Exercises, Week 41 (06–10 Oct, 2025)

DM580: Functional Programming, SDU

Learning Objectives

After doing these exercises, you will be able to:

- Understand and write type annotations involving polymorphic and overloaded types in Haskell.
- Write programs that use Haskell library functions.
- Read and write programs with guards in Haskell.
- Write and apply higher-order functions in Haskell.
- Write and debug simple pattern matching functions in Haskell.
- Prove simple properties of Haskell programs, using case analysis.

1 Fill in the Program (3.11.2 from the Book)

Write down definitions that have the following types; it does not matter what the definitions actually do as long as they are type correct.

```haskell
bools :: [Bool]

nums :: [[Int]]

add :: Int -> Int -> Int -> Int

copy :: a -> (a, a)

apply :: (a -> b) -> a -> b
```

2 What are the Types? (Based on 3.11.3 from the Book)

What are the types of the following functions? Make your types as general as possible; i.e., use polymorphism where you can.

Avoid using GHCi for now. Do it by hand first; then check against GHCi.

```haskell
second xs = head (tail xs)

swap (x, y) = (y, x)

pair x y = (x, y)

double x = x*2
```

```haskell
palindrome xs = reverse xs == xs

twice f x = f (f x)
```

Hint: take care to include the necessary class constrains in the types if the functions are defined using overloaded operators.

3 Fill in the Program 2

Write down definitions that have the following types. The functions should match the behavior summarized in the comment above each.

```haskell
-- `mySum` should compute the sum of three numbers
mySum :: Num a => a -> a -> a -> a

-- `myAvg` should compute the average (arithmetic mean) of adding two numbers
myAvg :: Fractional a => a -> a -> a

-- `myRead` should return `Just a` if the string can be fully parsed as an `a`
-- value.  Hint 1: use the `reads` function
-- https://hackage.haskell.org/package/base-4.21.0.0/docs/Prelude.html#v:reads
myRead :: Read a => String -> Maybe a

-- Hint 2: you can test your `myRead` function by either using explicit type
-- annotations, such as `λ> myRead "42" :: Int` in GHCi; or by using the
-- following test cases.
testMyRead :: Bool -- should be True
testMyRead =    myRead "True"    == Just True
             && myRead "False"   == Just False
             && myRead "3.14159" == Just 3.14159
```

Hint: use GHCi's `:info` to see the functions of a type class. E.g.:

```
λ> :info Fractional
type Fractional :: * -> Constraint
class Num a => Fractional a where
  (/) :: a -> a -> a
  recip :: a -> a
  fromRational :: Rational -> a
  {-# MINIMAL fromRational, (recip | (/)) #-}
        -- Defined in ‘GHC.Real’
instance Fractional Double -- Defined in ‘GHC.Float’
instance Fractional Float -- Defined in ‘GHC.Float’
```

4 Halve (4.8.1 from the Book)

Using library functions, define a function `halve :: [a] → ([a], [a])` that splits an even-lengthed list into two halves. For example:

```
λ> halve [1,2,3,4,5,6]
([1,2,3], [4,5,6])
```

5 List Deconstruction (4.8.2 from the Book)

Define a function `third :: [a] → a` that returns the third element in a list that contains at least this many elements using:

1. `head` and `tail`;
2. list indexing `!!`;
3. pattern matching.

6 Safe Tail (Based on 4.8.3 from the Book)

Define a function `safetail :: [a] → Maybe [a]` that behaves the same way as `tail`, except that it maps the empty list to `Nothing`, rather than producing an error, and returns `Just` the tail otherwise.

Implement it using:

1. a conditional expression and the function `null :: [a] → Bool`;
2. guarded equations;
3. pattern matching.

7 Equivalence of Logical Conjunction Functions (Based on 4.8.5 and 4.8.6 from the Book)

Prove that the following two functions are equivalent:

```haskell
and1 :: Bool -> Bool -> Bool
True `and1` True = True
_    `and1` _    = False

and2 :: Bool -> Bool -> Bool
True  `and2` b = b
False `and2` _ = False
```

That is, prove that for all Booleans `b` and `b'`, that `(b `and1` b') = (b `and2` b')`.

Hint: use case analysis.

8 Debugging Pattern Matching Functions 1

Debug and fix the following function.

```haskell
-- Should keep only the tail after the first occurrence of `x`
keepAfter :: Eq a => a -> [a] -> [a]
keepAfter _ []     = []
keepAfter x xs     = xs
keepAfter x (y:ys)
  | x == y         = ys
  | otherwise      = y : keepAfter x ys
```

9 Debugging Pattern Matching Functions 2

Debug and fix the following function.

```haskell
-- Should return `True` if all elements of the first list occur at least once in
-- the second, and `False` otherwise.
subList :: Eq a => [a] -> [a] -> Bool
subList []     ys = True
subList (x:xs) ys = go x ys
  where
    go x ys' = case ys of
      [] -> False
      (y:ys') | x == y -> subList xs ys
              | otherwise -> go x ys'
```

Hint: you can use the Haskell package `Debug.Trace` to print output during evaluation.

For example, the following Haskell program (where the `import` is at the top of the file) prints `loop: n` for each nth iteration of the loop.

```haskell
import Debug.Trace

loop n = trace ("loop: " ++ show n) (loop (n + 1))
```

10 Luhn Algorithm (4.8.8 from the Book)

The *Luhn algorithm* is used to check bank card numbers for simple errors such as mistyping a digit, and proceeds as follows:

- consider each digit as a separate number;
- moving left, double every other number from the second last;
- subtract 9 from each number that is now greater than 9;
- add all the resulting numbers together;
- if the total is divisible by 10, the card number is valid.

Define a function `luhnDouble :: Int → Int` that doubles a digit and subtracts 9 if the result is greater than 9. For example:

```
λ> luhnDouble 3
6
```

```
λ> luhnDouble 6
3
```

Using `luhnDouble` and the integer remainder function `mod`, define a function `luhn :: Int → Int → Int → Int → Bool` that decides if a four-digit bank card number is valid. For example:

```
λ> luhn 1 7 8 4
True

λ> luhn 4 7 8 3
False
```