

Discrete Structure Project

Deadline: 25th November, 2025 11:59PM

Automated Logical Reasoning System (ALRS)

Important Instructions

- Use **pure procedural C++** only (functions, arrays, loops, recursion; no OOP or STL).
- Program must compile without errors.
- Write clean, modular, and well-commented code with meaningful variable and function names.
- Use single uppercase letters (P, Q, R...) for variables and correct logical operators (!, &, |, >, =).
- Ensure proper use of parentheses; invalid input must show an error message and re-prompt.
- Provide a clear console menu for all operations with formatted output tables. After each operation, ask "Do you want to save results? (Y/N)" and return to the main menu. Test the program with at least three cases: truth table, argument validation, and equivalence check.
- Submit all files as a single ZIP named `Name_<Section>_<RollNo>.zip`. The ZIP must include source code, README.txt, three saved outputs, and a short project report.
- Projects must be done individually.
- Late submissions will receive mark deductions according to course policy.
- In case of plagiarism F grade will be awarded.

- If you have any query regarding assignment you can contact TA Muhammad Awais Rafique email: i222511@nu.edu.pk.

Overview

Students will design and implement a **procedural C++ program** named *Automated Logical Reasoning System (ALRS)*. ALRS is a mini logic engine that allows a user to:

- Input propositional variables and complex compound logical expressions.
- Parse and evaluate logical expressions using standard operators.
- Generate full truth tables (including intermediate/sub-expression columns).
- Validate multi-premise arguments using the truth-table method and provide counterexamples when invalid.
- Check logical equivalence between expressions.
- Simulate reasoning chains (proof steps) and output derived implications.
- Save results (truth tables, validation outcomes, counterexamples) to files in **.txt** and **.csv** formats.

Students will implement the entire system using **procedural programming only** (functions, arrays, loops, recursion permitted). No classes or OOP features should be used.

Required Features (Core)

Each student solution must implement the following modules:

1. User Input & Expression Syntax

- Accept user input for propositional variables (e.g., **P Q R**) and/or detect variables automatically from expressions.
- Accept compound expressions using these symbols:
 - **!** for NOT (unary)
 - **&** for AND

- `|` for OR
- `>` for IMPLIES (if $A > B$ then $A \rightarrow B$)
- `=` for EQUIVALENT ($A = B$)
- Parentheses `()` must be supported to dictate precedence.
- Allow multiple statements: premises and a conclusion for argument testing.

2. Expression Parser

- Implement a parser that converts an infix logical expression into a data structure that can be evaluated repeatedly for different truth assignments.
- Recommended approach: **Shunting-yard algorithm** to convert infix to postfix (Reverse Polish Notation). Use arrays or stack-simulation via arrays.
- Provide clear error messages for syntax errors (unmatched parentheses, unknown tokens, consecutive operators, etc.).

3. Expression Evaluator

- Evaluate postfix expressions for a given assignment of propositional variables.
- Use boolean values (`0` for false, `1` for true).
- Hint:
- Implement helper functions for each operator:
 - `bool op_not(bool a)`
 - `bool op_and(bool a, bool b)`
 - `bool op_or(bool a, bool b)`
 - `bool op_implies(bool a, bool b)` (interpretation: $A \rightarrow B$ is `!A || B`)
 - `bool op_equiv(bool a, bool b)`

4. Truth Table Generator

- For a given expression (or for a set of expressions/premises and a conclusion), generate the full truth table with `2^n` rows where `n` is the

number of unique propositional variables (support up to $n=5$ by requirement).

- Show columns for each propositional variable, intermediate sub-expressions (optional but **required for full credit** for at least one complex expression), and the final expression.
- Implement binary-counting style generation of truth assignments using loops.

5. Argument Validator (Multi-Premise)

- Let the user enter up to 5 premises and 1 conclusion.
- Use the truth-table method for validation:
 - For each row where **all premises** evaluate to true, check whether the conclusion is also true.
 - If you find any row where premises are all true and conclusion is false, the argument is **invalid**. Print that row as a **counterexample**.
 - If no such row exists, then the argument is **valid**.

6. Logical Equivalence Checker

- Allow the user to enter two expressions and check whether they are logically equivalent by comparing their truth columns across all combinations.
- If not equivalent, print rows where they differ.

7. Reasoning Chain / Proof Simulator

- Accept a user-specified set of implications/premises and attempt to derive direct implications (transitive closure-like inference) using basic inference rules such as **hypothetical syllogism** (if $P \rightarrow Q$ and $Q \rightarrow R$ then $P \rightarrow R$) and **modus ponens** (if P and $P \rightarrow Q$ then Q).
- Output the derived implications step-by-step until no new conclusions can be drawn or until a user-specified target conclusion is reached.

8. File Handling (Persistence)

- Save truth tables and results to a **.txt** or **.csv** file in a clear format.

- Required saved items:
 - Header with timestamp and list of variables.
 - The truth table rows.
 - Final verdict about validity/equivalence and any counterexamples.
- Provide a user option to save the output after each major operation.

9. User Interface (Console)

- Use clear, well-formatted console output (tables with column headers, dividers).
- Provide menus/options for the user to select tasks (Generate truth table, Validate argument, Check equivalence, Reasoning chain, View previous saved results).

Detailed Step-by-Step Implementation Plan

This section breaks the project into sequential implementation steps so students know what to build and test at each stage.

Step 1 — Tokeniser & Basic Input Handling

- Build a function to read a line and split it into tokens: variables (single uppercase letters), operators (`! & | > =`), parentheses (`(` and `)`).
- Normalize input: remove spaces, ensure operators are single-char tokens.
- Validate tokens and reject invalid characters.
- Test cases:
 - Valid: `(P&Q)>R` , `!P|Q` , `(A > (B & C))`
 - Invalid: `P && Q` (double-ampersand not supported), `P >` (trailing operator), `((P|Q)` (unmatched parenthesis)

Functions to implement: (Hint)

- `int tokenize(const char *input, char tokens[][4])` — fills `tokens` and returns token count.

Step 2 — Infix to Postfix Conversion (Shunting-yard)

- Implement the shunting-yard algorithm using arrays to simulate stacks.
- Define operator precedence: `!` highest (unary), `&`, `|`, `>`, `=` lowest; parentheses override.
- Output should be a postfix token list suitable for evaluation.

Functions to implement: (Hint)

- `int infix_to_postfix(char tokens[][4], int token_count, char output[][4])`
 - Error detection: mismatched parentheses, operator misuse.
-

Step 3 — Postfix Evaluator

- Implement evaluation of the postfix list for a given map of variable truth values.
- Use a boolean stack (array) to push operand values and apply operators.

Functions to implement: (Hint)

- `bool evaluate_postfix(char postfix[][4], int postfix_len, int var_count, char vars[], int values[])`
 - Helper operator functions (from Required Features).
-

Step 4 — Truth Table Generator

- Given `n` variables (maximum 5), loop from `0` to `2^n - 1` and use bit operations to assign truth values.
- Evaluate the expression for each assignment and print the row.
- Optionally compute and print intermediate sub-expression values by re-evaluating subexpressions or by storing sub-expression tokens during parsing.

Functions to implement: (Hint)

- `void generate_truth_table(char expr[], char vars[], int var_count, char postfix[][4], int postfix_len)`
-

Step 5 — Argument Validator

- Accept up to `m` premises and 1 conclusion.
- For each truth assignment, evaluate all premises and conclusion.
- If a row exists where all premises are `true` and conclusion is `false`, record that row as a counterexample and mark invalid.
- Print final verdict and counterexample (if invalid).

Functions to implement: (Hint)

- `bool validate_argument(char premises[][MAXTOK], int pcount, char conclusion[], char vars[], int var_count, ...)`

Step 6 — Logical Equivalence Checker

- Generate truth columns for both expressions and compare them row-by-row.
- Report equivalence or list differing rows.

Functions to implement: (Hint)

- `bool check_equivalence(char expr1[], char expr2[], char vars[], int var_count, ...)`

Step 7 — Reasoning Chain Simulator

- Represent implications (`P→Q`) from premises in a small adjacency matrix or adjacency list (2D array) among variables.
- Use derived relations to provide step-wise explanations (e.g., `From P→Q and Q→R infer P→R`).

Functions to implement: (Hint)

- `void build_implication_matrix(char premises[][MAXTOK], int pcount, char vars[], int var_count, int mat[][MAXVARS])`
- `void derive_transitive_closure(int mat[][MAXVARS], int var_count)`
- `void print_derivation_steps(...)`

Step 8 — File Handling (Saving Results)

Students should implement both **TXT** and **CSV** saving options. Examples and format below.

TXT File Format (human-readable): (Sample)

```
ALRS Output
Timestamp: 2025-11-09 17:00:00
Variables: P Q R
Expression: (P & Q) > R
-----
P Q R | (P & Q) | R | (P & Q) > R
T T T |   T   | T |      T
T T F |   T   | F |      F
... (other rows)
-----
Verdict: Invalid
Counterexample: P=T Q=T R=F
```

CSV File Format (spreadsheet-friendly): (Sample)

- First row: header with variable names and expression columns. Example:

```
P,Q,R,(P&Q),R,(P&Q)>R
1,1,1,1,1
1,1,0,1,0,0
...
```

- If multiple expressions/premises are saved, separate them with a blank line or write separate files.

Functions to implement: (Hint)

- `void save_as_txt(const char *filename, ...)`
- `void save_as_csv(const char *filename, ...)`

User Menu Flow (Hint: but you can use your own Menu)

1. Main Menu

- 1. Generate truth table for expression
 - 1. Validate an argument (premises \rightarrow conclusion)
 - 1. Check equivalence of two expressions
 - 1. Reasoning Chain / Derive implications
 - 1. View / Load saved results
 - 1. Exit
2. After selecting an operation, program asks for expressions and variables (or detects them automatically).
 3. Program displays results and asks whether to save them to TXT/CSV.
-

Input/Output Examples

Example 1 — Truth Table

Automated Logical Reasoning System (ALRS)

Enter number of statements: 3

Statement 1: $P > Q$

Statement 2: $Q > R$

Statement 3: $R > S$

Conclusion: $P > S$

Analyzing argument validity...

Truth Table (Partial View):

P	Q	R	S	P>Q	Q>R	R>S	P>S	Result
---	---	---	---	-----	-----	-----	-----	--------

T	T	T	T	T	T	T	T	VALID
---	---	---	---	---	---	---	---	-------

T	T	T	F	T	T	F	F	INVALID ❌
---	---	---	---	---	---	---	---	-----------

...

✅ Final Verdict: The argument is INVALID.

Counterexample found: P=T, Q=T, R=T, S=F

Error Handling & Edge Cases

- Reject more than 5 unique variables with a helpful message (Maximum of 5 propositional variables supported).
- Detect and reject malformed expressions.
- Handle empty input gracefully.
- When saving to a file, handle file opening errors and report status to the user.

Project Submission Requirements

Each student must submit:

1. Source code file(s) (.cpp and .h if used)
2. A README with:
 - Brief description of program features
 - How to compile and run (example commands)
 - Any known limitations
3. At least 3 sample saved result files (TXT or CSV) demonstrating:
 - A truth table save
 - An argument validation save
 - An equivalence test save
4. A short report summarizing implementation decisions and challenges.

Two Bonus Tasks (for extra credit)

Bonus 1 — Import / Export Expressions & Batch Mode

- Allow the program to import a batch of expressions/premises from a `.txt` or `.csv` file and process them in sequence, generating a results file for each. This is useful for automated testing and grading: instructors can create a single file containing 20 test expressions and run them all.

Bonus 2 — Create a “Logic Puzzle” mode that randomly generates **logical riddles** or **equivalence problems** for the user to solve.

For example:

Puzzle: Which of the following expressions is logically equivalent to $!(P \ \& \ Q)$?

A) $!P \mid !Q$

B) $P \mid Q$

C) $P \ \& \ !Q$

D) $!(P \mid Q)$

- The user selects an answer.
- The program checks the equivalence and shows the correct solution with explanation.