Administrivia

- Part I of the homework builds on the propositional logic package developed in class. For the problems of this part, add your code to a copy of: http://www.cis.syr.edu/courses/cis352/coursework/prop.hs.
- Part II consists of some problems involving higher type functions. For the Part II problems, use a fresh file. You may want to copy some of the import-statements from prop.hs to your fresh file.
- Let me know if any of my QuickCheck tests seem dodgy.
- You may do with homework with one other person. That is: No teams of size larger than two!
- If you pick up an idea from someone outside of your team or an internet site or a book, document it in your coversheet file.
- Use Blackboard to turn in the assignment. In your submission, include: (i) your coversheet, (ii) the source files, (iii) the transcripts of test runs.

Part I: More on Propositional Logic

**Problem 1 (10 points)**

Extend the Prop datatype and associated functions in prop.hs to handle the nand connective as follows.

- Find the declaration of the Prop datatype in prop.hs and extend it with the infix constructor #: for nand.¹
- Find the printer (showProp), evaluator (eval), name-extractor (names), and subformula-extractor (subformulas) functions and extend their definitions to cover, #: , the new nand constructor.²

*Testing this work:* Run (fullTable (vp :#: vq)) which should result in the truth-table of Figure 1.

---

¹ Obvious hint: What you do for :#: will be pretty similar to what is done for :&:, :|:, etc.

² See footnote 1. A tiny bit of thought is required for eval.

Typo fix. Previously I incorrectly had (fullTable (p :#: q)).
\textbf{Problem 2 (20 points)}

An important property of nand is that all the other propositional connectives can be expressed in terms of it.\textsuperscript{3} Thus, every propositional formula can be translated to an equivalent (if long-winded) formula that uses \# as its only logical connective. For example:

\begin{align*}
\sim P &\equiv (P \# P) \\
P \& (\sim Q) &\equiv ((P \# (Q \# Q)) \# (P \# (Q \# Q))) \\
P \leftrightarrow Q &\equiv (((P \# (Q \# Q)) \# (Q \# (P \# P))) \# ((P \# (Q \# Q))) \# (Q \# (P \# P))))
\end{align*}

Write a function

\begin{verbatim}
def nandify :: Prop -> Prop
\end{verbatim}

that recursively translates a proposition to an equivalent proposition that uses \# as its only logical connective. Make use of the equivalences from https://en.wikipedia.org/wiki/Sheffer_stroke.

Testing \texttt{nandify}: Run (quickCheck nand1_prop) and (quickCheck nand2_prop).

\begin{definition}
A propositional formula is in \textit{negation normal form} when:
\begin{itemize}
\item \& and \lor are the only binary connectives it uses and
\item the only place a \sim appears is before a variable.
\end{itemize}
\end{definition}

\textbf{Problem 3 (10 points)}

Write a function

\begin{verbatim}
def isNNF :: Prop -> Bool
\end{verbatim}

that tests whether a proposition is in negation normal form.

\textbf{Problem 4 (20 points)}

Write a function

\begin{verbatim}
def toNNF :: Prop -> Prop
\end{verbatim}

that recursively translates a proposition to an equivalent proposition in negation normal form. In doing this translation, the following equivalences will be useful:

\begin{align*}
\sim (P \& Q) &\leftrightarrow (\sim P) \lor (\sim Q) \\
(P \rightarrow Q) &\leftrightarrow (\sim P) \lor Q \\
\sim (\sim P) &\leftrightarrow P \\
\sim (P \# Q) &\leftrightarrow (P \& Q)
\end{align*}

My answer for this has 17 cases, including 9 of the form

\begin{verbatim}
\texttt{toNNF (Not stuff)} = \texttt{other-stuff}
\end{verbatim}

Your answer may have a few more or few less cases depending on how you do things.
Testing \textit{isNNF} and \textit{toNNF}: Run (quickCheck nnf1_prop) and (quickCheck nnf2_prop). Also add a few tests of your own. In particular, check that you get the following translations:

\[
\begin{align*}
toNNF(\text{Not } T) & \leadsto F \\
toNNF(\text{Not } F) & \leadsto T \\
toNNF(\text{Not } (\text{Not } (\text{Not } (\text{Var } "P")))) & \leadsto \sim P \\
toNNF(\text{Not } (\text{Not } (\text{Not } (\text{Var } "P")))) & \leadsto P \\
toNNF((\text{Not } (\text{Var } "P")) :\leftrightarrow: (\text{Not } (\text{Var } "Q"))) & \leadsto (((\sim P)\mid Q)\&((\sim Q)\mid P))
\end{align*}
\]

\textbf{Part II: More on Higher-Types}

\begin{itemize}
\item \textbf{Problem 5 (12 points)}
Write a function
\[
same :: [\text{Int}] \rightarrow \text{Bool}
\]
that returns True if and only if all of the elements of the list are equal. Your definition should be of the form
\[
same \text{ xs } = \text{ and } (\text{zipWith } \_ \_ \_ \text{ xs (tail xs)})
\]
To get an idea of what would be in the above blank, compute
\[
\text{zip [1..5] (tail [1..5])}
\]
\textbf{Testing for same}: Devise your own tests.\footnote{Problems with random tests: same will return False on most lists of length two or more. Experiment: Once you have same working, define: \text{same\_prop } \text{xs } = \text{(same } \text{xs } = \text{(length } \text{xs } < 2)) \text{ Now run quickCheck on same\_prop a few times. Some times this fails the equivalence test (as it should), but sometimes not.}}

\begin{itemize}
\item \textbf{Problem 6 (16 points)}
Write two functions
\[
squash, \text{squash'} :: (a\rightarrow a\rightarrow b) \rightarrow [a] \rightarrow [b]
\]
both of which apply a given function to adjacent elements of a list. E.g.
\[
squash f \ [x_1, x_2, x_3, x_4] \leadsto [f \ x_1 \ x_2, f \ x_2 \ x_3, f \ x_3 \ x_4 ]
\]
(a) (8 points) Fill in the blanks in:
\[
squash f \ [] = \_
\]
\[
squash f \ [x] = \_
\]
\[
squash f \ (x_1:x_2:x) = \_ : \text{squash } f \_
\]
to implement squash via explicit recursion and pattern matching.
(b) (8 points) Implement \textit{squash'} using \textit{zipWith}.\footnote{Hint: Use the trick employed in the definition of same above.}

\textbf{Testing squash and squash'}: Define the property squash\_prop:
\[
squash\_pred \text{ xs } = (\text{squash } (+) \text{ xs } \Rightarrow \text{squash' } (+) \text{ xs})
\]
Then run (quickCheck squash\_prop). Also add a few tests of your own.
Problem 7 (12 points)

Background:

```haskell
data BTree = Empty | Branch Int BTree BTree
             deriving (Show)
foldT :: (Int -> a -> a -> a) -> a -> BTree -> a
foldT brch emp Empty = emp
foldT brch emp (Branch n tl tr) = brch n vl vr
    where vl = foldT brch emp tl
          vr = foldT brch emp tr
```

The function `foldT` amounts to a version of `foldr` for the `BTree` datatype. For example, here is a version of the `bcount` function from Homework 2 defined via `foldT`:

```haskell
bcount t = foldT tally 0 t
    where tally _ numl numr = 1 + numl + numr
```

Your tasks:

(a) Define a function that sums all the `Int`-values in a `BTree` by filling in the blank in:

```haskell
bsum t = foldT f 0 t
    where f n vl vr = ________________
```

(b) Define a version of `bmaxDepth` from Problem 1 of Homework 2 by filling in the blanks in:

```haskell
bmaxDepth' t = foldT g ____ t
    where g n vl vr = ________________
```

Testing for `bsum` and `bmaxDepth'`: Devise your own tests.