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
```

This is proved in the book but the proof needs the distributivity property proved below.

The distributivity property is that `exec` distributes over lists, that executing two pieces of code appended together is the same as executing the two pieces of code in sequence:

```
exec (c ++ d) s = exec d $ exec c s
```

Inductive Case:

`exec ((PUSH n : c) ++ d) s`	=	`exec (PUSH n : c ++ d) s`	Apply `++`
	=	`exec (c ++ d) (n : s)`	Apply `exec`
	=	`exec d $ exec c (n : s)`	Apply induction
	=	`exec d $ exec (PUSH n : c) s`	Unapply `exec`

Other inductive case:

`exec ((ADD : c) ++ d) s`	=	`exec (Add : c ++ d) (x : y : s')`	Apply `++`
	=	`exec (c ++ d) (y + x : s')`	Apply `exec`
	=	`exec d $ exec c (y + x : s')`	Apply induction
	=	`exec d $ exec (ADD : c) s`	Unapply `exec`

Making append vanish with `comp`: a new `comp` that satisfies:

```
comp' e c = comp e ++ c
```

This will allow us to get rid of

```
comp (Add el er) = comp el ++ comp er ++ [ADD]
```

to replace it with

```
comp' (Add el er) c = comp' el $ comp' er $ ADD : c
```

In fact that's the definition:

```
comp' :: Expr -> Code -> Code
comp' (Val x) c = PUSH x : c
comp' (Add el er) c = comp' el $ comp' er $ ADD : c
```

With this new `comp'` the validity with respect to `eval` is:

$$\texttt{exec (comp' e c) s} = \texttt{exec c \$ eval e : s}$$

The advantage of this new definition is we do not need the distributivity lemma, but can prove this directly:

Base case:

$$\begin{aligned}
\texttt{exec (comp' (Val x) c) s} &= \texttt{exec (PUSH x : c) s} \\
&= \texttt{exec c \$ x : s} \\
&= \texttt{exec c \$ eval (Val x) : s}
\end{aligned}$$

Inductive case:

$$\begin{aligned}
&\texttt{exec (comp' (Add el er) c) s} \\
&\qquad = \texttt{exec (comp' el \$ comp' er \$ ADD : c) s} \\
&\qquad = \texttt{exec (comp' er \$ ADD : c) \$ eval el : s} \\
&\qquad = \texttt{exec (ADD : c) \$ eval er : eval el : s} \\
&\qquad = \texttt{exec c \$ eval el + eval er : s} \\
&\qquad = \texttt{exec c \$ eval (Add el er) : s}
\end{aligned}$$

Exercises

EXERCISE 1: Show that `add n $ Succ m` = `Succ $ add n m` by induction on `n`.