1. (Types of various functions)

   \[ f_{1a} :: (b, a) -> (a, b) \]
   \[ f_{1a} = \langle x, y \rangle -> (y, x) \]
   
   \[ f_{1b} :: a -> [a] -> [[a]] \]
   \[ f_{1b} = \langle x \rangle y -> [[x]], y \]
   
   \[ f_{1c} :: a -> a -> [a] -> [[a]] \]
   \[ f_{1c} = \langle x \rangle y z -> [x : z, y : z] \]
   
   \[ f_{1d} :: (a -> Bool) -> [a] -> Int \]
   \[ f_{1d} f = length . (filter f) \]

2. (Deriving Eq and Show) Given

   ```
   data D1 a = Const1 | Fcn1 a Int
   data D2 a = Const2 | Fcn2 a Int deriving (Eq, Show)
   ```

   The two differences are that values of type D2 can be tested for equality and can be printed out. (And since they can be printed out, you can type one into the interpreter read-eval-print loop and see the result.) For example, entering Const1 == Const1 (into the interpreter) produces an error message but Const2 == Const2 yields True. Similarly, Fcn1 1 2 yields an error but Fcn2 1 2 prints out Fcn2 1 2.

3. (Syntax issues) F3, X, and Abc are variables, so they have to begin in lower case. For the indentation, we need the a of abc = ... and the z of z = ... to line up. They both need to be indented relative to the f of f3 ...., so an example of what works is

   ```
   f3 x y = abc * z where
   abc = x + y
   z = abc + 2
   ```

   But only one space before abc = ... and z = ... is required

4. (Lambda function) \( \lambda b x -> b &\& \text{isLetter} x \) :: Bool -> Char -> Bool

5. (Trim list of numbers)

   ```
   trim n
   | n `mod` 5 == 0 = True
   | n `mod` 7 == 0 = True
   | otherwise = False
   ```

6. (Iterate function)

   With higher-order functions:

   ```
   iterate n f = foldl (.) id [f | i <- [1..n]]
   (This calculates \(((\ldots((id \cdot f) \cdot f) \ldots) \cdot f)\), where there are n copies of f.)
   ```

   Recursively:

   ```
   iterate_r n f x
   | n <= 0 = x
   | otherwise = iterate_r (n-1) f (f x)
   ```
7. (Preorder traversal)

\[
\text{pre'} :: \text{Tree } a -> [a] -> [a] \\
\text{pre'} (L x) xs = x : xs \\
\text{pre'} (Br r left right) xs = r : \text{pre'} left (\text{pre'} right xs) \\
\text{pre tree} = \text{pre'} tree []
\]

8. (Is a tree a heap?)

\[
\text{top} (L x) = x ; \text{top} (\text{Br } x _ _) = x \\
\text{isHeap} (L x) = \text{True} \\
\text{isHeap} (\text{Br } x \text{ left right}) = \\
\qquad x \geq \text{top left} \&\& x \geq \text{top right} \&\& \text{isHeap left} \&\& \text{isHeap right}
\]

9. (Stack operations)

\[
\text{type Stack a = [a]} \\
\text{data StackOp a = Push a | Pop deriving (Eq, Read, Show)}
\]

-- A configuration is a stack and list of results of Pop operations
-- (from oldest to newest result). Popping an empty stack yields
-- Nothing; popping a stack with head value x yields Just x
\[
\text{type Config a = (Stack a, [Maybe a])}
\]

-- apply applies a stack operation to a configuration and yields the
-- resulting configuration
\[
\text{apply :: StackOp a -> Config a -> Config a} \\
\text{apply (Push x) (q, vals) = (x : q, vals)} \\
\text{apply Pop ([], vals) = ([], Nothing : vals)} \\
\text{apply Pop (x : xs,vals) = (xs, Just x : vals)}
\]

-- Apply a list of stack operations
-- Tail-recursive solution
\[
\text{applyList ops = applyList' ops ([],[])} \text{ where} \\
\text{applyList' [] config = config} \\
\text{applyList' (cmd : cmds) config} \\
\qquad = \text{applyList' cmds (apply cmd config)}
\]

-- Higher-order function solution (two examples)
\[
\text{applyList ops = foldl (flip (\_)) id (map apply ops) ([], [])}
\]
\[
\text{applyList ops = foldl (\_) id (map apply (reverse ops)) ([], [])}
\]