```haskell
module Stack where

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

-- Definition des Stack-DatenTyps
data Stack a = Stack a (Stack a)
             | EmptyStack

-- Show-Instanz für Stack
instance Show a => Show (Stack a) where
  show EmptyStack = "EmptyStack" -- "===="
  show (Stack a rest) = show a ++ " -> " ++ show rest

-- FromList-Instanz für Stack
-- {-# MINIMAL fromList #-}
instance FromList Stack where
  fromList = foldr Stack EmptyStack

-- ToList-Instanz für Stack
-- {-# MINIMAL toList #-}
instance ToList Stack where
  toList = foldr (:) []

-- Functor-Instanz für Stack
-- {-# MINIMAL fmap #-}
instance Functor Stack where
  fmap _ EmptyStack = EmptyStack
  fmap f (Stack x rest) = Stack (f x) (fmap f rest)
  -- (<$) ist nicht unbedingt notwendig, weil es in Functor bereitgestellt
wird,
  -- aber wir können es für Vollständigkeit auch einfügen.
  (<$) :: b -> Stack a -> Stack b
  (<$) x EmptyStack = EmptyStack
  (<$) x (Stack _ rest) = Stack x (x <$ rest)


-- CanBeEmpty-Instanz für Stack
-- {-# MINIMAL isEmpty #-}
instance CanBeEmpty (Stack a) where
  isEmpty EmptyStack = True
  isEmpty _ = False

-- Semigroup-Instanz für Stack
-- {-# MINIMAL (<>) #-}
instance Semigroup (Stack a) where
  EmptyStack <> stack = stack
  stack <> EmptyStack = stack
  Stack x rest <> stack = Stack x (rest <> stack)
```

```haskell
-- Foldable-Instanz für Stack
-- {-# MINIMAL foldMap | foldr #-}
instance Foldable Stack where
  foldr _ z EmptyStack = z
  foldr f z (Stack x rest) = f x (foldr f z rest)
  foldl _ z EmptyStack = z
  foldl f z (Stack x rest) = foldl f (f z x) rest
  foldMap _ EmptyStack = mempty
  foldMap f (Stack x rest) = f x `mappend` foldMap f rest
  foldr1 _ EmptyStack = error "foldr1: empty structure"
  foldr1 f (Stack x rest) = foldr f x rest
  foldl1 _ EmptyStack = error "foldl1: empty structure"
  foldl1 f (Stack x rest) = foldl f x rest
  null EmptyStack = True
  null _ = False
  length EmptyStack = 0
  length (Stack _ rest) = 1 + length rest
  elem _ EmptyStack = False
  elem e (Stack x rest) = (e == x) || elem e rest
  maximum EmptyStack = error "maximum: empty structure"
  maximum (Stack x rest) = foldl max x rest
  minimum EmptyStack = error "minimum: empty structure"
  minimum (Stack x rest) = foldl min x rest
  sum EmptyStack = 0
  sum (Stack x rest) = foldl (+) x rest
  product EmptyStack = 1
  product (Stack x rest) = foldl (*) x rest

-- Monoid-Instanz für Stack
-- {-# MINIMAL mempty #-}
instance Monoid (Stack a) where
  mempty = EmptyStack
  mappend = (<>)
  mconcat = foldr mappend mempty

-- Applicative-Instanz für Stack
-- {-# MINIMAL pure, (<*>) #-}
instance Applicative Stack where
  pure x = Stack x EmptyStack
  EmptyStack <*> _ = EmptyStack
  _ <*> EmptyStack = EmptyStack
  (<*>) (Stack f restF) (Stack x restX) = Stack (f x) (restF <*> Stack x
restX)
  (*>) EmptyStack _ = EmptyStack
  (*>) _ EmptyStack = EmptyStack
  (*>) (Stack _ restF) stackX = restF *> stackX
  (<*) EmptyStack _ = EmptyStack
  (<*) _ EmptyStack = EmptyStack
  (<*) stackX@(Stack x restX) stackY = stackX <* stackY
```

```haskell
-- Monad-Instanz für Stack
-- {-# MINIMAL (>>=) #-}
instance Monad Stack where
  EmptyStack >>= _ = EmptyStack
  (Stack x rest) >>= f = (f x) <> (rest >>= f)
  EmptyStack >> _ = EmptyStack
  stack >> EmptyStack = EmptyStack
  stack >> (Stack y rest) = rest
  return = pure

-- Hilfsfunktionen für Stack
-- ein element hinzufügen
push :: a -> Stack a -> Stack a
push x stack = Stack x stack

-- element entfernen
pop :: Stack a -> Stack a
pop EmptyStack = EmptyStack
pop (Stack _ rest) = rest

top :: Stack a -> a
top EmptyStack = error "Empty stack has no top element"
top (Stack x _) = x

-- Beispiel-Stacks
exampleStack :: Stack Int
exampleStack = push 1 (push 2 (push 3 EmptyStack))

sf = Stack (+1) (Stack (*2) EmptyStack) :: Stack (Int -> Int)
sx = Stack 1 (Stack 2 (Stack 3 EmptyStack)) :: Stack Int
```