Programming I: Functional Programming in Haskell
Solutions to Unassessed Exercises

Set 2: Functions

1. ```
   addDigit :: Int -> Int -> Int
   addDigit i d
     = 10 * i + d
   ```

2. ```
   convert :: Float -> Float
   convert c
     = 9 / 5 * c + 32
   ```

3. ```
   distance :: Vertex -> Vertex -> Float
   distance (x, y) (x', y')
     = sqrt ((x - x') ^ 2 + (y - y') ^ 2)
   ```

4. ```
   triangleArea v1 v2 v3
     = sqrt (s * (s - a) * (s - b) * (s - c))
   where
     a = distance v1 v2
     b = distance v2 v3
     c = distance v3 v1
     s = (a + b + c) / 2
   ```

5. ```
   isPrime :: Int -> Bool
   isPrime n
     | n == 2          = True
     | n < 2 || even n = False
     | otherwise       = checkFactorsFrom 3
     where
        rootn = floor (sqrt (fromIntegral n))
        checkFactorsFrom a
          | a > rootn      = True
          | n `mod` a == 0 = False
          | otherwise      = checkFactorsFrom (a + 2)
   ```

   Or, using a list comprehension...

   ```
   isPrime n
     = null [m | m <- 2 : [3, 5 .. floor (sqrt (fromIntegral n))],
                 n `mod` m == 0]
   ```

6. ```
   fact :: Int -> Int
   fact n
     | n == 0    = 1
     | otherwise = n * fact (n - 1)
   ```

7. ```
   perm :: Int -> Int -> Int
   perm n r
     | r == 0    = 1
     | otherwise = perm n (r - 1) * (n - r + 1)
   ```

8. ```
   choose :: :: Int -> Int -> Int
   choose n r
     | n == r    = 1
     | otherwise = choose (n - 1) r * n `div` (n - r)
   ```

9. ```
   remainder :: Int -> Int -> Int
   remainder a b
     | a < b     = a
     | otherwise = remainder (a - b) b
   ```

10. ```
    quotient :: Int -> Int -> Int
    quotient a b
      | a < b     = 0
      | otherwise = 1 + quotient (a - b) b
    ```

11. ```
    binary :: Int -> Int
    binary n
      | n < 2     = n
      | otherwise = binary (div n 2) * 10 + mod n 2
    ```

    For a different base, simply replace the 2's above.

12. ```
    add :: Int -> Int -> Int
    Pre: m, n >= 0
    add m n
      | m == 0    = n
      | otherwise = succ (add (pred m) n)
    ```

    Alternatively, a tail-recursive solution...

    ```
    add m n
      | m == 0    = n
      | otherwise = add (pred m) (succ n)
    ```

    Or even...

    ```
    add m n
      | m == 0    = n
      | n == 0    = m
      | otherwise = succ (succ (add (pred m) (pred n)))
    ```

13. This is tricky...!

    ```
    chop :: Int -> (Int, Int)
    chop n
      | n < 10    = (0, n)
      | otherwise = (1 + q, r)
      where
        (q, r) = chop (n - 10)
    ```

14. Compare this with append (++)...

```
concatenate :: Int -> Int -> Int
concatenate m n
  | n == 0    = m
  | otherwise = addDigit (concatenate m q) r
  where
    (q, r) = chop n
```

15. We take the 0^{th} fibonacci number to be 0:

```
fib :: Int -> Int
fib n
  | n == 0    = 0
  | n == 1    = 1
  | otherwise = fib (n - 1) + fib (n - 2)
```

Alternatively, and much more efficiently,

```
fib n
  = fib' 0 1 0
  where
    fib' f f' k
      | k == n    = f
      | otherwise = fib' f' (f + f') (k + 1)
```

Note that in a call `fib' f f' k`, `f` is the $\texttt{k}^{th}$ fibonacci number and `f'` is the $(\texttt{k}+1)^{th}$.

16.
```
goldenRatio :: Float -> Float
goldenRatio e
  = gr 1 2 1
  where
    gr f f' r
     | abs ((r - r') / r) < e = r'
     | otherwise              = gr f' (f + f') r'
     where
       r' = fromIntegral f' / fromIntegral f
```

Note: we start with 1 2 and 1 (i.e. 1 / 1) to avoid division by zero.