G6021: Comparative Programming

Exercise Sheet 4

1 Data Types

In Haskell we can define our own types in two ways:

- Renaming old types, for example: `type Name = [Char]`.
- Constructing new types, for example: `data MyBool = T | F`.

  (Adding `deriving (Show)` means that Haskell will print these types.)

When constructing new types, new terms to the language are added also — `T` and `F` are two new terms of the language:

```
> T
T
> :type T
T :: MyBool
```

Pattern matching is automatically available for these new terms, so we can write things like this:

```
myand T T = T
myand T F = F
myand F T = F
myand F F = F
```

The order of the function cases for this example is not important, we can write the four cases in any order. We can also use patterns in abstractions like this:

```
(\T -> F)T
```

Using a variable as a pattern will always match, so we can simplify the above function `myand` to the following, as `x` and `y` will match against all the other cases.

```
myand T T = T
myand x y = F
```

The order of the cases is now very important. Can you think why? Try this:

```
myand x y = F
myand T T = T
```

If the variable is not needed in the body of the function, we can use a special symbol "`_`" as a pattern, which means the same as a variable, but we don't have to think of a name. This means we can write the above as:

```
myand T T = T
myand _ _ = F
```

Again, the order of the cases is very important.

We can also define recursive types, for example we can make our own lists:

```
data List = Nil | Cons Int List deriving (Show)
```

Typically in Haskell, when we have recursive types, we write recursive functions over them as we have seen in the notes. Here is a version of length for our definition of lists:

```
len Nil = 0
len (Cons _ t) = 1 + len t
```

2 Exercises

1. Using the type `MyBool` defined above, write the functions "or" and "not" that will compute the disjunction and negation respectively.

   Answer:

   ```
   myor F F = F
   myor _ _ = T

   mynot F = T
   mynot T = F
   ```

2. Define a new type `RPS` to represent rock, paper and scissors as used in the game. Write a function `beats :: RPS -> RPS -> Bool` that encodes that paper beats rock, rock beats scissors, and scissors beats paper.

   Answer:

   ```
   data RPS = Rock | Paper | Scissors deriving (Show)

   beats :: RPS -> RPS -> Bool
   beats Paper Rock = True
   beats Rock Scissors = True
   beats Scissors Paper = True
   beats _ _ = False
   ```

3. We can define natural numbers as Zero or the Successor of a natural number. Give a datatype `Nat` for natural numbers.

   Answer:

   ```
   data Nat = Z | S Nat deriving (Show)
   ```

4. Write a function `add :: Nat -> Nat -> Nat`, that will add two natural numbers.

   Answer:

   ```
   add Z y = y
   add (S x) y = S(add x y)
   ```

5. Write a function to multiply two natural numbers.

   Answer:

   ```
   mult Z y = Z
   mult (S x) y = add (mult x y) y
   ```

6. Write a function to test if two natural numbers are equal.

Answer:

```
equal Z Z = True
equal (S x) (S y) = equal x y
equal _ _ = False
```