LOGIC ASSIGNMENT-MANUAL WHICH EXPLAINS OUR CODE

Group Members: Punya Pratap Singh (2023B3A70280H), Tarun Pradeep Krishnamurthy (2023B3A70398H), Aditya Bhat (2023B3A70448H), Aryaman Srivastava (2023B5A70764H), Lohit Vijayabaskar (2023B5A70778H)

The given document helps the reader understand the various functions present in the code, the type of inputs each of the functions take and what is the actual work flow of the code. The descriptions of each of the functions has been divided into following sections;

1. **Function Header:** Shows what the return type of the function is.
2. **What the function takes in as input:** Shows the type of variable the function takes in as input.
3. **What the function generates as an output:** Shows the type of variable the function generates as an output.
4. **Example:** An example of the input the function takes in and the output it generates.

Apart from this, to test the speed of the respective tasks, we ran the respective functions for each of the tasks for the simple expression (P+Q)*R and calculated the computation and total time. We have also added an additional table in the end which helps understand the time and space complexity of each of the tasks and this helps understand the efficiency and memory usage of the given program.

Task 1: Infix to Prefix Conversion

🕒	**Computation Time:**45 micro seconds

Function: infixToPrefix

Function Name	Description
string infixToPrefix(string infix)	

Input:	**infix (string)** — An infix expression containing variables (`a-z`), operators (`+`, `*`, `>`, `~`), and parentheses
Output:	Returns a string representing the **prefix notation** of the input expression
Example 1:	**Input:** `"(p+q)"` **Output:** `"+pq"`
Example 2:	**Input:** `"~(p*q)"` **Output:** `"~*pq"`
Example 3:	**Input:** `"(p>q)"` **Output:** `">pq"`

Helper Function: getPrecedence

Function Name	Description
int getPrecedence(char op)	
Input:	**op (char)** — An operator character (`>`, `+`, `*`, `~`)
Output:	Returns an integer representing operator precedence (**1–4**, where **4** is highest)
Example 1:	**Input:** `'>'` **Output:** `1`
Example 2:	**Input:** `'~'` **Output:** `4`

Helper Function: findMainOperator

Function Name	Description
int findMainOperator(const string& infix)	
Input:	**infix (string)** — An infix expression to analyze

Output:	Returns the index position of the main operator (lowest precedence outside parentheses), or **-1** if none found
Example 1:	**Input:** `"p+q*r"` **Output:** `1` (position of `'+'`)
Example 2:	**Input:** `"(p+q)"` **Output:** `-1` (fully enclosed)

Helper Function: isFullyEnclosed

Function Name	Description
bool isFullyEnclosed(const string& infix)	
Input:	**infix (string)** — An expression to check for enclosing parentheses
Output:	Returns **true** if the entire expression is enclosed in matching outer parentheses, **false** otherwise
Example 1:	**Input:** `"(p+q)"` **Output:** `true`
Example 2:	**Input:** `"(p)+(q)"` **Output:** `false`

Task 2: Build Parse Tree from Prefix

🕒	• **Computation Time:** 8 micro seconds

Function: buildParseTree

Function Name	Description
Node* buildParseTree(const string& prefix)	
Input:	**prefix (string)** — A prefix expression without spaces
Output:	Returns a pointer to the **root Node** of the constructed parse tree
Example 1:	**Input:** `"+pq"` **Output:**Tree structure: Tree structure for +pq
Example 2:	**Input:** `"~p"` **Output:**Tree structure for ~p

Helper Function: buildParseTreeRecursive

Function Name	Description
Node* buildParseTreeRecursive(const string& prefix, int& index)	
Input:	**prefix (string)** — The prefix expression**index (int&)** — Reference to the current position being processed
Output:	Returns a **Node pointer** representing the subtree starting at the current index
Example:	**Input:** `prefix = "+pq"`, `index = 0` **Output:** Root node with `'+'`, left child `'p'`, right child `'q'`

Task 3: Convert Parse Tree to Infix

🕒	**Computation Time:** 21 micro seconds

Function: parseTreeToInfix

Function Name	Description
string parseTreeToInfix(Node* root)	
Input:	**root (Node*)** — Pointer to the root of a parse tree
Output:	Returns a **fully parenthesized infix expression** as a string
Example 1:	**Input:** Tree for `"+pq"` **Output:** `"(p+q)"`
Example 2:	**Input:** Tree for `"~p"` **Output:** `"(~p)"`
Example 3:	**Input:** Tree for `"*+pq~r"` **Output:** `"((p+q)*~r)"`

Task 4: Calculate Parse Tree Height

🕒	• **Computation Time:** 2 micro seconds • **Total Time (input + computation + output):** 54805 micro seconds

Function: getTreeHeight

Function Name	Description

int getTreeHeight(const Node* node)

Input:	**node (const Node*)** — Pointer to the root of a parse tree
Output:	Returns the **height of the tree** (number of levels from root to deepest leaf)
Example 1:	**Input:** Tree for `"+pq"` **Output:** `2`
Example 2:	**Input:** Tree for `"*+pqr"` **Output:** `3`

Task 5: Evaluate Truth Value

Computation Time: 1 micro seconds

Function: evaluate

Function Name	Description
bool evaluate(Node* root, const unordered_map<char, bool>& values)	
Input:	**root (Node*)** — Pointer to the root of a parse tree**values (unordered_map<char, bool>&)** — Map of variable names to their truth values
Output:	Returns **true** or **false** based on the logical evaluation of the formula
Example 1:	**Input:** Tree for `"+pq"` **Output:** The corresponding result will be T/F depending on the values given for p and q

Helper Function: getVariables

Function Name	Description
void getVariables(Node* root, set& vars)	
Input:	**root (Node*)** — Pointer to the root of a parse tree**vars (set&)** — Reference to a set to store variable names
Output:	Populates the **vars set** with all unique variables found in the tree (no return value)
Example:	**Input:** Tree for `"+*pqr"` **Output:** Gives the vars list with all unique variables which are p,q,r here

---

Task 6: Convert to CNF

🕒	**Computation Time:** 13 micro seconds

Function: impl_free

Function Name	Description
Node* impl_free(Node* root)	
Input:	**root (Node*)** — Pointer to the root of a parse tree containing implications (`>`)
Output:	Returns a pointer to a **modified tree** with all implications replaced by `(~P + Q)`
Example:	**Input:** Tree for `">pq"` (p → q)

Output: Tree for (~p+q)

Function: nnf

Function Name	Description
Node* nnf(Node* root)	
Input:	**root (Node*)** — Pointer to an implication-free parse tree
Output:	Returns a pointer to a tree in **Negation Normal Form (NNF)** — negations appear only at literals
Example:	**Input:** Tree for `"~*pq"` (~(p * q)) **Output:** Tree for (~p + ~q)

Function: cnf

Function Name	Description
Node* cnf(Node* root)	
Input:	**root (Node*)** — Pointer to a tree in **Negation Normal Form (NNF)**
Output:	Returns a pointer to a tree in **Conjunctive Normal Form (CNF)** — AND of ORs
Example:	**Input:** Tree for `"+p*qr"` (p + (q * r)) **Output:** Tree for (p*q) + (p*r)

Helper Function: distribute

Function Name	Description
Node* distribute(Node* a, Node* b)	
Input:	**a (Node*)** — Left subtree **b (Node*)** — Right subtree
Output:	Returns a **distributed tree** applying the law: (A + (B * C)) = ((A + B) * (A + C))
Example:	**Input:** `a = 'p'`, `b = `Tree for `"*qr"` **Output:** Tree for "*+pq+pr"

Task 7: Check CNF Validity

🕒	**Computation Time:** 50 micro seconds

Function: check_cnf_valid

Function Name	Description
bool check_cnf_valid(string formula)	
Input:	**formula (string)** — A CNF formula in string format (e.g., `"(p + ~p) * (q + ~q)"`)
Output:	Returns **true** if the formula is a **tautology** (always valid), **false** otherwise
Example 1:	**Input:** `"(p + ~p)"` **Output:** Number of false clauses `: 0` Number of true clauses `: 1` The CNF formula is valid (a tautology).

Example 2:	**Input:** `"(p + q) * (~p + q)"` **Output:** Number of false clauses `: 2` Number of true clauses `: 0` The CNF formula is NOT valid.

Helper Function: is_clause_true

Function Name	**Description**
bool is_clause_true(string clause)	
Input:	**clause (string)** — A single disjunctive clause (e.g., `"p + ~p"`)
Output:	Returns **true** if the clause is a **tautology** (contains both P and ~P), **false** otherwise
Example 1:	**Input:** `"p + ~p"` **Output:** `true`
Example 2:	**Input:** `"p + q"` **Output:** `false`

Utility Functions

Function: isOperator

Function Name	**Description**
bool isOperator(char c)	
Input:	**c (char)** — A character to check

Output:	Returns **true** if the character is an operator (`+`, `*`, `>`, `~`), **false** otherwise
Example 1:	**Input:** `'+'` **Output:** `true`
Example 2:	**Input:** `'p'` **Output:** `false`

Function: deleteTree

Function Name	Description
void deleteTree(Node* node)	
Input:	**node (Node*)** — Pointer to the root of a tree to delete
Output:	Recursively deallocates **all nodes** in the tree (no return value)
Example:	**Input:** Tree with 5 nodes **Output:** All memory freed, tree destroyed

Function: copyTree

Function Name	Description
Node* copyTree(Node* root)	
Input:	**root (Node*)** — Pointer to the root of a tree to copy
Output:	Returns a pointer to a **deep copy** of the tree
Example:	**Input:** Original tree for `"+pq"` **Output:** New independent tree with structure similar to this

Function: printTree

Function Name	Description
void printTree(const Node* root)	
Input:	**root (const Node*)** — Pointer to the root of a parse tree
Output:	Prints a **visual ASCII representation** of the tree to the console (no return value)
Example:	**Input:** Tree for `"+pq"` **Output:** The tree for given input is printed

---

Task 8 (a): Checking DIMACS CNF Validity using SAT Solver 2002 Data

Functionality	Description
Input:	The user enters a **DIMACS CNF formula** via console (paste multiple lines, end with a blank line). Each line follows the DIMACS format for CNF clauses.
Process:	1. Reads all lines of DIMACS input. 2. Parses the input using `readDIMACSCNF(dimacsInput, numVars, numClauses)` to extract clauses. 3. Measures computation time using `high_resolution_clock`.4. Checks if the parsed CNF formula is a **tautology** (always true) via `check_dimacs_valid(formula)`.
Output:	• Total number of parsed clauses. • Whether the formula is **valid (tautology)** or **not valid**. • **Computation Time** (in microseconds). • **Total Time** (in milliseconds).

Example: **Input (DIMACS CNF):** The input is lines of DIMACS CNF clauses which is of the usual SAT solver competition format (year is 2002).
Output: The output is a combination of number of parsed clauses, whether formula is valid or not and computation time.

TASK 8b): Analyzing the memory usage, Time taken and efficiency of the code

Task	Output	Time Complexity	Space Complexity
Infix → Prefix	Prefix string	$O(n^2)$	$O(n)$
Prefix → Parse Tree	Tree structure	$O(n)$	$O(n)$
Tree Printing	ASCII visual tree	$O(n^2)$	$O(n^2)$
Evaluate Formula	Truth value	$O(n)$	$O(n)$
Generate Truth Table	2^n evaluations	$O(n \cdot 2^n)$	$O(n)$
CNF Conversion	CNF formula	$O(2^n)$	$O(2^n)$
CNF Validity Check	Valid/Invalid + stats	$O(n^2)$	$O(n)$