

I'm going to learn some agda!

```
data Greeting : Set where
  hello : Greeting
```

```
greet : Greeting
greet = hello
```

Defining the natural numbers:

```
data Nat : Set where
  zero : Nat
  suc  : Nat → Nat
```

```
{-# BUILTIN NATURAL Nat #-}
```

```
_+_ : Nat → Nat → Nat
zero + y = y
suc x + y = suc (x + y)
```

```
infixl 6 _+_
```

EXERCISE 1.1: Define the function `halve : Nat → Nat` that computes the result of dividing the given number by 2 (rounded down). Test your definition by evaluating it for several concrete inputs.

```
halve : Nat → Nat
halve 0 = 0
halve 1 = 0
halve (suc (suc n)) = halve n + 1
```

EXERCISE 1.2: Define the function `_*_ : Nat → Nat → Nat` for multiplication of two natural numbers.

```
_*_ : Nat → Nat → Nat
0 * y = 0
suc x * y = y + (x * y)
```

```
infixl 7 _*_
```

EXERCISE 1.3: Define the type `Bool` with constructors `true` and `false`, and define the functions for negation `not : Bool → Bool`, conjunction `_&&_ : Bool → Bool → Bool`, and disjunction `_||_ : Bool → Bool → Bool` by pattern matching.

```
data Bool : Set where
  true  : Bool
  false : Bool
```

```
not : Bool → Bool
not true = false
not false = true
```

```

id : {A : Set} → A → A
id x = x

data List (A : Set) : Set where
  [] : List A
  _::_ : A → List A → List A

infixr 5 _::_

data _×_ (A B : Set) : Set where
  _,_ : A → B → A × B

fst : {A B : Set} → A × B → A
fst (x , _) = x

snd : {A B : Set} → A × B → B
snd (_, y) = y

```

#### EXERCISE 1.4:

```

length : {A : Set} → List A → Nat
length [] = zero
length (x :: xs) = suc (length xs)

_++_ : {A : Set} → List A → List A → List A
[] ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)

map : {A B : Set} → (A → B) → List A → List B
map f [] = []
map f (x :: xs) = f x :: map f xs

```

#### EXERCISE 1.5:

```

data Maybe (A : Set) : Set where
  nothing : Maybe A
  just : A → Maybe A

lookup : {A : Set} → List A → Nat → Maybe A
lookup [] _ = nothing
lookup (x :: xs) zero = just x
lookup (x :: xs) (suc i) = lookup xs i

data Vec (A : Set) : Nat → Set where
  [] : Vec A zero
  _::_ : {n : Nat} → A → Vec A n → Vec A (suc n)

replicateVec : {A : Set} → (n : Nat) → A → Vec A n
replicateVec zero x = []
replicateVec (suc n) x = x :: replicateVec n x

```

#### EXERCISE 2.1:

```

downFrom : (n : Nat) → Vec Nat n
downFrom zero = []
downFrom (suc x) = x :: downFrom x

```

```

_++Vec_ : {A : Set} {m n : Nat}
  → Vec A m → Vec A n → Vec A (m + n)
[] ++Vec ys = ys
(x :: xs) ++Vec ys = x :: (xs ++Vec ys)

head : {A : Set} {n : Nat} → Vec A (suc n) → A
head (x :: _) = x

```

EXERCISE 2.2:

```

tail : {A : Set} {n : Nat} → Vec A (suc n) → Vec A n
tail (_ :: xs) = xs

```

EXERCISE 2.3:

```

dotProduct : {n : Nat} → Vec Nat n → Vec Nat n → Nat
dotProduct [] [] = zero
dotProduct (x :: xs) (y :: ys) = x * y + dotProduct xs ys

data Fin : Nat → Set where
  zero : {n : Nat} → Fin (suc n)
  suc : {n : Nat} → Fin n → Fin (suc n)

zero3 : Fin 3
zero3 = zero

```

```

lookupVec : {A : Set} {n : Nat} → Vec A n → Fin n → A
lookupVec (x :: xs) zero = x
lookupVec (x :: xs) (suc i) = lookupVec xs i

```

EXERCISE 2.4:

```

putVec : {A : Set} {n : Nat} → Fin n → A → Vec A n → Vec A n
putVec zero x (_ :: xs) = x :: xs
putVec (suc i) x (x1 :: xs) = x1 :: putVec i x xs

data Σ (A : Set) (B : A → Set) : Set where
  _,_ : (x : A) → B x → Σ A B

fstΣ : {A : Set} {B : A → Set} → Σ A B → A
fstΣ (x , _) = x

sndΣ : {A : Set} {B : A → Set} → (z : Σ A B) → B (fstΣ z)
sndΣ (x , y) = y

_×'_ : (A B : Set) → Set
A ×' B = Σ A (λ _ → B)

```

EXERCISE 2.5:

```

fromProd : {A B : Set} → A × B → A ×' B
fromProd (x , y) = x , y

toProd : {A B : Set} → A ×' B → A × B
toProd (x , y) = x , y

```

```
List' : (A : Set) → Set
List' A =  $\Sigma$  Nat (Vec A)
```

#### EXERCISE 2.6:

```
[]' : {A : Set} → List' A
[]' = zero , []
```

```
_::'_ : {A : Set} → A → List' A → List' A
x ::' (n , xs) = suc n , (x :: xs)
```

```
fromList : {A : Set} → List A → List' A
fromList [] = []'
fromList (x :: xs) = x ::' fromList xs
```

```
fromList' : {A : Set} → List' A → List A
fromList' (zero , []) = []
fromList' (suc n , (x :: xs)) = x :: (fromList' (n , xs))
```

#### EXERCISE 3.1:

```
data Either (A : Set) (B : Set) : Set where
  left : A → Either A B
  right : B → Either A B
```

```
cases : {A B C : Set} → Either A B → (A → C) → (B → C) → C
cases (left x) f _ = f x
cases (right x) _ f = f x
```

```
data  $\tau$  : Set where
  tt :  $\tau$ 
```

```
data  $\perp$  : Set where
```

```
absurd : {A : Set} →  $\perp$  → A
absurd ()
```

#### EXERCISE 3.2

- If  $A$  then ( $B$  implies  $A$ )
 

```
p1 : {A B : Set} → A → (B → A)
p1 x =  $\lambda$  _ → x
```
- If ( $A$  and *true*) then ( $A$  or *false*)
 

```
p2 : {A : Set} → (A ×  $\tau$ ) → (Either A  $\perp$ )
p2 (x , tt) = left x
```
- If  $A$  implies ( $B$  implies  $C$ ), then ( $A$  and  $B$ ) implies  $C$ .
 

```
p3 : {A B C : Set} → (A → (B → C)) → ((A × B) → C)
p3 f (x , y) = f x y
```
- If  $A$  and ( $B$  or  $C$ ), then either ( $A$  and  $B$ ) or ( $A$  and  $C$ ).
 

```
p4 : {A B C : Set} → (A × (Either B C)) → (Either (A × B) (A × C))
p4 (x , left y) = left (x , y)
p4 (x , right z) = right (x , z)
```

- If  $A$  implies  $C$  and  $B$  implies  $D$ , then  $(A \text{ and } B)$  implies  $(C \text{ and } D)$ .

p5 : {A B C D : Set} → ((A → C) × (B → D)) → ((A × B) → (C × D))

p5 (f , g) (x , y) = f x , g y

proof3 : {P Q R : Set} → (Either P Q → R) → (P → R) × (Q → R)

proof3 f = (λ x → f (left x)) , (λ x → f (right x))

EXERCISE 3.3: Write a function of type {P : Set} → (Either P (P → ⊥) → ⊥) → ⊥.

Assuming (Either P (P → ⊥) → ⊥) then proof3 above says that P → ⊥ and (P → ⊥) → ⊥.

Applying (P → ⊥) → ⊥ to P → ⊥ results in ⊥ which proves the proposition.

constructive-P-or-not-P : {P : Set} → (Either P (P → ⊥) → ⊥) → ⊥

constructive-P-or-not-P {P} f =

(λ (x : P → ⊥) → f (right x)) (λ (x : P) → f (left x))

Some even stuff:

data IsEven : Nat → Set where

zeroIsEven : IsEven zero

sucsucIsEven : {n : Nat} → IsEven n → IsEven (suc (suc n))

6-is-even : IsEven 6

6-is-even = sucsucIsEven (sucsucIsEven (sucsucIsEven zeroIsEven))

7-is-even : IsEven 7 → ⊥

7-is-even (sucsucIsEven (sucsucIsEven (sucsucIsEven ())))

data IsTrue : Bool → Set where

TrueIsTrue : IsTrue true

\_ =Nat\_ : Nat → Nat → Bool

zero =Nat zero = true

suc x =Nat suc y = x =Nat y

\_ =Nat \_ = false

length-is-3 : IsTrue (length (1 :: 2 :: 3 :: []) =Nat 3)

length-is-3 = TrueIsTrue

double : Nat → Nat

double zero = zero

double (suc n) = suc (suc (double n))

double-is-even : (n : Nat) → IsEven (double n)

double-is-even zero = zeroIsEven

double-is-even (suc n) = sucsucIsEven (double-is-even n)

n-equals-n : (n : Nat) → IsTrue (n =Nat n)

n-equals-n zero = TrueIsTrue

n-equals-n (suc n) = n-equals-n n

half-a-dozen : Σ Nat (λ n → IsTrue ((n + n) =Nat 12))

half-a-dozen = 6 , TrueIsTrue

zero-or-suc : (n : Nat) → Either

(IsTrue (n =Nat zero))

(Σ Nat (λ m → IsTrue (n =Nat (suc m))))

zero-or-suc zero = left TrueIsTrue

zero-or-suc (suc m) = right (m , n-equals-n m)

## THE IDENTITY TYPE

```
data _≡_ {A : Set} : A → A → Set where
  refl : {x : A} → x ≡ x
```

```
infix 4 _≡_
```

```
n-equals-n-≡ : (n : Nat) → n ≡ n
n-equals-n-≡ n = refl
```

```
zero-not-one : 0 ≡ 1 → ⊥
zero-not-one ()
```

I am curious about an equivalency that must take an argument:

```
data _≡'_ {A : Set} : A → A → Set where
  refl : (x : A) → x ≡' x
```

```
infix 4 _≡'_
```

```
n-equals-n-≡' : (n : Nat) → n ≡' n
n-equals-n-≡' n = refl n
```

```
n-equals-n-≡'' : (n : Nat) → n ≡'' n
n-equals-n-≡'' = refl
```

Various laws of equivalency:

```
sym : {A : Set} {x y : A} → x ≡ y → y ≡ x
sym refl = refl
```

```
trans : {A : Set} {x y z : A} → x ≡ y → y ≡ z → x ≡ z
trans refl refl = refl
```

```
cong : {A B : Set} {x y : A} → (f : A → B) → x ≡ y → f x ≡ f y
cong f refl = refl
```

## EQUATIONAL REASONING

```
begin_ : {A : Set} → {x y : A} → x ≡ y → x ≡ y
begin p = p
```

```
_end : {A : Set} → (x : A) → x ≡ x
x end = refl
```

```
_≡(<_>_ : {A : Set} → (x : A) → {y z : A}
  → x ≡ y → y ≡ z → x ≡ z
x ≡(< p > q = trans p q
```

```
_≡(<_>_ : {A : Set} → (x : A) → {y : A} → x ≡ y → x ≡ y
x ≡(< q = x ≡(< refl > q
```

```
infix 1 begin_
infix 3 _end
infixr 2 _≡(<_>_
infixr 2 _≡(<_>_
```

Simple examples:

```
[_] : {A : Set} → A → List A
[ x ] = x :: []
```

```
reverse : {A : Set} → List A → List A
reverse [] = []
reverse (x :: xs) = reverse xs ++ [ x ]
```

```
reverse-singleton : {A : Set} (x : A) → reverse [ x ] ≡ [ x ]
reverse-singleton x =
  begin
    reverse [ x ]
  ≡⟨⟩
    reverse (x :: [])
  ≡⟨⟩
    reverse [] ++ [ x ]
  ≡⟨⟩
    [] ++ [ x ]
  ≡⟨⟩
    [ x ]
  end
```

Proof by induction and cases:

```
add-n-zero : (n : Nat) → n + zero ≡ n
add-n-zero zero = refl
add-n-zero (suc n) =
  begin
    suc n + zero
  ≡⟨⟩
    suc (n + zero)
  ≡⟨ cong suc (add-n-zero n) ⟩
    suc n
  end
```

EXERCISE 4.1: Prove that  $m + \text{suc } n = \text{suc } (m + n)$  for all natural numbers  $m$  and  $n$ . Next, use the previous lemma and this one to prove commutativity of addition, i.e. that  $m + n = n + m$  for all natural numbers  $m$  and  $n$ .

```

add-suc : (m n : Nat) → m + suc n ≡ suc (m + n)
add-suc zero n = refl
add-suc (suc m) n = cong suc (add-suc m n)

add-commut : (m n : Nat) → m + n ≡ n + m
add-commut zero n = sym (add-n-zero n)
add-commut (suc m) n =
  begin
    suc m + n
  ≡⟨ ⟩
    suc (m + n)
  ≡⟨ cong suc (add-commut m n) ⟩
    suc (n + m)
  ≡⟨ sym (add-suc n m) ⟩
    n + suc m
  end

add-commut' : (m n : Nat) → m + n ≡ n + m
add-commut' zero n = sym (add-n-zero n)
add-commut' (suc m) n =
  trans (cong suc (add-commut' m n)) (sym (add-suc n m))

add-assoc : (x y z : Nat) → x + (y + z) ≡ (x + y) + z
add-assoc zero y z = refl
add-assoc (suc x) y z = cong suc (add-assoc x y z)

```

EXERCISE 4.2: Consider the following function:

```

replicate : {A : Set} → Nat → A → List A
replicate zero x = []
replicate (suc n) x = x :: replicate n x

```

Prove that the length of `replicate n x` is always equal to  $n$ , by induction on the number  $n$ .

```

repl-length : {A : Set} → (n : Nat) → (x : A) → length (replicate n x) ≡ n
repl-length zero x = refl -- length [] ≡ zero
repl-length (suc n) x =
  begin
    length (replicate (suc n) x)
  ≡⟨ ⟩ -- Apply replicate
    length (x :: replicate n x)
  ≡⟨ ⟩ -- Apply length
    suc (length (replicate n x))
  ≡⟨ cong suc (repl-length n x) ⟩
    suc n
  end

```

EXERCISE 4.3: The proofs of the lemmas:



```

append-[] : {A : Set} → (xs : List A) → xs ++ [] ≡ xs
append-[] [] = begin ([] ++ []) ≡() [] end
append-[] (x :: xs) =
  begin
    (x :: xs) ++ []
  ≡() -- Apply ++
    x :: (xs ++ [])
  ≡( cong (x ::_) (append-[] xs) )
    x :: xs
  end

```

```

append-assoc : {A : Set} → (xs ys zs : List A)
  → (xs ++ ys) ++ zs ≡ xs ++ (ys ++ zs)

```

```

append-assoc [] ys zs =
  begin
    ([] ++ ys) ++ zs
  ≡() -- Apply ++
    ys ++ zs
  ≡() -- Unapply ++
    [] ++ (ys ++ zs)
  end
append-assoc (x :: xs) ys zs =
  begin
    ((x :: xs) ++ ys) ++ zs
  ≡() -- Apply ++
    (x :: (xs ++ ys)) ++ zs
  ≡() -- Apply ++
    x :: ((xs ++ ys) ++ zs)
  ≡( cong (x ::_) (append-assoc xs ys zs) )
    x :: (xs ++ (ys ++ zs))
  ≡() -- Unapply ++
    (x :: xs) ++ (ys ++ zs)
  end

```

Now we can prove distributivity of reverse:

```

reverse-distributivity : {A : Set} → (xs ys : List A)
  → reverse (xs ++ ys) ≡ reverse ys ++ reverse xs
reverse-distributivity [] ys =
  begin
    reverse ([] ++ ys)
  ≡() -- Apply ++
    reverse ys
  ≡( sym (append-[] (reverse ys)) ) -- Use the append-[] lemma
    reverse ys ++ []
  ≡() -- Unapply reverse to []
    reverse ys ++ reverse []
  end

```

```

reverse-distributivity (x :: xs) ys =
  begin
    reverse ((x :: xs) ++ ys)
  ≡⟨⟩ -- Apply ++
    reverse (x :: (xs ++ ys))
  ≡⟨⟩ -- Apply reverse
    reverse (xs ++ ys) ++ [ x ]
  ≡⟨ cong (λ_ ++ [ x ]) (reverse-distributivity xs ys) ⟩
    (reverse ys ++ reverse xs) ++ [ x ]
  ≡⟨ append-assoc (reverse ys) (reverse xs) [ x ] ⟩
    reverse ys ++ (reverse xs ++ [ x ])
  ≡⟨⟩ -- Unapply reverse
    reverse ys ++ reverse (x :: xs)
  end

```

And that reversing twice is idempotent:

```

reverse-reverse : {A : Set} → (xs : List A) → reverse (reverse xs) ≡ xs
reverse-reverse [] = refl
reverse-reverse (x :: xs) =
  begin
    reverse (reverse (x :: xs))
  ≡⟨⟩ -- Apply inner reverse
    reverse (reverse xs ++ [ x ])
  ≡⟨ reverse-distributivity (reverse xs) [ x ] ⟩
    reverse [ x ] ++ reverse (reverse xs)
  ≡⟨ cong ([ x ] ++_) (reverse-reverse xs) ⟩
    [ x ] ++ xs
  ≡⟨⟩ -- Apply ++
    x :: xs
  end

```