CS21001 : Discrete Structures (Autumn 2020)

Coding Assignment 1 : Propositional Logic – Representation and Deduction

Due Date: 01-November-2020, 11:59PM (IST) **Total Marks : 30**

Notations:

Propositions. Boolean variables with True ($\top$) and False ($\bot$) values

Literals. Propositions (p) or negated propositions ($\neg p$)

Connectives. Binary operators ($\bowtie$) such as, AND ($\wedge$), OR ($\vee$), IMPLY ($\rightarrow$) and IFF ($\leftrightarrow$)

Propositional Formula. Recursively defined as, $\varphi = p \mid (\varphi) \mid \neg\varphi \mid \varphi \bowtie \varphi$, where $\bowtie \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$

Problem Statement:

Input. Propositional Formula (φ) as strings with propositions, negations, connectives and brackets, '(' and ')'

Postfix Formula Representation. Propositional Formula (φ) as strings with propositions, negations, connectives in Postfix format (this will made available to you in code as a string for ready-made processing!)

Output. You will be asked to write separate functions for the following parts (in the already supplied code):

1. Represent the postfix propositional formula (φ) as a binary tree (τ) data structure, known as *expression tree*, which contains propositions as leaf nodes and operators $\{\wedge, \vee, \neg, \rightarrow, \leftrightarrow\}$ as internal nodes (refer to the left expression tree in Figure 1) **(Marks : 5)**
2. Print the expression tree (using in-order traversal of τ) and generate the formula (φ) **(Marks : 2)**
3. Given $\top/\bot$ values for all the propositions, find the outcome of the overall Formula (φ) from its expression tree (τ) **(Marks : 4)**
4. Transformation of the formula step-wise ($\varphi \rightsquigarrow \varphi_I \rightsquigarrow \varphi_N \rightsquigarrow \varphi_C/\varphi_D$) using the expression tree data structure ($\tau \rightsquigarrow \tau_I \rightsquigarrow \tau_N \rightsquigarrow \tau_C/\tau_D$) as follows:
   (a) Implication-Free Form (IFF): Formula (φ_I) after elimination of $\rightarrow$ and $\leftrightarrow$
   Procedure: Transform τ to τ_I and then print φ_I from τ_I **(Marks : 4)**
   (b) Negation Normal Form (NNF): Formula (φ_N) where $\neg$ appears only before propositions
   Procedure: Transform τ_I to τ_N and then print φ_N from τ_N **(Marks : 4)**
   (c) Conjunctive Normal Form (CNF): Formula (φ_C) with conjuction of *disjunctive-clauses* where each disjunctive-clause is a disjunction of literals
   Procedure: Transform τ_N to τ_C and then print φ_C from τ_C **(Marks : 3)**
   (d) Disjunctive Normal Form (DNF): Formula (φ_D) with disjuction of *conjuctive-clauses* where each conjunctive-clause is a conjunction of literals
   Procedure: Transform τ_N to τ_D and then print φ_D from τ_D **(Marks : 3)**
5. Given the expression tree (τ), using exhaustive search, check for the following – **(Marks : 5)**
   (a) the validity ($\top$) or the invalidity of the formula (whether it is a tautology or not), or
   (b) the satisfiability or the unsatisfiability ($\bot$) of the formula (whether it is a contradiction or not)

Algorithms:

Expression Tree Formation. Let the generated postfix string from the propositional formula (φ) be $\texttt{PS}[1..n]$. The recursive function ETF, i.e. $\tau \leftarrow \texttt{ETF}(\texttt{PS}[1..n])$, is as follows:

- If $n = 1$ (i.e. $\texttt{PS}[1]$ is a proposition), then $\tau = \texttt{CREATENODE}(\varphi)$;
- If $n > 1$ and $\texttt{PS}[n] = \neg$, then $\tau = \texttt{CREATENODE}(\neg); \tau \mapsto \texttt{rightChild} = \texttt{ETF}(\texttt{PS}[1..(n-1)])$;
- If $n > 2$ and $\texttt{PS}[n] = \bowtie$, then
  $\tau = \texttt{CREATENODE}(\bowtie); \tau \mapsto \texttt{leftChild} = \texttt{ETF}(\texttt{PS}[1..(k-1)]); \tau \mapsto \texttt{rightChild} = \texttt{ETF}(\texttt{PS}[k..(n-1)])$;
- return τ;

Here, the primary question is – *how to find k for the last step?* (this will be explained to you!)

Printing Expression Tree. The recursive function $\mathtt{ETP}(\tau)$ is as follows:

- If $\tau \mapsto \mathtt{element}$ is not NULL, then
  $\mathtt{PRINT}\big(($; $\mathtt{ETP}(\tau \mapsto \mathtt{leftChild})$; $\mathtt{PRINT}(\tau \mapsto \mathtt{element})$; $\mathtt{ETP}(\tau \mapsto \mathtt{rightChild})$; $\mathtt{PRINT}\big()\big)$;

Here, the PRINT subroutine displays the respective charater as output.

Formula Evaluation. The recursive function EVAL, i.e. $\{\top, \bot\} \leftarrow \mathtt{EVAL}(\tau, \mathtt{v_1}, \mathtt{v_2}, \ldots, \mathtt{v_n})$ (assuming n propositions where each proposition p_i $(1 \le i \le n)$ is assigned a value $\mathtt{v_i} \in \{\top, \bot\}$), is as follows:

- If $\tau \mapsto \mathtt{element}$ is proposition p_i, then return $(\mathtt{v_i} = \top)?\ \top : \bot$;
- If $\tau \mapsto \mathtt{element}$ is $\neg$, then return $(\mathtt{EVAL}(\tau \mapsto \mathtt{rightChild}) = \top)?\ \bot : \top$;
- If $\tau \mapsto \mathtt{element}$ is $\wedge$, then return $\mathtt{EVAL}(\tau \mapsto \mathtt{leftChild}) \wedge \mathtt{EVAL}(\tau \mapsto \mathtt{rightChild})$;
- If $\tau \mapsto \mathtt{element}$ is $\vee$, then return $\mathtt{EVAL}(\tau \mapsto \mathtt{leftChild}) \vee \mathtt{EVAL}(\tau \mapsto \mathtt{rightChild})$;
- If $\tau \mapsto \mathtt{element}$ is $\rightarrow$, then return $\big((\mathtt{EVAL}(\tau \mapsto \mathtt{leftChild}) = \top)$ and $(\mathtt{EVAL}(\tau \mapsto \mathtt{rightChild}) = \bot)\big)?\ \bot : \top$;
- If $\tau \mapsto \mathtt{element}$ is $\leftrightarrow$, then
  return $\Big(\big((\mathtt{EVAL}(\tau \mapsto \mathtt{leftChild}) = \top)$ and $(\mathtt{EVAL}(\tau \mapsto \mathtt{rightChild}) = \top)\big)$
  or $\big((\mathtt{EVAL}(\tau \mapsto \mathtt{leftChild}) = \bot)$ and $(\mathtt{EVAL}(\tau \mapsto \mathtt{rightChild}) = \bot)\big)\Big)?\ \top : \bot$;

IFF Transformation. The recursive function IFF, i.e. $\tau_\mathtt{I} \leftarrow \mathtt{IFF}(\tau)$, is as follows:

- If $\tau \mapsto \mathtt{element}$ is $\neg$, then
  /* $\mathtt{IFF}(\neg\varphi) = \neg\mathtt{IFF}(\varphi)$ */
- If $\tau \mapsto \mathtt{element}$ is $\{\wedge, \vee\}$, then
  /* $\mathtt{IFF}(\varphi_1 \wedge \varphi_2) = \mathtt{IFF}(\varphi_1) \wedge \mathtt{IFF}(\varphi_2)$
  $\mathtt{IFF}(\varphi_1 \vee \varphi_2) = \mathtt{IFF}(\varphi_1) \vee \mathtt{IFF}(\varphi_2)$ */
- If $\tau \mapsto \mathtt{element}$ is $\rightarrow$, then
  /* $\mathtt{IFF}(\varphi_1 \rightarrow \varphi_2) = \neg\mathtt{IFF}(\varphi_1) \vee \mathtt{IFF}(\varphi_2)$ */
- If $\tau \mapsto \mathtt{element}$ is $\leftrightarrow$, then
  /* $\mathtt{IFF}(\varphi_1 \leftrightarrow \varphi_2) = \mathtt{IFF}(\varphi_1 \rightarrow \varphi_2) \wedge \mathtt{IFF}(\varphi_1 \rightarrow \varphi_2)$ */
- return τ;

Here, φ_I can be obtained (as a string expression) by calling $\mathtt{ETP}(\tau_\mathtt{I})$.

NNF Transformation. The recursive function NNF, i.e. $\tau_\mathtt{N} \leftarrow \mathtt{NNF}(\tau_\mathtt{I})$, is as follows:

- If $\tau_\mathtt{I} \mapsto \mathtt{element}$ is $\neg$, then
  - if $(\tau_\mathtt{I} \mapsto \mathtt{rightChild}) \mapsto \mathtt{element}$ is $\neg$, then
    /* $\mathtt{NNF}(\neg\neg\varphi) = \mathtt{NNF}(\varphi)$ */
  - if $(\tau_\mathtt{I} \mapsto \mathtt{rightChild}) \mapsto \mathtt{element}$ is $\wedge$, then
    /* $\mathtt{NNF}(\neg(\varphi_1 \wedge \varphi_2)) = \neg\mathtt{NNF}(\varphi_1) \vee \neg\mathtt{NNF}(\varphi_2)$ */
  - if $(\tau_\mathtt{I} \mapsto \mathtt{rightChild}) \mapsto \mathtt{element}$ is $\vee$, then
    /* $\mathtt{NNF}(\neg(\varphi_1 \vee \varphi_2)) = \neg\mathtt{NNF}(\varphi_1) \wedge \neg\mathtt{NNF}(\varphi_2)$ */
- If $\tau_\mathtt{I} \mapsto \mathtt{element}$ is $\{\wedge, \vee\}$, then
  /* $\mathtt{NNF}(\varphi_1 \wedge \varphi_2) = \mathtt{NNF}(\varphi_1) \wedge \mathtt{NNF}(\varphi_2)$
  $\mathtt{NNF}(\varphi_1 \vee \varphi_2) = \mathtt{NNF}(\varphi_1) \vee \mathtt{NNF}(\varphi_2)$ */
- return $\tau_\mathtt{I}$;

Here, φ_N can be obtained (as a string expression) by calling $\mathtt{ETP}(\tau_\mathtt{N})$.

CNF Transformation. The recursive function CNF, i.e. $\tau_\mathtt{C} \leftarrow \mathtt{CNF}(\tau_\mathtt{N})$, is as follows:

- If $\tau_\mathtt{N} \mapsto \mathtt{element}$ is $\wedge$, then
  /* $\mathtt{CNF}(\varphi_1 \wedge \varphi_2) = \mathtt{CNF}(\varphi_1) \wedge \mathtt{CNF}(\varphi_2)$ */
- If $\tau_\mathtt{N} \mapsto \mathtt{element}$ is $\vee$, then /* Distribution Law enforcement */
  - if $(\tau_\mathtt{N} \mapsto \mathtt{leftChild}) \mapsto \mathtt{element}$ is $\wedge$, then
    /* $\mathtt{CNF}((\varphi_{1l} \wedge \varphi_{1r}) \vee \varphi_2) = \mathtt{CNF}(\varphi_{1l} \vee \varphi_2) \wedge \mathtt{CNF}(\varphi_{1r} \vee \varphi_2)$ */
  - if $(\tau_\mathtt{N} \mapsto \mathtt{rightChild}) \mapsto \mathtt{element}$ is $\wedge$, then
    /* $\mathtt{CNF}(\varphi_1 \vee (\varphi_{2l} \wedge \varphi_{2r})) = \mathtt{CNF}(\varphi_1 \vee \varphi_{2l}) \wedge \mathtt{CNF}(\varphi_1 \vee \varphi_{2r})$ */
- return $\tau_\mathtt{N}$;

Here, φ_C can be obtained (as a string expression) by calling $\mathtt{ETP}(\tau_\mathtt{C})$. The subroutine $\mathtt{DUPLICATE}(\tau)$ creates another exact replica of the expression tree rooted at τ.

DNF Transformation. The recursive function DNF, i.e. $\tau_{\mathtt{D}} \leftarrow \mathtt{DNF}(\tau_{\mathtt{N}})$, is as follows:

- If $\tau_{\mathtt{N}} \mapsto \mathtt{element}$ is $\vee$, then
  $/*\ \ \mathtt{DNF}(\varphi_1 \vee \varphi_2) = \mathtt{DNF}(\varphi_1) \vee \mathtt{DNF}(\varphi_2)\ \ */$
- If $\tau_{\mathtt{N}} \mapsto \mathtt{element}$ is $\wedge$, then /* Distribution Law enforcement */
  - if $(\tau_{\mathtt{N}} \mapsto \mathtt{leftChild}) \mapsto \mathtt{element}$ is $\vee$, then
    $/*\ \ \mathtt{DNF}((\varphi_{1\mathtt{l}} \vee \varphi_{1\mathtt{r}}) \wedge \varphi_2) = \mathtt{DNF}(\varphi_{1\mathtt{l}} \wedge \varphi_2) \vee \mathtt{DNF}(\varphi_{1\mathtt{r}} \wedge \varphi_2)\ \ */$
  - if $(\tau_{\mathtt{N}} \mapsto \mathtt{rightChild}) \mapsto \mathtt{element}$ is $\vee$, the
    $/*\ \ \mathtt{DNF}(\varphi_1 \wedge (\varphi_{2\mathtt{l}} \vee \varphi_{2\mathtt{r}})) = \mathtt{DNF}(\varphi_1 \wedge \varphi_{2\mathtt{l}}) \vee \mathtt{DNF}(\varphi_1 \wedge \varphi_{2\mathtt{r}})\ \ */$
- return τ_N;

Here, φ_D can be obtained (as a string expression) by calling $\mathtt{ETP}(\tau_{\mathtt{D}})$. The subroutine $\mathtt{DUPLICATE}(\tau)$ creates another exact replica of the expression tree rooted at τ.

Exhaustive Search for Validity/Satisfibility. The function $\mathtt{CHECK}(\tau)$ is as follows:

- For *every* value tuple $\{\mathtt{v}_1, \mathtt{v}_2, \ldots, \mathtt{v}_\mathtt{n}\}$ corresponding to n propositions $\{p_1, p_2, \ldots, p_n\}$, if $\mathtt{EVAL}(\tau, \mathtt{v}_1, \mathtt{v}_2, \ldots, \mathtt{v}_\mathtt{n}) = \top$, then print "$\langle \mathtt{VALID} + \mathtt{SATISFIABLE} \rangle$"
- For *every* value tuple $\{\mathtt{v}'_1, \mathtt{v}'_2, \ldots, \mathtt{v}'_\mathtt{n}\}$ corresponding to n propositions $\{p_1, p_2, \ldots, p_n\}$, if $\mathtt{EVAL}(\tau, \mathtt{v}'_1, \mathtt{v}'_2, \ldots, \mathtt{v}'_\mathtt{n}) = \bot$, then print "$\langle \mathtt{INVALID} + \mathtt{UNSATISFIABLE} \rangle$"
- Otherwise, *for any* pair of value tuples $\{\mathtt{v}_1, \mathtt{v}_2, \ldots, \mathtt{v}_\mathtt{n}\}$ and $\{\mathtt{v}'_1, \mathtt{v}'_2, \ldots, \mathtt{v}'_\mathtt{n}\}$ corresponding to n propositions $\{p_1, p_2, \ldots, p_n\}$ such that, $\mathtt{EVAL}(\tau, \mathtt{v}_1, \mathtt{v}_2, \ldots, \mathtt{v}_\mathtt{n}) = \top$ and $\mathtt{EVAL}(\tau, \mathtt{v}'_1, \mathtt{v}'_2, \ldots, \mathtt{v}'_\mathtt{n}) = \bot$, then print "$\langle \mathtt{SATISFIABLE} + \mathtt{INVALID} \rangle$", for $\{\mathtt{v}_1, \mathtt{v}_2, \ldots, \mathtt{v}_\mathtt{n}\}$ and $\{\mathtt{v}'_1, \mathtt{v}'_2, \ldots, \mathtt{v}'_\mathtt{n}\}$, respectively

---

Example:

Input Propositional Formula. $\varphi = (\neg p \wedge q) \rightarrow (p \wedge (r \rightarrow q))$

Postfix Formula Representation. $p\ \neg\ q\ \wedge\ p\ r\ q\ \rightarrow\ \wedge\ \rightarrow$ (YOUR INPUT STRING)

Expression Tree Formation. Depending on the recursive call, two types of parse tree (τ) can be formed. Figure 1 shows the representation of such expression trees.

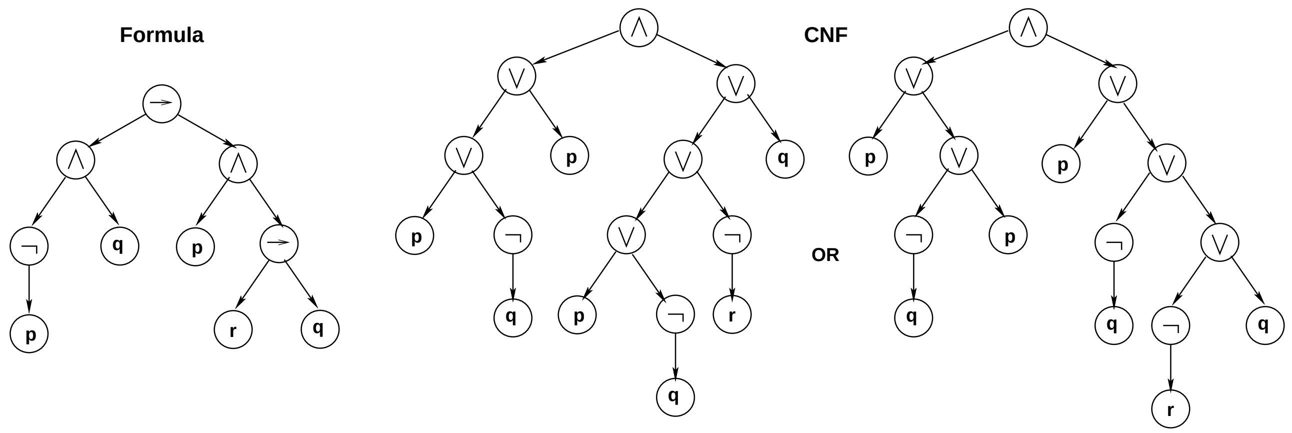

Figure 1: Expression Tree Structure for Original Formula and the Corresponding CNF

Formula Evaluation. $\{p = \bot, q = \top, r = \top\} \Rightarrow \varphi = \bot$; $\{p = \bot, q = \bot, r = \bot\} \Rightarrow \varphi = \top$

Formula Transformations. The path through which you shall be doing this is as follows:

$$\varphi \rightsquigarrow \mathtt{PostFix} \rightsquigarrow \tau\ (\mathtt{Print}\ \varphi) \rightsquigarrow \tau_{\mathtt{I}}\ (\mathtt{Print}\ \varphi_{\mathtt{I}}) \rightsquigarrow \tau_{\mathtt{N}}\ (\mathtt{Print}\ \varphi_{\mathtt{N}}) \rightsquigarrow \tau_{\mathtt{C}}/\tau_{\mathtt{D}}\ (\mathtt{Print}\ \varphi_{\mathtt{C}}/\varphi_{\mathtt{D}})$$

$$\begin{array}{lcl}
IFF : \varphi_I = \mathtt{IFF}(\varphi) & = & \mathtt{IFF}((\neg p \wedge q) \rightarrow (p \wedge (r \rightarrow q))) = \cdots = \neg(\neg p \wedge q) \vee (p \wedge (\neg r \vee q)) \\
NNF : \varphi_N = \mathtt{NNF}(\varphi_I) & = & \mathtt{NNF}(\neg(\neg p \wedge q) \vee (p \wedge (\neg r \vee q)))) = \cdots = (p \vee \neg q) \vee (p \wedge (\neg r \vee q)) \\
CNF : \varphi_C = \mathtt{CNF}(\varphi_N) & = & \mathtt{CNF}((p \vee \neg q) \vee (p \wedge (\neg r \vee q))) = \cdots = (p \vee \neg q \vee p) \wedge (p \vee \neg q \vee \neg r \vee q) \\
DNF : \varphi_D = \mathtt{DNF}(\varphi_N) & = & \mathtt{DNF}((p \vee \neg q) \vee (p \wedge (\neg r \vee q))) = \cdots = (p) \vee (\neg q) \vee (p \wedge \neg r) \vee (p \wedge q)
\end{array}$$

Check for Validity/Satisfibility.

$$\begin{array}{rl}
\mathtt{INVALID}: & \{p = \bot, q = \top, r = \times\} \quad (\times\ \textit{denotes don't care term}) \\
\mathtt{SATISFIABLE}: & \{p = \top, q = \times, r = \times\} \quad OR \quad \{p = \times, q = \bot, r = \times\}
\end{array}$$

Sample Execution:

Compile: (C Code) `gcc ROLLNO_CT1.c -lm` (Please follow the filename convention given!)

(C++ Code) `g++ ROLLNO_CT1.cpp -lm` (Please follow the filename convention given!)

Execution: `./a.out`

Sample Run:

```
Enter Propositional Logic Formula: (!p & q) -> (p & (r -> q))
Postfix Representation of Formula: p ! q & p r q -> & ->

++++ PostFix Format of the Propositional Formula ++++
('-' used for '->' and '~' used for '<->')
YOUR INPUT STRING: p!q&prq-&-

++++ Expression Tree Generation ++++
Original Formula (from Expression Tree): ( ( ! p & q ) -> ( p & ( r -> q ) ) )

++++ Expression Tree Evaluation ++++
Enter Total Number of Propositions: 3
Enter Proposition [1] (Format: Name <SPACE> Value): p 0
Enter Proposition [2] (Format: Name <SPACE> Value): q 1
Enter Proposition [3] (Format: Name <SPACE> Value): r 1

The Formula is Evaluated as: False

++++ IFF Expression Tree Conversion ++++
Formula in Implication Free Form (IFF from Expression Tree):
( ! ( ! p & q ) | ( p & ( ! r | q ) ) )

++++ NNF Expression Tree Conversion ++++
Formula in Negation Normal Form (NNF from Expression Tree):
( ( p | ! q ) | ( p & ( ! r | q ) ) )

++++ CNF Expression Tree Conversion ++++
Formula in Conjunctive Normal Form (CNF from Expression Tree):
( ( ( p | ! q ) | p ) & ( ( p | ! q ) | ( ! r | q ) ) )

++++ DNF Expression Tree Conversion ++++
Formula in Disjunctive Normal Form (DNF from Expression Tree):
( ( p | ! q ) | ( ( p & ! r ) | ( p & q ) ) )

++++ Exhaustive Search from Expression Tree for Validity / Satisfiability Checking ++++
Enter Number of Propositions: 3
Enter Proposition Names (<SPACE> Separated): p q r
Evaluations of the Formula:
    { (p = 0) (q = 0) (r = 0) }  :  1
    { (p = 0) (q = 0) (r = 1) }  :  1
    { (p = 0) (q = 1) (r = 0) }  :  0
    { (p = 0) (q = 1) (r = 1) }  :  0
    { (p = 1) (q = 0) (r = 0) }  :  1
    { (p = 1) (q = 0) (r = 1) }  :  1
    { (p = 1) (q = 1) (r = 0) }  :  1
    { (p = 1) (q = 1) (r = 1) }  :  1

The Given Formula is: < INVALID + SATISFIABLE >
```

Submit a single C/C++ source file following proper naming convention [`ROLLNO_CT1.c(.cpp)`].
Do not use any global/static variables. Use of STL is allowed.