HASKELL PROGRAMMING PROBLEM SET 2

LENNART JANSSON AND BRANDON AZAD

Read Chapter 4 of *Learn You a Haskell*, then implement the following functions. Use pattern matching, `let` and `where` bindings, and `case` expressions as appropriate to produce elegant code.

Function Syntax

Problem 1. Days of the week

Write a function `dayName :: Int -> String` that takes a number 0–6 and gives the corresponding name of the day of the week. If out of range, return an error message.

```
> dayName 0
"Sunday"
> dayName 4
"Thursday"
> dayName 11
"Day out of range"
```

Problem 2. Passwords

Write a function `validatePassword :: String -> Bool` that takes a potential password and returns `True` if it's a valid password and `False` if not. A password is valid if it is longer than 5 characters and shorter than 20 characters.

```
> validatePassword "bad"
False
> validatePassword "goodPassword"
True
> validatePassword "exceedinglyLongPassword"
False
```

To do the next problem you might need the function `sqrt :: (Floating a) => a -> a`, which does exactly what it says on the tin. `Float` is an instance of the typeclass `Floating`.

Problem 3. Triangles

Let's represent triangles as 3-tuples `(Float, Float, Float)` where the three `Float`s are the lengths of the sides of the triangle.

Write a function `analyzeTriangle :: (Float, Float, Float) -> (Float, Float)` that takes a triangle and returns a tuple of the triangle's area and its perimeter. If the triangle is invalid (does not respect the triangle inequality), then return `(-1.0, -1.0)` to signify an error.

```
> analyzeTriangle (1, 1, 1)
(0.4330127, 3.0)
> analyzeTriangle (1, 1, sqrt 2)
(0.5000001, 3.4142137)
```

```
> analyzeTriangle (3, 1, 1)
(-1.0, -1.0)
```

Don't worry if your numbers are very slightly off as shown here, `Floats` are imprecise, having the same internal representation as floats in C.

Recursive Functions

Read Chapter 5 of *Learn You a Haskell.*

Problem 4. Prelude

Implement the following functions that are defined in Prelude. Since they're part of every Haskell file and `ghci` by default, you'll have to name each one something else.

(1) `gcd :: Integral a => a -> a -> a`, which finds the greatest common denominator of two integral numbers.
(2) `(!!) :: [a] -> Int -> a`, which is used for indexing into a list. `foo !! 5` gets the 6th element in the list `foo`, for example.
(3) `init :: [a] -> [a]`, which takes a list and removes the last element.
(4) `cycle :: [a] -> [a]`, which makes a finite list into an infinite one by repeating the elements in the input list over and over.

This is a really great way to generate lots of exercises for yourself if you want more. You already know how to implement many functions in Prelude, and many in the module Data.List. Read the first bit of Chapter 7 of *Learn You a Haskell* for more information on Data.List.

Tail-call recursion

Tail-call recursion is a special recursive pattern that deserves special attention, even though it's not discussed in *Learn You a Haskell.* Tail-call recursive functions are a subset of recursive functions, with the property that for every recursive call, the result of that recursive call is returned directly rather than being modified before returned. For every recursive pattern, the function must be defined as a call to itself with different parameters. To make things clearer, here's a factorial function implemented in a non-tail-call recursive way.

```
fact1 :: Int -> Int
fact1 0 = 1
fact1 n = n * (fact1 (n - 1))
```

Notice how the result of the recursive call to `fact1` is multiplied by `n` before being returned, therefore this function is non-tail-call recursive. Here's the same function done tail-call recursively.

```
fact2 :: Int -> Int
fact2 n = fact2' n 1
  where fact2' 0 a = a
        fact2' n a = fact2' (n - 1) (a * n)
```

Notice how the recursive call to the inner helper function `fact2'` is returned directly by `fact2'` instead of being modified. Like in this example here, tail-call recursive functions often require an extra parameter to hold an accumulator value or the result of a partial computation while function is recursing. Since the value of a tail-call recursive function is returned directly without being modified first, the modification must be done to parameters which are passed back into the recursive function.

Why do we care about tail-call recursion? When Haskell programs are compiled, tail-call recursive functions can be optimized so that instead of requiring the use of the call stack in assembly to handle each recursive call, the recursive computation can be modeled by a while loop and thus there is no risk of consuming too much memory! This very convienent process is known as tail-call optimization or tail-call elimination. The reason is that with tail-call recursive functions, the parameters of the function can become local variables that are continually modified with each recursive call; when the base case is reached, no other computation has to be done since the value of the function is returned unmodified all the way up the call stack. With non-tail-call recursive functions, the value returned from the function might be modified by the parameters in every stack frame, and there's no clever way to optimize that out without storing every stack frame in memory separately.

If you have not taken CS107 yet, this may be confusing; don't worry, the internal representations of function calls and the stack is taught thoroughly in that class. The important thing to take away is that tail-call recursive functions are often faster and more efficient to calculate than other recursive functions in Haskell. Here are some problems to play with tail-call recursion.

Problem 5. Implementing sum

Here is a version of the `sum` function that is not tail-call recursive. Reimplement it so that it uses tail-call recursion.

```
sum1 :: Num a => [a] -> a
sum1 [] = 0
sum1 (n:ns) = n + (sum1 ns)
```

Problem 6. Implementing reverse

Here is a version of the `reverse` function that is not tail-call recursive. Reimplement it so that it uses tail-call recursion.

```
reverse1 :: [a] -> [a]
reverse1 [] = []
reverse1 (x:xs) = (reverse xs) ++ [x]
```

Actually, this implementation is very bad for other reasons as well. The operation `a ++ b` is $O(|\mathtt{a}|)$, while cons-ing elements to the front of a list is only $O(1)$. This is due to the way lists are represented internally in Haskell. More about this later!

Problem 7. Nested parentheses

Write a function `ppParens :: String -> String` that pretty prints a string of nested parentheses, so that every parenthesis gets its own line, and is indented with tabs corresponding to its nested depth. Use tail-call recursion.

In order to print a string in `ghci` to see if it's formatted correctly, you can use the function `putStrLn`, which is a function in the `IO` monad and therefore the details of which won't be discussed here.

```
> putStrLn "hello\nworld"
hello
world
> putStrLn (ppParens "(())((()())())")
(
    (
    )
)
(
    (
        (
        )
        (
        )
    )
    (
    )
)
```

Remember that building up a list by appending things to its back is much slower than building up a list by cons-ing things to its front.

Problem 8. Mergesort

Write a function `mergesort :: Ord a => [a] -> [a]` which implements the mergesort algorithm.

```
> mergesort [4, 2, 7, 6, 2]
[2, 2, 4, 6, 7]
```

Challenge problems

This one shouldn't be too difficult if you've taken CS106B/X. No need to limit yourself to tail-call recursion here, though it might still be useful. Don't just brute-force, be smart about your implementation.

Problem 9. Coprime permutations

Write a function `coprimePerm :: [Int] -> [Int]` that finds a permutation of the input list such that adjacent integers are coprime. If none exists, return `[-1]`. Recall that two integers are coprime iff they have no common factors greater than 1.

```
> coprimePerm [5, 8, 9, 3, 10, 2, 12, 25, 4]
[8, 5, 2, 9, 10, 3, 4, 25, 12]
```