Exercise 1

Given the definition of the operator (++)

1. [ ] ++ ys = ys
2. (x:xs) ++ ys = x : (xs ++ ys)

Property 1

I will show by induction on xs that the following holds: xs ++ [ ] = xs.

- **Base Case**: xs = [] :

  [] ++ []
  {`apply definition of (++) case 1` }
  [ ]

- **Inductive step**: xs = (w:ws) :

  (w:ws) ++ []
  { `apply definition of (++) case 2` }
  w : (ws ++ [])
  { `apply I.H. on the subexpression` ws ++ [] }
  w : ws ≡ xs

Property 2

I will show by induction on xs that xs ++ (ys ++ zs) = (xs ++ ys) ++ zs.

- **Base Case**: xs = [] .

  [] ++ (ys ++ zs)
  { `apply definition of (++) case 1` }
  ys ++ zs
  {`unapply definition of (++) case 1 to ys` }
  ([] ++ ys) ++ zs

- **Inductive step**: xs = w:ws .

  (w:ws) ++ (ys ++ zs)
  { `apply definition of (++) case 2` }
  w: (ws ++ (ys ++ zs)
  { `apply I.H. on` ws ++ (ys ++ zs) }
  w: ((ws ++ ys) ++ zs)
  {`unapply case 2 of (++)`}
  (w: (ws ++ ys)) ++ zs
  {`unapply case 2 of (++) to` (w: (ws ++ ys)) }
  ((w:ws) ++ ys) ++ zs
  {`substituting (w:ws) with xs` }
  (xs ++ ys) ++ zs

Exercise 2

Show that exec(c ++ d) s= exec d (exec c s)*, where* exec *is the function that executes the code consisting of sequences of PUSH n and ADD operations.* I will show by induction on c that the property holds:

- **Base Case**: c = []

```
exec ([] ++ d) s
 { Application of (++) }
exec d s
 { unapplying exec (case 1) }
exec d (exec [] s)
```

- **Inductive step** c = (w:ws), i.e. the list contains at least one element. Because the stack is composed of Push n and Add operations, I have to distinguish two cases:

  1. c = (Push n):ws

```
exec (((Push n):ws) ++ d) s
 { Application of (++) }
exec ((Push n) : (ws++d)) s
 { Application of exec }
exec (ws ++ d) (n : s)
 { Application of I.H. }
exec d (exec ws (n:s))
 { unapply inner exec }
exec d (exec (Push n:ws) s)
```

  2. c = ADD:ws

     exec ((ADD:ws) ++ d) s
      { Application of (++) }
     exec (ADD : (ws++d)) s
      { Application of exec }
     We assume that the stack is composed by at least 2 elements, i.e. s = m:n:s'.
     The previous line is:
     exec (ADD : (ws++d)) (m:n:s')

     exec (ws ++ d) ((m+n) : s')
      { Application of I.H. }
     exec d (exec ws ((m+n):s'))
      { unapply inner exec }
     exec d (exec (ADD:ws) m:n:s')

Exercise 3

Given the type and instance declarations below, verify the functor laws for the Tree type, by induction of trees:

```
data Tree a = Leaf a | Node (Tree a) (Tree a)

instance Functor Tree where
--fmap :: (a -> b) -> Tree a -> Tree b
fmap g (Leaf x) = Leaf (g x)
fmap g (Node l r) = Node (fmap g l) (fmap g r)
```

First Law

I have to show that fmap **id** = **id**. I have to distinguish two cases: the base case Leaf x and the step one Node l r

- **Base case**

  fmap **id** (Leaf x)=
  { `apply fmap` }
  Leaf (**id** x)=
  { `apply id` }
  Leaf x=
  { `unapply id` } **id** (Leaf x)

- **Inductive step**

  fmap **id** (Node l r) =
  { `apply fmap` }
  Node (<u>fmap id l</u>) (<u>fmap id r</u>) =
  { `I.H. on trees l and r` }
  Node (<u>id l</u>) (<u>id r</u>)=
  { `apply id on l and r` }
  Node l r=
  { `unaply id` }
  id (Node l r)

Second Law

I have to show that fmap (g.h) = fmap g. (fmap h). I have to distinguish two cases: the base case Leaf x and the step one Node l r

- **Base case** (Leaf x)

```
fmap (g.h) (Leaf x)=
 { apply fmap }
Leaf ((g.h) x)=
 { apply composition of g and h }
Leaf (g (h x))=
 { unapply fmap g }
fmap g (Leaf (h x)) =
 { unapply fmap h }
fmap g (fmap h (Leaf x)) =
 { unapply (.)     }
(fmap g . fmap h) (Leaf x)
```

- **Inductive step**

```
fmap (g.h) (Node l r) =
 { apply fmap }
Node (fmap (g.h) l) (fmap (g.h) r) =
 { I.H. on trees l and r }
Node (fmap g . fmap h l) (fmap g . fmap h r)=
 { Application of (.)  }
Node (fmap g (fmap h l)) (fmap g (fmap h r)) =
 { unapply fmap g }
fmap g (Node (fmap h l) (fmap h r))=
 { unapply fmap h }
fmap g (fmap h (Node l r))
 { unapply (.)  }
(fmap g . fmap h) (Node l r)
```

Exercise 4a

Verify the functor laws for the Maybe type. I recall the Maybe instantiation of the functor class:

```
data Maybe a = Nothing | Just a

instance Functor Maybe where
--fmap :: (a -> b) -> Tree a -> Tree b
fmap g Nothing = Nothing
fmap g (Just x) = Just (g x)
```

First Law

I have to show that fmap **id** = **id**. I have to distinguish two cases: **Nothing** and **Just** x (There is no induction here!)

- **Case A** (**Nothing**)

  fmap **id Nothing**=
  { apply fmap }
  Nothing=

- **Case B** (**Just** x)

  fmap id (Just x) =
  { apply fmap }
  Just (id x) =
  { apply id }
  Just x=
  { unapply id }
  id (**Just** x)=

Second Law

I have to show that fmap (g.h) = fmap g. (fmap h). I have to distinguish two cases: **Nothing** and **Just** x:

- **Case A** (**Nothing**)

```
fmap (g.h) Nothing =
{apply fmap}
Nothing =
{unapply fmap g}
fmap g Nothing} =
{unapply fmap h}
(fmap g.fmap h) Nothing
```

- **Case B** (**Just** x)

```
fmap (g.h) (Just x) =
{apply fmap}
Just ((g.h) x) =
{apply composition of g and h}
Just (g (h x))}=
{unapply fmap g}
fmap g (Just (h x)) =
{unapply fmap h}
(fmap g (fmap h (Just x))
{unapply (.)}
(fmap g . fmap h) (Just x)
```

Exercise 4

Verify the applicative law for the `Maybe` type.

Proof of the first law. We want to prove that `pure id <*> x = x` holds for `Maybe`.

`Nothing` case:

```
pure id <*> Nothing
=   { apply pure }
(Just id) <*> Nothing
=   { apply <*> }
fmap id Nothing
=   { apply fmap }
Nothing
```

`Just x` case:

```
pure id <*> (Just x)
=   { apply pure }
(Just id) <*> (Just x)
=   { apply <*> }
fmap id (Just x)
=   { apply fmap }
Just (id x)
=   { apply id }
Just x
```

□

Proof of the second law. We want to prove that `pure (g x) = pure g <*> pure x` holds for `Maybe`.

```
pure (g x)
=   { apply pure }
Just (g x)
=   { unapply fmap }
fmap g (Just x)
=   { unapply <*> }
(Just g) <*> (Just x)
=   { unapply pure }
pure g <*> pure x
```

□

Proof of the third law. We want to prove that `x <*> pure y = pure (\g -> g y) <*> x` holds for `Maybe`.

In order to prove the `Nothing` case, we prove that

```
x <*> Nothing = Nothing
```

Lemma. `x <*> Nothing = Nothing`

`Nothing` case:

```
Nothing <*> Nothing = Nothing
```

`Just g` case:

```
(Just g) <*> Nothing
=   { apply <*> }
fmap g Nothing
=   { apply fmap }
Nothing
```

□

We can conclude that `Nothing <*> _ = _ <*> Nothing = Nothing` because of the *Lemma* just proved. Therefore, the `Nothing` case for the third law is demonstrated.

`Just x` case:

```
(Just x) <*> pure y
=   { apply pure }
(Just x) <*> (Just y)
=   { apply <*> }
fmap x (Just y)
=   { apply fmap }
Just (x y)
=   { rewrite (x y) using application on a lambda expression }
Just ((\g -> g y) x)
=   { unapply fmap }
fmap (\g -> g y) (Just x)
=   { unapply <*> }
(Just (\g -> g y)) <*> (Just x)
=   { unapply pure }
pure (\g -> g y) <*> (Just x)
```

□

Proof of the fourth law. We want to prove that `x <*> (y <*> z) = (pure (.) <*> x <*> y) <*> z` holds for `Maybe`.

`Nothing` case:

```
(pure (.) <*> Nothing <*> y) <*> z
=   { apply inner right most <*> }
(pure (.) <*> Nothing) <*> z
=   { lemma on inner <*> }
```

```
Nothing <*> z
=   { apply <*> }
Nothing
=   { unapply <*> }
Nothing <*> (y <*> z)
```

In the next case we assume that `x`, `y` and `z` are `Just`:

```
(Just x) <*> ((Just y) <*> (Just z))
=   { apply inner <*> }
(Just x) <*> (fmap y (Just z))
=   { apply <*> }
fmap x (fmap y (Just z))
=   { unapply . }
(fmap x . fmap y) (Just z)
=   { unapply the second Functor law }
fmap (x . y) (Just z)
=   { unapply <*> }
(Just (x . y)) <*> (Just z)
=   { unapply fmap }
(fmap (x .) y) <*> (Just z)
=   { unapply <*> }
((Just (x .)) <*> y) <*> (Just z)
=   { unapply fmap }
(fmap (.) (Just x) <*> y) <*> (Just z)
=   { unapply <*> }
((Just (.)) <*> (Just x) <*> (Just y)) <*> (Just z)
=   { unapply first pure }
(pure (.) <*> (Just x) <*> (Just y)) <*> (Just z)
```

□

Exercise 5

Given the equation comp' e c = comp e ++ c, show how to construct the recursive definition for comp' by induction on e. Before trying to solve the exercise I will show a very trivial LEMMA, that simplifies the following proofs:

Lemma 1. *[x]++ys = x:ys*

Proof.

```
[x]++ys
 { Desugaring }
(x :[]) ++ ys
 { Apply definition of (++) step case }
x : ([] ++ ys)
 { Apply definition of (++) }
x:ys
```

□

- **Base Case**:

```
comp' (Val n) c
 { apply given equation }
comp (Val n) ++ c
 { apply comp }
[Push n] ++ c
 { Lemma 1 }
(Push n) : c
```

- **Inductive step**:

```
comp' (Add n m) c
 { apply given equation }
comp (Add n m) ++ c
 { apply comp }
comp n ++ comp m ++ [ADD] ++ c
 { apply associativity }
comp n ++ comp m ++ ([ADD] ++ c)
 { Lemma 1 }
comp n ++ comp m ++ (ADD : c)
 { I.H. }
comp n ++ (comp' m (ADD : c))
 { unapply comp' }
comp' n (comp' m (ADD : c))
```

```
comp' :: Expr -> Code -> Code
comp' (Val n) c = (Push n):c
comp' (Add m n) c = comp' m (comp' n (Add:c))
```