Polymorphism

Lists

Continued from previous

Recall the data constructors for a `[a]` (list):

```
[]  :: [a]
(:) :: a -> [a] -> [a]
```

`head` and `tail`

Let's look at a few common list functions, and see how we can use pattern matching to define them.

```
head :: [a] -> a
tail :: [a] -> [a]
```

Question

What do you think these functions do?

Answer:

- `head` returns first element of list
- `tail` returns all of list after first element

```
head (x:xs) = x
tail (x:xs) = xs
```

Think about the pattern matching we did with other constructors, like `Succ Peano`, `Point Float Float`, and tuples (e.g. `(x, y)`). We're doing the same thing here, except the constructor is now the infix operator `:`, so it looks kind of funny.

We can, of course, ignore some of these values on the LHS:

```
head :: [a] -> a
head (x:_) = x

tail :: [a] -> [a]
tail (_:xs) = xs
tail []     = []
```

Let's consider a few calls to these (these are *reductions*, not valid Haskell):

```
head [3, 4] => head (3:[4]) => 3
tail [3, 4] => tail (3:[4]) => 4
```

```
head [3] => head (3:[]) => 3
tail [3] => tail (3:[]) => []
```

More Pattern Matching Examples

TODO

Polymorphic Data Types

List

```
data List a = Nil | Cons a (List a)
```

Question

What are the types of the `List` constructors we just wrote?

Answer:

```
Nil  :: List a
Cons :: a -> List a -> List a
```

Compare these to the constructors for `[a]`:

```
[]  :: [a]
(:) :: a -> [a] -> [a]
```

So the constructors we wrote have the properties:

- `Nil` is the 'zero value' for the `List a` type
- `Cons a (List a)` means that the `Cons` constructor has two arguments:
  1. A value of any type (represented by the first `a`)
  2. A `List a`, where `a` is of the same type as the first `a`

The second constructor looks confusing, but just remember that the definition of a data constructor consists of an identifier (`Cons`, in this case) followed by a sequence of the *types of the values that can be used as arguments to the constructor*.

Constructing a `List a`

```haskell
empty :: List a
empty = Nil

emptyInt :: List Int
emptyInt = Nil

ints :: List Int
ints = Cons 1 (Cons 2 Nil) -- analogous to [1, 2]
```

Question

How would you write the `head` and `tail` functions that we write for `[a]` for our `List a` type? (Call them `headList` and `tailList`.)

Answer:

```haskell
headList :: List a -> a
headList (Cons x _) = x

tailList :: List a -> List a
tailList (Cons _ xs) = xs
tailList Nil         = Nil
```

Notice that our `headList` function does *not* handle the `Nil` case (just as the `head` function does not handle the `[]` case). We'll explore this more in the homework.