Problem 1. **Infinite Lists of Integers.** There are four independent parts.

(a) What is the infinite list defined by the following declaration?

```
infListA :: [Int]
infListA = 1 : (zipWith (+) ones infListA)
  where ones = 1 : ones
```

*Answer:* 1, 2, 3, 4, 5, ... (the positive integers)

(b) What is the infinite list defined by the following declaration?

```
infListB :: [Int]
infListB = 1 : (zipWith (+) infListB infListB)
```

*Answer:* 1, 2, 4, 8, 16, ... (the powers of 2)

(c) What is the infinite list defined by the following declaration?

```
inflistC :: [Int]
inflistC = 1 : (zipWith (*) (tail nats) inflistC)
  where nats = 0 : (map succ nats)
```

*Answer:* 1, 1, 2, 6, 24, 120, 720, ... (the list of factorials)

(d) The function *mysteryFn* below returns an infinite list. What is the infinite list returned by *mysteryFn 2*?

```
mysteryFn :: Int -> [Int]
mysteryFn x = x : (zipWith (+) (mysteryFn x) (mysteryFn x))
```

*Answer:* 2, 4, 8, 16, ... (the powers of 2 greater than 1)
Problem 2. Datatypes and Binary Trees. Consider the following polymorphic datatype, representing binary trees where every leaf and every internal node is labelled:

```haskell
data Tree a = Leaf a | Node a (Tree a) (Tree a)
```

In parts (a), (b) and (d) below, we ask you to define appropriate Haskell functions: To get full credit, you need to use pattern-matching and wildcards as much as possible.

(a) Define a function that counts the number of internal nodes (excluding leaves) in a tree:

**Answer:**

```haskell
numberOfNodes :: Tree a -> Int
numberOfNodes (Leaf _) = 0
numberOfNodes (Node _ t1 t2) = 1 + numberOfNodes t1 + numberOfNodes t2
```

(b) Define a function that counts the number of leaves in a tree:

**Answer:**

```haskell
numberOfLeaves :: Tree a -> Int
numberOfLeaves (Leaf _) = 1
numberOfLeaves (Node _ t1 t2) = numberOfLeaves t1 + numberOfLeaves t2
```

(c) For an arbitrary tree \( t \) of type \( \text{Tree} \ a \), it is a fact that:

\[
1 + \text{numberOfNodes}(t) = \text{numberOfLeaves}(t)
\]

Suppose we want to prove fact (##) by induction. Write the statement that we need to prove in the induction step of this induction – do not write the proof itself. In particular, write carefully what the induction is on – is it on the number of internal nodes? on the number of leaves? on the height of trees? on the structure of trees? etc. – and how the induction hypothesis will be invoked in this induction step:

**Answer:** The induction is on the number of internal nodes in trees. Let \( n \) be an arbitrary non-negative integer.

*Induction hypothesis:* Assume, for every tree with at most \( n \) internal nodes, that equation (##) holds, starting with \( n = 0 \).

*Induction step:* Let \( t \) be an arbitrary tree with \( n + 1 \) internal nodes. We have to prove that equation (##) holds for this \( t \), by invoking the induction hypothesis twice, once for the left subtree of \( t \) and once for the right subtree of \( t \).

(d) Recall the definition of the function \( \text{zip} \) on two lists \( \text{xs} \) and \( \text{ys} \): It pairs off corresponding elements in \( \text{xs} \) and \( \text{ys} \), and returns a list which is as long as the shorter of the two input lists. Generalize the definition of \( \text{zip} \) to work on trees, with the output being another tree that matches as much as possible the shapes of the two input trees:

**Answer:**
Problem 3. **Types and Type Checking.** The infix operator for composition is just “.” so that, for example, the expression “(even . ceiling) 4.2” is equivalent to “even (ceiling 4.2)”, which evaluates to False. Using composition as a prefix operator, the preceding expression is also equivalent to “(. even ceiling) 4.2”.

(a) Write the type of the prefix operator for composition.

*Answer:*

\[
(\cdot) :: (a -> b) -> (c -> a) -> c -> b
\]

The library function flip is a higher-order function which takes as input a curried function \(f\) of two arguments \(x\) and \(y\) such that the evaluation of the expression “\(f x y\)” produces the same result as the evaluation of the expression “(flip \(f\)) \(y x\)”. For example, the two expressions below:

\[
\text{map even [1,2,3]} \quad \text{and} \quad (\text{flip map}) [1,2,3] \text{ even}
\]

are equivalent and therefore evaluate to the same final result \([\text{False,True,False}]\).

(b) Write the type of the library function flip.

*Answer:*

\[
\text{flip} :: (a -> b -> c) -> (b -> a -> c)
\]

In the remaining parts of this problem, take the type of the library function even to be Int -> Bool, and the type of the library function ceiling to be Float -> Int.

(c) Does (\(\cdot\) even ceiling) type-check? If it does, write its type. If it does not, give a reason.

*Answer: The expression does type check, and its type is:*

\[
(\cdot\text{ even ceiling}) :: \text{Float -> Bool}
\]

(d) Does (\(\cdot\) ceiling even) type-check? If it does, write its type. If it does not, give a reason.

*Answer: The expression does not type check, because (\(\cdot\) ceiling :: (a->Float)->(a->Int) cannot be applied to even :: Int->Bool which follows from the fact that (a->Float) cannot be instantiated to (Int->Bool).*

(e) Does (flip (\(\cdot\) even ceiling)) type-check? If it does, write its type. If it does not, give a one-line reason.

*Answer: The expression does not type check, because it is equivalent to ((\(\cdot\) ceiling even) in part (d), which does not type check.
(f) Does (flip (.) ceiling even) type-check? If it does, write its type. If it does not, give a one-line reason.

Answer: The expression does type check, because it is equivalent to ((.) even ceiling) in part (c), which does type check. Hence,

\[(\text{flip (.) ceiling even}) :: \text{Float} \rightarrow \text{Bool}\]
Problem 4. **User-Defined Polymorphic Lists.** Instead of the native (i.e., pre-defined) datatype of polymorphic lists, we want to use the following user-defined datatype:

```haskell
data List a = Nil | Cons a (List a)
```

To get full credit, you need to use **pattern-matching** and **wildcards** as much as possible throughout. In parts (a), (b) and (c), you must use **recursion** explicitly; in parts (d), (e) and (f), you must use **foldList** explicitly.

(a) Define the function `toList` that translates native lists to user-defined lists, e.g., the list “[3, 2, 7]” (shorthand for “(3:(2:(7:[]))))” is translated to “(Cons 3 (Cons 2 (Cons 7 Nil))).”

**Answer:**

```haskell
toList :: [a] -> List a
toList (x:xs) = Cons x (toList xs)
toList [] = Nil
```

(b) Define the function `fromList` that translates user-defined lists to native lists, e.g., the user-defined list “(Cons ‘a’ (Cons ‘d’ (Cons ‘c’ Nil)))” is translated to “(‘a’:(‘d’:(‘c’:[]))).”

**Answer:**

```haskell
fromList :: List a -> [a]
fromList (Cons x xs) = x : (fromList xs)
fromList Nil = []
```

(c) Define the function `foldList` which acts on user-defined lists just as `foldr` acts on native lists.

**Answer:**

```haskell
foldList :: (a -> b -> b) -> b -> List a -> b
foldList f init Nil = init
foldList f init (Cons x xs) = f x (foldList f init xs)
```

(d) Define the function `mapList` on user-defined lists, the counterpart of `map` on native lists. You must use `foldList` to get credit.

**Answer:**

```haskell
mapList :: (a -> b) -> (List a) -> (List b)
mapList f = foldList (Cons . f) Nil
```

(e) Define the function `sumList` which adds up the entries in an argument of type `List Int`. You must use `foldList` to get credit.

**Answer:**

```haskell
sumList :: (List Int) -> Int
sumList = foldList (+) 0
```

(f) Define the function `toZeroOneList` which takes a list of type `List Int` then turns every **even** integer to 0 and every **odd** integer to 1. For example, the list “(Cons 3 (Cons 2 (Cons 7 Nil)))” is transformed to “(Cons 1 (Cons 0 (Cons 1 Nil))).” You must use the function `foldList` to get credit.

**Answer:**
toZeroOneList :: (List Int) -> (List Int)
toZeroOneList = foldList help Nil
    where
        help m
            | (even m)    = Cons 0
            | (odd m)     = Cons 1