

## 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 }  
 $[]$

- **Step Case:**  $xs = (w:ws)$  :

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

### 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  $ys$  }  
 $([] ++ ys) ++ zs$

- **Step Case:**  $xs = w:ws$  .

$(w:ws) ++ (ys ++ zs)$   
    { 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  $\text{exec}(c \ ++ \ d) \ s = \text{exec } d \ (\text{exec } c \ s)$ , 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)

```

- **Step Case**  $c = (w:ws)$ , 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.  $c = (\text{Push } n):ws$

```

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.  $c = \text{ADD}:ws$

```

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:

```

1 data Tree a = Leaf a | Node (Tree a) (Tree a)
2
3 instance Functor Tree where
4   --fmap :: (a -> b) -> Tree a -> Tree b
5   fmap g (Leaf x) = Leaf (g x)
6   fmap g (Node l r) = Node (fmap g l) (fmap g r)

```

## First Law

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

- Base case

```

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

```

- Step Case

```

fmap id (Node l r) =
{ apply fmap }
Node (fmap id l) (fmap id r) =
{ I.H. on trees l and r }
Node (id l) (id r)=
{ apply id on l and r }
Node l r=
{ unapply id }
id (Node l r)

```

## Second Law

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

- Base case ( $\text{Leaf } x$ )

```

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)

```

- Step Case

```
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) =
  { unapply fmap g }
fmap g (Node (fmap h l) (fmap h r)) =
  { unapply fmap h }
fmap g . fmap h (Node l r)
```

## Exercise 4

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

```
1 data Maybe a = Nothing | Just a
2
3 instance Functor Maybe where
4   -- fmap :: (a -> b) -> Tree a -> Tree b
5   fmap g Nothing = Nothing
6   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)
```

## Exercise 5

Given the equation  $\text{comp}' e c = \text{comp } e ++ c$ , show how to construct the recursive definition for  $\text{comp}'$  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
```

- **Step Case:**

```

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

```