Ad-hoc Polymorphism: Overloaded Types and Type Classes

Ad-hoc polymorphism is a bit trickier, especially in a language that performs type inference, as the system must be able to see an expression like x+y and infer some type information regarding x and y. This is accomplished by a couple of related ideas, namely *overloaded types* (often refered to as *bounded polymorphism*) and *type classes*.

A **overloaded type** is a type that comes with a certain constraint. For instance the type of an add function may look like this:

```
add :: Num t => t -> t -> t
add x y = x + y
```

What this tells us that is that the function add takes two arguments of a certain type and returns a value of that same type, but it can't just be any type. It has the constraint Num t, which says that it must be a "number type".

Even the type of a single number by itself has a similar constraint, because that number can be thought of as one of the many number types:

```
3 :: Num t => t
```

These constraints come from the so-called type-classes: A **type class** is a list of specifications for operations on a type. An **instance** of a type class is a specific type along with definitions for these operations.

A good example of a type-class is the Num type class for numbers. Any instance of this class must provide implementations for the following functions:

```
-- The Num class. An instance Num a must implement:
(+) :: a -> a -> a
(-) :: a -> a -> a
(*) :: a -> a -> a
negate :: a -> a
abs :: a -> a
signum :: a -> a          -- sign
```

If we wanted to, we could for instance make the Char type an instance of the Num class by specifying how each of these operations would work. From that point on we could be writing 'a' + 'b' and the system won't complain.

Standard Type Classes Implementing your own type class is a more advanced feature. But there are many standard type classes that are in constant use, and we will see more as we move on. Here are some of the standard ones:

Num We already encountered this earlier. It contains the following functions:

```
(+) :: a -> a -> a
(-) :: a -> a -> a
(*) :: a -> a -> a
negate :: a -> a
abs :: a -> a
signum :: a -> a
```

Eq The "equality" type class. Values of types that implement Eq can be compared to each other. This contains the following functions:

```
(==) :: a -> a -> Bool
(/=) :: a -> a -> Bool
```

You can see a "type error" if you try to compare two functions, as function types are not instances of the Eq class:

```
(+) == (-)     -- Look at the error
```

Ord This represents ordered types. These are an extension of Eq, and in addition to those functions must also implement these:

```
(<) :: a -> a -> Bool
(<=) :: a -> a -> Bool
(>) :: a -> a -> Bool
(>=) :: a -> a -> Bool
min :: a -> a -> Bool
max :: a -> a -> Bool
```

Show This represents types whose values have a string representation. These are the only values that Haskell will print out for you without complaining. They need to implement a single function:

```
show :: a -> String
```

Read This represents types that know how to turn a string into a value. They need to implement a single method:

```
read :: String -> a
```

Here's an example use of this, to read in a tuple from a string representation:

```
read "(True, 5)" :: (Bool, Int)    -- We must specify the return type.
```

Integral

This is an extension of the Num class. It further requires the implementation of integer division operations:

```
div :: a -> a -> a
mod :: a -> a -> a
```

Fractional This is an extension of the Num class that supports fractional division and reciprocation:

```
(/) :: a -> a -> a
recip :: a -> a
```

Many of these type classes extend to compound types if there is a specification on how to do so. For example tuples are instances of the class Ord as long as their components are, and the same for lists:

```
(3, 4) > (2, 5)
[3, 4, 5] > [2, 5, 6, 7]
```

Practice: Figure out the types of the following functions, including type class specifications:

1. posDiff defined by `posDiff x y = if x > y then x − y else y − x`.
2. maxList defined on lists by:

```
maxList (x:[]) = x
maxList (x:xs) = if x > restMax then x else restMax
            where restMax = maxList xs
```

3. has that checks for the existence of an element in a list, and is defined by:

```
has el []       = False
has el (x:rest) = el == x || has el rest
```

Default implementations:

Functions in a class definition can be provided with *default implementations*. This way someone trying to create an instance of such a class does not need to implement all the methods in the class. As an example, let's take a look at the Eq and and Ord classes, and their actual definitions (you can find them here for example[1]):

```
class  Eq a  where
    (==), (/=)              :: a -> a -> Bool

    x /= y                  = not (x == y)
    x == y                  = not (x /= y)

-- Must specify either (==) and (<=) or compare
class  (Eq a) => Ord a  where
    compare                 :: a -> a -> Ordering
    (<), (<=), (>), (>=)    :: a -> a -> Bool
    max, min                :: a -> a -> a

    -- compare defined in terms of == and <=
    compare x y
        | x == y    = EQ
        | x <= y    = LT
        | otherwise = GT

    -- all operators defined in terms of compare
    x <  y = case compare x y of { LT -> True;  _ -> False }
    x <= y = case compare x y of { GT -> False; _ -> True }
    x >  y = case compare x y of { GT -> True;  _ -> False }
    x >= y = case compare x y of { LT -> False; _ -> True }

    max x y = if x <= y then y else x
    min x y = if x <= y then x else y
```

[1] https://hackage.haskell.org/package/base-4.4.1.0/docs/src/GHC-Classes.html

Other important classes

Here is a list of other important built-in classes:

Enum Used for "enumerated types", i.e. types that we can list "one, two, three, etc". For a type that satisfies Enum, we can write [n,m,...] and similar expressions.

Bounded Used for "bounded types", whose values have a minimum bound and a maximum bound.

Show Used for types that can be "shown", i.e. turned to a string.

Read Used for types that can be "read", i.e. for which we can obtain a value from a string.

You can see the definition of Enum and Bounded here[2]:

```
class Bounded a where
    minBound, maxBound :: a

class Enum a where
    toEnum             :: Int -> a
    fromEnum           :: a -> Int

    succ               :: a -> a              -- next
    pred               :: a -> a              -- previous

    enumFrom           :: a -> [a]            -- [n ..]
    enumFromThen       :: a -> a -> [a]       -- [n,m ..]
    enumFromTo         :: a -> a -> [a]       -- [n .. m]
    enumFromThenTo     :: a -> a -> a -> [a]  -- [n,m, .. p]

    succ                   = toEnum . (+ 1)   . fromEnum
    pred                   = toEnum . (subtract 1) . fromEnum
    enumFrom x             = map toEnum [fromEnum x ..]
    enumFromThen x y       = map toEnum [fromEnum x, fromEnum y ..]
    enumFromTo x y         = map toEnum [fromEnum x .. fromEnum y]
    enumFromThenTo x1 x2 y = map toEnum [fromEnum x1, fromEnum x2 .. fromEnum y]
```

The Random module we have used before also defines two classes, called RandomGen and Random, as follows[3]:

```
class RandomGen g where
   next      :: g -> (Int, g)
   genRange  :: g -> (Int,Int)

   genRange _ = (minBound, maxBound)

class Random a where
  randomR :: RandomGen g => (a,a) -> g -> (a,g)
  random  :: RandomGen g => g -> (a, g)
  randomRs :: RandomGen g => (a,a) -> g -> [a]
```

[2] https://hackage.haskell.org/package/base-4.12.0.0/docs/src/GHC.Enum.html
[3] https://hackage.haskell.org/package/random-1.1/docs/src/System.Random.html#RandomGen

```
randoms   :: RandomGen g => g -> [a]

randomRIO :: (a,a) -> IO a
randomRIO range  = getStdRandom (randomR range)

randomIO  :: IO a
randomIO         = getStdRandom random
```

Practice:

1. Define randomRs based on randomR.
2. The Random module provides a function with the following signature:

   ```
   buildRandoms :: RandomGen g => (a -> as -> as) -> (g -> (a,g)) -> g -> as
   ```

   Understand this signature, and use this function to write randomRs instead of your previous version. Then implement this function.