```
module HuttonChap16 where

open import Haskell.Prelude
open import Haskell.Law.Equality using (sym; begin_; _≡⟨⟩_; step-≡; _∎; cong)

++-[] : {a : Set} → (xs : List a) → xs ++ [] ≡ xs
++-[] [] = begin ([] ++ []) ≡⟨⟩ [] ∎
++-[] (x ∷ xs) =
    begin
      (x ∷ xs) ++ []
    ≡⟨⟩ -- Apply ++
      x ∷ (xs ++ [])
    ≡⟨ cong (x ∷_) (++-[] xs) ⟩
      x ∷ xs
    ∎

++-assoc : {a : Set} → (xs ys zs : List a)
    → (xs ++ ys) ++ zs ≡ xs ++ (ys ++ zs)
++-assoc [] ys zs =
    begin
      ([] ++ ys) ++ zs
    ≡⟨⟩ -- Apply ++
      ys ++ zs
    ≡⟨⟩ -- Unapply ++
      [] ++ (ys ++ zs)
    ∎
++-assoc (x ∷ xs) ys zs =
    begin
      ((x ∷ xs) ++ ys) ++ zs
    ≡⟨⟩ -- Apply ++
      (x ∷ (xs ++ ys)) ++ zs
    ≡⟨⟩ -- Apply ++
      x ∷ ((xs ++ ys) ++ zs)
    ≡⟨ cong (x ∷_) (++-assoc xs ys zs) ⟩
      x ∷ (xs ++ (ys ++ zs))
    ≡⟨⟩ -- Unapply ++
      (x ∷ xs) ++ (ys ++ zs)
    ∎
```

Hutton's example of elimination of append from flattening a tree:

```
data Tree (a : Set) : Set where
    Leaf : a → Tree a
    Node : Tree a → Tree a → Tree a
{-# COMPILE AGDA2HS Tree #-}

flatten : {a : Set} → Tree a → List a
flatten (Leaf x) = x ∷ []
flatten (Node tl tr) = flatten tl ++ flatten tr
{-# COMPILE AGDA2HS flatten #-}

flatten' : {a : Set } → Tree a → List a → List a
flatten' (Leaf x) xs = x ∷ xs
flatten' (Node tₗ tᵣ) xs = flatten' tₗ (flatten' tᵣ xs)
{-# COMPILE AGDA2HS flatten' #-}
```

```
flatten'-flatten : {a : Set} → (t : Tree a) → (xs : List a)
    → flatten' t xs ≡ flatten t ++ xs
flatten'-flatten (Leaf x) xs = refl
flatten'-flatten (Node tₗ tᵣ) xs =
  begin
    flatten' (Node tₗ tᵣ) xs
  ≡⟨⟩ -- Apply flatten'
    flatten' tₗ (flatten' tᵣ xs)
  ≡⟨ cong (flatten' tₗ) (flatten'-flatten tᵣ xs) ⟩
    flatten' tₗ (flatten tᵣ ++ xs)
  ≡⟨ flatten'-flatten tₗ (flatten tᵣ ++ xs) ⟩
    flatten tₗ ++ (flatten tᵣ ++ xs)
  ≡⟨ sym (++-assoc (flatten tₗ) (flatten tᵣ) xs) ⟩
    (flatten tₗ ++ flatten tᵣ) ++ xs
  ≡⟨⟩ -- Unapply flatten
    flatten (Node tₗ tᵣ) ++ xs
  ∎

flatten'-≡-flatten : {a : Set} → (t : Tree a)
    → flatten' t [] ≡ flatten t
flatten'-≡-flatten (Leaf x) = refl
flatten'-≡-flatten (Node tₗ tᵣ) =
  begin
    flatten' (Node tₗ tᵣ) []
  ≡⟨⟩ -- Apply flatten'
    flatten' tₗ (flatten' tᵣ [])
  ≡⟨ cong (flatten' tₗ) (flatten'-flatten tᵣ []) ⟩ -- Apply the above equality
    flatten' tₗ (flatten tᵣ ++ [])
  ≡⟨ flatten'-flatten tₗ (flatten tᵣ ++ []) ⟩ -- Apply it again
    flatten tₗ ++ (flatten tᵣ ++ [])
  ≡⟨ cong (flatten tₗ ++_) (++-[] (flatten tᵣ)) ⟩ -- Remove trailing []
    flatten tₗ ++ flatten tᵣ
  ≡⟨⟩ -- Unapply flatten
    flatten (Node tₗ tᵣ)
  ∎
```

Compiler correctness

```
data Expr : Set where
    Val : Int → Expr
    Add : Expr → Expr → Expr
{-# COMPILE AGDA2HS Expr #-}

eval : Expr → Int
eval (Val n) = n
eval (Add x y) = eval x + eval y
{-# COMPILE AGDA2HS eval #-}

Stack = List Int
{-# COMPILE AGDA2HS Stack #-}

data Op : Set where
    PUSH : Int → Op
    ADD : Op
{-# COMPILE AGDA2HS Op #-}
```

```
Code = List Op
{-# COMPILE AGDA2HS Code #-}

exec : Code → Stack → Stack
exec [] s = s
exec (PUSH n ∷ c) s = exec c $ n ∷ s
exec (ADD ∷ c) (m ∷ n ∷ s) = exec c $ n + m ∷ s
exec (ADD ∷ c) _ = []
{-# COMPILE AGDA2HS exec #-}

comp : Expr → Code → Code
comp (Val n) c = PUSH n ∷ c
comp (Add x y) c = comp x $ comp y $ ADD ∷ c
{-# COMPILE AGDA2HS comp #-}

comp-exec-eval : (e : Expr) → (c : Code) → (s : Stack)
    → exec (comp e c) s ≡ exec c (eval e ∷ s)
comp-exec-eval (Val n) c s =
  begin
    exec (comp (Val n) c) s
  ≡⟨⟩ -- Apply comp
    exec (PUSH n ∷ c) s
  ≡⟨⟩ -- Apply exec
    exec c (n ∷ s)
  ≡⟨⟩ -- Unapply eval
    exec c (eval (Val n) ∷ s)
  ∎
comp-exec-eval (Add x y) c s =
  begin
    exec (comp (Add x y) c) s
  ≡⟨⟩ -- Apply comp
    exec (comp x $ comp y $ ADD ∷ c) s
  ≡⟨ comp-exec-eval x (comp y $ ADD ∷ c) s ⟩ -- Induction
    exec (comp y $ ADD ∷ c) (eval x ∷ s)
  ≡⟨ comp-exec-eval y (ADD ∷ c) (eval x ∷ s) ⟩ -- Induction Again
    exec (ADD ∷ c) (eval y ∷ eval x ∷ s)
  ≡⟨⟩ -- Apply exec
    exec c ((eval x) + (eval y) ∷ s)
  ≡⟨⟩ -- Unapply eval
    exec c (eval (Add x y) ∷ s)
  ∎

compile : Expr → Code
compile e = comp e []
{-# COMPILE AGDA2HS compile #-}

compile-exec-eval : (e : Expr) → exec (compile e) [] ≡ eval e ∷ []
compile-exec-eval e =
  begin
    exec (compile e) []
  ≡⟨⟩ -- Apply compile
    exec (comp e []) []
  ≡⟨ comp-exec-eval e [] [] ⟩
    exec [] (eval e ∷ [])
  ≡⟨⟩ -- Apply exec
    eval e ∷ []
  ∎
```

Exercise 1: Show that `add n (Suc m) = Suc (add n m)` by induction on `n`

```
+-suc : (n m : Nat) → n + (suc m) ≡ suc (n + m)
+-suc zero m = refl
+-suc (suc n) m =
  begin
    (suc n) + (suc m)
  ≡⟨⟩ -- Apply +
    suc (n + suc m)
  ≡⟨ cong suc (+-suc n m) ⟩
    suc (suc (n + m))
  ≡⟨⟩ -- Unapply +
    suc (suc n + m)
  ∎
```

Exercise 2: Using this property, together with `add n zero = n`, show that addition is commutative, `add n m = add m n`, by induction on `n`.

```
+-zero : (n : Nat) → n + zero ≡ n
+-zero zero = refl
+-zero (suc n) =
  begin
    suc n + zero
  ≡⟨⟩ -- Apply +
    suc (n + zero)
  ≡⟨ cong suc (+-zero n) ⟩
    suc n
  ∎
+-commut : (n m : Nat) → n + m ≡ m + n
+-commut zero m =
  begin
    zero + m
  ≡⟨⟩ -- Apply +
    m
  ≡⟨ sym (+-zero m) ⟩
    m + zero
  ∎
+-commut (suc n) m =
  begin
    suc n + m
  ≡⟨⟩ -- Apply +
    suc (n + m)
  ≡⟨ cong suc (+-commut n m) ⟩
    suc (m + n)
  ≡⟨ sym (+-suc m n) ⟩
    m + suc n
  ∎
```