CS119 – Module 7: Currying and Partial Application

Purpose: Another way of creating an abstraction is having a function return another function through partial application. This will allow your code to be more flexible and reusable.

Knowledge: This module will help you become familiar with the following content knowledge:

- Use of partial application to create a function that returns another function

Activity: With your group perform the following tasks and answer the questions. You will need to copy the `lab7` directory. You will be reporting your answers back to the class in 20 minutes.

Partial application of a function is using the function but not supplying all the arguments.

1. Enter the following directly into REPL:
   ```
   > let add x y = x + y
   > :type add
   ```

   The `Num a =>` part means that the type of `a` is a number so it could be an Int or Float or Double. Therefore the `a->a->a` indicates that x and y must be numbers and the return value is also a number.

2. Suppose we put in some parentheses on the type like `a -> (a -> a)`. If we write it this way, it looks like if we supply one argument of type a, we should get a function of type `(a-> a)`. That means that `add 3` should give us a function. Let's try it by naming f to be the function that add 3 returns:

   ```
   let f = add 3
   :type f
   ```

   Explain the type of `f` which is the result of partial application.

3. **Predict** what you would expect from:
   ```
    > f 4
   ```

4. Try partial application on the function `twice`.

   ```
   > let twice f x = f (f x)
   > let square x = x * x
   >:type twice
   > let g = twice (+1)
   > let h = twice square
   ```

   What are the types for the functions `g` and `h`?

5. Try partial application on function from the Words module
   `>:load Words.hs`

   First take a look at the types of `every` and `accumulate`:

   ```
   >:type every
   >:type accumulate
   ```

   Now we will try partial application on those functions:

   ```
   > let f = every firstItem
   > let g = accumulate (+++)
   ```

   What are the types of the functions `f` and `g` and what do those functions do?

Complete the following assignments to be submitted for grading. Each should be done individually but you can consult with a classmate to discuss your strategies or if you get an error message that you do not understand.
Write all of your functions in the file `Lab7.hs`.

Have you ever wondered how a sales representative knows when they have typed in an incorrect credit card number or a scanner knows when it fails to read a UPC barcode correctly? These are examples of *self-verifying numbers*. They are designed to have a certain property that will fail if a simple error occurs, like changing a digit. A self-verifying number with the rightmost digit d_1, second digit d_2, etc. will satisfy the following property for a specific function f and divisor m:

$$f(1, d_1) + f(2, d_2) + f(3, d_3) + \text{...is divisible by } m \tag{1}$$

We will write a function `makeVerifier` that takes as arguments f and m and returns a function that will perform a verification test for a given number. In other words `makeVerifier` is a function factory that makes the verifying functions for us.

Before we do this, let's consider a particular function that this factory is supposed to produce. Suppose we want to check that the sum of the digits is divisible by 17. That is, $f(i, d_i) = d_i$ and m is 17. The code in `Lab7.hs` gives you the functions to get the last digit of a number (by using the remainder of division by 10) and everything but the last digit (by using integer division). We can use these to create a function `sumOfDigits` (also provided). Make sure you understand this code and try it out.

Assignment 1:
Write a function `divisibleSum :: Integer -> Integer -> Bool` which takes a divisor m and a number *num* and determines whether the **sum of the digits** of *num* is divisible by m.

Criteria for Success: Test your function with various m and *num* values and verify that your function correctly determines whether **sum of the digits** of *num* is divisible by m. For example, the sum of the digits of 123 is 6 so would be divisible by 3 but not divisible by 4.

Consider defining:

```
divisibleSumBy17 = divisibleSum 17
```

We have used partial application to get a new function. The expression `divisibleSumBy17 num` will now verify that the sum of the digits of *num* is divisible by 17. We can now generalize this approach to achieve our function factory `makeVerifier`.

Assignment 2:
Write a function `digitSum` which takes a function f and a number num and computes $f(1, d_1) + f(2, d_2) + f(3, d_3) + ...$ for all the digits in num.

Criteria for Success: One example of using this function is `digitSum (*) 234` which should compute `1*4 + 2*3 + 3*2` which is 16.

Assignment 3:
Put the pieces together to write the function

```
makeVerifier :: (Integer->Integer->Integer) -> Integer -> (Integer->Bool)
makeVerifier f m = ...
```

Hint: Write a helper function `verify f m n` which returns true or false on whether the number `n` is verified or not. Then appropriately use partial application of this function to define `makeVerifier`

Criteria for Success: A simple example of a self verifying number is an ISBN number. The ISBN numbers use the function $f(i, d_i) = i * d_i$ and a divisor of 11. We can use our function factory to create a function which checks ISBN numbers:

```
checkISBN = makeVerifier (*) 11
```

For example, take the legal ISBN 0-201-53082-1 since 1*1 + 2*2 + 3*8 + 4*0 + 5*3 + 6*5 + 7*1 + 8*0 + 9*2 + 10*0 = 99 = 0 (mod 11)
You should now be able to verify this with `checkISBN 0201530821`
Change a digit and verify that it detects that you no longer have a legal ISBN.

Assignment 4:

8 92685 00100 3

UPC barcodes use a divisor of 10 and the function $f(i, d_i) = d_i$ when i is odd and $3d_i$ when i is even. Build a verifier `checkUPC`.

Criteria for Success: The UPC number consists of all the digits: the one to the left of the bars, the ones underneath the bars, and one on the right. Check out your function by using the valid UPC code above. Change a digit and verify that it detects that the code is no longer valid.

Assignment 5:
Credit card numbers also have a divisor of 10 and use the function $f(i, d_i) = d_i$ when i is odd. But when i is even, the function returns and $2d_i$ when $d_i < 5$ and $2d_i + 1$ otherwise. Build a verifier for checking credit card numbers.

Criteria for Success: Try out your function with the account number 79927398713. Change a digit and verify that it detects that the code is no longer valid.

Submit your `Lab7.hs` file in Canvas for grading.