Anonymous Functions and Sections

In this section we discuss two related and useful features, anonymous functions and sections. Anonymous functions allow us to quickly build ephemeral functions that may be used only once or twice. Sections allow us to create such functions from operators.

Reading

- Sections 4.5, 4.6
- Practice exercises (4.8): 7

Anonymous Functions

A common pattern that we will see in the future is the idea of "higher-order functions". These functions take as input some function, along with other values, and often apply this function.

Often times the functions passed as parameters to these higher-order functions are ephemeral, only used in this one instance and not really needing their own name. This is one of the motivations behind the idea of *anonymous functions*.

Anonymous functions, or **lambda expressions**, specify a, typically short, definition for a function and produce a function value. Unlike the named functions, this value will be lost if it is not stored somewhere.

> Anonymous functions can appear anywhere where a value/expression is expected.

We can think of most of the function we have seen as defined via a lambda expression. For example:

```
increment x = x + 1
-- It is the same as :
increment = \x -> x + 1
```

We get to see here the syntax for anonymous functions: They start with a backslash, followed by the function parameters, followed by an arrow and finally the function body. We can even have multiple parameters, and in so doing can do the same thing in multiple ways:

```
add x y = x + y
add x = \y -> x + y
add = \x y -> x + y
add = \x -> (\y -> x + y)
```

All of these are in effect the same function, written in different ways. They do mentally serve different roles however. For example the second form makes it clear that we can call add with one argument, and what we get back is a new function of the y argument.

In the above example we have saved all these functions with the name add. But we could instead use them directly, without ever giving a name to the function:

```
(\x y -> x + y) 5 2          -- Results in 7
```

Or similarly we can pass such a function directly into a higher-order function:

```
map (\x -> x * x) [1..10]    -- Results in the squares from 1 to 10
```

Practice: Write anonymous functions that perform the following tasks:

1. Given a number, check if the number is even by dividing modulo 2.
2. Given two numbers, compare them and return the larger, using an if−then−else construct.
3. Given a string, truncate it to its first 4 characters.

Operator Sections

Operator sections are a convenient syntax for creating anonymous functions out of operators.

First of all, recall that operators differ from "normal" functions in that they are written using symbols rather than letters, and are typically used inline. However, we can always think of an operator as a "normal" function by putting it in parentheses. For example, the following two have the same result:

```
3 + 5
(+) 3 5
```

So the operator + is technically really a function (+) :: Num t => t -> t -> t, and as such it can be called by following it up with the parameters as in the second example above.

Operator sections take this one step further: We can specify one of the two arguments to the operator by placing it on the correct side inside the parentheses:

```
(1+)        --- Same as: \x -> 1 + x
(1/)        --- Same as: \y -> 1 / y
(:lst)      --- Same as: \x -> x:lst
(4:)        --- Same as: \lst -> 4:lst
(`mod` 2)   --- Same as: \x -> x `mod` 2
(mod 2)     --- This is different! This is \x -> mod 2 x
```

Notice that the `mod` example demonstrates that we can even produce sections from normal functions of two arguments, if we think of them as operators by surrounding them with backticks.

Practice: Write sections for the following operations:

1. Decreasing a number by 1.
2. Testing if a number is greater than 2.
3. Appending the list `[1, 2]` to the front of another list.
4. Appending the list `[1, 2]` to the end of another list.
5. Raising 2 to a given power.