Code (Tuesday Week 8)

Phantom Types

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

data Stream = UG | PG
data StudentID (x :: Stream) = SID Int

-- data Either a b = Left a | Right b

postgrad :: [Int]
postgrad = [3253158]

makeStudentID :: Int -> Either (StudentID UG) (StudentID PG)
makeStudentID i | i `elem` postgrad = Right (SID i)
                | otherwise         = Left  (SID i)

enrollInCOMP3141 :: StudentID UG -> IO ()
enrollInCOMP3141 (SID x)
  = putStrLn (show x ++ " enrolled in COMP3141!")
```

GADTs

Untyped Evaluator

```haskell
data Expr t = BConst Bool
            | IConst Int
            | Times (Expr Int) (Expr Int)
            | Less (Expr Int) (Expr Int)
            | And (Expr Bool) (Expr Bool)
            | If (Expr Bool) (Expr t) (Expr t)
            deriving (Show, Eq)
data Value = BVal Bool | IVal Int
             deriving (Show, Eq)

eval :: Expr -> Value
eval (BConst b) = BVal b
eval (IConst i) = IVal i
eval (Times e1 e2) = case (eval e1, eval e2) of
                       (IVal i1, IVal i2) -> IVal (i1 * i2)
eval (Less e1 e2)  = case (eval e1, eval e2) of
                       (IVal i1, IVal i2) -> BVal (i1 < i2)
eval (And e1 e2)  = case (eval e1, eval e2) of
                       (BVal b1, BVal b2) -> BVal (b1 && b2)
eval (If ec et ee) =
  case eval ec of
    BVal True  -> eval et
    BVal False -> eval ee
```

Typed Evaluator

```haskell
{-# LANGUAGE GADTs, KindSignatures #-}
data Expr :: * -> * where
  BConst :: Bool -> Expr Bool
  IConst :: Int -> Expr Int
  Times  :: Expr Int -> Expr Int -> Expr Int
  Less   :: Expr Int -> Expr Int -> Expr Bool
  And    :: Expr Bool -> Expr Bool -> Expr Bool
  If     :: Expr Bool -> Expr a -> Expr a -> Expr a

eval :: Expr t -> t
eval (IConst i)    = i
eval (BConst b)    = b
eval (Times e1 e2) = eval e1 * eval e2
eval (Less e1 e2)  = eval e1 < eval e2
eval (And e1 e2)   = eval e1 && eval e2
eval (If ec et ee) = if eval ec then eval et else eval ee
```

Length-indexed vectors

```haskell
{-# LANGUAGE GADTs, KindSignatures #-}
{-# LANGUAGE DataKinds, StandaloneDeriving, TypeFamilies #-}

data Nat = Z | S Nat

plus :: Nat -> Nat -> Nat
plus Z n = n
plus (S m) n = S (plus m n)

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

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)

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

-- 0: Z
-- 1: S Z
-- 2: S (S Z)

hd :: Vec a (S n) -> a
hd (Cons x xs) = x


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