Task 1:

File 1 – Check Validity of Expression (flex + bison)

valid.l (LEX using flex)

```
Check if an arithmetic expression is valid

%{
#include "valid.tab.h"  // generated by
bison #include <stdlib.h>
%}

%%

[0-9]+        { return NUMBER; }      /*
integer number */
[+\-*/()]     { return yytext[0]; }   /*
operators and brackets */
[ \t]+        { /* ignore spaces and tabs */ }
\n            { return '\n'; }        /* newline as
end of input */
.             { return yytext[0]; }   /* any other
single character */

%%

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

File 2: valid.y (bison)

```
 valid.y : Check if expression is syntactically
valid

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

/* token from LEX */
%token NUMBER

/* operator precedence (optional but nice)
*/
%left '+' '-'
%left '*' '/'

%%

/* Start rule: one expression then newline
*/ input:      expr '\n'   { printf("Valid
expression\n");
}
   ;

expr:
     expr '+' expr
| expr '-' expr
   | expr '*' expr
   | expr '/' expr
   | '(' expr ')'
   | NUMBER
   ;

%%

void yyerror(const char *s) {
   printf("Invalid expression\n");
}

int main() {     printf("Enter an
expression: ");
   yyparse();
return 0;
}
```

Input:

(5+3)*2

Output:

Valid Expression

Input:

5+*3

Output:

Invalid Expression

Task 2 – Evaluate Value of Expression (flex + bison)

File 1: eval.l

```
/* eval.l : Evaluate arithmetic expression */


%{
#include "eval.tab.h"
#include <stdlib.h>
%}

%%

[0-9]+        { yylval = atoi(yytext); return
NUMBER;  }
[+\-*/()]     { return yytext[0];}
[ \t]+        { /* ignore spaces */ }
\n            { return '\n'; }
.             { return yytext[0];}

%%

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

File 2: eval.y

```
/* eval.y : Evaluate arithmetic expression */

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

int yylex(); void
yyerror(const char *s);
%}

/* all semantic values will be int */
%token NUMBER

%left '+' '-'
%left '*' '/'

%%

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

expr:
    expr '+' expr  { $$ = $1 + $3;}
| expr '-' expr  { $$ = $1 - $3;}
   | expr '*' expr  { $$ = $1 * $3;
}    | expr '/' expr  {
if ($3 == 0) {
                    printf("Error: division by
zero\n");
                    YYABORT;
                }
                $$ = $1 / $3;
            }
   | '(' expr ')'   { $$ = $2;}
   | NUMBER         { $$ = $1;}
   ;

%%

void yyerror(const char *s) {
   printf("Invalid expression\n");
}

int main() {    printf("Enter an
expression: ");
   yyparse();
return 0;}
```

Input

8/2*3

Output

Result = 12

Input

5+*3

Output

Invalid Expression

Task 3: Write a program to find whether a grammar is left recursive or not. If not then do left recursion.

```
#include
<bits/stdc++.h>
using namespace std;
int main() {
   string production;
   cout << "Enter production
(example: A->Aa|b): ";     getline(cin,
production);
   // Remove all spaces to make parsing
easy


production.erase(remove(production.be
gin(), production.end(), ' '),
production.end());

   // Left-hand side non-terminal (like
'A')     char A = production[0];     // Find
position of "->"
int posArrow = production.find("->");
if (posArrow == string::npos) {
      cout << "Invalid production
format!" << endl;

      return 0;

   }

   // Right-hand side (everything after -
>)

   string rhs =
production.substr(posArrow +
2);
  bool hasLeftRecursion = false;
   vector<string> alpha;

// for Aα (left recursive parts)

   vector<string> beta;

// for β  (nonleft-recursive parts)

// Split RHS by '|' → each part is one
production

   string current = "";

   for (int i = 0; i <= (int)rhs.size(); i++)
{

      if (i == (int)rhs.size() || rhs[i] ==
'|')
{          if
(!current.empty()) {


  // Check if this part is left
recursive: starts with A
  if (current[0] == A) {
hasLeftRecursion = true;
                // Save α (the part after A)
alpha.push_back(current.substr(1));

             } else {

                // Save β
beta.push_back(current);
             }

          }

          current = "";
} else {           current
+= rhs[i];
      }

   }

   // -------- Result part --------
if (!hasLeftRecursion) {
      cout << "\nThis grammar has NO
left recursion.\n";

   } else {

      cout << "\nThis grammar HAS left
recursion.\n";

// New productions after removing left
recursion

      cout << "\nAfter removing left
recursion:\n";

  // A → β A'
```

```
cout << A << " -> ";

      for (int i = 0; i <
(int)beta.size(); i++) {           cout
<< beta[i] << A << "'";
         if (i != (int)beta.size() - 1) cout <<
" | ";       }
      cout << endl;



      // A' → α A' | ε
cout << A << "' -> ";
      for (int i = 0; i <
(int)alpha.size(); i++) {           cout
<< alpha[i] << A << "'";
         if (i != (int)alpha.size() - 1) cout
<< " | ";
      }

      cout << " | ε" << endl;

   }

   return 0;

}
```

Input 1

A=Aa|b

Output

Left Recursion Detected in A
After Removing Left Recursion:
A → bA'
A' → aA' | ε

Task 4: Write a program to find whether a grammar is left factored or not. If not then do left factoring.

```
#

include <iostream>

#include <vector>
 #include <string>
using namespace std;
// Function to find common prefix of
two strings

string commonPrefix(const string &a,
const string &b) {    string pref = "";
int n = min(a.size(), b.size());
  for (int i = 0; i < n; i++) {
 if (a[i] == b[i]) {
 pref += a[i];       }
else {
  break;
     }

  }

  return pref;

}

int main() {
string nonTerminal;
  int n;

  cout << "Enter non-terminal (like
E, A, S): ";    cin >> nonTerminal;



  cout << "Enter number
of productions: ";    cin >>
n;



  vector<string> prod(n);

  cout << "Enter productions (right
side only, no spaces):\n";    for (int i =
0; i < n; i++) {       cout <<
nonTerminal << " -> ";       cin >>
prod[i];
  }

  // Step 1: Find common prefix of
//ALL productions
string prefix = prod[0];
```

```
  for (int i = 1; i < n; i++) {
      prefix = commonPrefix(prefix,
prod[i]);

      if (prefix == "") break; // no
common prefix

   }

   if (prefix == "") {

      cout << "\nGrammar is already
LEFT FACTORED (no common
prefix).\n";        cout << "Given
grammar:\n";
      for (int i = 0; i < n; i++) {

         cout << nonTerminal << " -> "
<< prod[i] << "\n";

      }
} else {
      cout << "\nGrammar is NOT left
factored.\n";

      cout << "Common prefix = \"" <<
prefix << "\"\n";

   // We will create a new nonterminal:
A'

   string newNonTerminal =
nonTerminal + "'";

  cout << "\nAfter LEFT
FACTORING:\n";

 // First production line: A -> prefix A'

 cout << nonTerminal << " -> " <<
prefix << newNonTerminal << "\n";

   // Build productions of A'

 cout << newNonTerminal << " -> ";

      bool first = true;        for
(int i = 0; i < n; i++) {
         string rest =
prod[i].substr(prefix.size());
//remaining part after prefix



         if (rest == "") {

            // If production is exactly the
prefix, we use epsilon

            if (!first) cout << " |
";              cout << "epsilon";
first = false;           } else {
if (!first) cout << " | ";
cout << rest;              first =
false;
         }

      }

      cout << "\n";

   }

   return 0;

}
```

Input

A=abc|abd

Output

Left Factoring Needed in A
After Left Factoring:
A → abA'
A'→ c | d

Task:5 : Writing a program to find the FIRST and FOLLOW from the given CFG

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

vector<string> prod[26];
set<char> FIRST[26], FOLLOW[26];
bool firstDone[26];
char startSymbol;

bool isNonTerminal(char c) {
   return (c >= 'A' && c <= 'Z');
}

set<char> firstOfString(const string &s);

set<char> firstOf(char X) {
   if (!isNonTerminal(X)) {
      set<char> res;
      res.insert(X);
      return res;
   }

   int idx = X - 'A';
   if (firstDone[idx]) return FIRST[idx];

   firstDone[idx] = true;
   set<char> result;

   for (string rhs : prod[idx]) {
      if (rhs == "e") {
         result.insert('e');
         continue;
      }

      bool allCanEpsilon = true;
      for (int i = 0; i < (int)rhs.size(); i++) {
         char ch = rhs[i];

         if (!isNonTerminal(ch)) {
            result.insert(ch);
            allCanEpsilon = false;
            break;
         } else {
            set<char> f = firstOf(ch);
            bool hasEpsilon = false;
            for (char x : f) {
               if (x == 'e') hasEpsilon = true;
```

```
                else result.insert(x);
            }
            if (!hasEpsilon) {
                allCanEpsilon = false;
                break;
            }
        }
    }
    if (allCanEpsilon) result.insert('e');
}

FIRST[idx] = result;
return result;
}

set<char> firstOfString(const string &s) {
set<char> result;
if (s.size() == 0) return result;

bool allCanEpsilon = true;

for (int i = 0; i < (int)s.size(); i++) {
    char ch = s[i];

    if (!isNonTerminal(ch)) {
        result.insert(ch);
        allCanEpsilon = false;
        break;
    } else {
        set<char> f = firstOf(ch);
        bool hasEpsilon = false;
        for (char x : f) {
            if (x == 'e') hasEpsilon = true;
            else result.insert(x);
        }
        if (!hasEpsilon) {
            allCanEpsilon = false;
            break;
        }
    }
}

if (allCanEpsilon) result.insert('e');
return result;
}

void computeFOLLOW() {
FOLLOW[startSymbol - 'A'].insert('$');

bool changed = true;
```

```
    while (changed) {
        changed = false;

        for (int A = 0; A < 26; A++) {
            if (prod[A].empty()) continue;
            char Achar = 'A' + A;

            for (string rhs : prod[A]) {
                int n = rhs.size();

                for (int i = 0; i < n; i++) {
                    char B = rhs[i];
                    if (!isNonTerminal(B)) continue;

                    int idxB = B - 'A';
                    set<char> toAdd;

                    if (i == n - 1) {
                        toAdd = FOLLOW[A];
                    } else {
                        string beta = rhs.substr(i + 1);
                        set<char> firstBeta = firstOfString(beta);

                        bool hasEpsilon = false;
                        for (char c : firstBeta) {
                            if (c == 'e') hasEpsilon = true;
                            else toAdd.insert(c);
                        }
                        if (hasEpsilon) {
                            for (char c : FOLLOW[A]) toAdd.insert(c);
                        }
                    }

                    int oldSize = FOLLOW[idxB].size();
                    for (char c : toAdd) FOLLOW[idxB].insert(c);
                    if ((int)FOLLOW[idxB].size() > oldSize) changed = true;
                }
            }
        }
    }
}

int main() {
    int n;
    cin >> n;

    for (int i = 0; i < n; i++) {
        string line;
        cin >> line;
        char lhs = line[0];
```

```
    if (i == 0) startSymbol = lhs;

    int idx = lhs - 'A';
    string rhsPart = line.substr(2);
    string current = "";
    for (char c : rhsPart) {
        if (c == '|') {
            prod[idx].push_back(current);
            current = "";
        } else {
            current += c;
        }
    }
    if (!current.empty()) prod[idx].push_back(current);
}

for (int i = 0; i < 26; i++) {
    if (!prod[i].empty()) {
        char A = 'A' + i;
        firstOf(A);
    }
}

computeFOLLOW();

for (int i = 0; i < 26; i++) {
    if (prod[i].empty()) continue;
    char A = 'A' + i;
    cout << "FIRST(" << A << ") = { ";
    for (char c : FIRST[i]) {
        if (c == 'e') cout << "ε ";
        else cout << c << ' ';
    }
    cout << "}\n";
}

cout << "\n";

for (int i = 0; i < 26; i++) {
    if (prod[i].empty()) continue;
    char A = 'A' + i;
    cout << "FOLLOW(" << A << ") = { ";
    for (char c : FOLLOW[i]) {
        if (c == '$') cout << "$ ";
        else cout << c << ' ';
    }
    cout << "}\n";
}

return 0;
```

```
}
```

Input:

```
Enter number of productions: 5
E=TA
A=+TA|e
T=FB
B=*FB|e
F=(E)|i
```

Output

```
FIRST sets:
FIRST(E) = { ( i }
FIRST(A) = { + ε }
FIRST(T) = { ( i }
FIRST(B) = { * ε }
FIRST(F) = { ( i }

FOLLOW sets:
FOLLOW(E) = { ) $ }
FOLLOW(A) = { ) $ }
FOLLOW(T) = { + ) $ }
FOLLOW(B) = { + ) $ }
FOLLOW(F) = { * + ) $ }
```

Task 6: Writing a program to implement Recursive Descent Parsing for the following grammar

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

vector<string> tokens;   // token list:
"id", "+", "*", "(", ")"

int pos = 0;            // current position in
tokens

// helper: return current token or ""
if finished string current() {
   if (pos < (int)tokens.size()) return
tokens[pos];

   return "";

}

// helper: try to match a token bool
match(const string &expected) {    if
(current() == expected) {
pos++;        return true;
   }

   return false;

}

// ---------- Recursive Descent Functions
----------

// Grammar:

// E  → T E'

// E' → + T E' | ε

// T  → F T'

// T' → * F T' | ε // F  → (E) | id
bool E();    // forward declarations
bool Eprime(); bool T(); bool
Tprime(); bool F(); // E → T E'
bool E() {    if (!T()) return false;
if (!Eprime()) return false;
return true; }
```

```
// E' → + T E' | ε bool Eprime()
{    if (current() == "+") {       if
(!match("+")) return false;       if
(!T()) return false;       if
(!Eprime()) return false;
   }

   // if current is not "+", E' -> ε (do
nothing, just return true)

   return true; }
// T → F T' bool T() {    if (!F())
return false;    if (!Tprime()) return
false;    return true; }
// T' → * F T' | ε bool Tprime() {    if
(current() == "*") {       if (!match("*"))
return false;       if (!F()) return false;
if (!Tprime()) return false;
   }

   // if current is not "*", T' -> ε
return true; }
// F → (E) | id bool F() {    if (current()
== "id") {       return match("id");    }
else if (current() == "(") {       if
(!match("(")) return false;       if (!E())
return false;       if (!match(")")) return
false;       return true;

   }

   return false;   // neither id nor '('E')'

}

// ---------- Tokenizer ----------

// Convert string like "(id+id)*id" into
tokens: "(", "id", "+", "id", ")", "*", "id"
bool tokenize(const string &s) {
tokens.clear();    int n = s.size();    for
(int i = 0; i < n; ) {       if
(isspace(s[i])) { i++; continue; }
      if (s[i] == '(' || s[i] == ')' || s[i] ==
'+' || s[i] == '*') {
tokens.push_back(string(1, s[i]));
         i++;

      } else if (i + 1 < n && s[i] == 'i' &&
s[i+1] == 'd') {
tokens.push_back("id");
         i += 2;
      } else {

         // invalid character
return false;
      }

   }

   return true;

}

// ---------- main ----------

int main() {    string input;
cout << "Enter expression: ";
getline(cin, input);    if
(!tokenize(input)) {
      cout << "Invalid characters in
input.\n";

      return 0;

   }

   pos = 0;    bool ok = E();              //
start symbol
   // expression is valid only if:

   // 1) E() returned true
```

```
    // 2) all tokens are consumed (pos ==
tokens.size())     if (ok && pos ==
(int)tokens.size()) {
        cout << "Input string is
accepted.\n";

    } else {

        cout << "Input string is
rejected.\n";

    }

    return 0;
}
```

Input 1

id+id*id$

Output

Input string is accepted.

Input 2

(id+id)*id$

Output

Input string is accepted.

Task 7: Write a program to implement LL(1) parsing for the following grammar

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

bool isNonTerminal(char c) {
    return (c >= 'A' && c <= 'Z');
}

map<pair<char,char>, string> table;

void buildTable() {
    table[{ 'E', 'i' }] = "TA";
    table[{ 'E', '(' }] = "TA";

    table[{ 'A', '+' }] = "+TA";
    table[{ 'A', ')' }] = "e";
    table[{ 'A', '$' }] = "e";

    table[{ 'T', 'i' }] = "FB";
    table[{ 'T', '(' }] = "FB";

    table[{ 'B', '*' }] = "*FB";
    table[{ 'B', '+' }] = "e";
    table[{ 'B', ')' }] = "e";
    table[{ 'B', '$' }] = "e";

    table[{ 'F', 'i' }] = "i";
    table[{ 'F', '(' }] = "(E)";
}

string tokenize(const string &s) {
```

```
    string tokens;
    int n = s.size();
    for (int i = 0; i < n; ) {
        if (i + 1 < n && s[i] == 'i' && s[i+1] == 'd') {
            tokens.push_back('i');
            i += 2;
        } else {
            tokens.push_back(s[i]);
            i++;
        }
    }
    if (tokens.empty() || tokens.back() != '$') tokens.push_back('$');
    return tokens;
}

int main() {
    buildTable();

    cout << "Enter input (example: id+id*id$): ";
    string input;
    cin >> input;

    string tokens = tokenize(input);

    stack<char> st;
    st.push('$');
    st.push('E');

    int idx = 0;
    char a = tokens[idx];

    cout << "\nSteps:\n";
    cout << "Stack\t\tInput\t\tAction\n";
    cout << "-----------------------------------------\n";

    while (!st.empty()) {
        stack<char> temp = st;
        string stackContent;
        while (!temp.empty()) {
            stackContent.push_back(temp.top());
            temp.pop();
        }
        reverse(stackContent.begin(), stackContent.end());

        string remainingInput = tokens.substr(idx);

        cout << stackContent << "\t\t" << remainingInput << "\t\t";
```

```
    char X = st.top();

    if (!isNonTerminal(X)) {
        if (X == a) {
            if (X == '$') {
                cout << "Accept\n";
                break;
            }
            st.pop();
            idx++;
            a = tokens[idx];
            cout << "Match " << X << "\n";
        } else {
            cout << "Error: expected " << X << ", found " << a << "\n";
            break;
        }
    } else {
        auto it = table.find({X, a});
        if (it == table.end()) {
            cout << "Error: no rule for (" << X << ", " << a << ")\n";
            break;
        }

        string rhs = it->second;
        st.pop();

        if (rhs == "e") {
            cout << X << " -> epsilon\n";
        } else {
            cout << X << " -> " << rhs << "\n";
            for (int i = (int)rhs.size() - 1; i >= 0; i--) {
                st.push(rhs[i]);
            }
        }
    }
}

if (st.empty() && a == '$') {
    cout << "\nInput string is accepted.\n";
} else {
    cout << "\nInput string is rejected.\n";
}

return 0;
}
```

Input 1

id+id*id$

Output

Input string is accepted.

Input 2

id+*id$

Output

Error: no rule for (A, *)
Input string is rejected.