CDL

LAB 1 :

Design and implement a lexical analyzer for given language using C/C++/Python and the lexical analyzer should ignore redundant spaces, tabs and new line

---

Pseudocode:

```
Algorithm LexicalAnalyzer
Input: Program source code as text
Output: Tokens with categories

1. Read the input program as a string
2. Initialize left and right pointers at start of string
3. While right < length of input
    a. If current character is operator → print "operator"
       move left and right forward
    b. Else if current character is space/tab/newline
       - extract substring from left to right
       - if substring matches keyword → print "keyword"
       - else if substring is number → print "number"
       - else → print "identifier"
       move left to next position
    c. Else → increment right
4. End While
End Algorithm
```

Code :

```
#include <bits/stdc++.h>
using namespace std;

// operators, keywords, numbers, variables

static const unordered_set<string> KEYWORDS = {
    "INTEGER","BEGIN","READ","IF","ELSE","THEN","ENDIF",
    "WHILE","DO","ENDWHILE","WRITE","PROG","END"
};

bool isKeyword( string &s) {
    return KEYWORDS.find(s) != KEYWORDS.end();
}

bool isOperator(char ch)
{
    if (
        ch == '+' || ch == '-' || ch == '*' ||
        ch == '/' || ch == '>' || ch == '<' ||
        ch == '=' || ch == ';')
    {
        return true;
    }

    return false;
}


bool isNumber(const string_view &sv) {
    if (sv.empty()) return false;
    int dots = 0;
    for (size_t i = 0; i < sv.size(); ++i) {
        char c = sv[i];
        if (c == '.') {
            if (++dots > 1) return false;
        }
        else if (!(isdigit(c) || (c=='-' && i==0))) {
            return false;
        }
    }
    return true;
}

char *substringExtraction(char *realString, int l, int r)
{
    int i;
    char *str = (char *)malloc(sizeof(char) * (r - l + 2));
    for (int i = l; i <= r; i++)
    {
        str[i - l] = realString[i];
        str[r - l + 1] = '\0';
    }
    return str;
}

void parser(string &input) {
    size_t left = 0, right = 0, N = input.size();
    while (right <= N && left <= right) {
        if (right < N && isOperator(input[right])) {
            cout << input[right] << " is an operator\n";
            ++right;
            left = right;
        }
        else if (right == N || isspace((unsigned char)input[right])) {
            if (left != right) {
                string_view raw{ input.data() + left, right - left };

                string up{ raw };
                transform(up.begin(), up.end(), up.begin(), ::toupper);

                if (isKeyword(up)) {
                    cout << raw << " is a keyword\n";
                }
                else if (isNumber(raw)) {
                    cout << raw << " is a number\n";
                }
                else {
                    cout << raw << " is an identifier\n";
                }
            }
            ++right;
            left = right;
        }
        else {
            ++right;
        }
    }
}


int main() {
    cout << "Working dir: " << filesystem::current_path() << "\n";

    ifstream fin("program.txt");
    if (!fin.is_open()) {
        cerr << "❌ error opening program.txt\n";
        return 1;
    }


    string fileContent, line;
    while (getline(fin, line)) {
        fileContent += line;
        fileContent += ' ';
    }
    fin.close();

    cout << "Input program:\n" << fileContent << "\n\nTokens:\n";
    parser(fileContent);
    return 0;
}
```

LAB 2 :

Implementation of Lexical Analyzer using Lex Tool. In this given program, you are tokenizing a custom language using lex tool. In output, it must give regular expressions and their corresponding return value.

Pseudocode

```
Algorithm LexicalAnalyzerWithLex
Input: Source program
Output: Tokens with categories

1. Define token patterns using regular expressions:
   - DIGIT, ID, OP, SEMI, COMMA, KEYWORDS
2. For each token matched:
   a. Print corresponding token type and value
3. Ignore whitespace and newlines
4. If unknown symbol found → print "Unknown symbol"
5. Repeat until end of input
End Algorithm
```

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

int line_no = 1;
%}

DIGIT       [0-9]+
ID          [a-zA-Z_][a-zA-Z0-9_]*
OP          (\+|\-|\*|\/|:=|<|>|<=|>=|==)
SEMI        ;
COMMA       ,
WS          [ \t]+
NEWLINE     \n

%%

"Prog"          { printf("Regex: \"Prog\"\t\tToken: 1 (PROGRAM)\n"); }
"Integer"       { printf("Regex: \"Integer\"\tToken: 2 (INTEGER)\n"); }
"Begin"         { printf("Regex: \"Begin\"\t\tToken: 3 (BEGIN)\n"); }
"read"          { printf("Regex: \"read\"\t\tToken: 4 (READ)\n"); }
"write"         { printf("Regex: \"write\"\t\tToken: 5 (WRITE)\n"); }
"if"            { printf("Regex: \"if\"\t\tToken: 6 (IF)\n"); }
"then"          { printf("Regex: \"then\"\t\tToken: 7 (THEN)\n"); }
"else"          { printf("Regex: \"else\"\t\tToken: 8 (ELSE)\n"); }
"endif"         { printf("Regex: \"endif\"\tToken: 9 (ENDIF)\n"); }
"while"         { printf("Regex: \"while\"\tToken: 10 (WHILE)\n"); }
```

```
"do"            { printf("Regex: \"do\"\t\tToken: 11 (DO)\n"); }
"endwhile"        { printf("Regex: \"endwhile\"\tToken: 12 (ENDWHILE)\n"); }
"end"            { printf("Regex: \"end\"\t\tToken: 13 (END)\n"); }

{DIGIT}          { printf("Regex: [0-9]+\t\tToken: 14 (NUMBER) → %s\n", yytext); }
{ID}            { printf("Regex: [a-zA-Z_][a-zA-Z0-9_]*\tToken: 15 (IDENTIFIER) → %s\n", yytext);
}
{OP}            { printf("Regex: OPERATOR\t\tToken: 16 (OPERATOR) → %s\n", yytext); }
{SEMI}           { printf("Regex: ;\t\tToken: 17 (SEMICOLON)\n"); }
{COMMA}          { printf("Regex: ,\t\tToken: 18 (COMMA)\n"); }

{WS}            { /* ignore spaces */ }
{NEWLINE}         { line_no++; }

.               { printf("Unknown symbol: %s at line %d\n", yytext, line_no); }

%%

int yywrap(void) {
    return 1;
}

int main() {
    printf("Lexical Analysis Output:\n");
    yylex();
    return 0;
}
```

2. Write a program to find closure of all states of any given NFA with-transitions

PseudoCode:

```
Algorithm EpsilonClosure
Input: NFA with epsilon transitions
Output: ε-closure for each state

1. For each state i in NFA:
   a. Mark all states as unvisited
   b. Initialize closure set as empty
   c. Call recursive function epsilon_closure(i)
2. In epsilon_closure(state):
   a. If state not visited:
       - Mark state visited
       - Add state to closure set
       - For each next state reachable by ε-transition:
           call epsilon_closure(next)
```

```
3. Print closure set for state i
End Algorithm
```

```
#include <bits/stdc++.h>

#define MAX_STATES 20

int nfa[MAX_STATES][MAX_STATES];
int closure[MAX_STATES];
int visited[MAX_STATES];
int n;

void epsilon_closure(int state) {
    if (!visited[state]) {
        visited[state] = 1;
        closure[state] = 1;
        for (int next = 0; next < n; next++) {
            if (nfa[state][next]) {
                epsilon_closure(next);
            }
        }
    }
}

int main() {
    int i, j;

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

    printf("Enter ε-transition table (1 if ε-transition exists, else 0):\n");
    printf("Row = from state, Column = to state\n");
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            scanf("%d", &nfa[i][j]);
        }
    }

    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            visited[j] = 0;
            closure[j] = 0;
        }

        epsilon_closure(i);

        printf("ε-closure(q%d) = { ", i);
        for (j = 0; j < n; j++) {
            if (closure[j]) {
                printf("q%d ", j);
            }
        }
        printf("}\n");
    }

    return 0;
}
```

LAB 3:

1. Floating point or not:

pseudocode:

```
Algorithm FloatingPointCheck
Input: String of numbers
Output: Classification as integer or float

1. Define regex for integer → digits only
2. Define regex for float → digits '.' digits
3. If string matches float regex → print "floating point"
4. Else if string matches integer regex → print "not floating point"
5. Else → invalid
End Algorithm
```

```
%{
#include <cstdio>
%}

INT     [0-9]+
FLOAT   [0-9]+\.[0-9]+

%%
{FLOAT}     { printf("%s is a floating-point number\n", yytext); }
{INT}       { printf("%s is not a floating-point number\n", yytext); }
[ \t\r]+
\n
.
%%

int yywrap(void){ return 1; }
int main(void){
    printf("Enter Number(s):\n");
    yylex();
    return 0;
}
```

---

2. Number or tokens :

pseudocode:

```
Algorithm TokenCounter
Input: Source code
Output: Number of keywords, identifiers, constants, operators, punctuations, invalids

1. Define regex patterns for each token type
2. For each lexeme in input:
   a. If matches KEYWORD → increment keyword count
   b. Else if IDENTIFIER → increment identifier count
   c. Else if CONSTANT → increment constant count
   d. Else if OPERATOR → increment operator count
   e. Else if PUNCTUATION → increment punctuation count
   f. Else → increment invalid token count
3. Maintain total tokens counter
4. Print counts at end
End Algorithm
```

```
%{
#include <stdio.h>
int Total_Tokens = 0;
int NumberOfKeywords = 0;
int NumberOfIdentifiers = 0;
int NumberOfConstants = 0;
int NumberOfOperators = 0;
int NumberOfInvalidToken = 0;
int NumberOfPunctuations = 0;
%}

KEYWORD (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)
IDENTIFIER [a-zA-Z_][a-zA-Z0-9_]*
CONSTANT [0-9]+
OPERATOR (:=|<=|>=|==|[+\-*/<>=%])
PUNCTUATION [\(\)\;\,\{\}\[\]\.]
INVALID [0-9a-zA-Z_]+
%%

{KEYWORD}      { printf("%s is a keyword\n", yytext); NumberOfKeywords++; Total_Tokens++;
}
{IDENTIFIER}   { printf("%s is an identifier\n", yytext); NumberOfIdentifiers++; Total_Tokens++;
}
{CONSTANT}      { printf("%s is a constant\n", yytext); NumberOfConstants++; Total_Tokens++;
}
{OPERATOR}      { printf("%s is an operator\n", yytext); NumberOfOperators++; Total_Tokens+
+; }
{PUNCTUATION}   { printf("%s is a punctuation\n", yytext); NumberOfPunctuations++; Total_To
kens++; }
{INVALID}       { printf("%s is an invalid token\n", yytext); NumberOfInvalidToken++;}
[ \t\r\n]+
.               { printf("%s is unknown\n", yytext); Total_Tokens++; }

%%

int yywrap(void) { return 1; }

int main(void){
    printf("Lexical Output:\n");
    yylex();
    printf("Number of Keywords : %d\n", NumberOfKeywords);
    printf("Number of Identifiers : %d\n", NumberOfIdentifiers);
    printf("Number of Constants : %d\n", NumberOfConstants);
    printf("Number of Operators : %d\n", NumberOfOperators);
    printf("Number of Punctuations : %d\n", NumberOfPunctuations);
    printf("Number of Invalid Tokens : %d\n", NumberOfInvalidToken);
    printf("Total Tokens : %d\n", Total_Tokens);

    return 0;
}
```

3. Valid Expression

pseudocode:

```
Algorithm ValidExpressionCheck
Input: Expression string
Output: Valid or Not Valid

1. Read input expression
2. Match against regex:
   a. If expression contains identifier/operator/identifier pattern → Valid
   b. Else → Not Valid
3. Print result
End Algorithm
```

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

```
%%
^[ \t]*[a-zA-Z]([ \t]*[+\-*/][ \t]*[a-zA-Z])*[ \t]*(\n|$)    { printf("It is a Valid expression\n"); }
^[^\n]*(\n|$)                               { printf("It is not a Valid Expression\n"); }
%%

int yywrap (void){
    return 1;
}

int main(void){
    yylex();
    return 0;
}
```

LAB 4:

1. Making in a single line with spaces

psuedocode:

```
Algorithm SingleLineWithSpaces
Input: Multiline text
Output: Single line text with single spaces

1. Initialize variable spaced = true
2. For each character in input:
   a. If character is space, tab, newline:
      - If spaced is false → print one space
      - Set spaced = true
   b. Else:
      - Print character
      - Set spaced = false
3. End For
End Algorithm
```

```
%{
#include <stdio.h>
int spaced = 1;
%}

%option noyywrap

%%
[ \t\n]+     { if (!spaced) { putchar(' '); spaced = 1; } }
.            { ECHO; spaced = 0; }
%%
```

```
int main(void) {
    yylex();
    return 0;
}
```

2. some language vala question agar vowel hai toh...

Pseudocode

```
Algorithm PigLatinConversion
Input: Word
Output: Transformed word

1. Read word
2. If first letter is vowel:
   - Print word + "ay"
3. Else:
   - Print word[1..end] + first letter + "ay"
End Algorithm
```

```
%{
#include <stdio.h>
#include <ctype.h>
%}

%option noyywrap

%%

[A-Za-z]+   {
                char *yytext_copy = yytext;
                int len = yyleng;
                char first = yytext_copy[0];
                if (strchr("AEIOUaeiou", first)) {
                    printf("%say", yytext_copy);
                } else {
                    printf("%s%cay", yytext_copy+1, first);
                }
            }

.           { ECHO; }

%%

int main(void) {
    yylex();
    return 0;
}
```

3. First and Follow

pesudocode:

```
Algorithm FirstAndFollow
Input: Grammar productions
Output: FIRST and FOLLOW sets for each non-terminal

1. Read number of productions
2. For each production:
   - Parse LHS and RHS
   - Store grammar rules
3. Compute FIRST sets:
   a. For each non-terminal A:
      - For each production A → α:
        - If α starts with terminal → add to FIRST(A)
        - If α starts with non-terminal B → add FIRST(B) to FIRST(A)
        - If ε in FIRST(B) → continue to next symbol
4. Compute FOLLOW sets:
   a. Place $ in FOLLOW(start symbol)
   b. For each production A → αBβ:
      - Add FIRST(β) – {ε} to FOLLOW(B)
      - If β ⇒* ε → add FOLLOW(A) to FOLLOW(B)
5. Print FIRST and FOLLOW sets
End Algorithm
```

```
#include <bits/stdc++.h>
using namespace std;

string trim(const string &s){
    size_t a = s.find_first_not_of(" \t\r\n");
    if (a==string::npos) return "";
    size_t b = s.find_last_not_of(" \t\r\n");
    return s.substr(a, b-a+1);
}

vector<string> splitAlternatives(string rhs){
    string t;
    for(char c: rhs) if(c!=' ' && c!='\t') t.push_back(c);
    vector<string> out;
    string cur;
    for(char c: t){
        if(c=='|'){ out.push_back(cur); cur.clear(); }
        else cur.push_back(c);
    }
    if(!cur.empty()) out.push_back(cur);
    return out;
}

int main(){
    // ios::sync_with_stdio(false);
    // cin.tie(nullptr);
    int n;
    cout<<"Enter number of productions: ";
    if(!(cin>>n)) return 0;
    string line;
    getline(cin, line);
    cout<<"Use # for epsilon.\n";

    map<char, vector<string>> grammar;
    set<char> nonTerminals, terminals;
    char start = 0;

    for(int i=0;i<n;i++){
        getline(cin, line);
        line = trim(line);
        if(line.empty()){ i--; continue; }
        size_t arrow = line.find("->");
        if(arrow==string::npos){ cerr<<"Bad production: "<<line<<"\n"; return 1; }
        string L = trim(line.substr(0, arrow));
        string R = line.substr(arrow+2);
        if(L.empty()){ cerr<<"Bad LHS: "<<line<<"\n"; return 1; }
        char lhs = L[0];
        if(i==0) start = lhs;
        nonTerminals.insert(lhs);
        auto alts = splitAlternatives(R);
        for(auto &alt: alts){
            grammar[lhs].push_back(alt);
            for(char c: alt){
                if(isupper((unsigned char)c)) nonTerminals.insert(c);
                else if(c!='#') terminals.insert(c);
            }
        }
    }

    map<char, set<char>> FIRST;
    bool changed = true;
    while(changed){
        changed = false;
        for(char A: nonTerminals){
            for(auto &prod: grammar[A]){
                if(prod == "#"){
                    if(FIRST[A].insert('#').second) changed = true;
                continue;
            }
            bool allEps = true;
            for(char sym: prod){
                if(!isupper((unsigned char)sym)){
                    if(FIRST[A].insert(sym).second) changed = true;
                    allEps = false;
                    break;
                } else {
                    for(char t: FIRST[sym]){
                        if(t != '#' && FIRST[A].insert(t).second) changed = true;
                    }
                    if(FIRST[sym].count('#')) {
                    } else {
                        allEps = false;
                        break;
                    }
                }
            }
            if(allEps){
                if(FIRST[A].insert('#').second) changed = true;
            }
        }
    }
}

map<char, set<char>> FOLLOW;
if(start) FOLLOW[start].insert('$');
changed = true;
while(changed){
    changed = false;
    for(char A: nonTerminals){
        for(auto &prod: grammar[A]){
            int L = (int)prod.size();
            for(int i=0;i<L;i++){
                char B = prod[i];
                if(!isupper((unsigned char)B)) continue;
                bool betaAllEps = true;
                if(i+1 < L){
                    for(int j=i+1;j<L;j++){
                        char sym = prod[j];
                        if(!isupper((unsigned char)sym)){
                            if(FOLLOW[B].insert(sym).second) changed = true;
                            betaAllEps = false;
                            break;
                        } else {
                            for(char t: FIRST[sym]){
                                if(t != '#' && FOLLOW[B].insert(t).second) changed = true;
                        }
                        if(FIRST[sym].count('#')) {

                        } else {
                            betaAllEps = false;
                            break;
                        }
                    }
                }
            } else betaAllEps = true;

            if(betaAllEps){
                for(char t: FOLLOW[A]){
                    if(FOLLOW[B].insert(t).second) changed = true;
                }
            }
        }
    }
}
}

vector<char> nts(nonTerminals.begin(), nonTerminals.end());
sort(nts.begin(), nts.end());
cout<<"\nFIRST sets:\n";
for(char A: nts){
    cout<<"FIRST("<<A<<") = { ";
    vector<string> elems;
    for(char x: FIRST[A]){
        if(x == '#') elems.push_back("ε");
        else { string s; s.push_back(x); elems.push_back(s); }
    }
    sort(elems.begin(), elems.end());
    for(size_t i=0;i<elems.size();++i){
        if(i) cout<<", ";
        cout<<elems[i];
    }
    cout<<" }\n";
}

cout<<"\nFOLLOW sets:\n";
for(char A: nts){
    cout<<"FOLLOW("<<A<<") = { ";
    vector<string> elems;
    for(char x: FOLLOW[A]){
        if(x == '#') elems.push_back("ε");
        else { string s; s.push_back(x); elems.push_back(s); }
    }
    sort(elems.begin(), elems.end());
```

```
        for(size_t i=0;i<elems.size();++i){
            if(i) cout<<", ";
            cout<<elems[i];
        }
        cout<<" }\n";
    }

    return 0;
}
```

Lab 5

Question 1

Program 1: Write a program to implement (in CPP)
pseudocode:

```
Algorithm RecursiveDescentParser
Input: Input string
Output: ACCEPT or REJECT

1. Define function parseS():
   - Match 'c'
   - Call parseA()
   - Match 'd'
   - If all succeed → return true
   - Else backtrack → return false

2. Define function parseA():
   Case 1: Match "ab" → return true
   Case 2: Match "a"  → return true
   Else → return false

3. In main:
   - Read input string
   - Call parseS()
   - If parse succeeds and entire string consumed → ACCEPT
   - Else → REJECT
End Algorithm
```

part a.

(a) Recursive Descent Parsing with back tracking (Brute Force Method)
S → cAd
A → ab / a

```
#include <bits/stdc++.h>
using namespace std;
```

```
string s;
int n;
long long calls = 0, backtracks = 0;

bool match(char ch, int &pos) {
    if (pos < n && s[pos] == ch) { pos++; return true; }
    return false;
}

bool parseA(int &pos) {
    calls++;
    int save = pos;
    if (match('a', pos) && match('b', pos)) {
        return true;
    }
    pos = save; backtracks++;
    if (match('a', pos)) {
        return true;
    }
    pos = save;
    return false;
}

bool parseS(int &pos) {
    calls++;
    int save = pos;
    if (!match('c', pos)) { pos = save; return false; }
    if (!parseA(pos)) { pos = save; return false; }
    if (!match('d', pos)) { pos = save; return false; }
    return true;
}

int main() {
    while (getline(cin, s)) {
        if (s.size() && s.back() == '\r') s.pop_back();
        n = s.size();
        calls = backtracks = 0;
        int pos = 0;
        bool ok = parseS(pos) && pos == n;
        cout << "Input: \"" << s << "\" => " << (ok ? "ACCEPT" : "REJECT") << "\n";
        cout << "calls: " << calls << "  backtracks: " << backtracks << "\n\n";
    }
    return 0;
}
```

part b

Recursive Descent Parsing with back tracking (Brute Force Method)
S → cAd
A → a /ab

```
#include <bits/stdc++.h>
using namespace std;

string s;
int n;
long long calls = 0, backtracks = 0;

bool match(char ch, int &pos) {
    if (pos < n && s[pos] == ch) { pos++; return true; }
    return false;
}

bool parseA(int &pos) {
    calls++;
    int save = pos;
    if (match('a', pos)) {
        return true;
    }
    pos = save; backtracks++;
    if (match('a', pos) && match('b', pos)) {
        return true;
    }
    pos = save;
    return false;
}

bool parseS(int &pos) {
    calls++;
    int save = pos;
    if (!match('c', pos)) { pos = save; return false; }
    if (!parseA(pos)) { pos = save; return false; }
    if (!match('d', pos)) { pos = save; return false; }
    return true;
}

int main() {
    while (getline(cin, s)) {
        if (s.size() && s.back() == '\r') s.pop_back();
        n = s.size();
        calls = backtracks = 0;
        int pos = 0;
        bool ok = parseS(pos) && pos == n;
        cout << "Input: \"" << s << "\" ⇒ " << (ok ? "ACCEPT" : "REJECT") << "\n";
        cout << "calls: " << calls << "  backtracks: " << backtracks << "\n\n";
    }
    return 0;
}
```

question 2

Write a LEX program to recognize the following tokens over the alphabets
{0,1,..,9}
a) The set of all string ending in 00.
b) The set of all strings with three consecutive 222's.
c) The set of all string such that every block of five consecutive symbols contains at least two 5's.
d) The set of all strings beginning with a 1 which, interpreted as the binary representation of an integer, is congruent to zero modulo 5.
e) The set of all strings such that the 10th symbol from the right end is 1.
f) The set of all four digits numbers whose sum is 9
g) The set of all four digital numbers, whose individual digits are in ascending order from left to right.

```
Algorithm TokenRecognition
Input: String over digits {0-9}
Output: Classification of string according to given rules

1. Rule a: If string length ≥ 2 and last two chars = "00"
        → match rule a
2. Rule b: If string contains substring "222"
        → match rule b
3. Rule c: For each block of 5 consecutive characters:
        - Count number of '5's
        - If any block has < 2 fives → reject
        If all blocks valid → match rule c
4. Rule d: If string starts with '1' AND string is binary
        - Convert to decimal
        - If decimal % 5 = 0 → match rule d
5. Rule e: If string length ≥ 10 AND 10th char from right = '1'
        → match rule e
6. Rule f: If string length = 4 AND sum of digits = 9
        → match rule f
7. Rule g: If string length = 4 AND digits strictly ascending
        → match rule g
8. If no rule matched → print "no rule matched"
End Algorithm
```

```
%option noyywrap
%{
#include <stdio.h>
#include <string.h>
```

```
#include <stdlib.h>
#include <bits/stdc++.h>

//strings ending with 00
int rule_a(const char *s){ int n=strlen(s); return n>=2 && s[n-1]=='0' && s[n-2]=='0'; }

//strings with the three consecutive 222's
int rule_b(const char *s){ return strstr(s,"222") != NULL; }

//set of all strings such that every block of five consecutive symbols contains at least two 5’s.
int rule_c(const char *s){
    int n=strlen(s);
    if(n < 5) return 0; /* requires at least one block of length 5 */
    for(int i=0;i<=n-5;i++){
        int cnt=0;
        for(int j=0;j<5;j++) if(s[i+j]=='5') cnt++;
        if(cnt < 2) return 0;
    }
    return 1;
}

//The set of all strings beginning with a 1 which, interpreted as the binary
//representation of an integer, is congruent to zero modulo 5.
int rule_d(const char *s){
    int n=strlen(s);
    if(n<1 || s[0] != '1') return 0;
    for(int i=0;i<n;i++) if(s[i] != '0' && s[i] != '1') return 0;
    int v=0;
    for(int i=0;i<n;i++) v = (v*2 + (s[i]-'0')) % 5;
    return v == 0;
}

//The set of all strings such that the 10th symbol from the right end is 1.
int rule_e(const char *s){
    int n=strlen(s);
    if(n < 10) return 0;
    return s[n-10] == '1';
}

// The set of all four digits numbers whose sum is 9
int rule_f(const char *s){
    if(strlen(s) != 4) return 0;
    int sum=0;
    for(int i=0;i<4;i++) sum += s[i]-'0';
    return sum == 9;
}

// The set of all four digital numbers, whose individual digits are in ascending order
// from left to right.
int rule_g(const char *s){
    if(strlen(s) != 4) return 0;
    return s[0] < s[1] && s[1] < s[2] && s[2] < s[3];
}

void classify(const char *tok){
    if(rule_a(tok)) printf("%s match rule a\n", tok);
    else if(rule_b(tok)) printf("%s match rule b\n", tok);
    else if(rule_c(tok)) printf("%s match rule c\n", tok);
    else if(rule_d(tok)) printf("%s match rule d\n", tok);
    else if(rule_e(tok)) printf("%s match rule e\n", tok);
    else if(rule_f(tok)) printf("%s match rule f\n", tok);
    else if(rule_g(tok)) printf("%s match rule g\n", tok);
    else printf("%s : no rule matched\n", tok);
}
%}

%%

[0-9]+    { classify(yytext); }
[ \t\r\n]+    { /* skip whitespace */ }
.         { printf("%s : invalid token\n", yytext); }

%%

int main(int argc, char **argv){
    yylex();
    return 0;
}
```

question 3

Suppose an image is encodes as an n×m matrix M of ''light intensifies.'' Mij is a number from 0 to 15, with 0=black and 15=white. M may be stored row by row, with rows terminated by newline (n) characters. Call this string VM. For example, if M is

0 0 2 6
0 1 4 7
1 8 8 6

then VM is 0026 n 0147 n 1886. We can also encode M by its differences along rows, dM. For Example, for M above dM is 0+2+4 n +1+3+3 n+70-2. If we replace positive numbers by + and negative numbers by - in dM we get the sequence of changes, d'M. In this case, d'M=0++ n +++ n +0-. Suppose a ‘feature’ is defined to be a nonempty sequence of increasing value of intensity, followed by zero to three unchanging value of intensity followed by at least one decreasing value of intensity, all on one row.
a. Write a LEX program to find maximal features in d'M and surround them by

parentheses. A maximal feature is not a proper substring of any other feature.
b. Display the longest maximal feature of each row.

```
Algorithm MatrixFeatureExtraction
Input: n × m matrix M of intensities (0–15)
Output: Maximal features and longest feature in each row

1. Represent M as string VM (row by row, separated by newline)
2. For each row in M:
   a. Compute differences between consecutive values → dM
   b. Replace:
      - Positive differences with '+'
      - Negative differences with '-'
      - Zero with '0'
      → get d'M
   c. Identify features:
      - One or more '+'
      - Followed by 0 to 3 '0'
      - Followed by one or more '-'
   d. Mark maximal features (not contained in larger ones)
   e. Surround each maximal feature with parentheses in output
   f. Among all features, find longest feature
3. Print:
   - Original row
   - d' sequence
   - d' with features marked
   - Longest maximal feature of row
End Algorithm
```

```
%option noyywrap
%{
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>

typedef struct { int s,e; } Feature;

/* process one input line (one row of intensities) */
void process_line(const char *raw, int rowno){
    char line[4096];
    char orig[4096];
    strncpy(line, raw, sizeof(line)-1); line[sizeof(line)-1]=0;
    /* remove trailing newline/cr */
    int L = strlen(line);
    while(L>0 && (line[L-1]=='\n' || line[L-1]=='\r')) line[--L]=0;
    strncpy(orig, line, sizeof(orig)-1); orig[sizeof(orig)-1]=0;
```

```
/* parse intensities */
int vals[2048], vcnt=0;
int has_space = 0;
for(int i=0;i<L;i++) if(isspace((unsigned char)line[i])) { has_space=1; break; }

if(has_space){
    char buf[4096];
    strncpy(buf, line, sizeof(buf)-1); buf[sizeof(buf)-1]=0;
    char *p = strtok(buf, " \t\r\n");
    while(p){
        vals[vcnt++] = atoi(p);
        p = strtok(NULL, " \t\r\n");
    }
} else {
    for(int i=0;i<L;i++){
        if(isdigit((unsigned char)line[i])){
            vals[vcnt++] = line[i]-'0';
        }
    }
}

if(vcnt < 2){
    printf("Row %d original: %s\n", rowno, orig);
    printf("Row %d d': <none>\n", rowno);
    printf("Row %d d' with maximal features: <none>\n", rowno);
    printf("Row %d longest maximal feature: <none>\n\n", rowno);
    return;
}

/* build d' */
char dprime[4096]; int dlen=0;
for(int i=0;i<vcnt-1;i++){
    int diff = vals[i+1] - vals[i];
    dprime[dlen++] = (diff>0? '+': (diff<0? '-':'0'));
}
dprime[dlen]=0;

/* find all candidate features matching +{1,} 0{0,3} -{1,} */
Feature feats[4096]; int fcnt=0;
for(int i=0;i<dlen;i++){
    if(dprime[i] != '+') continue;
    int p = i;
    while(p < dlen && dprime[p] == '+') p++;
    int plusEnd = p-1;
    for(int z=0; z<=3; z++){
        int zstart = plusEnd+1;
        int zend = zstart + z - 1;
        if(z>0){
            if(zend >= dlen) break;
            int okz = 1;
            for(int k=zstart;k<=zend;k++) if(dprime[k] != '0'){ okz = 0; break; }
            if(!okz) break;
        }
        int mstart = plusEnd + 1 + z;
        if(mstart >= dlen) continue;
        if(dprime[mstart] != '-') continue;
        int m = mstart;
        while(m < dlen && dprime[m] == '-') m++;
        int mend = m-1;
        feats[fcnt].s = i;
        feats[fcnt].e = mend;
        fcnt++;
    }
}

/* remove proper-substring features → keep only maximal features */
int keep[4096]; for(int i=0;i<fcnt;i++) keep[i]=1;
for(int i=0;i<fcnt;i++){
    if(!keep[i]) continue;
    for(int j=0;j<fcnt;j++){
        if(i==j || !keep[j]) continue;
        if(feats[i].s >= feats[j].s && feats[i].e <= feats[j].e){
            if(feats[i].s > feats[j].s || feats[i].e < feats[j].e){
                keep[i] = 0; break;
            }
        }
    }
}

/* collect maximal features */
Feature maxf[4096]; int mcnt=0;
for(int i=0;i<fcnt;i++) if(keep[i]) maxf[mcnt++] = feats[i];

if(mcnt==0){
    printf("Row %d original: %s\n", rowno, orig);
    printf("Row %d d': %s\n", rowno, dprime);
    printf("Row %d d' with maximal features: <none>\n", rowno);
    printf("Row %d longest maximal feature: <none>\n\n", rowno);
    return;
}

/* sort by start asc, length desc */
for(int i=0;i<mcnt;i++) for(int j=i+1;j<mcnt;j++){
    int li = maxf[i].e - maxf[i].s + 1;
    int lj = maxf[j].e - maxf[j].s + 1;
```

```
        if(maxf[i].s > maxf[j].s || (maxf[i].s==maxf[j].s && li < lj)){
            Feature t = maxf[i]; maxf[i]=maxf[j]; maxf[j]=t;
        }
    }

    /* choose non-overlapping set preferring earlier start and longer length */
    int chosen[4096]; for(int i=0;i<mcnt;i++) chosen[i]=0;
    int last_end = -1;
    for(int i=0;i<mcnt;i++){
        if(maxf[i].s > last_end){
            chosen[i]=1;
            last_end = maxf[i].e;
        }
    }

    /* build output with parentheses */
    char out[8192]; int oi=0;
    int idx = 0;
    for(int i=0;i<mcnt;i++){
        if(!chosen[i]) continue;
        int s = maxf[i].s, e = maxf[i].e;
        for(int t=idx;t<s;t++) out[oi++]=dprime[t];
        out[oi++]='(';
        for(int t=s;t<=e;t++) out[oi++]=dprime[t];
        out[oi++]=')';
        idx = e+1;
    }
    for(int t=idx;t<dlen;t++) out[oi++]=dprime[t];
    out[oi]=0;

    /* find longest chosen maximal feature */
    int best_len=0, bs=-1, be=-1;
    for(int i=0;i<mcnt;i++) if(chosen[i]){
        int Lf = maxf[i].e - maxf[i].s + 1;
        if(Lf > best_len){ best_len = Lf; bs = maxf[i].s; be = maxf[i].e; }
    }

    printf("Row %d original: %s\n", rowno, orig);
    printf("Row %d d': %s\n", rowno, dprime);
    printf("Row %d d' with maximal features: %s\n", rowno, out);
    if(best_len>0){
        char best[256]; int bi=0;
        for(int t=bs;t<=be;t++) best[bi++]=dprime[t];
        best[bi]=0;
        printf("Row %d longest maximal feature: %s (length %d)\n\n", rowno, best, best_len);
    } else {
        printf("Row %d longest maximal feature: <none>\n\n", rowno);
    }
```

```
}
%}

%%

.*\n    { static int row_counter = 0; process_line(yytext, ++row_counter); }
<<EOF>> { /* end */ }

%%

int main(int argc, char **argv){
    yylex();
    return 0;
}
```