Chapter 2. Equational Reasoning

---

equations in Haskell are true mathematical equations—they are not assignment statements. p.38

Theorem 1 (length (++))**.** Let `xs, ys :: [a]` be arbitrary lists. Then `length (xs ++ ys) = length xs + length ys`. pp.38-39

Theorem 2 (length map)**.** Let `xs :: [a]` be arbitrary list and `f :: a -> b` an arbitraty function. Then `length (map f xs) = length xs`.

Theorem 3 (map (++))**.** Let `xs, ys :: [a]` be arbitrary lists and `f :: a -> b` an arbitrary function. Then `map f (xs ++ ys) = map f xs ++ map f ys`.

Theorem 4. For arbitrary lists `xs, ys :: [a]`, and arbitrary `f :: a -> b`, the following equation holds: `length (map f (xs ++ ys)) = length xs + length ys`.

Proof. We prove the theorem by equational reasoning, starting with the left hand side of the equation and transforming it into the right hand side.

```
length (map f (xs ++ ys))
  = length (map f xs ++ map f ys)              { map (++) }
  = length (map f xs) + length (map f ys)    { length (++) }
  = length xs + length ys                      { length map }
```

□

Assignments have to be understood in the context of time passing as a program executes. To underrstand `n := n + 1` assignment, you need to talk about the old value of the variable, and its new value. In contrast, equations are timeless, as there is no notion of of changing the value of a variable. p.44

Chapter 3. Recursion

---

p.48

```
factorial :: Int -> Int
factorial 0       = 1
factorial (n + 1) = (n + 1) * factorial n
```

Recursive definitions consist of a collection of equations that state properties od the function being degfined. There are a number of algebraic properties of the factorial function, and one of them is used as the second equation of the recursive definition. In fact, the definition doesn't consist of a set of commands to be obeyed; it consists of a set of true equations describing the salient properties of the function being defined.

Exercises

pp.52-53

Exercise 1. Write a recursive function `copy :: [a] -> [a]` that copies its list argument. For example, `copy [2]` ⇒ `[2]`.

Solution a

```
copy :: [a] -> [a]
copy []     = []
copy (x:xs) = x : copy xs
```

Solution b

```
copy :: [a] -> [a]
copy = map id
```

Exercise 2. Write a function `inverse` that takes a list of pairs and swaps the pair elements. For example,

```
inverse [(1,2), (3,4)] ==> [(2,1), (4,3)]
```

Solution a

```
inverse :: [a] -> [a]
inverse []          = []
inverse ((a,b):xs) = (b,a) : inverse xs
```

Solution b

```
inverse :: [a] -> [a]
inverse = map (\(a, b) -> (b, a))
```

Solution c

```
import Data.Tuple
inverse :: [a] -> [a]
inverse = map swap
```

Exercise 3. Write a function

```
merge :: Ord a => [a] -> [a] -> [a]
```

which takes two sorted lists and returns a sorted list containing the elements of each.

Solution

```
merge :: Ord a :: [a] -> [a] -> [a]
merge [] ys = ys
merge (x:xs) ys =
  let
    smaller = [ y | y <- ys, y <= x ]
    bigger  = [ y | y <- ys, y > x ]
  in
    merge xs (smaller ++ [x] ++ bigger)
```

Exercise 4. Write `(!!)`, a function that takes a natural number `n` and a list and selects the nth element of the list. List elements are indexed from 0, not 1, and since the type of the incoming number does not prevent it from being out of range, the result should be a `Maybe` type. For example,

```
[1,2,3]!!0 ==> Just 1
[1,2,3]!!2 ==> Just 3
[1,2,3]!!5 ==> Nothing
```

Solution

```
(!!) :: [a] -> Int -> Maybe a
(!!) [] _      = Nothing
(!!) (x:xs) 0 = Just x
(!!) (x:xs) n = (!!) xs (n-1)
```

Exercise 5. Write a function `lookup` that takes a value and a list of pairs, and returns the second element of the pair that has the value as its first element. Use a Maybe type to indicate whether the lookup succeeded. For example,

```
lookup 5 [(1,2),(5,3)] ==> Just 3
lookup 6 [(1,2),(5,3)] ==> Nothing
```

Solution

```
lookup :: Int -> [(Int, a)] -> Maybe a
lookup _ [] = Nothing
lookup n ((k,v):xs)
  | n == k    = Just v
  | otherwise = lookup n xs
```

Exercise 6. Write a function that counts the number of times an element appears in a list.

Solution

```
count :: Eq a => [a] -> a -> Int
count []     n  = 0
count (x:xs) n
  | x == n    = 1 + count xs n
  | otherwise = count xs n
```

Exercise 7. Write a function that takes a value `e` and a list of values `xs` and removes all occurrences of `e` from `xs`.

Solution

```
remove :: Eq a => [a] -> a -> [a]
remove []     _ = []
remove (x:xs) e
  | e == x      = remove xs e
  | otherwise   = x : remove xs e
```

Exercise 8. Write a function

```
f :: [a] -> [a]
```

that removes alternating elements of its list argument, starting with the first one. For examples, `f [1,2,3,4,5,6,7]` returns `[2,4,6]`.

Solution

```
alternating :: [a] -> [a]
alternating (x:[])    = []
alternating (x:xx:[]) = xx : []
alternating (x:xx:xs) = xx : alternating xs
```