COMP3258: Functional Programming

Assignment 1

Deadline: 23:59, Mar 05, 2025 (HKT)

---

1. Please do not import other modules in your code.
2. For absurd cases (don't be confused with base case), feel free to raise exception (e.g. using `undefined`) or give a default value for it.
3. Please submit a single Haskell file, named as `A1_XXX.hs`, with `XXX` replaced by your UID, and follow all the type signatures strictly. In the case that Moodle rejects your .hs file, please submit it as `A1_XXX.zip`, which includes only one file `A1_XXX.hs`.
4. Please do this assignment on your own; if, for a small part of an exercise, you use something from the Internet or were advised by your classmate, please mark and attribute the source in a comment, and explain it in a detailed manner. Do not use publicly accessible code sharing websites for your assignment to avoid being suspected of plagiarism.
5. The style and quality of the code will also be assessed. Please follow the Haskell style guide to write your code.

Part 1: Warm-up (20 pts)

Problem 1 (10 pts)

A *Tetranacci sequence* is a number sequence that starts with four predetermined terms. Afterwards, each term is the sum of the preceding four terms. To simplify:

```
0, 0, 0, 1, 1, 2, 4, 8, 15, 29, 56, 108, 208, 401, ...
```

Your task is to implement a function `tetranacci` that outputs the n-th tetranacci number.

```haskell
tetranacci :: Int -> Int
```

Expected Results

```haskell
tetranacci 1 == 0
tetranacci 4 == 1
tetranacci 6 == 2
tetranacci 10 == 29
```

Problem 2 (10 pts)

Sorting a list in ascending order is quite boring, what about making it a pyramid (the first half in ascending order, and the second half in descending order)? Define the function:

```haskell
pyramid :: [Int] -> [Int]
```

As the name suggested, this function should make a "pyramid" out of its input list:

- The first element is the smallest one.
- The last element is the second smallest one.
- The second element is the third smallest one.
- The second to last element is the fourth smallest one.
- …

Notice

- The return list should have the same length as the input, and contain all the elements in the input list.
- Duplication is allowed.

Expected Results

```
pyramid [] == []
pyramid [1,2] == [1,2]
pyramid [4,1,2,0,3] == [0,2,4,3,1]
pyramid [1,2,3,4,5,6] == [1,3,5,6,4,2]
```

Part 2: Work with Lists (20 pts)

Problem 3 (10 pts)

Define a list to be a Martian list if the number of 1s is greater than the number of 2s and no two adjacent elements are equal. Write a function named `isMartian` that returns `True` if its list argument is a Martian list, otherwise it returns `False`.

```
isMartian :: [Int] -> Bool
```

Expected Results

```
isMartian [] == False
isMartian [1,3] == True
isMartian [1,2] == False
isMartian [1,1,1,2,3] == False
```

Problem 4 (10 pts)

You are going to implement a simple transposition cipher. To encrypt the plain text `HELLOWORLD`, we first put down the plain text in a zig-zag shape like the following:

```
H . L L . W O . L D
. E . . O . . R . .
```

Then by reading off the text horizontally, we can get the upper part `HLLWOLD` and the lower part `EOR`. The cipher text will be the concatenation of them, which is `HLLWOLDEOR`.

Similarly, encrypting the plain text `WUTHERINGHEIGHTS` gives us the cipher text `WTHRIGHIGTSUENEH`.

```
W . T H . R I . G H . I G . T S
. U . . E . . N . . E . . H . .
```

To decrypt the cipher text, you first need to split the cipher text at the correct position, such that the first half corresponding to the upper part and the second half corresponds to the lower part. The exact place to split the list can be done by analyzing the length of the cipher text.

For example, we need to split the cipher text `FENHFIERCRS` into the upper part `FENHFIE`, and the lower part `RCRS`.

```
F . E N . H F . I E . | R C R S
                      ^ position to split
```

After splitting the cipher text, we can put the two substrings at the correct place. By reading the text in a zig-zag way, we can restore the plain text `FRENCHFRIES`.

```
F . E N . H F . I E .
. R . . C . . R . . S
```

Your task is to implement the encryption function `cipher` and the decryption function `decipher`. Type signatures of both functions are given below:

```haskell
cipher :: String -> String
decipher :: String -> String
```

Expected Results

```haskell
cipher "FRENCHFRIES" == "FENHFIERCRS"
decipher "FENHFIERCRS" == "FRENCHFRIES"
cipher "ABCDEF" == "ACDFBE"
decipher "ACDFBE" == "ABCDEF"
```

> *Hint 1: a `String` in Haskell is just a `List` of `Char`(s).*
>
> *Hint 2: how many possible cases (respect to the length of the input) are there?*
>
> *Hint 3: you may find the function **`splitAt`** and **`divMod`** from Prelude helpful :D*

Part 3: Adding Two Numbers (20 pts)

In this part, we will learn how to add two numbers using long addition. (A version slightly different from the one you've learnt in primary school).

1. Align the two numbers by their least significant digits (the rightmost nonzero digit), and assume the initial *carry* to be 0.
2. Start from the rightmost column, and add up the two numbers and the carry.
   - If the sum is smaller than 9, put down the sum as the answer digit, and set the carry to 0.
   - If the sum is greater than 9, put down the sum *modulo 10* as the answer digit, and the carry now becomes 1.
3. Move leftwards and repeat step 2 until there are no numbers left.

```
   4  5  6
+     2  9
____0__1__0_
   4  8  5
```

Take $456 + 29$ as an example:

1. Let the initial the carry $c = 0$, and we compute $6 + 9 + 0 = 15$. Since $15 > 9$, we put $15 \bmod 10 = 5$ as the answer digit and carry $c = 1$.
2. For the second pair of digit, we have $5 + 2 + 1 = 8$, so we have 8 as the answer and $c = 0$.
3. For the last one, we get $4 + 0 + 0 = 4$, so we put down 4.

Problem 5 (10 pts)

To start, we need to convert integers to digits and vice versa. Define the functions:

```haskell
toDigits :: Int -> [Int]
fromDigits :: [Int] -> Int
```

The function `toDigits` should convert an integer to a list of digits. The least significant digit should be the first element of the list (i.e. writing the number backwards). For 0 and negative inputs, `toDigits` should return an empty list.

The function `fromDigits` should convert a list of digits back to an integer. If the input is an empty list, just return 0. You can assume the input of this function is always valid.

Expected Results

```haskell
toDigits (-1) == []
toDigits 1234 == [4,3,2,1]
```

```
fromDigits [] == 0
fromDigits [1,2,3,4] == 4321
```

Problem 6 (5 pts)

The next step is to implement addition with carry. Define the function:

```
addCarry :: Int -> Int -> Int -> (Int, Int)
```

The function `addCarry` takes 3 inputs (the *carry*, a digit from the first number and a digit from the second number) and returns a tuple. The first element of the tuple should be the *carry*, and the second element of the tuple should the *answer* digit.

Expected Results

```
addCarry 0 2 2 == (0, 4) -- carry = 0, answer = 4
addCarry 0 7 6 == (1, 3) -- carry = 1, answer = 3
addCarry 1 2 3 == (0, 6)
addCarry 1 5 5 == (1, 1)
```

Problem 7 (5 pts)

Finally, we can implement the actual addition. Define the function:

```
addDigits :: [Int] -> [Int] -> [Int]
```

The function `addDigits` should perform the addition operation described at the beginning of this part.

Expected Results

```
addDigits [6,5,4] [9,2] == [5,8,4]
addDigits [9,9] [8,8] == [7,8,1]
```

Just one more thing...

After finishing the previous problems, you can put down the following definition:

```
add :: Int -> Int -> Int
add m n = fromDigits $ addDigits (toDigits m) (toDigits n)
```

Now you can check if everything is implemented correctly.

```
ghci> add 12 34 == 12 + 34
True
```

Part 4: Define a High-order Function (20 pts)

Problem 8 (10 pts)

Consider a function `unique :: Eq a => [a] -> [a]` that removes all the duplicated elements in a list. The equality test of this function relies on the function `(==)` from the class `Eq`.

```
unique [1,1,2,2,3] == [1,2,3]
unique [True, False, False] == [True, False]
```

Here we consider a generalized version of it which lets us to use a predicate `p :: a -> a -> Bool` for testing the "equality". Any two elements satisfying the predicate are considered as duplicates. Try to implement the following function using recursion.

```
uniqueBy :: (a -> a -> Bool) -> [a] -> [a]
```

Expected Results

```haskell
uniqueBy (==) [] == []
uniqueBy (==) [1,1,2,2,3,4] == [1,2,3,4]
uniqueBy (\a b -> (a `mod` 3) == (b `mod` 3)) [2,3,5,9] == [2,3]
uniqueBy (\a b -> length a == length b) ["hello", "world", "bye"] == ["hello","bye"]
```

Problem 9 (10 pts)

In *Prelude*, `scanl` is a function similar to `foldl`, but returns a list of successive reduced values from the left (including the initial value):

```haskell
scanl :: (b -> a -> b) -> b -> [a] -> [b]
-- example
scanl (+) 0 [1,2,3] == [0,1,3,6] -- [0, 0 + 1, (0 + 1) + 2, ((0 + 1) + 2) + 3]
```

For infinite lists, expressions like `scanl (+) 0 [1..]` will produce an infinite list as a result, which can be inconvenient sometimes. We want to implement an improved version of `scanl` which accepts a predicate (if the current reduced value doesn't satisfy the predicate, the function terminates and returns all previous values) to deal with infinite lists.

```haskell
scanlTill :: (b -> Bool) -> (b -> a -> b) -> b -> [a] -> [b]
```

Consider running the example `scanlTill (< 10) (+) 0 [1..]` :

```haskell
[0, 1, 3, 6, 10, ...]
         --  ^ 10 < 10 == False, so the function terminates
```

And we get the result `[0, 1, 3, 6]`.

Expected Results

```haskell
scanlTill (< 0) (+) 0 [] == []
scanlTill (>= 0) (+) 0 [] == [0]
scanlTill (< 30) (+) 0 [1..] == [0,1,3,6,10,15,21,28]
scanlTill (< 50) (*) 1 [1..] == [1,1,2,6,24]
```

Part 5: Hanoi (20 pts)

The Towers of Hanoi is a classic puzzle with a solution that can be described recursively. Disks of different sizes are stacked on three pegs; the goal is to get from a starting configuration with all disks stacked on the first peg to an ending configuration with all disks stacked on the last peg.

The only rules are:

- you may only move one disk at a time, and
- a larger disk may never be stacked on top of a smaller one.

For example, as the first move all you can do is move the topmost, smallest disk onto a different peg, since only one disk may be moved at a time.

To move n discs (stacked in increasing size) from peg A to peg B using peg C as temporary storage,

1. move $n - 1$ discs from A to C using B as temporary storage.
2. move the top disc from A to B.
3. move $n - 1$ discs from C to B using A as temporary storage.

Define a function `hanoi` as follows:

```haskell
type Peg = String
type Move = (Peg, Peg)
```

```haskell
hanoi :: Integer -> Peg -> Peg -> Peg -> [Move]
hanoi discs fromPeg toPeg storePeg = undefined
```

Given the number of discs and names for the three pegs, `hanoi` should return a list of moves to be performed to move the stack of discs from the first peg to the second.

Expected Results

```haskell
hanoi 0 "a" "b" "c" == []
hanoi 1 "a" "b" "c" == [("a", "b")]
hanoi 2 "a" "c" "b" == [("a","b"),("a","c"),("b","c")]
```