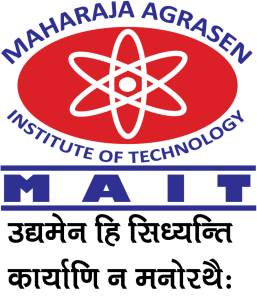
***Compiler Design***

**Faculty Name – Ms. Garima Gupta Student Name – Kamarushi Rachit Kumar**

**Branch – CSE (Shift-1) Semester – Fifth Semester**

**Enrollment no – 00414802722 Group – C1**



Maharaja Agrasen Institute of Technology, PSP Area, Sector – 22, Rohini, New Delhi - 110083

Logo

Description automatically generated

**MAHARAJA AGRASEN INSTITUTE OF TECHNOLOGY**

**COMPUTER SCIENCE & ENGINEERING DEPARTMENT**

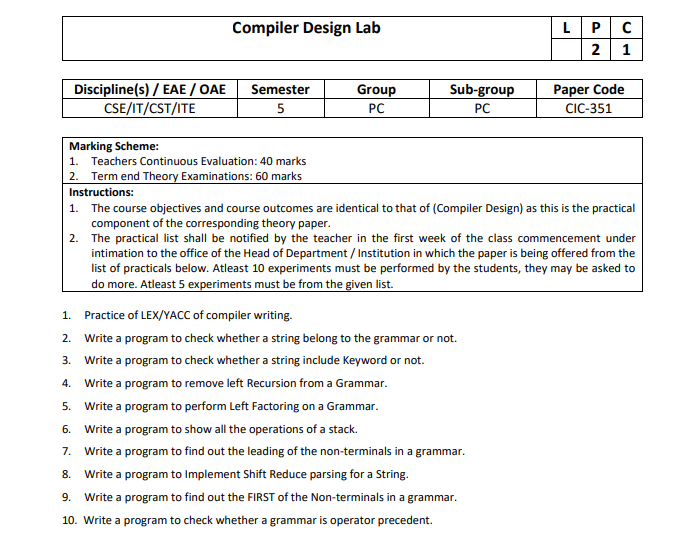
# VISION

“To be center of excellence in education, research and technology transfer in the field of computer engineering and promote entrepreneurship and ethical values.”

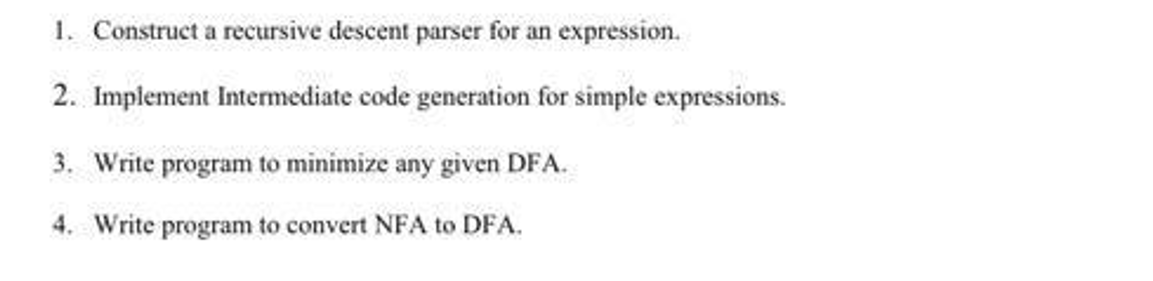
# MISSION

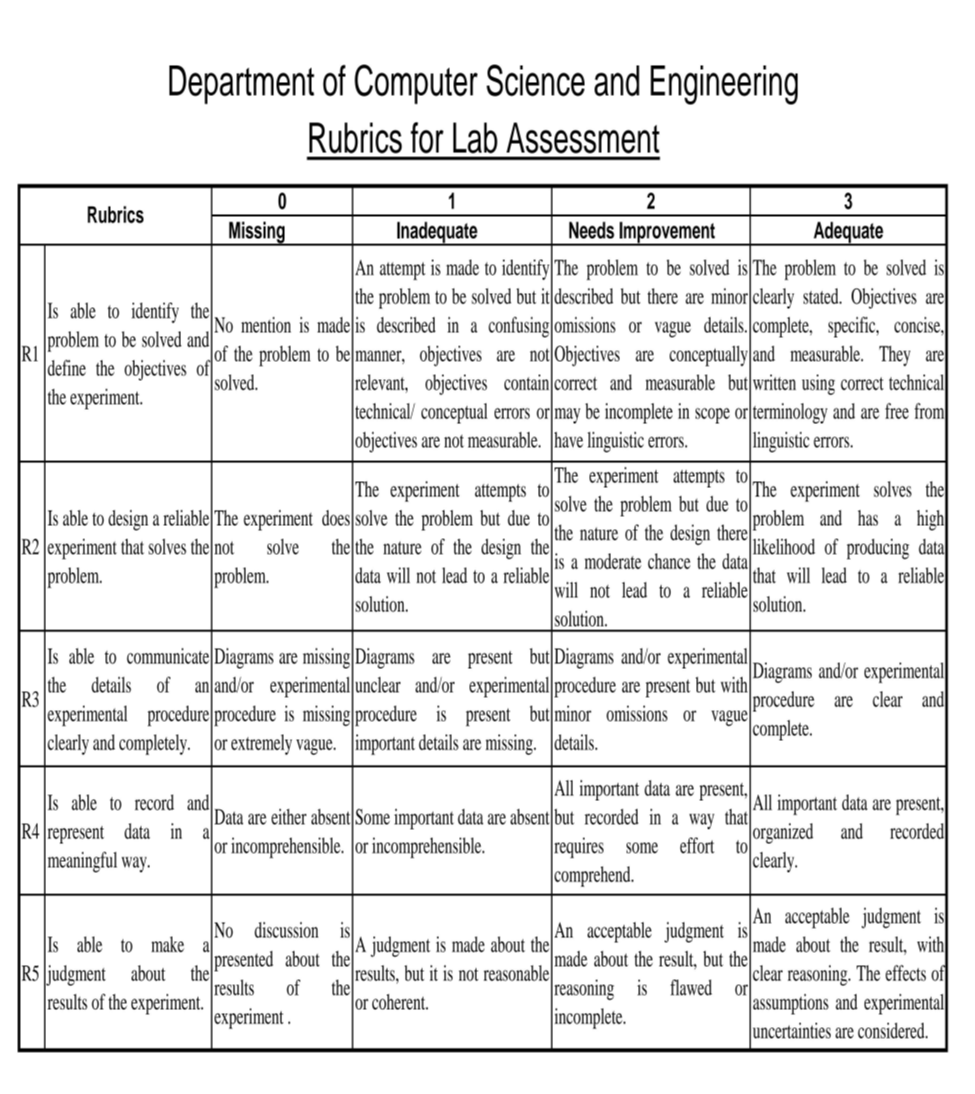
“To foster an open, multidisciplinary and highly collaborative research environment to produce world-class engineers capable of providing innovative solutions to real life problems and fulfill societal needs.”

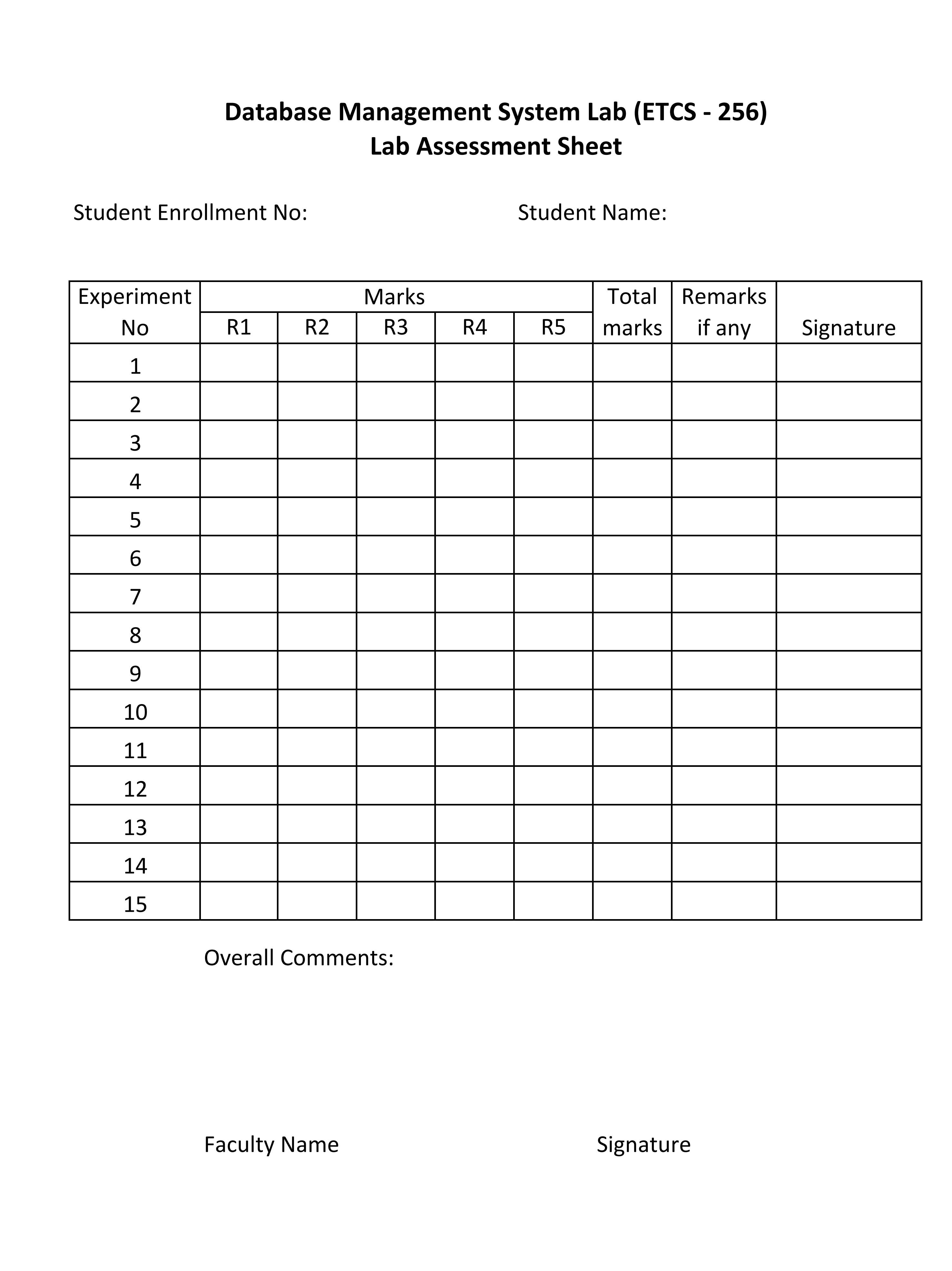
**LIST OF EXPERIMENTS (As prescribed by GGSIPU)**

****

**LIST OF EXPERIMENTS (Beyond the prescribed list of GGSIPU)**

****

****



**Kamarushi Rachit Kumar**

**00414802722**

**INDEX**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| SNO. | NAME OF EXPERIMENT | R1 | R2 | R3 | R4 | R5 | TOTAL MARKS | DATE OF CHECKING | SIGN |
|  |  |  |  |  |  |  |  |  |  |

Experiment 2

**AIM: -** Write a program to check whether a string belong to grammar or not.

**Production rules: -**

a) S -> aS

S -> Sb

S -> ab String form: aab

b) S -> aSa

S -> bSb

S -> a

S -> b String form: all length pallindrome

c) S -> aSbb

S -> abb String form: a^nb^2n

d) S -> aSb

S -> ab String form: anbn

**Theory: -**

Grammars:-

A grammar is a formal system that defines the syntax of a language through a set of production rules. These rules determine how strings in the language can be formed from an initial symbol, often called the start symbol. The string generation process involves repeatedly replacing symbols with other symbols or terminal symbols (which represent actual characters in the language).

Grammars are essential in the field of compiler design as they define the syntactic structure of programming languages. They are used in various stages of compilation, such as parsing and syntax analysis, to ensure that the source code adheres to the correct syntax before further processing.

Types of Grammars: -

Grammars are classified into the following types based on their complexity and generative power:

Regular Grammars: -

Regular grammars define regular languages, which can be recognized by finite automata. These grammars have production rules where the left-hand side contains a single non-terminal, and the right-hand side contains a terminal, possibly followed by a non-terminal. Regular grammars are equivalent to regular expressions.

Context-Free Grammars (CFGs): -

Context-Free Grammars define context-free languages, which can be recognized by pushdown automata. CFGs allow production rules where the left-hand side contains a single non-terminal and the right-hand side contains a string of terminals and/or non-terminals. CFGs are widely used to describe the syntax of programming languages, as they can model nested structures like parentheses or block statements.

Context-Sensitive Grammars (CSGs): -

Context-Sensitive Grammars describe context-sensitive languages, which can be recognized by linear-bounded automata. In CSGs, production rules can have context-sensitive restrictions, meaning that the replacement of a non-terminal may depend on its surrounding symbols.

Unrestricted Grammars: -

Unrestricted grammars describe recursively enumerable languages, which can be recognized by Turing machines. These grammars have the most general form of production rules, allowing for any combination of symbols on both sides of a rule.

Checking Whether a String Belongs to a Grammar: -

To determine whether a string belongs to a grammar, we follow these general steps:

1. Identify the Production Rules.

2. Start with the Initial Symbol.

3. Apply Production Rules Recursively.

4. Compare with the Input String.

5. Recursive Descent Parsing.

Example:

Let's consider the grammar `S → aSb | ab` and the string `aabb`:

1. Initial Symbol: Start with `S`.

2. Apply Production Rules:

- Apply `S → aSb`, resulting in `aSb`.

- Apply `S → ab` to `S` in the middle, resulting in `aabb`.

3. Comparison: The derived string `aabb` matches the input string exactly.

4. Conclusion: The string `aabb` belongs to the grammar.

And if the input string were `aaabb`, no sequence of applying the rules `S → aSb | ab` would produce this string, so `aaabb` does not belong to the grammar.

Code: -

#include <iostream>

#include <string>

bool checkGrammarA(const std::string &str) {

int aCount = 0, bCount = 0;

bool foundAb = false;

for (char ch : str) {

if (ch == 'a') {

if (foundAb) return false;

aCount++;

} else if (ch == 'b') {

if (aCount == 0) return false;

bCount++;

} else {

return false;

}

}

if (aCount > 0 && bCount > 0 && str.substr(str.length() - 2) == "ab") {

return true;

}

return false;

}

bool checkGrammarB(const std::string &str) {

int left = 0, right = str.length() - 1;

while (left < right) {

if (str[left] != str[right]) {

return false;

}

left++;

right--;

}

return true;

}

bool checkGrammarC(const std::string &str) {

int aCount = 0, bCount = 0;

int i = 0;

while (i < str.length() && str[i] == 'a') {

aCount++;

i++;

}

while (i < str.length()) {

if (str[i] != 'b') {

return false;

}

bCount++;

i++;

}

return (bCount == 2 \* aCount);

}

bool checkGrammarD(const std::string &str) {

int aCount = 0, bCount = 0;

int i = 0;

while (i < str.length() && str[i] == 'a') {

aCount++;

i++;

}

while (i < str.length() && str[i] == 'b') {

bCount++;

i++;

}

return (i == str.length()) && (aCount == bCount);

}

int main() {

std::string inputString;

char grammarChoice;

std::cout << "Enter the string to check: ";

std::cin >> inputString;

std::cout << "Enter the grammar to check (a, b, c, d): ";

std::cin >> grammarChoice;

switch (grammarChoice) {

case 'a':

std::cout << "Grammar A: " << (checkGrammarA(inputString) ? "Accepted" : "Rejected") << std::endl;

break;

case 'b':

std::cout << "Grammar B: " << (checkGrammarB(inputString) ? "Accepted" : "Rejected") << std::endl;

break;

case 'c':

std::cout << "Grammar C: " << (checkGrammarC(inputString) ? "Accepted" : "Rejected") << std::endl;

break;

case 'd':

std::cout << "Grammar D: " << (checkGrammarD(inputString) ? "Accepted" : "Rejected") << std::endl;

break;

default:

std::cout << "Invalid grammar choice!" << std::endl;

break;

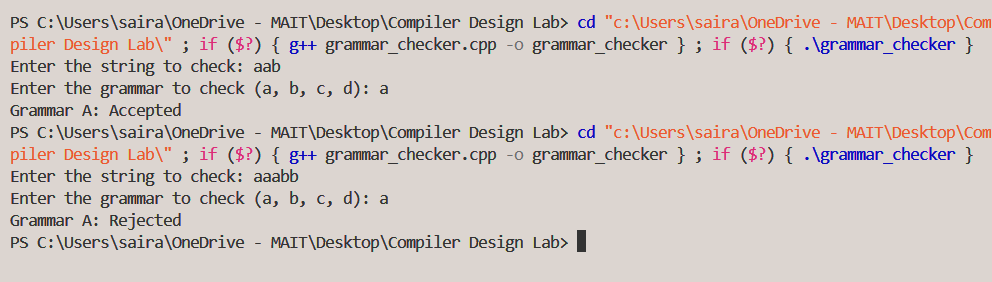
}

return 0;

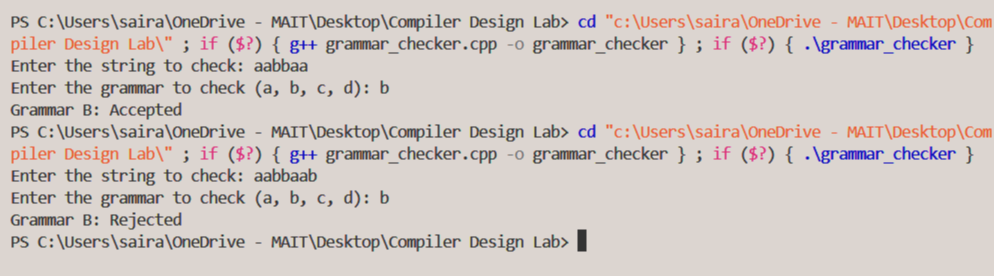
}

Output: -

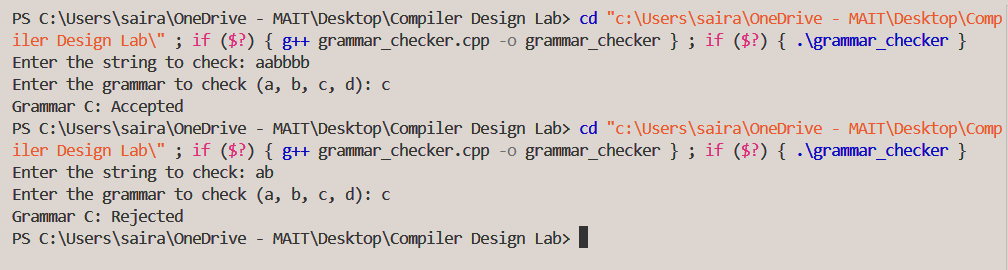
a)



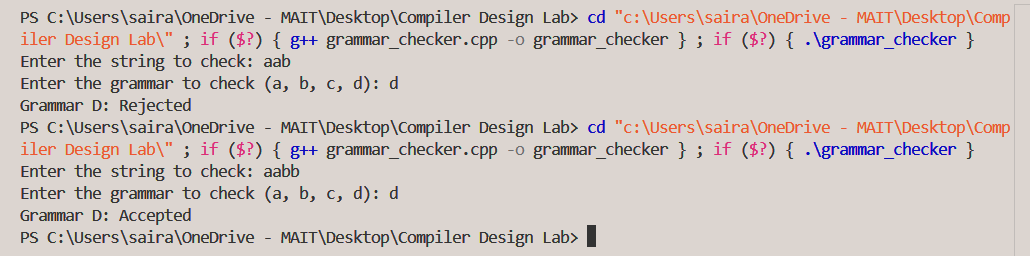
b)



c)



d)



Experiment 3

AIM: - Write a program to check whether a string include Keyword or not.

Theory: -

In C++, a keyword is a reserved word that has a special meaning to the compiler. Keywords are used to define the syntax and structure of C++ programs. They cannot be used as identifiers (such as variable names, function names, etc.) in a C++ program.

Common C++ keywords include:

int, float, double, char, void, bool

if, else, switch, case, default

for, while, do

break, continue, return

class, struct, public, private, protected

new, delete, try, catch, throw

namespace, using, static, const, typedef

To determine if a string contains a C++ keyword, we need to compare the input string against a predefined list of keywords. The comparison should be case-sensitive as C++ keywords are case-sensitive.

Code: -

#include <iostream>

#include <string>

#include <unordered\_set>

#include <sstream>

#include <vector>

void findKeywords(const std::string& input, const std::unordered\_set<std::string>& keywords,

std::unordered\_set<std::string>& foundKeywords, int& keywordCount) {

std::istringstream stream(input);

std::string word;

while (stream >> word) {

if (keywords.find(word) != keywords.end()) {

if (foundKeywords.find(word) == foundKeywords.end()) {

foundKeywords.insert(word);

keywordCount++;

}

}

}

}

int main() {

// Define a set of C++ keywords

std::unordered\_set<std::string> cppKeywords = {

"alignas", "alignof", "and", "and\_eq", "asm", "atomic\_cancel", "atomic\_commit",

"atomic\_noexcept", "auto", "bitand", "bitor", "bool", "break", "case", "catch",

"char", "char16\_t", "char32\_t", "class", "compl", "concept", "const", "consteval",

"constexpr", "constinit", "const\_cast", "continue", "decltype", "default", "delete",

"do", "double", "dynamic\_cast", "else", "enum", "explicit", "export", "extern",

"false", "float", "for", "friend", "goto", "if", "inline", "int", "long", "mutable",

"namespace", "new", "noexcept", "not", "not\_eq", "nullptr", "operator", "or",

"or\_eq", "private", "protected", "public", "register", "reinterpret\_cast", "requires",

"return", "short", "signed", "sizeof", "static", "static\_assert", "static\_cast", "struct",

"switch", "template", "this", "thread\_local", "throw", "true", "try", "typedef", "typeid",

"typename", "union", "unsigned", "using", "virtual", "void", "volatile", "wchar\_t",

"while", "xor", "xor\_eq"

};

std::string input;

std::cout << "Enter a string to check for C++ keywords: ";

std::getline(std::cin, input);

std::unordered\_set<std::string> foundKeywords;

int keywordCount = 0;

findKeywords(input, cppKeywords, foundKeywords, keywordCount);

if (keywordCount > 0) {

std::cout << "The string contains " << keywordCount << " C++ keywords:" << std::endl;

for (const auto& keyword : foundKeywords) {

std::cout << "- " << keyword << std::endl;

}

} else {

std::cout << "The string does not contain any C++ keywords." << std::endl;

}

return 0;

}

Output: -

