Exercises, Week 44 (27–31 Oct, 2025)

DM580: Functional Programming, SDU

Learning Objectives

After doing these exercises, you will be able to:

- Write Haskell programs using higher-order functions (see Section 1).
- Write Haskell programs that declare and use custom data types (see Section 2).
- Write Haskell programs involving custom type class instances (see Section 3).

If you do not have time to finish all assignments, make sure to finish some from each section.

1 Higher-Order Functions

1.1 Map and Filter (7.9.1 from the Book)

Show how the list comprehension `[f x | x <- xs, p x]` can be re-expressed using the higher-order functions `map` and `filter`.

1.2 Higher-Order Functions (7.9.2 from the Book)

Without looking at the definitions from Haskell's standard prelude, define the following higher-order library functions on lists.

1. Decide if all elements of a list satisfy a predicate.

   ```haskell
   allB :: (a -> Bool) -> [Bool] -> Bool
   ```

   Note: in the prelude, this function is called `all` and is generic in the type of elements in the input list.

2. Decide if any element of a list satisfies a predicate.

   ```haskell
   anyB :: (a -> Bool) -> [Bool] -> Bool
   ```

   Note: in the prelude, this function is called `all` and is generic in the type of elements in the input list.

3. Select elements from a list while they satisfy a predicate:

   ```haskell
   takeWhile' :: (a -> Bool) -> [a] -> [a]
   ```

   For example,

   ```
   λ> takeWhile' (<3) [1,2,3,4]
   [1,2]

   λ> takeWhile' (==1) [1,1,1,2,1,2]
   [1,1,1]
   ```

4. Remove elements from a list while they satisfy a predicate:

   ```
   dropWhile' :: (a -> Bool) -> [a] -> [a]
   ```

   For example,

   ```
   λ> dropWhile' (<3) [1,2,3,4]
   [3,4]

   λ> dropWhile' (==1) [1,1,1,2,1,2]
   [2,1,2]
   ```

1.3 Using `foldl` (7.9.4 from the Book)

Using `foldl`, define a function `dec2int :: [Int] → Int` that converts a decimal number into an integer. For example,

```
λ> dec2int [2,3,4,5]
2345
```

1.4 Unfolding (7.9.6 from the Book)

A higher-order function `unfold` that encapsulates a simple pattern of recursion for preducing a list can be defined as follows:

```
unfold p h t x | p x       = []
               | otherwise = h x : unfold p h t (t x)
```

That is, the function `unfold p h t` produces the empty list if the predicate `p` is true of the argument value, and otherwise produces a non-empty list by applying the function `h` to this value to give the head, and the function `t` to generate another argument that is recursively processed in the same way to produce the tail of the list. For example, the function `int2bin` can be rewritten more compactly using `unfold` as follows:

```
int2bin = unfold (== 0) (`mod` 2) (`div` 2)
```

Redefine the functions `chop8`, `map f`, and `iterate f` (see Chapter 7.6) using `unfold`.

2 Data Types

2.1 Operations on Natural Numbers (8.9.1 from the Book)

Chapter 8.4 declares the following data type `Nat` of natural numbers.

```
data Nat = Zero | Succ Nat
```

Using this type, define a recursive multiplication function `mult :: Nat → Nat → Nat`.

Hint: make use of the `add` function given in Chapter 8.4.

2.2 Tree Occurrence (8.9.2 from the Book)

Chapter 8.4 declares the following data type of trees:

```
data Tree a = Leaf a | Node (Tree a) a (Tree a)
```

It also defines the following `occurs` function:

```
occurs :: Ord a => a -> Tree a -> Bool
occurs x (Leaf y)                 = x == y
occurs x (Node l y r) | x == y    = True
                      | x < y     = occurs l
                      | otherwise = occurs r
```

Redefine this function using the following data type from the standard prelude:

```
data Ordering = LT | EQ | GT
```

The prelude also defines a function `compare :: Ord a => a -> a -> Ordering` that decides if one value in an ordered type is less that (`LT`), equal to (`EQ`), or greater than (`GT`) another value.

Redefine `occurs` to use `compare` instead.

Explain why this new definition is more efficient than the original version.

2.3 Tree Balance Checking (8.9.3 from the Book)

Consider the following type of binary trees (no values in nodes):

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

Let us say that such a tree is *balanced* if the number of leaves in the left and right subtree of every node differs by at most one, with leaves themselves being trivially balanced. Define a function `balanced :: Tree a -> Bool` that decides if a binary tree is balanced or not.

Hint: first define a function that returns the number of leaves in a tree.

2.4 Expression Folding (8.9.5 from the Book)

Given the data type declaration

```
data Expr = Val Int | Add Expr Expr
```

define a higher-order function

```
folde :: (Int -> a) -> (a -> a -> a) -> Expr -> a
```

such taht `folde f g` replaces each `Val` constructor in an expression by the function `f`, and each `Add` constructor by the function `g`.

2.5 Expression Evaluation (9.8.6 from the Book)

Using `folde`, define a function `eval :: Expr -> Int` that evaluates an expression to an integer value, and a function `size :: Expr -> Int` that calculates the number of values in an expression.

2.6 Extend Abstract Machine (9.8.9 from the Book)

Consider the abstract machine given in Chapter 8.7.

Extend it to support multiplication.

3 Type Classes

3.1 Instances for Equality

Complete the following instance declarations:

```
instance Eq a ⇒ Eq (Maybe a) where
  {- ... -}

instance Eq a ⇒ Eq [a] where
  {- ... -}
```

3.2 Instance for Showing

Using the `Expr` data type declared in exercise 2.4 in this exercise set, complete the following instance declaration:

```
instance Show Expr where
  {- ... -}
```

3.3 Pretty Printing

Refine your `Show Expr` instance to pretty-print expressions as arithmetic expressions.

Your pretty-printer should insert parentheses to indicate that addition expressions are nested to the left; e.g.:

```
λ> Add (Val 1) (Add (Val 2) (Val 3))
1 + 2 + 3

λ> Add (Add (Val 0) (Val 1)) (Add (Val 2) (Val 3))
(0 + 1) + 2 + 3
```

3.4 Prettier Printing

Refine your `Expr` data type to include `Mul` expressions, and ensure that pretty-printing respects precendences.

For example,

```
λ> Add (Val 1) (Mul (Val 2) (Val 3))
1 + (2 * 3)

λ> Add (Mul (Val 1) (Val 2)) (Val 3)
(1 * 2) + 3
```

```
λ> Add (Val 0) (Add (Val 1) (Mul (Val 2) (Val 3)))
0 + 1 + (2 * 3)

λ> Add (Add (Val 0) (Val 1)) (Mul (Val 2) (Val 3))
(0 + 1) + (2 * 3)
```