    printf("Total LALR states: %d\n", nLALRStates);
    printf("States saved: %d\n", nLR1States - nLALRStates);
}

void printLALRTable() {
    printf("\n=== LALR PARSING TABLE ===\n");
    printf("STATE\t");
    for(int i = 0; i < nTerms; i++) printf("%c\t", terminals[i]);
    printf("$\t");
    for(int i = 0; i < nNonTerms; i++) printf("%c\t", nonTerminals[i]);
    printf("\n");

    for(int i = 0; i < nLALRStates; i++) {
        printf("%d\t", i);

        for(int t = 0; t < nTerms; t++) {
            char a = terminals[t];
            char entry[10] = "-";
```

```
    if(lalrTransitions[i][symbolIndex(a)] != -1) {
        sprintf(entry, "s%d", lalrTransitions[i][symbolIndex(a)]);
    } else {
        for(int j = 0; j < lalrStates[i].count; j++) {
            ItemLR1 it = lalrStates[i].items[j];
            char *rhs = prods[it.prodIndex].rhs;

            if(it.dotPos == strlen(rhs) && contains(it.lookahead, a)) {
                if(it.prodIndex == 0 && a == '$') {
                    strcpy(entry, "A");
                } else {
                    sprintf(entry, "r%d", it.prodIndex);
                }
            }
        }
    }
    printf("%s\t", entry);
}

char entry[10] = "-";
for(int j = 0; j < lalrStates[i].count; j++) {
    ItemLR1 it = lalrStates[i].items[j];
    char *rhs = prods[it.prodIndex].rhs;
```

```
            if(it.dotPos == strlen(rhs) && contains(it.lookahead, '$')) {
                if(it.prodIndex == 0) {
                    strcpy(entry, "A");
                } else {
                    sprintf(entry, "r%d", it.prodIndex);
                }
            }
        }
        printf("%s\t", entry);

        for(int nt = 0; nt < nNonTerms; nt++) {
            int idx = lalrTransitions[i][nt];
            if(idx != -1) {
                printf("%d\t", idx);
            } else {
                printf("-\t");
            }
        }
        printf("\n");
    }
}
```

```
int main() {
    printf("Enter number of productions: ");
    scanf("%d", &nProds);

    printf("Enter productions (e.g., A->CC):\n");
    for(int i = 0; i < nProds; i++) {
        char buf[50];
        scanf("%s", buf);
        prods[i].lhs = buf[0];
        strncpy(prods[i].rhs, buf + 3, MAX_LEN - 1);
        prods[i].rhs[MAX_LEN - 1] = '\0';

        if(!contains(nonTerminals, prods[i].lhs)) {
            nonTerminals[nNonTerms++] = prods[i].lhs;
        }

        for(int j = 0; j < strlen(prods[i].rhs); j++) {
            if(!isNonTerminal(prods[i].rhs[j]) &&
               !contains(terminals, prods[i].rhs[j])) {
                terminals[nTerms++] = prods[i].rhs[j];
            }
        }
    }
```

```
    for(int i = nProds; i > 0; i--) {
        prods[i] = prods[i-1];
    }
    prods[0].lhs = 'S';
    sprintf(prods[0].rhs, "%c", prods[1].lhs);
    nProds++;

    computeFirstSets();
    buildLR1States();
    buildLALRStates();

    printLALRStates():
    printMergeTable();
    printLALRTable();

    return 0;
}
```

Output:

```
Enter number of productions: 3
Enter productions (e.g., A->CC):
A->CC
C->cC
C->d

=== LALR STATES (After Merging) ===

LALR State I0:
S -> .A, lookahead: {$}
A -> .CC, lookahead: {$}
C -> .cC, lookahead: {c,d}
C -> .d, lookahead: {c,d}

LALR State I1:
S -> A., lookahead: {$}

LALR State I2:
A -> C.C, lookahead: {$}
C -> .cC, lookahead: {$}
C -> .d, lookahead: {$}

LALR State I3:
C -> c.C, lookahead: {c,d,$}
C -> .cC, lookahead: {c,d,$}
C -> .d, lookahead: {c,d,$}

LALR State I4:
C -> d., lookahead: {c,d,$}

LALR State I5:
A -> CC., lookahead: {$}

LALR State I6:
C -> cC., lookahead: {c,d,$}
```

```
=== LALR STATE MERGING INFORMATION ===
LALR State  Merged from LR(1) States
--------------------------------------
I0      I0
I1      I1
I2      I2
I3      I3, I6
I4      I4, I7
I5      I5
I6      I8, I9

Total LR(1) states: 10
Total LALR states: 7
States saved: 3

=== LALR PARSING TABLE ===
S   c   d   $   A   C
0   s3  s4  -   1   2
1   -   -   A   -   -
2   s3  s4  -   -   5
3   s3  s4  -   -   6
4   r3  r3  r3  -   -
5   -   -   r1  -   -
6   r2  r2  r2  -   -
```

(b)

Aim: Implement the LALR parsing algorithm, get parse table and input string are inputs. Use C language for implementation.

Algorithm:

- Input Grammar and Productions
- Input Terminals and Non-terminals
- Augment Grammar
- Input ACTION Table
- Input GOTO Table
- Input String to Parse
- Parse String using tables
- Print Stack Configuration, Input String and Action Taken for each step
- Display Final Result

Source Code:

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>


#define MAX_PRODS 20

#define MAX_TERMS 20

#define MAX_NT 20

#define MAX_STATES 50

#define MAX_INPUT 50

#define MAX_ACTION_LEN 10


typedef struct {

   char lhs;

   char rhs[50];

} Production;


Production prods[MAX_PRODS];

int nProds;

char terminals[MAX_TERMS];

int nTerms;

char nonTerminals[MAX_NT];

int nNonTerms;

int nStates;
```

```
char action[MAX_STATES][MAX_TERMS+1][MAX_ACTION_LEN];
int gotoTable[MAX_STATES][MAX_NT];

int termIndex(char c) {
   if(c == '$') return nTerms;
   for(int i = 0; i < nTerms; i++)
      if(terminals[i] == c) return i;
   return -1;
}

int nonTermIndex(char c) {
   for(int i = 0; i < nNonTerms; i++)
      if(nonTerminals[i] == c) return i;
   return -1;
}


void parseInputString(char *input) {
   int stateStack[MAX_INPUT];
   char symbolStack[MAX_INPUT];
   int top = 0;
   int inputPtr = 0;
   int inputLen = strlen(input);
```

```
stateStack[0] = 0;
symbolStack[0] = '$';

printf("\n=== LALR PARSING ===\n");
printf("%-20s %-20s %-30s\n", "Stack", "Input", "Action");
printf("----------------------------------------------------------------\n");

while(1) {
  int currentState = stateStack[top];
  char currentInput = (inputPtr < inputLen) ? input[inputPtr] : '$';
  int termIdx = termIndex(currentInput);

  if(termIdx == -1) {
    printf("Error: Invalid input symbol '%c'\n", currentInput);
    return;
  }

  char stackStr[100] = "";
  for(int i = 0; i <= top; i++) {
    char buf[10];
    sprintf(buf, "%d", stateStack[i]);
    strcat(stackStr, buf);
    if(i < top) {
```

```
        char symBuf[3];
        sprintf(symBuf, "%c", symbolStack[i+1]);
        strcat(stackStr, symBuf);
    }
}

printf("%-20s %-20s ", stackStr, &input[inputPtr]);

char *actionStr = action[currentState][termIdx];

if(strcmp(actionStr, "-") == 0 || strlen(actionStr) == 0) {
    printf("%-30s\n", "ERROR - No action defined");
    printf("\nResult: INPUT REJECTED\n");
    return;
}

if(actionStr[0] == 's') {
    int nextState = atoi(&actionStr[1]);
    printf("%-30s\n", actionStr);

    top++;
    stateStack[top] = nextState;
    symbolStack[top] = currentInput;
    inputPtr++;
```

```
} else if(actionStr[0] == 'r') {
    int prodNum = atoi(&actionStr[1]);

    if(prodNum < 0 || prodNum >= nProds) {
        printf("ERROR - Invalid production number\n");
        return;
    }

    Production prod = prods[prodNum];
    int rhsLen = strlen(prod.rhs);

    char reduceStr[50];
    sprintf(reduceStr, "Reduce by P%d: %c->%s", prodNum, prod.lhs, prod.rhs);
    printf("%-30s\n", reduceStr);

    for(int i = 0; i < rhsLen; i++) {
        top--;
    }
    int ntIdx = nonTermIndex(prod.lhs);
    if(ntIdx == -1) {
        printf("ERROR - Invalid non-terminal in production\n");
        return;
    }
```

```
        int gotoState = gotoTable[stateStack[top]][ntIdx];
        if(gotoState == -1) {
            printf("ERROR - No GOTO entry for state %d, symbol %c\n",
                   stateStack[top], prod.lhs);
            return;
        }
        top++;
        stateStack[top] = gotoState;
        symbolStack[top] = prod.lhs;

    } else if(strcmp(actionStr, "A") == 0) {
        printf("%-30s\n", "ACCEPT");
        printf("\nResult: INPUT ACCEPTED\n");
        return;

    } else {
        printf("%-30s\n", "ERROR");
        printf("\nResult: INPUT REJECTED\n");
        return;
    }
  }
}
```

```c
int main() {
    printf("Enter number of productions: ");
    scanf("%d", &nProds);
    printf("Enter productions (format: A->BC):\n");
    for(int i = 0; i < nProds; i++) {
        char buf[50];
        scanf("%s", buf);
        prods[i].lhs = buf[0];
        strcpy(prods[i].rhs, buf + 3);
    }

    printf("\nEnter number of terminals: ");
    scanf("%d", &nTerms);
    for(int i = 0; i < nTerms; i++) {
        scanf(" %c", &terminals[i]);
    }

    printf("Enter number of non-terminals: ");
    scanf("%d", &nNonTerms);
    for(int i = 0; i < nNonTerms; i++) {
        scanf(" %c", &nonTerminals[i]);
    }
```

```
for(int i = nProds; i > 0; i--) {
    prods[i] = prods[i-1];
}
prods[0].lhs = 'S';
sprintf(prods[0].rhs, "%c", prods[1].lhs);
nProds++;
//printf("\nAugmented Grammar:\n");
//for(int i = 0; i < nProds; i++) {
//    printf("P%d: %c -> %s\n", i, prods[i].lhs, prods[i].rhs);
//}

printf("\nEnter number of LALR states: ");
scanf("%d", &nStates);

printf("\nEnter LALR Action Table (rows = states, columns = terminals + $):\n");
for(int i = 0; i < nStates; i++) {
    for(int j = 0; j <= nTerms; j++) {
        scanf("%s", action[i][j]);
    }
}

printf("\nEnter LALR GOTO Table (rows = states, columns = non-terminals, use - for no transition):\n");
for(int i = 0; i < nStates; i++) {
    for(int j = 0; j < nNonTerms; j++) {
```

```
            char buf[10];
            scanf("%s", buf);
            if(buf[0] == '-')
                gotoTable[i][j] = -1;
            else
                gotoTable[i][j] = atoi(buf);
        }
    }




    char input[MAX_INPUT];
    printf("\nEnter input string to parse (without $): ");
    scanf("%s", input);
    strcat(input, "$");

    parseInputString(input);

    return 0;
}
```

OUTPUT

```
Enter number of productions: 3
Enter productions (format: A->BC):
A->CC
C->cC
C->d

Enter number of terminals: 2
c d
Enter number of non-terminals: 2
A C

Enter number of LALR states: 7
```

```
Enter LALR Action Table (rows = states, columns = terminals + $):
s3 s4 -
- - A
s3 s4 -
s3 s4 -
r3 r3 r3
- - r1
r2 r2 r2

Enter LALR GOTO Table (rows = states, columns = non-terminals, use - for no transition):
1 2
- -
- 5
- 6
- -
- -
- -
```

```
Enter input string to parse (without $): cdcd

=== LALR PARSING ===
Stack                Input                Action
--------------------------------------------------------------
0                    cdcd$                s3
0c3                  dcd$                 s4
0c3d4                cd$                  Reduce by P3: C->d
0c3C6                cd$                  Reduce by P2: C->cC
0C2                  cd$                  s3
0C2c3                d$                   s4
0C2c3d4              $                    Reduce by P3: C->d
0C2c3C6              $                    Reduce by P2: C->cC
0C2C5                $                    Reduce by P1: A->CC
0A1                  $                    ACCEPT

Result: INPUT ACCEPTED
```

BCSE307P Compiler Design Lab
ASSESSMENT 4
22BCE1462 Guha Pranav Yelchuru

EXPERIMENT 12

(a)

Aim: Implementation of a simple calculator using LEX and YACC tools

Algorithm:

- Read input using LEX and convert characters into tokens (numbers, operators, parentheses, newline).
- Parse tokens in YACC according to grammar rules (expr, term, factor) respecting operator precedence.
- Evaluate expressions during parsing using semantic actions ($$).
- Handle errors (syntax errors, division by zero).
- Print the result after each expression.
- Repeat until end-of-file (Ctrl+D).

Calc.l:

```
%{

#include "y.tab.h"

%}



%%

[0-9]+     { yylval = atoi(yytext); return NUMBER; }

[ \t]+     ;

"\n"       { return '\n'; }

"+"        { return PLUS; }

"-"        { return MINUS; }

"*"        { return MUL; }

"/"        { return DIV; }

"("        { return LPAREN; }
```

```
")"       { return RPAREN; }

.         { return yytext[0]; }

%%

int yywrap() { return 1; }
```

Calc.y:

```
%{
#include <stdio.h>
#include <stdlib.h>
int yylex();
void yyerror(const char *s);
%}

%token NUMBER
%token PLUS MINUS MUL DIV LPAREN RPAREN

%%

input:
   /* empty */
   | input line
   ;

line:
```

```
    expr '\n' { printf("Result = %d\n", $1); }
    ;

expr: expr PLUS term   { $$ = $1 + $3; }
    | expr MINUS term  { $$ = $1 - $3; }
    | term             { $$ = $1; }
    ;

term: term MUL factor  { $$ = $1 * $3; }
    | term DIV factor  {
        if ($3 == 0) {
            printf("Error: Division by zero\n");
            exit(1);
        }
        $$ = $1 / $3;
      }
    | factor           { $$ = $1; }
    ;

factor: NUMBER         { $$ = $1; }
      | LPAREN expr RPAREN { $$ = $2; }
      ;

%%
```

```
int main() {

   printf("Enter arithmetic expressions (Ctrl+D to end):\n");

   yyparse();

   return 0;

}



void yyerror(const char *s) {

   fprintf(stderr, "Error: %s\n", s);

}
```

OUTPUT:

```
Asus@LAPTOP-N4CPSGA2 /cygdrive/c/Users/Asus/Desktop/GuhaPranavYelchuru/VITC/sem7/CompilerLab/ass4
$ yacc -d calc.y

Asus@LAPTOP-N4CPSGA2 /cygdrive/c/Users/Asus/Desktop/GuhaPranavYelchuru/VITC/sem7/CompilerLab/ass4
$ lex calc.l

Asus@LAPTOP-N4CPSGA2 /cygdrive/c/Users/Asus/Desktop/GuhaPranavYelchuru/VITC/sem7/CompilerLab/ass4
$ gcc lex.yy.c y.tab.c -o calc

Asus@LAPTOP-N4CPSGA2 /cygdrive/c/Users/Asus/Desktop/GuhaPranavYelchuru/VITC/sem7/CompilerLab/ass4
$ ./calc
Enter arithmetic expressions (Ctrl+D to end):
3+4*(2-1)
Result = 7
10/(2+3)
Result = 2
```

EXPERIMENT 13

Aim: Implementation of Abstract Syntax Tree. Infix to Postfix using LEX and YACC tools

Algorithm:

- Read the arithmetic expression from input.
- Use LEX to convert characters into tokens: numbers, operators, parentheses.
- Ignore whitespace and handle newlines.
- Use YACC to parse tokens according to grammar rules: expr, term, factor.
- Respect operator precedence (*, / higher than +, -).
- For each number/operator, create an AST node.
- Assign left and right children to operator nodes based on grammar.
- After parsing an expression, traverse AST in post-order.
- Print numbers and operators during traversal → generates postfix expression.
- Handle syntax errors using yyerror().
- Repeat the process for multiple expressions until end-of-file (Ctrl+D).

Infix.l:

```
%{

#include <stdlib.h>



typedef struct Node {

   char op;

   int val;

   struct Node *left;

   struct Node *right;

} Node;



#define YYSTYPE Node*
```

```
#include "y.tab.h"

Node* createNode(char op, int val, Node* left, Node* right);
%}

%%

[0-9]+      { yylval = createNode('\0', atoi(yytext), NULL, NULL); return NUMBER; }
[ \t]+      ;
"+"         { return PLUS; }
"-"         { return MINUS; }
"*"         { return MUL; }
"/"         { return DIV; }
"("         { return LPAREN; }
")"         { return RPAREN; }
\n          { return '\n'; }
.           { return yytext[0]; }

%%

int yywrap() { return 1; }
```

Infix.y:

```
%{
#include <stdio.h>
#include <stdlib.h>

typedef struct Node {
   char op;
   int val;
   struct Node *left;
   struct Node *right;
} Node;

int yylex();
void yyerror(const char *s);
Node* createNode(char op, int val, Node* left, Node* right);
void printPostfix(Node* node);

#define YYSTYPE Node*
%}

%token NUMBER
%token PLUS MINUS MUL DIV LPAREN RPAREN

%%
```

```
input:
   /* empty */
   | input line
   ;

line:
   expr '\n' {
      printPostfix($1);
      printf("\n");
   }
   ;

expr:
   expr PLUS term   { $$ = createNode('+', 0, $1, $3); }
   | expr MINUS term { $$ = createNode('-', 0, $1, $3); }
   | term         { $$ = $1; }
   ;

term:
   term MUL factor  { $$ = createNode('*', 0, $1, $3); }
   | term DIV factor { $$ = createNode('/', 0, $1, $3); }
   | factor       { $$ = $1; }
   ;
```

```
factor:
    NUMBER          { $$ = $1; }
    | LPAREN expr RPAREN { $$ = $2; }
    ;

%%

Node* createNode(char op, int val, Node* left, Node* right) {
    Node* node = (Node*)malloc(sizeof(Node));
    node->op = op;
    node->val = val;
    node->left = left;
    node->right = right;
    return node;
}

void printPostfix(Node* node) {
    if (node == NULL) return;
    printPostfix(node->left);
    printPostfix(node->right);
    if (node->op == '\0')
        printf("%d ", node->val);
    else
```

```
    printf("%c ", node->op);
}


int main() {
  printf("Enter arithmetic expressions (Ctrl+D to end):\n");
  yyparse();
  return 0;
}


void yyerror(const char *s) {
  fprintf(stderr, "Error: %s\n", s);
}
```

Output:

```
Asus@LAPTOP-N4CPSGA2 /cygdrive/c/Users/Asus/Desktop/GuhaPranavYelchuru/VITC/sem7/CompilerLab/ass4
$ yacc -d infix.y
Asus@LAPTOP-N4CPSGA2 /cygdrive/c/Users/Asus/Desktop/GuhaPranavYelchuru/VITC/sem7/CompilerLab/ass4
$ lex infix.l

Asus@LAPTOP-N4CPSGA2 /cygdrive/c/Users/Asus/Desktop/GuhaPranavYelchuru/VITC/sem7/CompilerLab/ass4
$ gcc lex.yy.c y.tab.c -o infix

Asus@LAPTOP-N4CPSGA2 /cygdrive/c/Users/Asus/Desktop/GuhaPranavYelchuru/VITC/sem7/CompilerLab/ass4
$ ./infix
Enter arithmetic expressions (Ctrl+D to end):
3+4*(2-1)
3 4 2 1 - * +
10/(2+3)
10 2 3 + /
```

EXPERIMENT 14

Aim: Using LEX and YACC tools to recognize the strings of the following context-free languages:

- L(G) = { anbm / m ≠ n}
- L(G) = { ab (bbaa)n bba (ba)n / n ≥ 0}

Algorithm:

- Start program.
- Read input string (line by line).
- **Lexical analysis**:
  - Identify tokens (a, b, or others).
- **Parsing** using grammar rules:
  - **Pattern1:** Count as and bs → Accept if count_a ≠ count_b.
  - **Pattern2:** Match grammar ab (bbaa)^n bba (ba)^n → Accept if fully matched.
- **Semantic check** (if needed):
  - Pattern1: Compare counts.
  - Pattern2: Already ensured by grammar.
- **Print result**:
  - "Accepted" if string matches language.
  - "Rejected: syntax error" if invalid.
- Repeat for next line until EOF.

(A)

Pattern1.l:

```
%{

#include "y.tab.h"

#include <stdio.h>

#include <stdlib.h>

int count_a = 0;

int count_b = 0;

%}



%%
```

```
a      { count_a++; return A; }

b      { count_b++; return B; }

[ \t]+  ;

\n     { return '\n'; }

.      { printf("Invalid character\n"); exit(1); }


%%


int yywrap() { return 1; }
```

Pattern1.y:

```
%{

#include <stdio.h>

#include <stdlib.h>

extern int count_a, count_b;

int yylex();

void yyerror(const char *s);

%}


%token A B


%%


lines:
```

```
    /* empty */
    | lines line
    ;

line: seq '\n' {
        if (count_a != count_b)
            printf("Accepted\n");
        else
            printf("Rejected: m = n\n");
        count_a = count_b = 0;
     }
    ;

seq: /* empty */
   | seq A
   | seq B
   ;

%%
int main() {
    printf("Enter strings (Ctrl+D to end):\n");
    yyparse();  // parse all lines at once
    return 0;
}
```

```
void yyerror(const char *s) {
    printf("Rejected: syntax error\n");
}
```

Output:

```
Asus@LAPTOP-N4CPSGA2 /cygdrive/c/Users/Asus/Desktop/GuhaPranavYelchuru/VITC/sem7/CompilerLab/ass4
$ yacc -d pattern1.y

Asus@LAPTOP-N4CPSGA2 /cygdrive/c/Users/Asus/Desktop/GuhaPranavYelchuru/VITC/sem7/CompilerLab/ass4
$ lex pattern1.l

Asus@LAPTOP-N4CPSGA2 /cygdrive/c/Users/Asus/Desktop/GuhaPranavYelchuru/VITC/sem7/CompilerLab/ass4
$ gcc lex.yy.c y.tab.c -o pattern1

Asus@LAPTOP-N4CPSGA2 /cygdrive/c/Users/Asus/Desktop/GuhaPranavYelchuru/VITC/sem7/CompilerLab/ass4
$ ./pattern1
Enter strings (Ctrl+D to end):
aaabb
Accepted
aabbb
Accepted
aabb
Rejected: m = n
aaa
Accepted
bb
Accepted
abc
Invalid character
```

(B)

Pattern2.l:

```
%{
#include "y.tab.h"
#include <stdio.h>
#include <stdlib.h>
%}

%%

a      { return A; }
b      { return B; }
\n     { return '\n'; }
.      { printf("Invalid character\n"); exit(1); }

%%

int yywrap() { return 1; }
```

Pattern2.y:

```
%{
#include <stdio.h>
#include <stdlib.h>
int yylex();
```

```
void yyerror(const char *s);

%}


%token A B

%start lines


%%


lines:

     /* empty */

   | lines line

   ;


line: A B repeat1 B B A repeat2 '\n' { printf("Accepted\n"); }

   | A B B B A '\n'              { printf("Accepted\n"); }

   ;


repeat1:

     B B A A repeat1

   | /* empty */

   ;


repeat2:

     B A repeat2
```

```
    | /* empty */

    ;


%%


int main() {

    printf("Enter strings (Ctrl+D to end):\n");

    yyparse();

    return 0;

}


void yyerror(const char *s) {

    printf("Rejected: syntax error\n");

}
```

OUTPUT

```
Asus@LAPTOP-N4CPSGA2 /cygdrive/c/Users/Asus/Desktop/GuhaPranavYelchuru/VITC/sem7
/CompilerLab/ass4
$ lex pattern2.l

Asus@LAPTOP-N4CPSGA2 /cygdrive/c/Users/Asus/Desktop/GuhaPranavYelchuru/VITC/sem7
/CompilerLab/ass4
$ gcc lex.yy.c y.tab.c -o pattern2

Asus@LAPTOP-N4CPSGA2 /cygdrive/c/Users/Asus/Desktop/GuhaPranavYelchuru/VITC/sem7
/CompilerLab/ass4
$ ./pattern2
Enter strings (Ctrl+D to end):
abbba
Accepted
aba
Rejected: syntax error

Asus@LAPTOP-N4CPSGA2 /cygdrive/c/Users/Asus/Desktop/GuhaPranavYelchuru/VITC/sem7
/CompilerLab/ass4
$
```

BCSE307P Compiler Design Lab
ASSESSMENT 5
22BCE1462 Guha Pranav Yelchuru

EXPERIMENT 15

Aim: Implementation of three address codes of a simple program using LEX and YACC tools

Algorithm:

- Start the program.
- Initialize LEX and YACC to handle lexical and syntax analysis.
- Prompt the user: “Enter statement”.
- Read the input statement
- LEX analyzes tokens such as identifiers, operators, and delimiters.
- YACC parses the input according to grammar rules.
- Apply semantic actions in YACC rules to generate intermediate three-address code for each operation.
- Display the generated 3AC
- Repeat steps 3–8 for multiple statements until end of input.
- Stop the program.

three.l:

```
%{

#include "y.tab.h"

#include <string.h>

#include <stdio.h>



int yylex(void);

%}



%%

[a-zA-Z][a-zA-Z0-9]*   { yylval.s = strdup(yytext); return ID; }

[0-9]+                 { yylval.s = strdup(yytext); return NUM; }

"="                    { return ASSIGN; }
```

```
"+"                  { return PLUS; }

"-"                  { return MINUS; }

"*"                  { return MUL; }

"/"                  { return DIV; }

";"                  { return SEMI; }

\n                   ;   /* Ignore newlines */

[ \t\r]              ;   /* Ignore whitespace */

.                    ;

%%


int yywrap() { return 1; }
```

three.y:

```
%{

#include <stdio.h>

#include <stdlib.h>

#include <string.h>


int yylex(void);

int yyerror(char *s);


int tempCount = 0;

char* newTemp();


struct expr {
```

```
    char *addr;
    char code[512];
};
%}

%union {
    char *s;
    struct expr *E;
}

%token <s> ID NUM
%token ASSIGN PLUS MINUS MUL DIV SEMI
%type <E> expr term factor

%%

program:
      /* empty */
    | program stmt
    ;

stmt:
    ID ASSIGN expr SEMI {
        printf("%s", $3->code);
```

```
        printf("%s = %s\n", $1, $3->addr);

        printf("\nEnter statement: ");

    }

    ;


expr:

    expr PLUS term {

        $$ = malloc(sizeof(struct expr));

        char *t = newTemp();

        sprintf($$->code, "%s%s%s = %s + %s\n", $1->code, $3->code, t, $1->addr, $3->addr);

        $$->addr = strdup(t);

    }

    | expr MINUS term {

        $$ = malloc(sizeof(struct expr));

        char *t = newTemp();

        sprintf($$->code, "%s%s%s = %s - %s\n", $1->code, $3->code, t, $1->addr, $3->addr);

        $$->addr = strdup(t);

    }

    | term { $$ = $1; }

    ;


term:

    term MUL factor {

        $$ = malloc(sizeof(struct expr));
```

```
        char *t = newTemp();
        sprintf($$->code, "%s%s%s = %s * %s\n", $1->code, $3->code, t, $1->addr, $3->addr);
        $$->addr = strdup(t);
    }
    | term DIV factor {
        $$ = malloc(sizeof(struct expr));
        char *t = newTemp();
        sprintf($$->code, "%s%s%s = %s / %s\n", $1->code, $3->code, t, $1->addr, $3->addr);
        $$->addr = strdup(t);
    }
    | factor { $$ = $1; }
    ;

factor:
    ID {
        $$ = malloc(sizeof(struct expr));
        $$->addr = strdup($1);
        $$->code[0] = '\0';
    }
    | NUM {
        $$ = malloc(sizeof(struct expr));
        $$->addr = strdup($1);
        $$->code[0] = '\0';
    }
```

```
    ;
%%

char* newTemp() {
    static char buffer[10];
    sprintf(buffer, "t%d", tempCount++);
    return strdup(buffer);
}

int main() {
    printf("Enter statement: ");
    yyparse();
    printf("\nAll statements processed. Exiting.\n");
    return 0;
}

int yyerror(char *s) {
    fprintf(stderr, "Error: %s\n", s);
    return 0;
}
```

OUTPUT:

```
Asus@LAPTOP-N4CPSGA2 /cygdrive/c/Users/Asus/Desktop/GuhaPranavYelchuru/VITC/sem7
/CompilerLab/ass5
$ yacc -d three.y

Asus@LAPTOP-N4CPSGA2 /cygdrive/c/Users/Asus/Desktop/GuhaPranavYelchuru/VITC/sem7
/CompilerLab/ass5
$ lex three.l

Asus@LAPTOP-N4CPSGA2 /cygdrive/c/Users/Asus/Desktop/GuhaPranavYelchuru/VITC/sem7
/CompilerLab/ass5
$ gcc lex.yy.c y.tab.c -o three

Asus@LAPTOP-N4CPSGA2 /cygdrive/c/Users/Asus/Desktop/GuhaPranavYelchuru/VITC/sem7
/CompilerLab/ass5
$ ./three
Enter statement: a = 2 * 3 + b * 4 + c * 1 + d + e * 8;
t0 = 2 * 3
t1 = b * 4
t2 = t0 + t1
t3 = c * 1
t4 = t2 + t3
t5 = t4 + d
t6 = e * 8
t7 = t5 + t6
a = t7

Enter statement:
All statements processed. Exiting.
```

EXPERIMENT 16

Aim: Implement simple code optimization techniques (Constant folding, Strength reduction and Algebraic transformation)

Algorithm:

- Input TAC statements from the user.
- For each TAC statement:
    - Parse the statement into result, operand1, operator, operand2.
    - Apply Constant Folding: If both operands are constants, compute result at compile time.
    - Apply Strength Reduction: Replace multiplication/division by powers of 2 with shift operations (* 2^n → << n, / 2^n → >> n).
    - Apply Algebraic Transformation: Simplify operations like x * 1 → x, x + 0 → x, x - 0 → x. If no optimization applies, keep the original TAC.
- Print the optimized TAC.
- Repeat until all statements are processed.
- End.

Code:

```
#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <math.h>



#define MAX_LINES 100

#define LEN 100



int isNumber(char *str) {

   for(int i=0; str[i]; i++) {

       if(str[i] < '0' || str[i] > '9') return 0;

   }
```

```
    return 1;
}

int powerOf2(int n) {
    if(n <= 0) return -1;
    int exp = 0;
    while(n > 1) {
        if(n % 2 != 0) return -1;
        n /= 2;
        exp++;
    }
    return exp;
}

void optimizeTAC(char *result, char *op1, char op, char *op2) {
    // Constant Folding
    if(op != '\0' && isNumber(op1) && isNumber(op2)) {
        int val1 = atoi(op1);
        int val2 = atoi(op2);
        int res = 0;
        switch(op) {
            case '+': res = val1 + val2; break;
            case '-': res = val1 - val2; break;
            case '*': res = val1 * val2; break;
```

```
        case '/': res = val1 / val2; break;
    }
    printf("%s = %d\n", result, res);
    return;
}

// Strength Reduction
if(op == '*') {
    if(isNumber(op2)) {
        int exp = powerOf2(atoi(op2));
        if(exp != -1) {
            printf("%s = %s << %d\n", result, op1, exp);
            return;
        }
    }
    if(isNumber(op1)) {
        int exp = powerOf2(atoi(op1));
        if(exp != -1) {
            printf("%s = %s << %d\n", result, op2, exp);
            return;
        }
    }
}
if(op == '/') {
```

```
    if(isNumber(op2)) {
        int exp = powerOf2(atoi(op2));
        if(exp != -1) {
            printf("%s = %s >> %d\n", result, op1, exp);
            return;
        }
    }
}

// Algebraic Transformations
if(op == '*') {
    if(isNumber(op1) && atoi(op1) == 1) {
        printf("%s = %s\n", result, op2);
        return;
    }
    if(isNumber(op2) && atoi(op2) == 1) {
        printf("%s = %s\n", result, op1);
        return;
    }
}
if(op == '+') {
    if(isNumber(op1) && atoi(op1) == 0) {
        printf("%s = %s\n", result, op2);
        return;
```

```
        }
        if(isNumber(op2) && atoi(op2) == 0) {
            printf("%s = %s\n", result, op1);
            return;
        }
    }
    if(op == '-') {
        if(isNumber(op2) && atoi(op2) == 0) {
            printf("%s = %s\n", result, op1);
            return;
        }
    }

    if(op != '\0') {
        printf("%s = %s %c %s\n", result, op1, op, op2);
    } else {
        printf("%s = %s\n", result, op1);
    }
}

int main() {
    char input[MAX_LINES][LEN];
    char result[20], op1[20], op2[20], op;
    int lineCount = 0;
```

```
    printf("Enter TAC statements (type 'exit' to stop):\n");
    while(1) {
        fgets(input[lineCount], LEN, stdin);
        input[lineCount][strcspn(input[lineCount], "\n")] = 0;
        if(strcmp(input[lineCount], "exit") == 0) break;
        lineCount++;
    }

    printf("\n--- Optimized TAC ---\n");
    for(int i=0;i<lineCount;i++) {
        if(sscanf(input[i], " %s = %s %c %s", result, op1, &op, op2) == 4) {
            optimizeTAC(result, op1, op, op2);
        } else if(sscanf(input[i], " %s = %s", result, op1) == 2) {
            optimizeTAC(result, op1, '\0', "");
        } else {
            printf("Invalid TAC format: %s\n", input[i]);
        }
    }

    return 0;
}
```

Output:

```
PS C:\Users\Asus\Desktop\GuhaPranavYelchuru\VITC\sem7\CompilerLab\ass5> gcc optimizer.c -o optimizer
PS C:\Users\Asus\Desktop\GuhaPranavYelchuru\VITC\sem7\CompilerLab\ass5> ./optimizer
Enter TAC statements (type 'exit' to stop):
t0 = 2 * 3
t1 = b * 4
t2 = t0 + t1
t3 = c * 1
t4 = t2 + t3
t5 = d + 0
t6 = t4 + t5
t7 = e * 8
t8 = t5 + t6
a = t8
exit

--- Optimized TAC ---
t0 = 6
t1 = b << 2
t2 = t0 + t1
t3 = c << 0
t4 = t2 + t3
t5 = d
t6 = t4 + t5
t7 = e << 3
t8 = t5 + t6
a = t8
```

EXPERIMENT 17

Aim: Implement Back-End of the compiler for which three address code is given as input and the 8086 assembly language is produced as output.:

Algorithm:

- Input TAC Statements
- Process Each TAC Line
    - Split into Destination and Expression
    - Map Temporary Variables (t0, t1 … → R2, R3 …)
    - Identify Operation Type
        - Addition (+)
        - Subtraction (-)
        - Multiplication (*)
        - Division (/)
        - Left Shift (<<)
        - Simple Assignment
- Generate Assembly Instructions
- Print/Output Assembly
- End

Code:

```
#include <stdio.h>

#include <string.h>

#include <stdlib.h>



#define MAX_REGISTERS 4

#define MAX_LINES 100



int registers[MAX_REGISTERS] = {0};



int allocateRegister() {

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

      if (registers[i] == 0) {
```

```
        registers[i] = 1;

        return i;

      }

    }

    printf("Error: Out of registers\n");

    return -1;

}


void freeRegister(int reg) {

    if (reg >= 0 && reg < MAX_REGISTERS)

      registers[reg] = 0;

}


void generateAssembly(char* result, char* op1, char op, char* op2) {

    int reg1, reg2;


    if (op == '\0') {

      reg1 = allocateRegister();

      printf("MOV R%d, %s\n", reg1, op1);

      printf("MOV %s, R%d\n", result, reg1);

      freeRegister(reg1);

      return;

    }
```

```
    reg1 = allocateRegister();

    reg2 = allocateRegister();

    printf("MOV R%d, %s\n", reg1, op1);

    printf("MOV R%d, %s\n", reg2, op2);


    switch (op) {

        case '+': printf("ADD R%d, R%d\n", reg1, reg2); break;

        case '-': printf("SUB R%d, R%d\n", reg1, reg2); break;

        case '*': printf("MUL R%d, R%d\n", reg1, reg2); break;

        case '/': printf("DIV R%d, R%d\n", reg1, reg2); break;

        case '<': printf("SHL R%d, R%d\n", reg1, reg2); break;  // << operator

        default: printf("Invalid operator\n"); freeRegister(reg1); freeRegister(reg2); return;

    }


    printf("MOV %s, R%d\n", result, reg1);

    freeRegister(reg1);

    freeRegister(reg2);

}


int main() {

    char input[MAX_LINES][30];

    char result[10], op1[10], op2[10], op;

    int lineCount = 0;
```

```
    printf("Enter three-address code statements (type 'exit' to stop):\n");
    while (1) {
        fgets(input[lineCount], sizeof(input[lineCount]), stdin);
        input[lineCount][strcspn(input[lineCount], "\n")] = '\0';
        if (strcmp(input[lineCount], "exit") == 0)
            break;
        lineCount++;
    }

    printf("\n\n");

    for (int i = 0; i < lineCount; i++) {
        if (sscanf(input[i], " %s = %s %c %s", result, op1, &op, op2) == 4) {
            generateAssembly(result, op1, op, op2);
        } else if (sscanf(input[i], " %s = %s", result, op1) == 2) {
            generateAssembly(result, op1, '\0', "");
        } else {
            printf("Invalid format in line: %s\n", input[i]);
        }
    }

    return 0;
}
```

Output:

```
PS C:\Users\Asus\Desktop\GuhaPranavYelchuru\VITC\sem7\C
ompilerLab\ass5> gcc code.c -o code
PS C:\Users\Asus\Desktop\GuhaPranavYelchuru\VITC\sem7\C
ompilerLab\ass5> ./code
Enter three-address code statements (type 'exit' to sto
p):
t0 = 2 * 3
t1 = t0 + b
a = t1
exit


MOV R0, 2
MOV R1, 3
MUL R0, R1
MOV t0, R0
MOV R0, t0
MOV R1, b
ADD R0, R1
MOV t1, R0
MOV R0, t1
MOV a, R0
```