Compile with @pandoc -fhaddock+lhs fpcomplete.lhs -o fpcomplete.pdf

```
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE UndecidableInstances #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE ExistentialQuantification #-}
{-# LANGUAGE StandaloneDeriving #-}
```

Type functions

With `DataKinds` language extension, the following defines a data kind named `Nat` and two type constructor `Z :: Nat` and `S :: Nat -> 'Nat`

Check with:

```
> :set -XDataKinds
> :k 'Z
```

```
data Nat = Z | S Nat
```

Note that data kinds other than * cannot have values.

To define type function, we need the `TypeFamilies` extension. A type family can be regarded as the type signature of a data function. And the type instances is the implementation of the type function.

```
type family Plus (m :: Nat) (n :: Nat) :: Nat
type instance Plus Z n = n
type instance Plus (S m) n = S (Plus m n)
```

We want to make the plus function look more naturally, so we can introduce `TypeOperators` extension for us to define some syntatic sugars.

Then we can define it in this way:

```
infixl 6 :+
```

```
type family (m :: Nat) :+ (n :: Nat) :: Nat
type instance  Z    :+ n = n
type instance (S m) :+ n = S (m :+ n)
```

We can also define multiplication in similar way, note we need `UndecidableInstances` lang ext for this type function to type check.

```
infixl 7 :*
type family (m :: Nat) :* (n :: Nat) :: Nat
type instance  Z    :* n = Z
type instance (S m) :* n = (m :* n) :+ n
```

GHC can resolve the name `Z`/`S` as type constructors instead of value constructors normally, but whenever there's an ambiguity, we can always use `'Z`/`'S` to explicitly denote them as type constructors.

GADTs

GADTs (Generalized Algebraic Data-Types) enable us to define types function that depends on types with kinds other than *. To use GADTs, we need to enable it as GHC language extension.

We define our lengthed-vector like this:

```
data Vec n a where
   Nil   :: Vec Z a
   Cons  :: a -> Vec n a -> Vec (S n) a
```

Alternatively, we can define the same thing with type level equality and existential qualitification (language ext `ExistentialQuantification`):

```
data Vec' n a = (n ~ Z)              => Nil'
              | forall m. (n ~ S m) => Cons' a (Vec' m a)
```

This **forall m** here should actually be read as **there is some m** (thus is an existential quantification).

We now defined Vec data type, and we want it to derive some standard typeclasses. For this purpose, we need to use `StandaloneDeriving` language extension instead of the `deriving` clause.

```
deriving instance Eq a => Eq (Vec n a)
deriving instance Show a => Show (Vec n a)
```

Let's implement some operations on them.

```
head' :: Vec (S n) a -> a
head' (Cons x xs) = x
```

```
tail' :: Vec (S n) a -> Vec n a
tail' (Cons x xs) = xs
```

How fun!

```
-- The following code won't even compile!
-- head' Nil
```

Let's play with some type level arithmetic. Define the `append` function:

```
append :: Vec n a -> Vec m a -> Vec (n :+ m) a
append Nil         ys = ys
append (Cons x xs) ys = Cons x (append xs ys)
```

Happily this type checks! This means GHC checked the logic we encoded in the types and concluded this implementation works. (It only checks the length of the resulting vector though, since we didn't put other info in the type, e.g. the order of elements)

We can also implement these functions:

```
toList :: Vec n a -> [a]
toList Nil         = []
toList (Cons x xs) = x : toList xs
```

```
fromList :: [a] -> Vec n a
fromList []     = Nil
fromList (x:xs) = Cons x $ fromList xs
```

```
map' :: (a -> b) -> Vec n a -> Vec n b
map' f Nil         = Nil
map' f (Cons x xs) = Cons (f x) $ map' f xs
```

```
uncons :: Vec (S n) a -> (a, Vec n a)
uncons (Cons x xs) = (x, xs)
```

```
init' :: Vec (S n) a -> Vec n a
init' (Cons x Nil) = Nil
init' (Cons x xs)  = Cons x (init xs)
```

```
last' :: Vec (S n) a -> a
last' (Cons x Nil) = x
last' (Cons x xs)  = last' xs
```

```
zipWithSame :: (a -> b -> c) -> Vec n a -> Vec n b -> Vec n c
zipWithSame f Nil         Nil         = Nil
zipWithSame f (Cons x xs) (Cons y ys) = Cons (f x y) $ zipWithSame f xs ys
```

```
type family Min (m :: Nat) (n :: Nat) :: Nat
type instance Min Z n         = Z
type instance Min m Z         = Z
type instance Min (S m) (S n) = S (Min m n)
```

Then we can use `Min` to implement a more general version of `zipWith`:

```
zipWith' :: (a -> b -> c) -> Vec m a -> Vec n b -> Vec (Min m n) c
zipWith' f Nil         ys          = Nil
zipWith' f xs          Nil         = Nil
zipWith' f (Cons x xs) (Cons y ys) = Cons (f x y) (zipWith' f xs ys)
```

Singleton patterns