Programming I: Functional Programming in Haskell

Unassessed Exercises 2: **Functions** (Solutions)

These exercises are unassessed so you do not need to submit them. They are designed to help you master the language, so you should do as many as you can at your own speed.

There are probably more questions on these sheets than you may need in order to get the hang of a particular concept, so feel free to skip over some of the questions. You can always go back to them later if you need to.

Model answers to each set will be handed out throughout the course.

For each function you should also include its type signature. As an additional exercise, when you are happy that your function is correct, comment out the type signature and then return to the GHCi prompt. The system will automatically work out, i.e. *infer*, the type of your function and you can check this by typing `:type <fun>`, or `:t <fun>` where `<fun>` is the name of your function. In some cases you may be surprised to see that the inferred type is different to the type you originally wrote down. You may be able to work out what it's doing in these cases; if not the later lectures will help to explain.

1. Write a two-argument function `addDigit` which will add a single-digit integer onto the right-hand end of an arbitrary-size integer. For example, `addDigit 123 4` should evaluate to `1234`. Ignore the possibility of integer overflow.

   Solution:

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

2. Write a function convert which converts a temperature given in degrees Celcius to the equivalent temperature in Fahrenheit. (To convert from Fahrenheit to Celcius the rule is to subtract `32` from the temperature in Fahrenheit and then multiply the result by `5/9`.)

   Solution:

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

3. Given the type synonym type `Vertex = (Float, Float)` write a function `distance :: Vertex -> Vertex -> Float` that will calculate the distance between two points, each represented as a `Vertex`.

Solution:

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

4. Using the `distance` function write a function `triangleArea :: Vertex -> Vertex -> Vertex -> Float` that will calculate the area of the triangle formed by the three given vertices. Note that a triangle whose sides are of length a, b and c has an area given by

$$\sqrt{s(s-a)(s-b)(s-c)}$$

where $s = (a+b+c)/2$. Use a `where` clause to define the value of a, b, c and s.

Solution:

```
triangleArea :: Vertex -> Vertex -> Vertex
triangleArea u v w = sqrt (s * (s - a) * (s - b) * (s - c))
  where a = distance u v
        b = distance v w
        c = distance w u
        s = (a + b + c) / 2
```

5. Define a function `turns :: Float -> Float -> Float -> Float` such that `turns start end r` computes the number of turns of a wheel of radius `r` would have to make to travel the distance between `end` and `start`. You should use a `where` clause and good names to keep your function clearly understandable.

Solution:

```
turns :: Float -> Float -> Float -> Float
turns start end r = totalDistance / distancePerTurn
  where totalDistance = kmToMetres * (end - start)
        distancePerTurn = 2 * pi * r
        kmToMetres = 1000
```

6. Define a function `isPrime :: Int -> Bool` that indicates whether a positive number is prime or not. A number a is prime if it has exactly 2 distinct divisors that are 1 and a.

Hint: a is prime if and only if $a > 1$ and each number in the interval $[2, \sqrt{a}\,]$ does not divide a.

Solution: The function `null` will check if a list is empty: the list comprehension checks all the candidate factors, returning the list of divisors: if this list is empty and `n` is greater than 1, it must be prime.

```
isqrt :: Int -> Int
isqrt n = floor (sqrt (fromIntegral n))

isPrime :: Int -> Bool
isPrime n = n > 1 && null [x | x <- [2..isqrt n], n `mod` x == 0]
```

7. The factorial of a non-negative integer n is denoted as $n!$ and defined as:

$$n \times (n-1) \times (n-2) \times (n-3) \times \ldots \times 2 \times 1$$

0! is defined to be 1. Write a function `fact :: Integer -> Integer` which returns the factorial of a given non-negative integer.

Solution: While, you could use guards to check `n == 0`, it is much cleaner to just use pattern matching: when you can, pattern match!

```
fact :: Integer -> Integer
fact 0 = 1
fact n = n * fact (n - 1)
```

8. The number of arrangements (permutations) of r objects selected from a set of $n \geq r$ objects is denoted as nPr and defined as:

$$n \times (n-1) \times (n-2) \times \ldots \times (n-r+1) \quad \text{or} \quad \frac{n!}{(n-r)!}$$

Define a recursive function `perm` such that `perm n r` evaluates nPr for all non-negative integers n, r, $n \geq r$, without using the earlier `fact` function.

Solution:

```
perm :: Int -> Int -> Int
perm n 0 = 1
perm n r = perm n (r - 1) * (n - r + 1)
```

9. The number of ways of choosing r objects from a set of $n \geq r$ objects is denoted nCr, or $\binom{n}{r}$, and is defined as

$$\binom{n}{r} = \frac{n!}{r!(n-r)!}$$

Define $\binom{n}{r}$ in terms of $\binom{n-1}{r}$. Hence, define a recursive function `choose` such that `choose n r` computes $\binom{n}{r}$ for all non-negative integers n, r, $n \geq r$. Do not use the earlier `fact` or `perm` functions.

Solution:

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

10. Write a function `remainder` which computes the remainder after integer division. Implement the division by repeated subtraction, i.e. do not use the pre-defined `mod` function.

Solution:

```
remainder :: Int -> Int -> Int
remainder x y
  | x < y     = x
  | otherwise = remainder (x - y) y
```

11. Write a function `quotient` which similarly defines integer division using repeated subtraction. Ignore division by zero. Use guards to distinguish the base and recursive cases.

Solution:

```
quotient :: Int -> Int -> Int
quotient x y
  | x < y     = 0
  | otherwise = 1 + quotient (x - y) b
```

12. The binary representation of an integer n can be obtained by generating another integer whose digits, in base 10 representation, comprise only 0s and 1s thus:

    - If n is less than 2, its binary representation is just n.
    - Otherwise, divide n by 2. The remainder gives the last (rightmost) digit of the binary representation, which is either 0 or 1.
    - The preceding digits of the binary representation are given by the binary representation of the quotient from the previous step.

    Write a function `binary` that computes the binary representation of a given integer as described. As an example, `binary 19` should produce the number `10011`. How would you adapt this to compute the representation of an integer in an arbitrary given base $b \geq 2$?

Solution: For another arbitrary base, replace `base = 2`.

```
binary :: Int -> Int
binary n
  | n < base     = n
  | otherwise = binary n' * 10 + bit
  where (n', bit) = divMod n base
        base = 2
```

13. Given an integer argument the built-in "successor" function `succ` returns the next largest integer. For example, `succ 4` returns `5`. The function `pred` returns the next smallest. Using just `succ` and `pred`, i.e. without using `+` or `-`:

    (a) Write a recursive function `add` which will add two non-negative numbers.

    Solution: This is induction on integers!

    ```
    add :: Int -> Int -> Int
    add 0 n = n
    add m n = add (pred m) (succ n)
    ```

    (b) Write a recursive function `larger` to determine the larger of two positive integers.

    Solution: Here, both numbers should decrease, when one hits zero we know the other one is larger. However, since value has been removed during the recursion, we must be careful to add it back on as we unwind, hence the `succ` in the last case.

    ```
    larger :: Int -> Int -> Int
    -- Pre: m, n >= 0
    larger 0 n = n
    larger m 0 = m
    larger m n = succ (larger (pred m) (pred n))
    ```

    Use guards or multiple recursion equations and pattern matching. Remember that there are two parameters in each case, each of which can be either zero or non-zero. Do you need four equations in each case?

14. Write a recursive function `chop` which uses repeated subtraction to define a pair of numbers which represent the first `n-1` digits and the last digit respectively of the decimal represenation of a given number. Note that this is equivalent to the quotient and remainder after division by 10. However, you are not allowed to use `div`, `mod`, `remainder`, or `quotient`!

    Solution: We have already implemented this shape of function in `remainder` and `quotient`, so first step is to merge these together. The way to do this is to make a pair of results, one for the quotient, and the other for the remainder. Then, the base case is the pair of the base cases,

and the recursive case deconstructs the results of the recursive call and forms a new pair as a result. Notice how these are the two base cases respectively.

```
chop :: Int -> (Int, Int)
chop n = quotRem n 10

quotRem :: Int -> Int -> (Int, Int)
quotRem x y
  | x < y     = (0, x)
  | otherwise = let (q, r) = quotRem (x - y) y in (1 + q, r)
```

15. Using `chop`, write a function `concatenate` which concatenates the digits of two non-zero integers. For example, `concatenate 123 456` should evaluate to `123456` and `concatenate 123 0` should evaluate to `123`. Estimate the cost of your concatenate function in terms of the number of multiplications and subtractions it performs.

Solution: This is similar in idea to the definition of `(++)`, but `concatenate` does induction on the second number, and `(++)` does it on the first list.

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

16. The Fibonacci numbers are defined by the recurrence

$$\begin{aligned} f(0) &= 1 \\ f(1) &= 1 \\ f(n) &= f(n-1) + f(n-2), \quad n > 1 \end{aligned}$$

Write a Haskell function `fib` that, given an integer `n=0, 1, 2, ...` returns the `n`th Fibonacci number by encoding the above recurrence directly.

If you think about the way a call to this function will be evaluated you will notice that there is an enormous degree of redundancy. For a given n, $f(n-2)$ is evaluated twice, $f(n-3)$ three times, $f(n-4)$ five times etc. Notice any pattern?! A much more efficient way to compute the n^{th} Fibonacci number is to use an auxiliary function that includes as arguments the k^{th} and the $(k+1)^{th}$ Fibonacci numbers for some number k. The idea is that the recursive call is made with the $(k+1)^{th}$ and the $(k+2)^{th}$ Fibonacci numbers - can you see how to generate them from the k^{th} and the $(k+1)^{th}$? You will also need to carry a third argument which records either k or the number of remaining calls to make before one of the arguments is the value, $f(n)$, you need. Define an auxiliary function `fib'` which works in this way and redefine the original `fib` function to call `fib'` appropriately.

Solution: As usual, pattern matching is our friend:

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

This is called "tail recursion", where arguments are fed-through as accumulators to be returned at the end. If you are familiar with looping constructs in other programming languages, tail-recursion is equivalent.

```
fib' :: Int -> Integer
fib' n = go n 1 1
  where go :: Int -> Integer -> Integer -> Integer
        go 0 cur _ = cur
        go n cur next = go (n - 1) next (cur + next)
```

17. The Golden Ratio, G, is a beautiful number discovered by the ancient Greeks and used to proportion the Parthenon and other ancient buildings. The golden ratio has this property: take a rectangle whose sides (long:short) are in the Golden Ratio. Cut out the largest square you can and the sides of the rectangle that remains are also in the Golden Ratio. From this you should be able to work out that $G = \frac{1+\sqrt{5}}{2}$. Curiously, it can also be shown that the ratio of consecutive Fibonacci numbers converges on the Golden Ratio. Defining $r_n = \frac{f(n)}{f(n-1)}$, this says:

$$lim_{n\to\infty}\frac{f(n)}{f(n-1)} \;=\; lim_{n\to\infty} r_n \;=\; G$$

where f is the fibonacci function defined above. Inspired by your earlier `fib'` function build a function `goldenRatio` which takes an accuracy threshold value, `e`, of type `Float` and which returns r_n, where `n` is the smallest integer satisfying

$$|r_n - r_{n-1}| < e$$

Hint: you should start with your optimised `fib'` function from the previous question. To save re-calculating the same ratio twice, you should carry round the the ratio of the previous two numbers you have just computed, i.e. r_k for some k, as a third argument. Use a where clause to define the ratio of the two fibonacci numbers you are carrying round. Note that the absolute value of a number can be obtained using Haskell's `abs` function. Also, to perform the division above, you must first *cast* the integers representing the various Fibonacci numbers into floating-point numbers using the built-in function `fromIntegral`. For now you can think of this function as having type `fromIntegral :: Int -> Float`. This isn't quite right, however: the true picture will emerge later in the course.

Solution: We start with the ratio between the first two numbers `1` and `1`, which is `1`, and then the initial "current fib" is the second of these `1`s, and the "next fib" is then `2`. This gives the initial seeding of `go 1 2 (1/1)`.

```
goldenRatio :: Float -> Float
goldenRatio threshold = go 1 2 1
  -- nobody said Haskell names cannot be descriptive
  where go fibCur fibNext prevRatio
          | abs (prevRatio - ratio) < threshold = ratio
          | otherwise = go fibNext (fibCur + fibNext) ratio
          where ratio = fromIntegral fibNext / fromIntegral fibCur

-- 1.618033988749895
g = goldenRatio 0.0000000000000001
```