Haskell – Lab 3

Prepared by Dr. Wooi Ping Cheah

Solution for the Exercises from Chapter 3 – Types and Classes

Haskell Chapter 3 Answers

(1)

Expression	Type
`['a','b','c']`	`[Char]`
`('a','b','c')`	`(Char, Char, Char)`
`[(False, '0'),(True, '1')]`	`[(Bool, Char)]`
`([False,True],[ '0', '1'])`	`([Bool], [Char])`
`[tail,init,reverse]`	`[ [a] -> [a] ]`

(2)

```
second :: [a] -> a
swap :: (a,b) -> (b,a)
pair :: a -> b -> (a, b)
double :: Num a => a -> a
palindrome :: Eq a => [a] -> Bool
twice :: (a -> a) -> a -> a
```

How to infer the type of **twice f x = f (f x)**:

Start at most general type definition and infer more concrete type.

Supposing the argument of function **f** has type **a** (i.e., **x :: a**), and the return value of function **f** has type **b** (i.e., **f :: b**), then function **twice** has type:

twice :: (a -> b) -> a -> b

Since (**f x**) is also an argument of function **f** (i.e., **f (f x)**), hence (**f x**) must also have type **a** (i.e., **b = a**). Substitute **b** with **a** in the above:

twice :: (a -> a) -> a -> a

Solution for the Exercises from Chapter 4 – Defining Functions

```haskell
-- Slide 14
add = \x -> (\y -> x + y)

-- Slide 16
odds n = map (\x -> x*2 + 1) [0..n-1]

(1)
safetail1 :: [a] -> [a]
safetail1 xs = if null xs then [] else tail xs

safetail2:: [a] -> [a]
safetail2 xs | null xs    = []
             | otherwise = tail xs

safetail3:: [a] -> [a]
safetail3 [] = []
safetail3 xs = tail xs
```

(2)

```
(||!) :: Bool -> Bool -> Bool
True  ||! True  = True
True  ||! False = True
False ||! True  = True
False ||! False = False

(||!!) :: Bool -> Bool -> Bool
False ||!! False = False
_  ||!! _ = True

(||!!!) :: Bool -> Bool -> Bool
False ||!!!  b = b
True  ||!!! _ = True
```

(3)

```
(&&!) :: Bool -> Bool -> Bool
x &&! y = if x then
                    if  y then True else False
               else False
```

(4)

```
(&&!!) :: Bool -> Bool -> Bool
x &&!! y = if x then y else False
```

Exercises
from
Chapter 5 – List Comprehensions

(1) A triple (x,y,z) of positive integers is called pythagorean if $x^2 + y^2 = z^2$. Using a list comprehension, define a function

```
pyths :: Int → [(Int,Int,Int)]
```

that maps an integer n to all such triples with components in [1..n]. For example:

```
> pyths 5
[(3,4,5),(4,3,5)]
```

Hints:

This question is relatively simple. Please try it yourself before you look at the code skeleton, as shown in the next page.

```
pyths :: Int -> [(Int,Int,Int)]
pyths n = [(x,y,z) | x <- r, y <- r, z <- r,                    ]
                    where r = [1..n]
```

An equation involving x, y, and z

(2) A positive integer is perfect if it equals the sum of all of its factors, excluding the number itself. Using a list comprehension, define a function

```
perfects :: Int → [Int]
```

that returns the list of all perfect numbers up to a given limit. For example:

```
> perfects 500

[6,28,496]
```

Hints:

(1) A perfect number is defined as a positive integer that can be expressed as the sum of its proper factors (factors except for the number itself). For example, the factors of 6 are 1, 2, 3 and 6; and the proper factors of 6 are 1, 2, 3 (excluding 6 itself). We say that 6 is a perfect number because 6 = 1+2+3.

(2) For this question, you may write your function called **perfect** by using the function **factors** defined on page 9 of Chapter 5.

```haskell
factors :: Int -> [Int]
factors n = [x | x <- [1..n], n `mod` x == 0]

perfects :: Int -> [Int]
perfects n = [x | x <- [1..n], x ==            - x]
```

Sum of the number's proper factors

To exclude the number itself

(3) The <u>scalar product</u> of two lists of integers xs and ys of length n is give by the sum of the products of the corresponding integers:

$$\sum_{i=0}^{n-1} (xs_i * ys_i)$$

Using a list comprehension, define a function that returns the scalar product of two lists.

Hints:
For example, the scalar product of two lists: [1,2,3] and [4,5,6], will be 32, which is the sum of the products (1 * 4) + (2 * 5) + (3 * 6). You may like to use the **zip** function for this task (Page 12-13 of Chapter 5).

```
scalar :: [Int] -> [Int] -> Int
scalar xs ys =       [x*y | (x,y) <-      xs ys]
```

Two built-in functions

Exercises
from
Chapter 6 – Recursive Functions

(1) Without looking at the standard prelude, define the following library functions using recursion:

- Decide if all logical values in a list are true:

```
and :: [Bool] → Bool
```

- Concatenate a list of lists:

```
concat :: [[a]] → [a]
```

Hints:

(1) Applying **myAnd** function to a list of logical values will return true if the head of the list is true, and applying **myAnd** function to the tail of the list will also return true.
Example: **ghci> myAnd [True, True, True] returns True**

(2) To **myConcat** (i.e, concatenate) a list containing **n** element lists is to first **myConcat** the tail of the list (containing **n-1** element lists) then the result will be appended to the head of the list using **++** standard function.
Example: **ghci> myConcat [[1,2,3],[4,5,6],[7,8,9]] returns [1,2,3,4,5,6,7,8,9]**

- Produce a list with n identical elements:

```
replicate :: Int → a → [a]
```

- Select the nth element of a list:

```
(!!) :: [a] → Int → a
```

- Decide if a value is an element of a list:

```
elem :: Eq a ⇒ a → [a] → Bool
```

Hints:

Please work out the solution yourself, without any specific hint, for the three questions above.

(1) Example: **ghci> myReplicate 3 [1,2,3,4] returns [[1,2,3,4], [1,2,3,4], [1,2,3,4]]**

(2) Example: **ghci> [1,2,3,4,5,6] !!! 3 returns 4**

(3) Example: **ghci> myElem 4 [1,2,3,4,5,6] returns True**

(2) Define a recursive function

```
merge :: Ord a ⇒ [a] → [a] → [a]
```

that merges two sorted lists of values to give a single sorted list. For example:

```
> merge [2,5,6] [1,3,4]

[1,2,3,4,5,6]
```

```haskell
myMerge :: Ord a => [a] -> [a] -> [a]
myMerge xs [] = xs
myMerge [] ys = ys
myMerge (x:xs) (y:ys) | x<y          = x:(myMerge xs (y:ys))
                      | otherwise =
```

Try to work out this part by comparing it to the above, when x<y.
You may also refer to the two base cases above.

(3) Define a recursive function

```
msort :: Ord a ⇒ [a] → [a]
```

that implements <u>merge sort</u>, which can be specified by the following two rules:

- Lists of length $\leq$ 1 are already sorted;
- Other lists can be sorted by sorting the two halves and merging the resulting lists.

Hints:

(1) Example: **<u>ghci> msort [1,6,2,5,3,4] returns [1,2,3,4,5,6]</u>**

```
myMsort :: Ord a => [a] -> [a]
myMsort [] = []
myMsort [x] = [x]
myMsort xs = myMerge (myMsort (take          xs)) (myMsort (drop          xs))
                  where halflen = length xs `div` 2
```

This should return an integer