Problem 1 (Boolean Blasters)

Consider the following top-level Racket definitions over booleans that give direct definitions of the standard boolean operations \texttt{not}, \texttt{and}, and \texttt{or}:

\begin{verbatim}
(define my-not
  (lambda (b)
    (if b #f #t)))

(define my-and
  (lambda (b1 b2)
    (if b1 b2 #f)))

(define my-or
  (lambda (b1 b2)
    (if b1 #t b2)))
\end{verbatim}

Prove the following claims of program equivalence about these definitions:

\textbf{Claim 1.} \( (\text{my-not (my-and #t #f)}) \equiv \#t \).

\textbf{Claim 2.} \textit{There exists a boolean }\texttt{b1} \textit{such that for all }\texttt{b2}, \( (\text{my-or b1 b2}) \equiv \#t \).

\textit{(Hint: Handle each of the quantified variables in sequence!)}

\textbf{Claim 3 (Negation is an Involution).} \textit{For all booleans }\texttt{b}, \( (\text{my-not (my-not b)}) \equiv b \).

For this final claim, you may reason about the standard \texttt{not}, \texttt{and}, and \texttt{or} functions and assume that they evaluate in a single step, e.g., \texttt{and \#t \#f} \(\rightarrow\) \texttt{\#f}.

\textbf{Claim 4 (De Morgan’s Theorem).} \textit{For all booleans }\texttt{b1} \textit{and }\texttt{b2}, \( (\text{not (and b1 b2)}) \equiv (\text{or (not b1) (not b2)}) \).

\textit{(Hint: Consider all the cases!)}
Problem 2 (What Are You Implying?)

Consider the following Racket function:

\[
\begin{align*}
\text{define } f &= \\
& (\lambda (x y) \\
& \quad (\text{if } (\lt x y) \\
& \quad \quad 0 \\
& \quad \quad (- x y))))
\end{align*}
\]

Suppose that you would like to prove that \( f \) always produces a positive (non-negative, non-zero) result.

(a) What preconditions on \( f \) are necessary to enforce this claim?
(b) Write a formal proposition that formalizes this claim. Make sure that your proposition includes the preconditions you listed in the previous part.
(c) Formally prove this proposition.
Problem 3 (Down the Drain)

For each of the following descriptions of recursive Racket functions:

(i) Write a high-level definition of the behavior of the function in terms of the recursive definition of a list. In the recursive case of your definition, you should mention a recursive invocation of the function somewhere, e.g., “We add one to the length of the tail of the list.”

(ii) Give an implementation of that high-level definition as a Racket function.

(a) Write a recursive Racket function \( \text{stutter } l \) that takes a list \( l \) as input and returns \( l \) but with every element duplicated. For example, \( \text{stutter } '(1 2 3) \rightarrow '(1 1 2 2 3 3) \).

(b) Write a recursive Racket function \( \text{sorted? } l \) that takes a list of numbers as input and returns true iff the list is in sorted order. For example, \( \text{sorted? } '(1 2 3 4 5) \rightarrow #t \) whereas \( \text{sorted? } '(1 3 9 4 6) \rightarrow #f \).

(Hint: Consider multiple base cases. How many elements must the list contain before you safely extract out elements to compare with <?)
Problem 4 (First Steps)

Consider the following claim over the stutter function you wrote in the previous section:

Claim 5 (Stutter Doubles Length). For all lists \( l \), \((\text{length} \ (\text{stutter} \ l)) \equiv (+ \ (\text{length} \ 1) \ (\text{length} \ 1))\).

This claim also depends on the standard definition of list length:

\[
\begin{align*}
\text{(define length} \\
\quad (\text{lambda} \ (l) \\
\quad \quad (\text{if} \ (\text{null?} \ l) \\
\quad \quad \quad 0 \\
\quad \quad (+ \ 1 \ (\text{length} \ (\text{cdr} \ l)))))
\end{align*}
\]

Below is a skeleton of an inductive proof of the claim. Fill in the skeleton with appropriate prose and derivations where necessary.

Proof. We proceed by induction on \( l \).

- \( l \) is empty. (FILL ME IN!).
- \( l \) is non-empty. Our induction hypothesis is:

  (FILL ME IN!).

And in the non-empty case, we are trying to prove:

  (FILL ME IN!).

The left-hand side of the claim, \((\text{length} \ (\text{stutter} \ l))\) evaluates as follows:

\[
(\text{length} \ (\text{stutter} \ l)) \rightarrow^* \text{(FILL ME IN!)}
\]

The right-hand side of the claim, \((+ \ (\text{length} \ 1) \ (\text{length} \ 1))\) evaluates as follows:

\[
(+ \ (\text{length} \ 1) \ (\text{length} \ 1)) \rightarrow^* \text{(FILL ME IN!)}
\]

Our induction hypothesis allows us to rewrite (FILL ME IN!) to (FILL ME IN!). Thus, our goal equivalence can be rewritten to:

  (FILL ME IN!)

completing the proof.