Advanced Haskell Typing concepts

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:

```haskell
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:

```haskell
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:

```haskell
-- 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:

```haskell
(+) :: a -> a -> a
(-) :: a -> a -> a
(*) :: a -> a -> a
negate :: a -> a
```

```haskell
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:

```haskell
(==) :: 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:

```haskell
(+) == (-)      -- 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:

```haskell
(<) :: 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 value 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:

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

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

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

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

```haskell
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:

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

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

```haskell
(/) :: 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 elem []       = False
   has elem (x:rest) = elem == x || has elem rest
   ```