Chapter 16

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

add :: Nat -> Nat -> Nat
add Zero m = m
add (Succ n) m = Succ $ add n m
```

```
reverse' :: [a] -> [a] -> [a]
reverse' xs ys = foldl (flip (:)) ys xs
```

```
data Tree = Leaf Int | Node Tree Tree

flatten :: Tree -> [Int]
flatten (Leaf n) = [n]
flatten (Node l r) = flatten l ++ flatten r
```

We define a relation

$$\texttt{flatten' t ns = flatten t ++ ns}$$

and use this to derive `flatten'` as follows. Base case:

$$\begin{aligned}\texttt{flatten' (Leaf n) ns} &= \texttt{flatten (Leaf n) ++ ns} \\ &= \texttt{[n] ++ ns} \\ &= \texttt{n :: ns}\end{aligned}$$

Induction:

$$\begin{aligned}\texttt{flatten' (Node l r) ns} &= \texttt{flatten l ++ flatten r ++ ns} \\ &= \texttt{flatten l ++ flatten' r ns} \\ &= \texttt{flatten' l \$ flatten' r ns}\end{aligned}$$

So we define:

```
flatten' :: Tree -> [Int] -> [Int]
flatten' (Leaf n) ns = n : ns
flatten' (Node l r) ns = flatten' l $ flatten' r ns

flatten2 :: Tree -> [Int]
flatten2 t = flatten' t []
```

Compiler correctness

```
data Expr = Val Int | Add Expr Expr deriving Show

eval :: Expr -> Int
eval (Val n) = n
eval (Add x y) = eval x + eval y
```

```
type Stack = [Int]
type Code = [Op]
data Op = PUSH Int | ADD deriving Show

exec :: Code -> Stack -> Stack
exec [] s = s
exec (PUSH x : ops) s = exec ops (x : s)
exec (ADD : ops) (x : y : s) = exec ops (y + x : s)
```

```
comp :: Expr -> Code
comp (Val x) = [PUSH x]
comp (Add el er) = comp el ++ comp er ++ [ADD]

e :: Expr
e = Add (Add (Val 2) (Val 3)) (Val 4)
```

Compiler law version 1:

```
exec (comp e) [] = [eval e]
```

Compiler law version 2:

```
exec (comp e) s = eval e : s
```