```
                  Monad transformers, další monádové třídy
                  ----------------------------------------
class (Monad m) => MonadIO m where
    liftIO :: IO a -> m a
instance MonadIO IO where
    liftIO = id

class MonadTrans t where
    lift :: Monad m => m a -> t m a
```

♣ Identity

```
newtype Identity a = Identity { runIdentity :: a }
instance Monad Identity where
    return a = Identity a
    m >>= k  = k (runIdentity m)
```

♣ Reader

```
class Monad m => MonadReader r m | m -> r where
  ask :: m r
  local :: (r -> r) -> m a -> m a

newtype ReaderT r m a = ReaderT { runReaderT :: r -> m a }
instance (Monad m) => Monad (ReaderT r m) where
    return   = lift . return
    m >>= k  = ReaderT $ \ r -> do
        a <- runReaderT m r
        runReaderT (k a) r
    fail msg = lift (fail msg)

ask :: (Monad m) => ReaderT r m r
ask = ReaderT return

local :: (Monad m) => (r -> r) -> ReaderT r m a -> ReaderT r m a
local f m = ReaderT $ runReaderT m . f

type Reader r = ReaderT r Identity
instance (Monad m) => MonadReader r (ReaderT r m) where
    ask = ReaderT.ask
    local = ReaderT.local

runReader :: Reader r a -> r -> a
runReader m = runIdentity . runReaderT m
```

Případně pomocí funkčních závislostí

```
class Monad m => MonadReader m where
  type EnvType m
  ask   :: m (EnvType m)
  local :: (EnvType m -> EnvType m) -> m a -> m a

eval :: Expr -> Reader Variables Int
eval (Var x) = do vars <- ask; lookup x in vars

runReader (eval x) variables
```

♣ State

```
class Monad m => MonadState s m | m -> s where
    get :: m s
    put :: s -> m ()

newtype StateT s m a = StateT { runStateT :: s -> m (a,s) }
evalStateT m s = fst `liftM` runStateT m s

instance Monad m => Monad (StateT s m) where
    return a = StateT $ \s -> return (a, s)
    m >>= k  = StateT $ \s -> do
        (a, s') <- runStateT m s
        runStateT (k a) s'
    fail str = StateT $ \_ -> fail str
```

```
get :: (Monad m) => StateT s m s
get = StateT $ \s -> return (s, s)
put :: (Monad m) => s -> StateT s m ()
put s = StateT $ \_ -> return ((), s)

type State s = StateT s Identity
instance Monad m => MonadState s (StateT s m) where
    get = StateT.get
    put = StateT.put
runState :: State s a -> s -> (a, s)
runState m = runIdentity . runStateT m
evalState m s = fst (runState m s)

tick :: State Int Int
tick = do n <- get
          put (n+1)
          return n

data Tree a = Nil | Node a (Tree a) (Tree a) deriving (Show, Eq)
type Table a = [a]

numberTree :: Eq a => Tree a -> State (Table a) (Tree Int)
numberTree Nil = return Nil
numberTree (Node x t1 t2)
       =  do num <- numberNode x
             nt1 <- numberTree t1
             nt2 <- numberTree t2
             return (Node num nt1 nt2)
    where
    numberNode :: Eq a => a -> State (Table a) Int
    numberNode x
       = do table <- get
            (newTable, newPos) <- return (nNode x table)
            put newTable
            return newPos
    nNode::  (Eq a) => a -> Table a -> (Table a, Int)
    nNode x table
       = case (findIndexInList (== x) table) of
         Nothing -> (table ++ [x], length table)
         Just i  -> (table, i)
numTree :: (Eq a) => Tree a -> Tree Int
numTree t = evalState (numberTree t) []
```

♣ Writer: *MonadWriter*

```
class (Monoid w, Monad m) => MonadWriter w m | m -> w where
  tell   :: w -> m ()
  listen :: m a -> m (a, w)
  pass   :: m (a, w -> w) -> m a

class Monoid a where
  mempty  :: a
  mappend :: a -> a -> a
  mconcat :: [a] -> a
  mconcat = foldr mappend mempty
```

instances:

```
Monoid (), Monoid [a]
Monoid b => Monoid (a -> b)
Monoid a, Monoid b => Monoid (a -> b)
```

♣ RWS: *MonadRWS*

```
class (Monoid w, MonadReader r m, MonadWriter w m, MonadState s m) =>
      MonadRWS r w s m | m -> r, m -> w, m -> s
```

♣ Error: *MonadError*

```
class (Monad m) => MonadError e m | m -> e where
    throwError :: e -> m a
    catchError :: m a -> (e -> m a) -> m a

class Error a where
    noMsg  :: a                 ; noMsg    = strMsg ""
    strMsg :: String -> a       ; strMsg _ = noMsg
instance (Error e) => MonadError e (Either e)
```