Program 1: Write a LEX program to recognize the following tokens over the alphabets {0,1,..,9}

a) The set of all strings ending with 00.

b) The set of all strings with three consecutive 222's.

c) The set of all strings 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.

Solution:

vim tokens.l

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

// Function to compute digit sum
int digit_sum(const char *s) {
   int sum = 0;
   for (int i = 0; i < strlen(s); i++)
      sum += s[i] - '0';
   return sum;
```

```
}

// Check if digits are in ascending order
int is_ascending(const char *s) {
    for (int i = 0; i < strlen(s) - 1; i++)
        if (s[i] > s[i+1])
            return 0;
    return 1;
}

// Check if binary string is divisible by 5
int modulo5(const char *s) {
    int rem = 0;
    for (int i = 0; i < strlen(s); i++) {
        rem = (rem * 2 + (s[i] - '0')) % 5;
    }
    return rem;
}

// Check every block of 5 consecutive symbols has at least two 5s
int at_least_two_fives(const char *s) {
    int len = strlen(s);
    if (len < 5)
        return 0; // Invalid if not enough digits

    for (int i = 0; i <= len - 5; i++) {
```

```
        int count = 0;
        for (int j = 0; j < 5; j++) {
            if (s[i + j] == '5') count++;
        }
        if (count < 2)
            return 0; // A block failed the rule
    }
return 1;
}
%}

%%

[0-9]+ {
    char *s = strdup(yytext);
    int len = strlen(s);

    // i. Ends with 00
    if (len >= 2 && s[len - 2] == '0' && s[len - 1] == '0')
        printf("Ends with 00: %s\n", s);

    // ii. Contains three consecutive 222's (i.e., "222222")
    if (strstr(s, "222222"))
        printf("Contains three consecutive 222's: %s\n", s);

    // iii. Every block of five has at least two 5's
```

```
    if (at_least_two_fives(s))
        printf("Valid: every block of 5 contains at least two 5's: %s\n", s);

    // iv. Starts with 1 and binary value divisible by 5
    if (s[0] == '1' && modulo5(s) == 0)
        printf("Binary string divisible by 5: %s\n", s);

    // v. 10th symbol from right is 1
    if (len >= 10 && s[len - 10] == '1')
        printf("10th symbol from the right is 1: %s\n", s);

    // vi. 4-digit number with digit sum 9
    if (len == 4 && digit_sum(s) == 9)
        printf("4-digit number with digit sum 9: %s\n", s);

    // vii. 4-digit number with digits in ascending order
    if (len == 4 && is_ascending(s))
        printf("4-digit number with digits in ascending order: %s\n", s);

    free(s);
}

.|\n    { /* Ignore other characters */ }

%%
```

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

vim input1.txt (type the below in the text file)

```
100
12222222
155551111
1010
1234567890
1233
1234
1357
```

Output:

```
csbs@csbs-Precision-3660:~$ vim tokens.l
csbs@csbs-Precision-3660:~$ vim input1.txt
csbs@csbs-Precision-3660:~$ lex tokens.l
csbs@csbs-Precision-3660:~$ gcc lex.yy.c -o tokens -ll
csbs@csbs-Precision-3660:~$ ./tokens < input1.txt
Ends with 00: 100
Contains three consecutive 222's: 12222222
Binary string divisible by 5: 1010
10th symbol from the right is 1: 1234567890
Binary string divisible by 5: 1233
4-digit number with digit sum 9: 1233
4-digit number with digits in ascending order: 1233
4-digit number with digits in ascending order: 1234
4-digit number with digits in ascending order: 1357
```

Program 2: Write a LEX program that copies a file, replacing each nonempty sequence of white spaces by a single blank.

Solution:

vim space.l

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

%%

[ \t\n]+    { printf(" "); }    // Replace whitespace sequences with a single space
[^ \t\n]+   { printf("%s", yytext); } // Print other characters as they are

%%

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

vim input.txt

```
This        is
        a                test.
```

Output:

```
csbs@csbs-Precision-3660:~$ vim space.l
csbs@csbs-Precision-3660:~$ vim input.txt
csbs@csbs-Precision-3660:~$ lex space.l
csbs@csbs-Precision-3660:~$ gcc lex.yy.c -o space -ll
csbs@csbs-Precision-3660:~$ vim output.txt
csbs@csbs-Precision-3660:~$ ./space <input.txt> output.txt
csbs@csbs-Precision-3660:~$ vim output.txt
```

input.txt

output.txt

Program 3: Write a program to implement

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

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

Note: What is the difference that you have experienced w.r.t. the language accepted for these two CFGs.

Solution:

vim parser_a.cpp

```
include <iostream>
#include <string>

using namespace std;

string input;
int i = 0;

bool A() {
  int back = i;

  // Try A → ab
  if (input[i] == 'a') {
    i++;
    if (input[i] == 'b') {
      i++;
      return true;
```

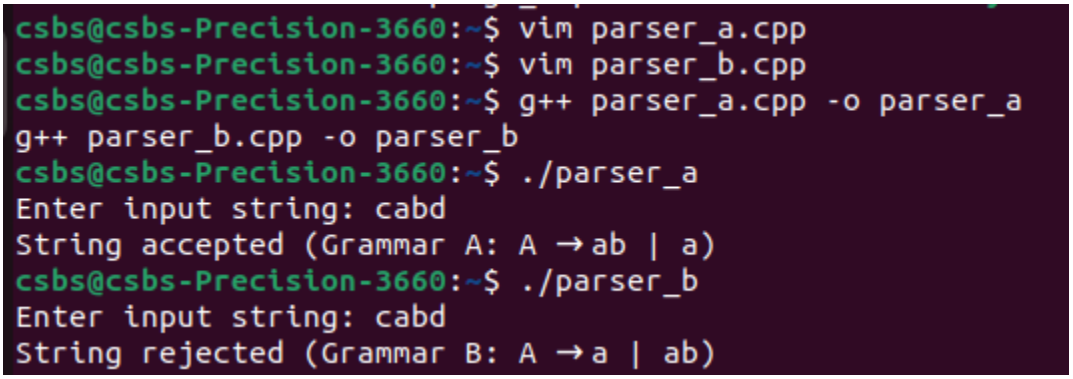
```
    }
  }

  // Backtrack
  i = back;

  // Try A → a
  if (input[i] == 'a') {
    i++;
    return true;
  }

  return false;
}

bool S() {
  if (input[i] == 'c') {
    i++;
    if (A()) {
      if (input[i] == 'd') {
        i++;
        return true;
      }
    }
  }
  return false;
```

```
}
int main() {
   cout << "Enter input string: ";
   cin >> input;
   i = 0;

   if (S() && i == input.length()) {
      cout << "String accepted (Grammar A: A → ab | a)\n";
   } else {
      cout << "String rejected (Grammar A: A → ab | a)\n";
   }

   return 0;
}

vim parser_b.cpp
#include <iostream>
#include <string>

using namespace std;

string input;
int i = 0;

bool A() {
   int back = i;
```

```
    // Try A → a
    if (input[i] == 'a') {
        i++;
        return true;
    }

    i = back;

    // Try A → ab
    if (input[i] == 'a') {
        i++;
        if (input[i] == 'b') {
            i++;
            return true;
        }
    }

    return false;
}

bool S() {
    if (input[i] == 'c') {
        i++;
        if (A()) {
            if (input[i] == 'd') {
```

```
            i++;
            return true;
          }
        }
    }
    return false;
}

int main() {
    cout << "Enter input string: ";
    cin >> input;
    i = 0;

    if (S() && i == input.length()) {
        cout << "String accepted (Grammar B: A → a | ab)\n";
    } else {
        cout << "String rejected (Grammar B: A → a | ab)\n";
    }

    return 0;
}
```

Output:



g++ parser_a.cpp -o parser_a

g++ parser_b.cpp -o parser_b

csbs@csbs-Precision-3660:~$ ./parser_a

Enter input string: cabd

String accepted (Grammar A: A → ab | a)

csbs@csbs-Precision-3660:~$ ./parser_b

Enter input string: cabd

String rejected (Grammar B: A → a | ab)

Program 4: Write a program to implement
(a) Write a program to find FIRST
(b) Write a program to find FOLLOW
(iii) Implementation of LL(1) Parsing Table
Give the parsing process to show that it is neither using backtracking nor using other alternatives.
Outcome: This assignment provides implementation of computing two important sets First and Follow, and also the implementation of LL (1) parser. These functions are very important for developing not only LL (1) parser but also for LR (1) parsers, that are used extensively in practical compilers.

Solution:

```
#include <iostream>
#include <vector>
#include <map>
#include <set>
#include <string>
#include <algorithm>

using namespace std;

// Grammar representation:
// Non-terminals mapped to list of productions (each production is vector<string>)
map<string, vector<vector<string>>> grammar;

// FIRST and FOLLOW sets
map<string, set<string>> FIRST, FOLLOW;

// Terminals and Non-terminals
set<string> terminals;
set<string> nonTerminals;

// Utility: Check if symbol is terminal
bool isTerminal(const string &symbol) {
    return terminals.count(symbol) > 0;
}

// Compute FIRST of a symbol (terminal or non-terminal)
set<string> computeFirst(const string &symbol);

// Compute FIRST of a production
set<string> computeFirstOfProduction(const vector<string> &prod) {
    set<string> result;
    bool epsilon_in_prefix = true;

    for (const string &sym : prod) {
        set<string> first_sym = computeFirst(sym);
```

```
            result.insert(first_sym.begin(), first_sym.end());
            if (first_sym.count("ε")) {
                // Remove epsilon temporarily
                result.erase("ε");
                epsilon_in_prefix = true;
            } else {
                epsilon_in_prefix = false;
                break;
            }
    }

    if (epsilon_in_prefix) {
        result.insert("ε");
    }

    return result;
}

// Compute FIRST sets for all symbols
set<string> computeFirst(const string &symbol) {
    // Terminal: FIRST is the symbol itself
    if (isTerminal(symbol) || symbol == "ε") {
        return {symbol};
    }

    if (FIRST.count(symbol)) {
        return FIRST[symbol];
    }

    set<string> firstSet;

    for (auto &prod : grammar[symbol]) {
        // Compute FIRST of production
        set<string> first_prod = computeFirstOfProduction(prod);
        firstSet.insert(first_prod.begin(), first_prod.end());
    }

    FIRST[symbol] = firstSet;
    return firstSet;
}

// Compute FOLLOW sets
void computeFollow() {
    // Initialize FOLLOW sets empty
    for (auto &nt : nonTerminals) {
        FOLLOW[nt] = {};
```

```
    }
    // Start symbol's FOLLOW contains $
    FOLLOW["S"].insert("$");

    bool changed = true;
    while (changed) {
        changed = false;
for (auto &nt : nonTerminals) {
            for (auto &prod : grammar[nt]) {
                for (int i = 0; i < (int)prod.size(); i++) {
                    string B = prod[i];
                    if (nonTerminals.count(B)) {
                        // Compute FIRST of beta
                        set<string> first_beta;
                        bool epsilon_in_beta = true;

                        for (int j = i + 1; j < (int)prod.size(); j++) {
                            set<string> first_sym = computeFirst(prod[j]);
                            first_beta.insert(first_sym.begin(), first_sym.end());
                            if (first_sym.count("ε")) {
                                first_beta.erase("ε");
                            } else {
                                epsilon_in_beta = false;
                                break;
                            }
                        }
                        if (epsilon_in_beta) {
                            // Add FOLLOW of nt to FOLLOW of B
                            int before_size = FOLLOW[B].size();
                            FOLLOW[B].insert(FOLLOW[nt].begin(), FOLLOW[nt].end());
                            if (FOLLOW[B].size() > before_size)
                                changed = true;
                        }

                        // Add FIRST(beta) \ {ε} to FOLLOW(B)
                        int before_size = FOLLOW[B].size();
                        first_beta.erase("ε");
                        FOLLOW[B].insert(first_beta.begin(), first_beta.end());
                        if (FOLLOW[B].size() > before_size)
                            changed = true;
                    }
                }
            }
        }
    }
}
```

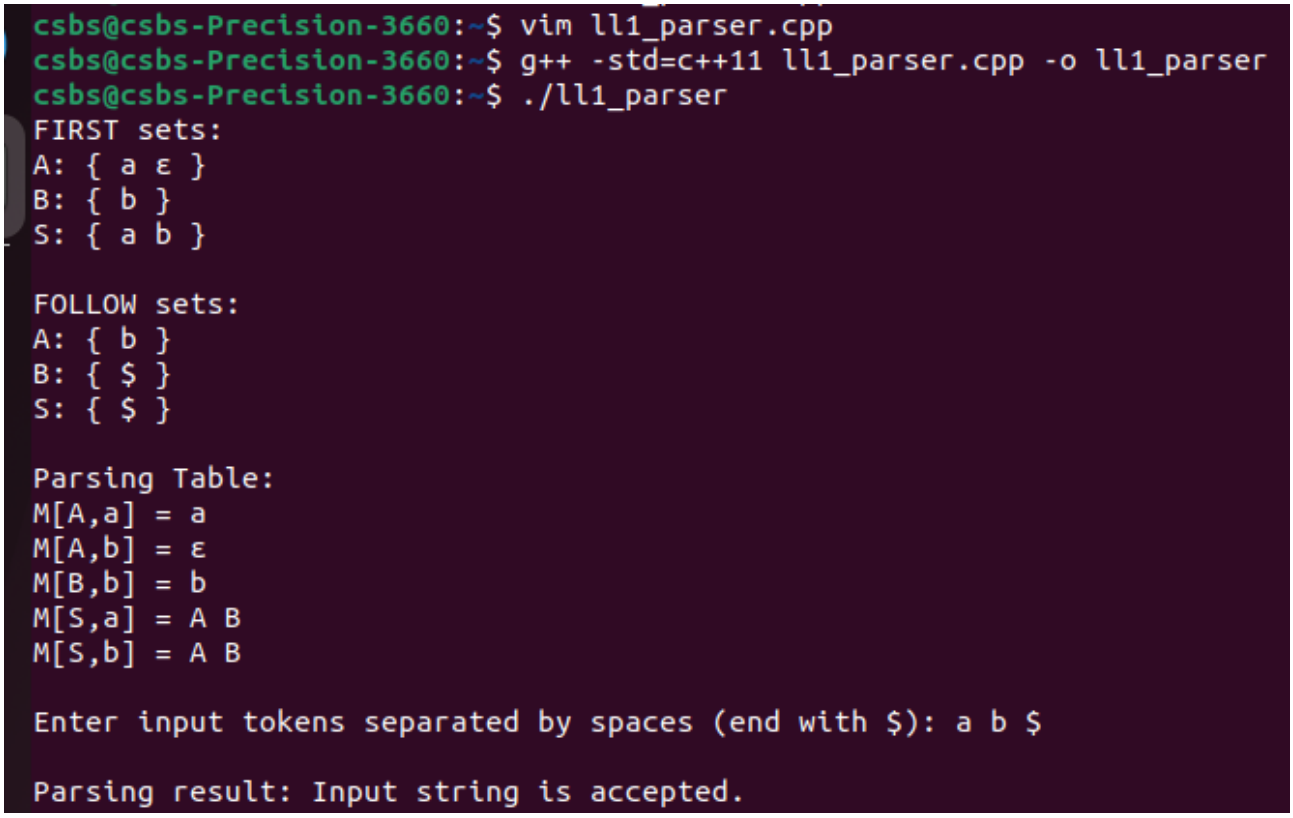
```
// LL(1) Parsing Table (map: Non-terminal -> Terminal -> production index)
map<string, map<string, int>> parsingTable;

// Build LL(1) parsing table
bool buildParsingTable() {
    for (auto &nt : nonTerminals) {
        for (int idx = 0; idx < (int)grammar[nt].size(); idx++) {
            vector<string> prod = grammar[nt][idx];
            set<string> first_prod = computeFirstOfProduction(prod);

            for (const string &terminal : first_prod) {
                if (terminal != "ε") {
                    if (parsingTable[nt].count(terminal)) {
                        cout << "Grammar is not LL(1) (conflict at " << nt << "," << terminal << ")\n";
                        return false;
                    }
                    parsingTable[nt][terminal] = idx;
                }
            }
            if (first_prod.count("ε")) {
                for (const string &foll : FOLLOW[nt]) {
                    if (parsingTable[nt].count(foll)) {
                        cout << "Grammar is not LL(1) (conflict at " << nt << "," << foll << ")\n";
                        return false;
                    }
                    parsingTable[nt][foll] = idx;
                }
            }
        }
    }
    return true;
}

// Parse input using LL(1) parsing table
bool parseInput(const vector<string> &inputTokens) {
    vector<string> stack = {"$", "S"};
    int idx = 0;

    while (!stack.empty()) {
        string top = stack.back();

stack.pop_back();

        string current = (idx < (int)inputTokens.size()) ? inputTokens[idx] : "$";

        if (top == current) {
```

```
            if (top == "$") {
                return true; // accept
            }
            idx++;
        }
        else if (isTerminal(top)) {
            // Terminal mismatch
            cout << "Error: unexpected token " << current << ", expected " << top << "\n";
            return false;
        }
        else {
            // Non-terminal: consult table
            if (!parsingTable[top].count(current)) {
                cout << "Error: no rule for [" << top << ", " << current << "]\n";
                return false;
            }
            int prodIdx = parsingTable[top][current];
            vector<string> prod = grammar[top][prodIdx];
            // Push RHS in reverse order (except ε)
            if (!(prod.size() == 1 && prod[0] == "ε")) {
                for (auto it = prod.rbegin(); it != prod.rend(); ++it) {
                    stack.push_back(*it);
                }
            }
        }
    }
    return false;
}
int main() {
    // Define terminals and non-terminals
    terminals = {"a", "b", "$"};
    nonTerminals = {"S", "A", "B"};

    // Define grammar
    // S -> A B
    grammar["S"].push_back({"A", "B"});

    // A -> a | ε
    grammar["A"].push_back({"a"});
    grammar["A"].push_back({"ε"});

    // B -> b
    grammar["B"].push_back({"b"});

    // Compute FIRST sets
    for (const auto &nt : nonTerminals) {
```

```
    computeFirst(nt);
  }

  cout << "FIRST sets:\n";
  for (const auto &nt : nonTerminals) {
    cout << nt << ": { ";
    for (const auto &s : FIRST[nt]) cout << s << " ";
    cout << "}\n";
  }

  // Compute FOLLOW sets
  computeFollow();

  cout << "\nFOLLOW sets:\n";
  for (const auto &nt : nonTerminals) {
    cout << nt << ": { ";
    for (const auto &s : FOLLOW[nt]) cout << s << " ";
    cout << "}\n";
  }

  // Build parsing table
  if (!buildParsingTable()) {
    cout << "\nGrammar is not LL(1), aborting parsing.\n";
return 1;
  }

  cout << "\nParsing Table:\n";
  for (auto &nt : nonTerminals) {
    for (auto &t : terminals) {
      if (parsingTable[nt].count(t)) {
        int idx = parsingTable[nt][t];
        cout << "M[" << nt << "," << t << "] = ";
        for (auto &sym : grammar[nt][idx]) cout << sym << " ";
        cout << "\n";
      }
    }
  }

  // Input string tokens (space separated)
  cout << "\nEnter input tokens separated by spaces (end with $): ";
  vector<string> inputTokens;
  string token;
  while (cin >> token) {
    inputTokens.push_back(token);
    if (token == "$") break;
  }
```

```
    cout << "\nParsing result: ";
    if (parseInput(inputTokens)) {
        cout << "Input string is accepted.\n";
    } else {
        cout << "Input string is rejected.\n";
    }

    return 0;
}
```

Output:



Program 5: Use YACC to implement, evaluator for arithmetic expressions (Desktop calculator).
Outcome: This assignment gives how an expression is parsed and how syntax-directed translation is carried out on the same. It is found to be a very useful problem as an expression is a very common construct in programming languages.
Solution:

vim calc.l

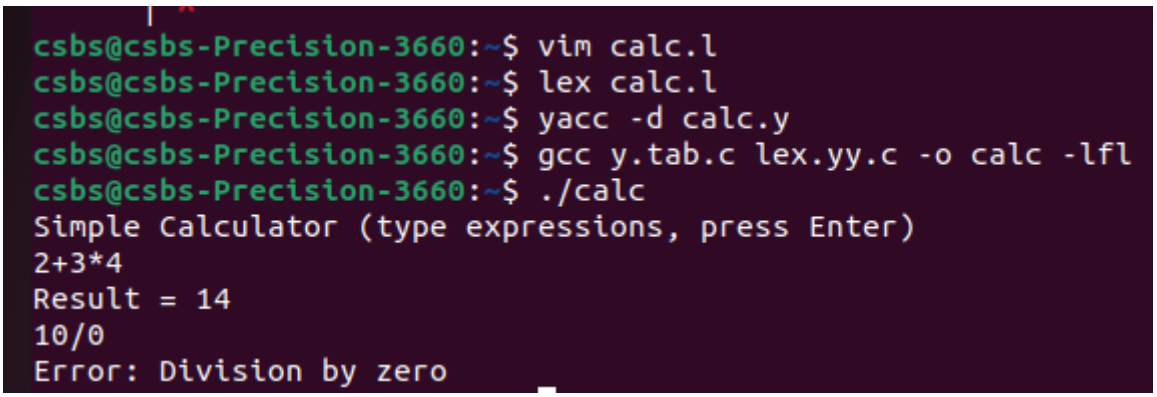
```
%{
#include "y.tab.h"
%}

%%
[0-9]+      { yylval = atoi(yytext); return NUMBER; }
[ \t]       ;  // Ignore whitespace
\n          { return '\n'; }
.           { return yytext[0]; }
%%

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

vim calc.y

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

int yylex();
int yyerror(const char *msg);
%}

%token NUMBER

%%
program:
   program expr '\n'   { printf("Result = %d\n", $2); }
  | /* empty */
  ;

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

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

```
        }
        $$ = $1 / $3;
    }
  | factor              { $$ = $1; }
  ;

factor:
    '(' expr ')'        { $$ = $2; }
  | NUMBER              { $$ = $1; }
  ;
%%

int main() {
    printf("Simple Calculator (type expressions, press Enter)\n");
    yyparse();
    return 0;
}

int yyerror(const char *msg) {
    fprintf(stderr, "Parse error: %s\n", msg);
    return 1;
}
```

Output:



Program 6: Use YACC to generate 3-Address code for a given expression.
Outcome: The students can obtain intermediate code: three-address code, which is commonly used intermediate code in practical implementation of compilers
Solution:
vim expr.l

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

%%
[ \n]  {return '\n';}            ;  // Ignore whitespace
[a-zA-Z_][a-zA-Z0-9_]* {
   yylval.str = strdup(yytext);
   return ID;
}
"="                 return '=';
"+"                 return '+';
"-"                return '-';
"*"                 return '*';
"/"                return '/';
"("                 return '(';
")"                 return ')';
.                  return yytext[0];
%%

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

vim expr.y

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

int yylex();
void yyerror(const char *s);
int tempCount = 1;

char* newTemp() {
   static char temp[10];
   sprintf(temp, "t%d", tempCount++);
   return strdup(temp); // ensure new memory
}

%}

%union {
   char* str;
}
```

```
%token <str> ID
%type <str> expr

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

%%
stmt: ID '=' expr '\n' {
    printf("%s = %s\n", $1, $3);
}
;

expr: expr '+' expr {
    char* temp = newTemp();
    printf("%s = %s + %s\n", temp, $1, $3);
    $$ = temp;
}
| expr '-' expr {
    char* temp = newTemp();
    printf("%s = %s - %s\n", temp, $1, $3);
    $$ = temp;
}
| expr '*' expr {
 char* temp = newTemp();
    printf("%s = %s * %s\n", temp, $1, $3);
    $$ = temp;
}
| expr '/' expr {
    char* temp = newTemp();
    printf("%s = %s / %s\n", temp, $1, $3);
    $$ = temp;
}
| '(' expr ')' {
    $$ = $2;
}
| ID {
    $$ = $1;
}
;

%%

int main() {
    printf("Enter an assignment expression (e.g., a = b + c * d):\n");
    return yyparse();
}

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

Output:

```
csbs@csbs-Precision-3660:~$ vim expr.l
csbs@csbs-Precision-3660:~$ yacc -d expr.y
csbs@csbs-Precision-3660:~$ lex expr.l
csbs@csbs-Precision-3660:~$ gcc y.tab.c lex.yy.c -o expr
csbs@csbs-Precision-3660:~$ ./expr
Enter an assignment expression (e.g., a = b + c * d):
a=b+c*d
t1 = c * d
t2 = b + t1
a = t2
```