0.1 Small categories

To define a small category, we will need the following definition:

- Ob : Set — Set of all objects
- Hom : Ob → Ob → Set — Set of all morphisms
- id : ∀ X : Ob .Hom X X — Identity morphism
- _;_ : ∀ X, Y, Z : Ob . Hom X Y → Hom Y Z → Hom X Z — Composition of morphisms

and the following laws:

- left-identity law
- right-identity law
- associativity law

This can be implemented in Agda:

```
record Cat : Set₁ where
  field Ob  : Set
        Hom : Ob → Ob → Set
        id  : {X : Ob} → Hom X X
        _▶_ : {X Y Z : Ob} → Hom X Y → Hom Y Z → Hom X Z

        left_id  : {X Y : Ob} {f : Hom X Y} → id ▶ f ≡ f
        right_id : {X Y : Ob} {f : Hom X Y} → f ▶ id ≡ f
        assoc    : {W X Y Z : Ob}
                   {f : Hom W X}
                   {g : Hom X Y}
                   {h : Hom Y Z} → f ▶ (g ▶ h) ≡ (f ▶ g) ▶ h
```

With this, an instance of any small category can be made.

For instance, the category of endofunctions on Booleans.

This category is one which rises from a monoid, so we use the same definition previously seen to make this category in Agda.

First, we define the data types:

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

data Unit : Set where
  unit : Unit
```

Unit is the singleton set I will be using as the objects of the category of endofunctions on Booleans. Bool is the set of Booleans which I will be using to define all endofunctions.

With this in place, we can define the category:

```
setℙB : Cat
setℙB = record
        { Ob  = Unit
        ; Hom = λ unit unit → Bool → Bool
        ; id  = λ b → b
        ; _▶_ = λ f g → λ x → g (f x)
        ; left_id  = refl
        ; right_id = refl
        ; assoc    = refl
        }
```

0.2 Set in Agda

With the previous definition, we were unable to define **Set**, however. To do this, we must define a universe level polymorphic record.

```
record Category {l : Level} : Set (lsuc l) where
  field Ob  : Set l
        Hom : Ob → Ob → Set i
        id  : {X : Ob} → Hom X X
        _▶_ : {X Y Z : Ob} → Hom X Y → Hom Y Z → Hom X Z

        left_id  : {X Y : Ob} {f : Hom X Y} → id ▶ f ≡ f
        right_id : {X Y : Ob} {f : Hom X Y} → f ▶ id ≡ f
        assoc    : {W X Y Z : Ob}
                   {f : Hom W X}
                   {g : Hom X Y}
                   {h : Hom Y Z} → f ▶ (g ▶ h) ≡ (f ▶ g) ▶ h
```

With this definition, we can finally define **Set**:

```
set : Category
set = record
        { Ob  = Set
        ; Hom = λ X Y → Set → Set
        ; id  = λ X → X
        ; _▶_ = λ {X Y Z : Set} f g → λ x → g (f x)
        ; left_id  = refl
        ; right_id = refl
        ; assoc    = refl
        }
```

0.3 Defining monads

It is possible to define a monad in the traditional way:

- T an endofunctor on C
- η a unit natural transformation
- id μ a multiplication natural transformation

The downside is that it would also require the definition of a functor and all the proves associated. Alternatively, a monad can also be defined as a Kleisli Triple:

- T : Ob C → Ob C a morphism in C
- η : X : Ob C → Hom_C(X,T X) a unit transformation
- $(\_)^*$: X Y : Ob C → Hom_C(X,T Y) → Hom_C(T X, T Y) the Kleisli operator

This is easier to implement since no definition of a functor is required, and no mention of naturality is given either. Hence I implemented monads as a Kleisli triple.

Given the following projections:

```
Ob      = Category.Ob
Hom     = Category.Hom
id      = Category.id
compose = Category._▶_
```

The definition of a Kleisli Triple is given as follows:

```
record Monad {l : Level}{C : Category {l}} : Set (lsuc l) where
  field T  : Ob C → Ob C
        η  : {X : Ob C} → Hom C X (T X)
        _* : {X Y : Ob C} → Hom C X (T Y) → Hom C (T X) (T Y)

        monad_lid   : {X Y : Ob C}
                      {f : Hom C X (T Y)} →
                      compose C η (f *) ≡ f
        monad_rid   : {X : Ob C} →
                      (η {X})* ≡ id C {T X}
        monad_assoc : {X Y Z : Ob C}
                      {f : Hom C X (T Y)}
                      {g : Hom C Y (T Z)} →
                      compose C (f *) (g *) ≡ (compose C f (g *)) *
```

0.4 State monad

Now that monads are defined, it is possible to check the answer to the exercise at the end of the last chapter. We can show—in Agda—that the state monad defined previously is indeed a valid monad.

First, we need products defined:

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

π₁ : {A B : Set} → (A × B) → A
π₁ (a , b) = a

π₂ : {A B : Set} → (A × B) → B
π₂ (a , b) = b

πid : {A B : Set}{p : A × B} → π₁ p , π₂ p ≡ p
πid {A}{B}{a , b} = refl
```

Assuming function extensionality:

```
postulate exten : {X Y : Set}{f g : X → Y} →
                  ((x : X) → f x ≡ g x) → (f ≡ g)

exten2 : {X Y Z : Set}{f g : X → Y → Z} →
         ((x : X)(y : Y) → f x y ≡ g x y) → (f ≡ g)
exten2 h = exten (λ x → exten(h x))
```

We can now define the state monad—on **Set**—and prove its laws:

```
state : Set → Monad {lsuc lzero}{set}
state S = record
          { T  = λ A → S → (S × A)
          ; η = λ a → λ s → (s , a)
          ; _⋆ = λ g f → λ s → g (π₂ (f s)) (π₁ (f s))
          ; monad_lid   = refl
          ; monad_rid   = lemma_rid
          ; monad_assoc = refl
          }
          where
            lemma_rid : {A : Ob set} →
                        (λf s → π₁ (f s) , π₂ (f s))
                        ≡ id set {S → (S × A)}
            lemma_rid {A} =
              begin
                (λ f s → π₁ (f s) , π₂ (f s))
                ≡⟨ exten2 (λ f s → πid {p = f s}) ⟩
                (λ f s → f s)
                ≡⟨ refl ⟩
                (λ f → f)
              ■
```