```haskell
module List where

import Lib (CanBeEmpty(..), FromList(..), ToList(..))

data List a = Cons a (List a)
            | EmptyList
            deriving (Eq)

-- :k Show, Show :: * -> Constraint
-- {-# MINIMAL showsPrec | show #-}
instance Show a => Show (List a) where
    -- | Show the list
    show EmptyList = "[]"
    show (Cons a b) = show a ++ ":" ++ show b

-- :k FromList, FromList :: (* -> *) -> Constraint
-- {-# MINIMAL fromList #-}
instance FromList List where
    fromList = foldr Cons EmptyList

-- :k Functor, Functor :: (* -> *) -> Constraint
-- {-# MINIMAL fmap #-}
instance Functor List where
    fmap _ EmptyList = EmptyList
    fmap f (Cons x xs) = Cons (f x) (fmap f xs)
    (<$) :: b -> List a -> List b
    (<$) x EmptyList = EmptyList
    (<$) x (Cons _ xs) = Cons x (x <$ xs)

-- k: ToList, ToList :: (* -> *) -> Constraint
-- {-# MINIMAL toList #-}
instance ToList List where
    toList EmptyList = []
    toList (Cons x xs) = x : toList xs

-- :k Foldable, Foldable :: (* -> *) -> Constraint
-- {-# MINIMAL foldMap | foldr #-}
instance Foldable List where
    foldr _ z EmptyList = z
    foldr f z (Cons x xs) = f x (foldr f z xs)
    foldl _ z EmptyList = z
    foldl f z (Cons x xs) = foldl f (f z x) xs
    foldMap _ EmptyList = mempty
    foldMap f (Cons x xs) = f x `mappend` foldMap f xs
    foldr1 _ EmptyList = error "foldr1: empty structure"
    foldr1 f (Cons x xs) = foldr f x xs
    foldl1 _ EmptyList = error "foldl1: empty structure"
    foldl1 f (Cons x xs) = foldl f x xs
    null EmptyList = True
    null _ = False
    length EmptyList = 0
    length (Cons _ xs) = 1 + length xs
    elem _ EmptyList = False
    elem e (Cons x xs) = (e == x) || elem e xs
    maximum EmptyList = error "maximum: empty structure"
    maximum (Cons x xs) = foldl max x xs
    minimum EmptyList = error "minimum: empty structure"
    minimum (Cons x xs) = foldl min x xs
    sum EmptyList = 0
    sum (Cons x xs) = x + sum xs
    product EmptyList = 1
    product (Cons x xs) = x * product xs

-- :k CanBeEmpty, CanBeEmpty :: * -> Constraint
-- {-# MINIMAL isEmpty #-}
instance CanBeEmpty (List a) where
    isEmpty EmptyList = True
    isEmpty _ = False

-- :k Semigroup, Semigroup :: (* -> *) -> Constraint
-- {-# MINIMAL (<>) #-}
instance Semigroup (List a) where
    EmptyList <> xs = xs
    xs <> EmptyList = xs
    (Cons x xs) <> ys = Cons x (xs <> ys)

-- :k Monoid, Monoid :: * -> Constraint
-- {-# MINIMAL mempty #-}
instance Monoid (List a) where
    mempty = EmptyList
    mappend = (<>)
    mconcat = foldr mappend mempty

-- :k Applicative, Applicative :: (* -> *) -> Constraint
-- {-# MINIMAL pure, ((<*>) | liftA2) #-}
instance Applicative List where
    pure x = Cons x EmptyList
    EmptyList <*> _ = EmptyList
    _ <*> EmptyList = EmptyList
    (Cons f fs) <*> xs = (fmap f xs) <> (fs <*> xs)

-- :k Monad, Monad :: (* -> *) -> Constraint
-- {-# MINIMAL (>>=) #-}
instance Monad List where
    EmptyList >>= _ = EmptyList
    (Cons a as) >>= f = (f a) <> (as >>= f)
    return = pure

insert :: a -> List a -> List a
insert x xs = Cons x xs

delete :: Eq a => a -> List a -> List a
delete _ EmptyList = EmptyList
delete y (Cons x xs)
    | x == y = xs
    | otherwise = Cons x (delete y xs)

search :: Eq a => a -> List a -> Bool
search _ EmptyList = False
search y (Cons x xs) = (x == y) || search y xs

exampleList :: List Int
exampleList = insert 3 (insert 2 (insert 1 EmptyList))

lf = Cons (+1) (Cons (*2) EmptyList) :: List (Int -> Int)
lx = Cons 1 (Cons 2 (Cons 3 EmptyList)) :: List Int
```