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.

Peano Arithmetic

p.57

```
data Peano = Zero | Succ Peano deriving Show

decrement :: Peano -> Peano
decrement Zero     = Zero
decrement (Succ a) = a

add :: Peano -> Peano -> Peano
add Zero     b = a
add (Succ a) b = Succ (add a b)

sub :: Peano -> Peano -> Peano
sub a        Zero     = a
sub Zero     _        = Zero
sub (Succ a) (Succ b) = sub a b

equals :: Peano -> Peano -> Bool
equals Zero     Zero = True
equals Zero     b    = False
equals a        Zero = False
equals (Succ a) (Succ b) = equals a b

lt :: Peano -> Peano -> Bool
lt a        Zero = False
lt Zero     b    = True
lt (Succ a) (Succ b) = lt a b
```