1[9] Prove the following law of Binary Theory using the proof format of the course, and any laws listed in the handout. Do not use the Completion Rule.

\[(a \Rightarrow (p=x)) \land (\neg a \Rightarrow p) = p=(x \lor \neg a)\]

$\begin{align*}
\text{OR} & \quad p=(x \lor \neg a) \\
& \equiv \quad \text{case idempotent law} \\
& \equiv \quad \text{if } a \text{ then } p=(x \lor \neg a) \text{ else } p=(x \lor \neg a) \text{ fi} \\
& \equiv \quad \text{context} \\
& \equiv \quad \text{if } a \text{ then } p=(x \lor \neg T) \text{ else } p=(x \lor \neg \bot) \text{ fi} \\
& \equiv \quad \text{if } a \text{ then } p=(x \lor \bot) \text{ else } p=(x \lor T) \text{ fi} \\
& \equiv \quad \text{if } a \text{ then } p=x \text{ else } p \text{ fi} \\
& \equiv \quad \text{case analysis} \\
\end{align*}$

So yes, \((a \Rightarrow (p=x)) \land (\neg a \Rightarrow p)\)

$\begin{align*}
\text{OR} & \quad p=(a \Rightarrow (p=x)) \land (\neg a \Rightarrow p) \\
& \equiv \quad \text{case analysis} \\
& \equiv \quad \text{if } a \text{ then } p=x \text{ else } p \text{ fi} \\
& \equiv \quad \text{identity} \\
& \equiv \quad \text{if } a \text{ then } p=x \text{ else } p=\top \text{ fi} \\
& \equiv \quad \text{case distributive} \\
& \equiv \quad p = \text{if } a \text{ then } x \text{ else } \top \text{ fi} \\
& \equiv \quad \text{one-case} \\
& \equiv \quad p = (a \Rightarrow x) \\
& \equiv \quad \text{material implication} \\
& \equiv \quad p = (\neg a \lor x) \\
& \equiv \quad \text{symmetry} \\
& \equiv \quad p = (x \lor \neg a) \\
\end{align*}$

2[9] A list is bitonic if it is monotonic up to some index, and antimonotonic after that. For example, \([1; 3; 4; 5; 5; 6; 4; 4; 3]\) is bitonic. One or both of the segments could be empty, in which case the list is monotonic or antimonotonic. Express formally that \(L\) is bitonic. (You are not being asked to write a program.)

$\exists n: 0,..\#L+1: (\forall i, j: 0,..n \cdot i \leq j \Rightarrow L_i \leq L_j) \land (\forall i, j: n,..\#L \cdot i \leq j \Rightarrow L_i \geq L_j)$

3 Let \(n\) be a natural state variable. Is the following specification implementable? Proof required.

(a)[9] \(n:= n-1\)

$\begin{align*}
\forall n: \text{nat} \cdot \exists n': \text{nat} \cdot n := n-1 \\
& \equiv \quad \text{expand assignment} \\
& \equiv \quad \forall n: \text{nat} \cdot \exists n': \text{nat} \cdot n' = n-1 \\
& \equiv \quad \text{specialization} \\
& \equiv \quad \exists n': \text{nat} \cdot n' = 0-1 \\
& \equiv \quad \exists n': \text{nat} \cdot n' = -1 \\
& \equiv \quad \bot \\
\end{align*}$

So no, \(n:= n-1\) is not implementable. From the line

\(\forall n: \text{nat} \cdot \exists n': \text{nat} \cdot n' = n-1\)

we can use an identity law

\(\forall n: \text{nat} \cdot \exists n': \text{nat} \cdot n' = n-1 \land T\)

but now we cannot use the one-point law to get

\(\forall n: \text{nat} \cdot \exists n': \text{nat} \cdot T\)

because the one-point law requires \(n-1: \text{nat}\)

(b)[9] \((n>0) \Rightarrow (n:= n-1)\)

$\begin{align*}
\forall n: \text{nat} \cdot \exists n': \text{nat} \cdot n>0 \Rightarrow (n:= n-1) \\
& \equiv \quad \text{expand assignment} \\
& \equiv \quad \forall n: \text{nat} \cdot \exists n': \text{nat} \cdot n>0 \Rightarrow n' = n-1 \\
& \equiv \quad \text{distributive and identity} \\
& \equiv \quad \forall n: \text{nat} \cdot n>0 \Rightarrow \exists n': \text{nat} \cdot n' = n-1 \land T \\
& \equiv \quad \text{In the context } n>0, n-1: \text{nat}. \\
& \equiv \quad \text{So we can use one-point.} \\
& \equiv \quad \forall n: \text{nat} \cdot n>0 \Rightarrow T \\
& \equiv \quad \text{base and identity} \\
& \equiv \quad T \\
\end{align*}$

So yes, \((n>0) \Rightarrow (n:= n-1)\) is implementable.
4[7] Let $s$ and $i$ be integer variables, and let $L$ be a list of integers (not a variable). What is the exact precondition for $s' = \Sigma L[0..i']$ to be refined by $(s := s + Li. \ i := i + 1)$?

\[ \forall s', i'. (s' = \Sigma L[0..i']) \iff (s := s + Li. \ i := i + 1) \]

Expand assignments
\[ = \forall s', i'. (s' = \Sigma L[0..i']) \iff s' = s + Li \land i' = i + 1 \]

One-point, twice
\[ = s + Li = \Sigma L[0..i + 1] \]

Separate final term in sum
\[ = s + Li = (\Sigma L[0..i]) + Li \]

Cancel
\[ = s = \Sigma L[0..i] \]

5 Let $s$ and $n$ be integer variables. Let $Q$ be a specification defined as $Q = s' = s + n \times (n - 1)/2$

(a)[9] Prove the refinement $Q \iff n := n - 1. \ s := s + n. \ Q$

\[ = n := n - 1. \ s := s + n. \ Q \]

Expand $Q$
\[ = n := n - 1. \ s := s + n. \ s' = s + n \times (n - 1)/2 \]

Substitution law twice
\[ = s' = s + n - 1 + (n - 1) \times (n - 1)/2 \]

Arithmetic
\[ = s' = s + n \times (n - 1)/2 \]

(b)[6] Add time according to the recursive measure, replace $Q$ by a timing specification, and reprove the refinement.

\[ t' = \infty \iff n := n - 1. \ s := s + n. \ t := t + 1. \ t' = \infty \]

Substitution law three times
\[ = t' = \infty \]

6 A theory of widgets is presented in the form of some new syntax and some axioms. An implementation of widgets is written. In a couple of sentences, state:

(a)[6] How do we know whether the theory of widgets is consistent or inconsistent?

If we implement it, and we prove the implementation is correct, and the theories used in the implementation are consistent, then we know that the theory of widgets is consistent also. If we can prove $\bot$ from the theory of widgets, then it is inconsistent.

(b)[6] How do we know whether the theory of widgets is complete or incomplete?

To show completeness, show the result of all combinations of the new functions and operations. To show incompleteness, implement the theory twice so that some binary expression is a theorem according to one implementation and an antitheorem according to the other.

(c)[6] How do we know whether the implementation of widgets is correct or incorrect?

To prove it correct, prove that each of the axioms of widget theory becomes a theorem using the definitions and theories of the implementation. To say the same thing differently, prove that the implementation implies the theory. To prove incorrectness, find a behavior (values for initial and final state, time, ...) that satisfy the implementation but not the theory.
Let \( u \) be a binary user's variable. Let \( a \) and \( b \) be old binary implementer's variables. We replace \( a \) and \( b \) by new integer implementer's variables \( x \) and \( y \) using the convention (from the C language) that \( 0 \) stands for \( \bot \) and non-zero integers stand for \( T \).

(a) What is the transformer?
\[
\begin{align*}
\forall a, b \cdot a &= (x\neq 0) \land b = (y\neq 0) & \text{replace asmt} \\
\Rightarrow \exists a', b' \cdot a' &= (x'\neq 0) \land b' = (y'\neq 0) \land (a := \neg a) & 1\text{-pt } a' \text{ and } b'
\end{align*}
\]

(b) Transform and implement \( a := \neg a \).
\[
\begin{align*}
\forall a, b \cdot a &= (x\neq 0) \land b = (y\neq 0) & \Rightarrow \exists a', b' \cdot a' &= (x'\neq 0) \land b' = (y'\neq 0) \land a' = \neg a \land b' = b \land u' = u & 1\text{-pt } a' \text{ and } b'
\end{align*}
\]

Express the program
\[
(x := 1. \  x := x+y) \parallel (y := 2. \ y := x+y)
\]
without using any programming notations (no assignment, no dependent composition, no independent composition), where \( t \) is time, and \( x \) and \( y \) are

(a) boundary variables and assignment takes time 0.
\[
\begin{align*}
& x' = y+1 \land y' = x+2 \land t' = t
\end{align*}
\]

(b) interactive variables and assignment takes time 1.
\[
\begin{align*}
& x(t+1) = 1 \land x(t+2) = 3 \land y(t+1) = 2 \land y(t+2) = 3 \land t' = t+2
\end{align*}
\]