Programming I: Functional Programming in Haskell
Solutions to Unassessed Exercises

Set 4: Higher-order functions

1. (a) Error: `n` undefined
   (b) `[""]`
   (c) `True`
   (d) Error: `(:)` used with wrong types
   (e) `1`
   (f) Error: folded functions must be binary
   (g) `[True]`

2. There are two versions listed in each case. The second is "point free", i.e. is a pure function, or *combinator*, with no named arguments.

   (a)
   ```
   depunctuate s
     = filter p s
     where
       p c = not (elem c ".,:")

   depunctuate
     = filter (not . flip elem ".,:")
   ```

   (b)
   ```
   makeString ns
     = map chr ns

   makeString
     = map chr
   ```

   (c)
   ```
   enpower ns
     = foldr1 (flip (^)) ns

   enpower
     = foldr1 (flip (^))
   ```

   (d)
   ```
   revAll xs
     = concatMap reverse xs

   revAll
     = concatMap reverse
   ```

   (e)
   ```
   rev xs
     = foldl (flip (:)) [] xs

   rev
     = foldl (flip (:)) []
   ```

(f)
```
dezip ps
  = foldr f ([], []) ps
  where
    f (x, y) (xs, ys) = (x : xs, y : ys)

dezip
  = foldr (\(x, y) (xs, ys) -> (x : xs, y : ys)) ([], [])
```

Note: the point-free version is a bit of a cheat, as the inner function is a lambda expression with names arguments. You can make this point-free as well, but it gets very messy.

3.
```
same :: [Int] -> Bool
same xs
  = and (zipWith (==) xs (tail xs))
```

4. (a)
```
scanl (*) 1 [2..]
```

(b)
```
sum (map (1/) (scanl (*) 1 [1..5]))
```

(c) This generates the infinite list of fibonacci numbers by building a circularly-defined list. Lazy evaluation is essential for it to work. Try sketching out the evaluation of `xs`.

5.
```
squash :: (a -> a -> b) -> [a] -> [b]
squash f (x : y : ys)
  = f x y : squash f (y : ys)
squash f xs
  = []
```

But much neater:

```
squash f xs
  = zipWith f xs (tail xs)
```

Note: if `squash` is called with an empty list the expression `tail []` is never evaluated (check out the implementation of `zipWith` in the prelude).

6.
```
-- Pre: the list is non-empty
converge :: (a -> a -> Bool) -> [a] -> a
converge f [x]
  = x
converge f (x : y : ys)
  | f x y     = y
  | otherwise = converge f (y : ys)
```

whereupon

```
e :: Float
e = converge lim (scanl (+) 0 (map (1/) facts))
  where facts   = scanl (*) 1 [1..]
        lim x y = abs (x - y) / x < 0.00001
```

7.
```
limit :: (a -> a -> Bool) -> [a] -> [a]
limit f (x : y : ys)
  | f x y     = [x, y]
  | otherwise = x : limit f (y : ys)
limit f xs
  = xs
```

8. ```
   map :: (a -> b) -> [a] -> [b]
   map f
     = foldr g []
     where
       g x ys = f x : ys

   -- Alternatively...

   map f
     = foldr (\x ys -> f x : ys) []

   map f
     = foldr ((:) . f) []

   -- Or, point-free:

   map
     = flip foldr [] . ((:) .)

   filter :: (a -> Bool) -> [a] -> [a]
   filter p
     = foldr g [] xs
     where
       g x ys
         | p x       = x : ys
         | otherwise = ys

   zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
   zipWith f xs ys
     = foldr g [] (zip xs ys)
     where
       g (x, y) zs = f x y : zs
   ```

9. ```
   repeatUntil :: (a -> Bool) -> (a -> a) -> a -> a
   repeatUntil p f
     = head . filter p . iterate f
   ```

10. ```
    any' p
      = or  . map p
    all' p
      = and . map p
    ```

11. ```
    isElem :: Eq a => a -> [a] -> Bool
    isElem
      = any . (==)
    ```

12. (a) ```
        infixl 9 <.>
        (<.>) :: (a -> b) -> (c -> d -> a) -> (c -> d -> b)
        f <.> g
          = h
          where
            h x y = f (g x y)
        ```
        Alternatively,
        ```
        (f <.> g) x y
           = f (g x y)
        ```
        ```

Alternatively, using "lambda" notation for defining ambiguous functions:

```
f <.> g = \x -> \y -> f (g x y)
```

Alternatively, we can express the right-hand side in point-free form:

```
(f <.> g)
   = (f .) . g
```

(b) The point free form of `<.>` itself is:

```
(<.>) = (.) . (.)
```

This is sometimes called the 'dot' operator (e.g. see https://wiki.haskell.org/Pointfree).

(c)
```
any = or  <.> map
all = and <.> map
```

(d)
```
any = (or .) . map
all = (and .) . map
```

13.
```
pipeline :: [a -> a] -> [a] -> [a]
pipeline = map . foldr (.) id
```