Code (Wednesday Week 8)

Printf

Haskell

```
{-# LANGUAGE GADTs, KindSignatures, DataKinds #-}

data Format :: * -> * where
  End :: Format ()
  Str :: Format t -> Format (String, t)
  Dec :: Format t -> Format (Int, t)
  L :: String -> Format t -> Format t

printf :: Format ts -> ts -> IO ()
printf End () = pure ()
printf (Str fmt) (s,ts) =
   do
     putStr s;
     printf fmt ts
printf (Dec fmt) (i,ts) =
   do
     putStr (show i);
     printf fmt ts
printf (L s fmt) ts =
  do
    putStr s;
    printf fmt ts
```

Length-indexed vectors

```
{-# LANGUAGE GADTs, KindSignatures, DataKinds #-}
{-# LANGUAGE TypeFamilies, UndecidableInstances #-}
{-# LANGUAGE StandaloneDeriving #-}
data Nat = Z | S Nat

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

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

type family Plus (a :: Nat) (b :: Nat) :: Nat where
  Plus Z n = n
  Plus (S m) n = S (Plus m n)

appendVec :: Vec a m -> Vec a n -> Vec a (Plus m n)
appendVec Nil ys = ys
appendVec (Cons x xs) ys = Cons x (appendVec xs ys)

type family Times (a :: Nat) (b :: Nat) :: Nat where
  Times Z n = Z
  Times (S m) n = Plus n (Times m n)

concatVec :: Vec (Vec a m) n -> Vec a (Times n m)
concatVec Nil = Nil
concatVec (Cons v vs) = v `appendVec` concatVec vs

filterVec :: (a -> Bool) -> Vec a n -> [a]
filterVec p Nil = []
filterVec p (Cons x xs) | p x       = x : filterVec p xs
                        | otherwise = filterVec p xs
```