**Programming Assignment-3**

1. Logical Consequences: The logical consequences of the knowledge base (KB) are:

a, b, c, d, e, f, g, h, j, k

1. Model where f is false: A model in which 'f' is false can be constructed by assigning truth values to the propositions while still satisfying the given KB. For example, in the model where 'f' is false:

a = True, b = True, c = True, d = True, e = True, f = False, g = True, h = True, j = True, k = True

1. Top-down derivation for ask a: Using a top-down approach, we successfully derived the conclusion that 'a' is true based on the rules and facts in the KB.

**Code Explaination:**  
#include <iostream>

#include <unordered\_set>

#include <vector>

#include <string>

#include <map>

#include <unordered\_map>

using namespace std;

/\*\*

\* @brief Represents a rule in the knowledge base.

\*

\* A rule consists of a head (the fact to be inferred) and a body (the conditions that need

\* to be true for the head to be inferred). The rule can be thought of as "If all the conditions

\* in the body are true, then the head is also true."

\*/

struct Rule {

vector<string> body; // Conditions that need to be true for the head to be inferred.

string head; // The fact (head) that can be inferred when all conditions (body) are true.

};

/\*\*

\* @brief Checks if all conditions (body) of a rule are true (i.e., present in the set of known facts).

\*

\* @param body The conditions (body) of the rule.

\* @param facts The set of known facts (truths) that have been inferred so far.

\* @return true If all conditions in the body are true.

\* @return false If any condition in the body is not true.

\*/

bool bodySatisfied(const vector<string>& body, const unordered\_set<string>& facts) {

for (const string& literal : body) {

// If any condition in the body is not found in the known facts, return false.

if (facts.find(literal) == facts.end()) {

return false;

}

}

// If all conditions are found in the known facts, return true.

return true;

}

/\*\*

\* @brief Bottom-up proof procedure to infer all logical consequences from a knowledge base.

\*

\* This function takes the knowledge base (a set of rules) and known initial facts. It iteratively

\* applies the rules to infer new facts until no more new facts can be inferred.

\*

\* @param kb The knowledge base, represented as a list of rules.

\* @return The set of all facts that can be inferred from the knowledge base.

\*/

unordered\_set<string> bottomUpProof(const vector<Rule>& kb) {

// Set of known facts initialized with some basic facts (c, e, h, and k are known to be true).

unordered\_set<string> facts = {"c", "e", "h", "k"};

cout << "Initial facts: c, e, h, k" << endl;

// Track whether new facts are inferred in the current iteration.

bool addedNewFact = true;

// Keep iterating while new facts are being added.

while (addedNewFact) {

addedNewFact = false; // Reset flag for each iteration.

// Iterate over all rules in the knowledge base.

for (const Rule& rule : kb) {

// Check if all conditions in the rule's body are true (i.e., if the body is "satisfied").

if (bodySatisfied(rule.body, facts)) {

// If the rule's head is not already known (i.e., not in the set of facts).

if (facts.find(rule.head) == facts.end()) {

// Add the rule's head (fact) to the known facts.

facts.insert(rule.head);

// Print the newly inferred fact and the rule that was applied.

cout << "Inferred new fact: " << rule.head << " using rule: " << rule.head << " ← ";

for (size\_t i = 0; i < rule.body.size(); ++i) {

cout << rule.body[i];

if (i < rule.body.size() - 1) {

cout << " ∧ "; // Print "and" symbol between conditions.

}

}

cout << endl;

// Set flag to true since a new fact was added.

addedNewFact = true;

}

}

}

}

// Return the complete set of inferred facts.

return facts;

}

/\*\*

\* @brief Top-down proof procedure to prove a query by working backwards through the rules.

\*

\* This function attempts to prove a specific query (fact) by recursively proving the conditions

\* (body) of rules that infer the query. This is also known as "backward chaining."

\*

\* @param kb The knowledge base, represented as a list of rules.

\* @param query The fact (head) that we are trying to prove.

\* @param visited The set of facts that have already been proven or assumed to be true.

\* @return true If the query can be proven from the knowledge base.

\* @return false If the query cannot be proven.

\*/

bool topDownProof(const vector<Rule>& kb, const string& query, unordered\_set<string>& visited) {

// If the query is already known (i.e., it has been proven or is a known fact).

if (visited.find(query) != visited.end()) {

return true; // The query is true.

}

// Try to find a rule where the head matches the query.

for (const Rule& rule : kb) {

if (rule.head == query) {

// Print the rule being used to try to prove the query.

cout << "Attempting to prove: " << query << " using rule: " << rule.head << " ← ";

for (size\_t i = 0; i < rule.body.size(); ++i) {

cout << rule.body[i];

if (i < rule.body.size() - 1) {

cout << " ∧ "; // Print "and" symbol between conditions.

}

}

cout << endl;

// Check if all conditions in the body of the rule can be proven.

bool canProve = true;

for (const string& literal : rule.body) {

// Recursively attempt to prove each condition.

if (!topDownProof(kb, literal, visited)) {

cout << "Failed to prove: " << literal << endl;

canProve = false; // If any condition can't be proven, the query fails.

break;

}

}

// If all conditions can be proven, add the query to the set of known facts.

if (canProve) {

cout << "Successfully proved: " << query << endl;

visited.insert(query); // Mark the query as proven.

return true; // The query is true.

}

}

}

// If no rule can be found to prove the query, return false.

cout << "Failed to prove: " << query << endl;

return false;

}

int main() {

// Define the knowledge base (KB) as a list of rules. Each rule has a head (fact) and a body (conditions).

// For example, "a ← b ∧ c" means "a is true if both b and c are true."

vector<Rule> kb = {

{{"b", "c"}, "a"}, // Rule: a ← b ∧ c

{{"d"}, "b"}, // Rule: b ← d

{{"e"}, "b"}, // Rule: b ← e

{{"c", "k"}, "g"}, // Rule: g ← c ∧ k

{{"g", "b"}, "f"}, // Rule: f ← g ∧ b

{{"a", "b"}, "j"}, // Rule: j ← a ∧ b

{{"h"}, "d"} // Rule: d ← h

};

// Step 1: Perform the bottom-up proof procedure to infer all facts from the knowledge base.

cout << "---- Bottom-up Proof Procedure ----" << endl;

unordered\_set<string> facts = bottomUpProof(kb);

// Output all the facts that were inferred.

cout << "\nAll logical consequences of KB:" << endl;

for (const string& fact : facts) {

cout << fact << endl;

}

// Step 2: Provide a model where the fact 'f' is false.

cout << "\n---- Model where f is false ----" << endl;

unordered\_map<string, bool> model = {

{"a", true}, {"b", true}, {"c", true}, {"d", true}, {"e", true},

{"f", false}, {"g", true}, {"h", true}, {"j", true}, {"k", true}

};

// Print the truth value of each fact in the model.

for (const auto& item : model) {

cout << item.first << " = " << (item.second ? "True" : "False") << endl;

}

// Step 3: Perform the top-down proof procedure to prove the query 'a'.

cout << "\n---- Top-down Proof for Query 'a' ----" << endl;

string query = "a"; // The fact we are trying to prove.

unordered\_set<string> visited = {"c", "e", "h", "k"}; // Known initial facts.

// Try to prove the query 'a' using the top-down proof procedure.

if (topDownProof(kb, query, visited)) {

cout << "The query '" << query << "' is a logical consequence of KB." << endl;

} else {

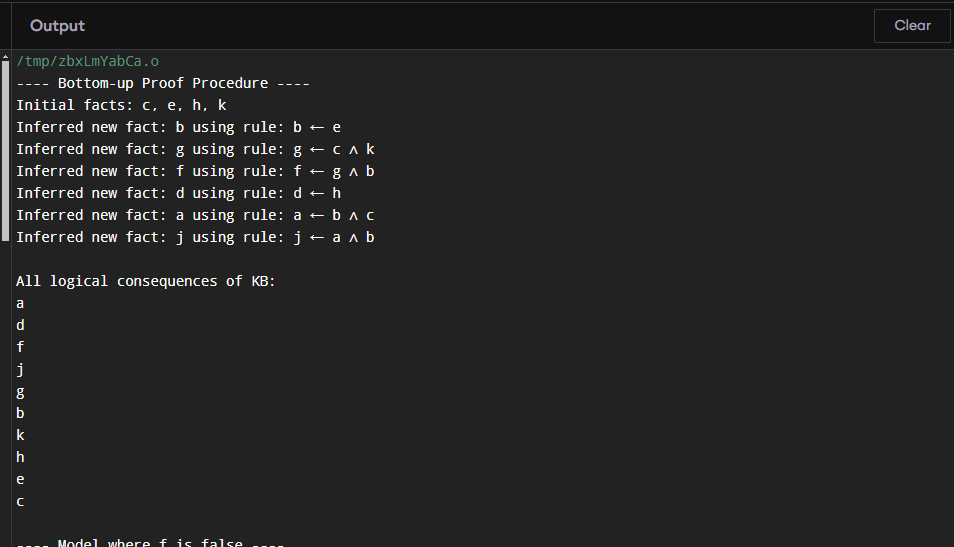
cout << "The query '" << query << "' is NOT a logical consequence of KB." << endl;

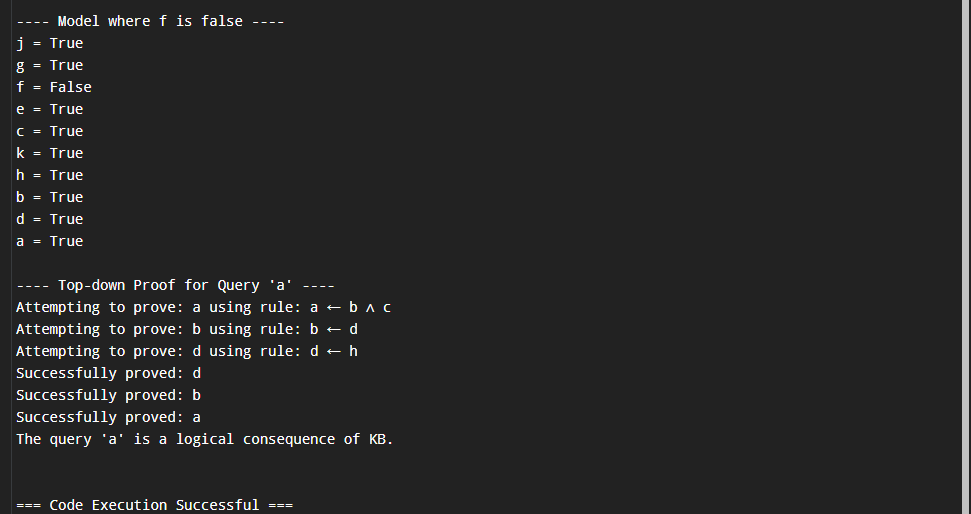
}

return 0;

}  
**How to Run:**

* Copy the C++ programs into the file.
* Compile and run using a C++ compiler (g++ or any other).
* The bottom-up proof procedure will output all logical consequences of the knowledge base.
* The top-down proof procedure will check whether the query ‘a’ is a logical consequence.

**Output:  
**

****