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)
```

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)
```

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 [] = 0
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
```