COMP105: Programming Paradigms
Lab Sheet 8

This lab covers material on custom types.

1. **Check the hidden tests from last week.** Did any of them trip up your submission?

2. **Custom types.** Copy the following type declaration into your file.

   ```
   data Direction = North | East | South | West deriving (Show)
   ```

   After you load the code into ghci, you can use the new type. For example:

   ```
   ghci> North
   ghci> :t South
   ```

   (a) Modify the type definition so that the type also derives `Eq`. After reloading, test that your new definition works by running `North == North` and `North /= South`.

   (b) Modify the type definition so that the type also derives `Read`. Test that your new definition works by running `read "North" :: Direction`. Make sure that you don't remove the `Read` derivation, because otherwise Codegrade's tests for questions (d) and (e) won't work.

   (c) Modify the type definition so that the type also derives `Ord`. test that your new definition works by running `North < East` and `max South West`. Do you understand why these queries return the results that they do?

   (d) Write a function `is_north :: Direction -> Bool` that returns `True` if the argument is `North` and `False` otherwise.

   (e) Write a function `dir_to_int :: Direction -> Int` that returns `1` if the argument is `North`, `2` if the argument is `East`, `3` if the argument is `South`, and `4` if the argument is `West`.

3. **Types with data.**

   Copy the following type declaration into your file.

   ```
   data Point = Point Int Int deriving (Show, Read)
   ```

   You can now build instances of point like so:

   ```
   ghci> Point 2 4
   Point 2 4
   ```

(a) Write a function `same :: Int -> Point` that takes an integer `x` and returns `Point x x`.

(b) Write a function `is_zero :: Point -> Bool` that returns `True` if the input is `Point 0 0` and `False` otherwise. When you call your function, make sure that you use brackets around the argument, like so: `is_zero (Point 0 0)`.

(c) Write a function `mult_point :: Point -> Int` that takes `Point x y` and returns `x * y`.

(d) Write a function `up_two :: Point -> Point` that takes `Point x y` and returns `Point x (y + 2)`.

(e) Write a function `add_points :: Point -> Point -> Point` that adds two points together, so if the inputs are `Point a b` and `Point c d` then the output should be `Point (a+c) (b+d)`.

4. **Maybe.**

(a) Recall the `Maybe a` type from the lectures. Try it out by typing the following into ghci.

```
ghci> Just "hello"
ghci> Just False
ghci> Just 3
ghci> Nothing
```

Use the `:t` command to inspect the types of each of these values.

(b) Write a function `not_nothing :: Eq a => Maybe a -> Bool` that returns `False` if the input `Nothing` and `True` otherwise. Note that the `Eq a` constraint is necessary if you intend to do something like `input == Nothing`, because `Maybe a` is only in `Eq` if `a` is also in `Eq`. Equality tests can be avoided by using pattern matching.

(c) Write a function `safe_tail :: [a] -> Maybe [a]` that takes a list `list`, and returns `Nothing` if the list is empty, or `Just (tail list)` otherwise.

(d) Write a function `mult_maybe :: Maybe Int -> Maybe Int -> Maybe Int` that returns `Just (x*y)` if the inputs are `Just x` and `Just y`, and returns `Nothing` if one or more of the inputs is `Nothing`.

5. **Either.**

(a) Recall the `Either a b` type from the lectures. Try it out by typing the following into ghci.

```
ghci> Left 'a'
ghci> Left False
ghci> Right "hello"
```

Again, you can use the `:t` command to inspect the types of each of these values.

(b) Write a function `return_two :: Int -> Either Bool Char` that takes one argument `n` and returns `Left True` if `n == 1` and returns `Right 'a'` otherwise.

(c) Write a function `show_right :: Either String Int -> String` that returns `x` if the input is `Left x` and `show y` if the input is `Right y`.

6. **Custom lists.**

   (a) Copy the custom list type that we used in the lectures into your file:

   ```
   data List a = Cons a (List a) | Empty deriving (Show, Read)
   ```

   We can now write down instances of our custom list like so

   ```
   ghci> Empty
   ghci> True ‘Cons‘ Empty
   ghci> False ‘Cons‘ (True ‘Cons‘ Empty)
   ```

   Here the ‘ symbol refers to the backtick.

   (b) In ghci, write an instance of `List Int` that is equivalent to `[1,2,3,4]`.

   (c) Write a function `is_empty :: List a -> Bool` that returns `True` if the list is empty, and `False` otherwise.

   (d) Write a function `is_single :: List a -> Bool` that returns `True` if the list has exactly one element, and `False` otherwise.

   (e) Write a function `mult :: List Int -> Int` that takes a list of integers and returns the product $(x \times y \times z \ldots)$ of the list.

   (f) (*) Write a function `our_map :: (a -> b) -> List a -> List b` that implements the map function for our list type.

7. **Custom trees.**

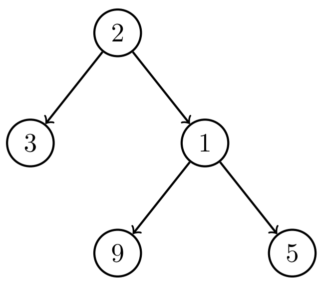
   (a) Copy the custom data-tree type that we used in the lectures into your file:

   ```
   data DTree a = Leaf a | Branch a (DTree a) (DTree a) deriving (Show, Read)
   ```

   We can now write down instances of our custom tree like so

   ```
   ghci> Leaf 1
   ghci> Branch 2 (Leaf 1) (Leaf 3)
   ```

   (b) In ghci, write an instance of `DTree Int` that represents the following tree.



   (c) Write a function `tree_mult :: DTree Int -> Int` that multiplies all of the numbers in the tree together.

(d) Write a function `sum_leafs :: DTree Int -> Int` that sums the values stored in the leafs of the tree. The values stored in the branch nodes should be ignored.

(e) Write a function `count_threes :: DTree Int -> Int` that counts the number of nodes in the tree that store `3`.

(f) Write a function `get_leafs :: DTree Int -> [Int]` that returns a list containing all of the values stored in the leafs. The order should be determined by the structure of the tree: the leftmost leaf should be the first in the list, and the rightmost leaf should be last.

Lab complete.