```
                              Control.Monad
                              -------------
class Functor f where                          class Monad m where
  fmap :: (a -> b) -> f a -> f b                 (>>=)  :: m a -> (a -> m b) -> m b
                                                 return :: a -> m a
Funktor by měl splňovat                          fail   :: String -> m a
 ♣ fmap id  ==  id
 ♣ fmap (f . g)  ==  fmap f . fmap g             (>>)   :: m a -> m b -> m b
                                                 f >> g = f >>= \_ -> g

instance Functor [] (Array i) Maybe ((,) a) (Either a) ((->) r) IO STM (ST s) Id
instance Monad [] Maybe (Either e) ((->) r) IO STM (ST s)
instance Monad Maybe where                        instance Monad [] where
    (Just x) >>= k  =  k x                           m >>= k   =  concat (map k m)
    Nothing  >>= k  =  Nothing                       return x  =  [x]
    return          =  Just                          fail s    =  []
    fail s          =  Nothing

class Monad m => MonadPlus m where
    mplus :: m a -> m a -> m a
    mzero :: m a
instance MonadPlus [] Maybe STM
instance  MonadPlus Maybe  where               instance  MonadPlus []  where
    mzero                  = Nothing               mzero =  []
    Nothing `mplus` ys     =  ys                   mplus = (++)
    xs      `mplus` ys     =  xs

liftM :: Monad m => (a1 -> r) -> m a1 -> m r
liftM2 :: Monad m => (a1 -> a2 -> r) -> m a1 -> m a2 -> m r
liftM3 :: Monad m => (a1 -> a2 -> a3 -> r) -> m a1 -> m a2 -> m a3 -> m r
when :: Monad m => Bool -> m () -> m ()
unless :: Monad m => Bool -> m () -> m ()
guard :: MonadPlus m => Bool -> m ()

mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM_ :: Monad m => (a -> m b) -> [a] -> m ()
forM :: Monad m => [a] -> (a -> m b) -> m [b]
forM_ :: Monad m => [a] -> (a -> m b) -> m ()
sequence :: Monad m => [m a] -> m [a]
sequence_ :: Monad m => [m a] -> m ()
(=<<) :: Monad m => (a -> m b) -> m a -> m b
(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> a -> m c
(<=<) :: Monad m => (b -> m c) -> (a -> m b) -> a -> m c
forever :: Monad m => m a -> m b
void :: Functor f => f a -> f ()

join :: Monad m => m (m a) -> m a
msum :: MonadPlus m => [m a] -> m a
mfilter :: MonadPlus m => (a -> Bool) -> m a -> m a
filterM :: Monad m => (a -> m Bool) -> [a] -> m [a]
mapAndUnzipM :: Monad m => (a -> m (b, c)) -> [a] -> m ([b], [c])
zipWithM :: Monad m => (a -> b -> m c) -> [a] -> [b] -> m [c]
zipWithM_ :: Monad m => (a -> b -> m c) -> [a] -> [b] -> m ()
foldM :: Monad m => (a -> b -> m a) -> a -> [b] -> m a
foldM_ :: Monad m => (a -> b -> m a) -> a -> [b] -> m ()
replicateM :: Monad m => Int -> m a -> m [a]
replicateM_ :: Monad m => Int -> m a -> m ()

                              PreludeIO
                              ---------
data IO a = ...
type FilePath = String

putChar                :: Char -> IO ()
putStr, putStrLn       :: String -> IO ()
print                  :: Show a = > a -> IO ()
getChar                :: IO Char
getLine, getContents   :: IO String
interact               :: (String -> String) -> IO ()
```

```
readFile            :: FilePath -> IO String
writeFile           :: FilePath -> String -> IO ()
appendFile          :: FilePath -> String -> IO ()
readIO              :: Read a = > String -> IO a
readIO s =  case [x | (x,t) <- reads s, ("","") <- lex t] of
              [x] -> return x
              []  -> ioError (userError "Prelude.readIO: no parse")
              _   -> ioError (userError "Prelude.readIO: ambiguous parse")
readLn :: Read a => IO a
readLn =  do l <- getLine
             r <- readIO l
             return r
data IOError          -- IO exceptions
catch :: IO a -> (IOError -> IO a) -> IO a

                              Module System.IO
                              ----------------
data Handle = ... -- implementation-dependent
data HandlePosn = ... -- implementation-dependent
data IOMode      =  ReadMode | WriteMode | AppendMode | ReadWriteMode
data BufferMode  =  NoBuffering | LineBuffering |  BlockBuffering (Maybe Int)
data SeekMode    =  AbsoluteSeek | RelativeSeek | SeekFromEnd
stdin, stdout, stderr :: Handle

openFile           :: FilePath -> IOMode -> IO Handle
hClose             :: Handle -> IO ()

hFileSize          :: Handle -> IO Integer
hIsEOF             :: Handle -> IO Bool
isEOF              =  hIsEOF stdin

hSetBuffering      :: Handle  -> BufferMode -> IO ()
hGetBuffering      :: Handle  -> IO BufferMode
hFlush             :: Handle -> IO ()
hGetPosn           :: Handle -> IO HandlePosn
hSetPosn           :: HandlePosn -> IO ()
hSeek              :: Handle -> SeekMode -> Integer -> IO ()

hWaitForInput      :: Handle -> Int -> IO Bool     hReady h = hWaitForInput h 0
hGetChar           :: Handle -> IO Char
hGetLine           :: Handle -> IO String
hLookAhead         :: Handle -> IO Char
hGetContents       :: Handle -> IO String
hPutChar           :: Handle -> Char -> IO ()
hPutStr, hPutStrLn :: Handle -> String -> IO ()
hPrint             :: Show a = > Handle -> a -> IO ()
hIsOpen, hIsClosed, hIsReadable, hIsWritable, hIsSeekable :: Handle -> IO Bool

-- GHC extensions
hSetBinaryMode        :: Handle -> Bool -> IO ()
hSetEncoding          :: Handle -> TextEncoding -> IO ()
localeEncoding        :: TextEncoding
latin1, utf8, utf8_bom, utf16{,le,be}, utf32{,le,be} :: TextEncoding
mkTextEncoding        :: String -> IO TextEncoding

hSetNewlineMode       :: Handle -> NewlineMode -> IO ()
data NewlineMode      = NewlineMode { inputNL :: Newline, outputNL :: Newline }
data Newline          = LF | CRLF
nativeNewline         :: Newline
noNewlineTranslation  = NewlineMode { inputNL  = LF, outputNL = LF }
universalNewlineMode  = NewlineMode { inputNL  = CRLF, outputNL = nativeNewline}
nativeNewlineMode     = NewlineMode { inputNL  = nativeNewline
                                    , outputNL = nativeNewline }

                         Modul System.Environment
                         ------------------------
getArgs :: IO [String]                 withArgs :: [String] -> IO a -> IOa
getProgName :: IO String               withProgName :: String -> IO a -> IO a
getEnvironment :: IO [(String, String)]
```

Obecná pole Data.Array.IArray

class *IArray* a e **where** ...
funkce array, listArray, accumArray, !, bounds, //, amap, ixmap
instance *IArray Array* e
v *Data.Array.Unboxed* je **instance** *IArray UArray* boxed
boxed může být *Bool Double Float Char Int Int*{8,16,32,64} *Word Word*{8,16,32,64}
♣ Rozdílová pole v *Data.Array.Diff*, přístup a modifikace nejnovější verze je v O(1)
instance *IArray DiffArray* e
instance *IArray DiffUArray* boxed

Rychlá pole v monádě

```
class (Monad m) => MArray a e m where
    getBounds :: Ix i => a i e -> m (i,i)
    newArray  :: Ix i => (i,i) -> e -> m (a i e)
    newArray_ :: Ix i => (i,i) -> m (a i e)
newListArray :: (MArray a e m, Ix i) => (i, i) -> [e] -> m (a i e)
readArray :: (MArray a e m, Ix i) => a i e -> i -> m e
writeArray :: (MArray a e m, Ix i) => a i e -> i -> e -> m ()
getelems, getAssocs, mapArray, mapIndices
freeze,unsafeFreeze :: (Ix i, MArray a e m, IArray b e) => a i e -> m (b i e)
thaw, unsafeThaw :: (Ix i, IArray a e, MArray b e m) => a i e -> m (b i e)

permute :: (MArray a e m) => [Int] -> a Int e -> m (a Int e)
permute p arr = let pa = listArray (0, length p - 1) p
                    pi = (pa !)
                in do bnds <- getBounds arr
                      mapIndices bnds pi arr

permute p a = do res <- getBounds a >>= newArray_
                 permute' 0 p res
 where permute' _ []    res = return res
       permute' i (j:p) res = do readArray a j >>= writeArray res i
                                 permute' (i+1) p res
```

ST monáda

Existuje monáda *Control.Monad.ST* s typem **data** *ST* s a. Také existuje funkce
runST :: (forall s. ST s a) -> a, která provede výpočet. Všimněte si toho forall...
stToIO :: ST RealWorld a -> IO a

```
module Data.STRef
newSTRef :: a -> ST s (STRef s a)
readSTRef :: STRef s a -> ST s a
writeSTRef :: STRef s a -> a -> ST s ()
modifySTRef :: STRef s a -> (a -> a) -> ST s ()   readSTRef >> writeSTRef
swap a b = do a'<-readSTRef a; b'<-readSTRef b; writeSTRef a b'; writeSTRef b a'
```

A v modulu *Data.Array.ST* jsou pole uvnitř *ST* monády:

```
data STArray s i e;                       data STUArray s i e
instance MArray (STArray s) e (ST s);     instance MArray (STUArray s) boxed (ST s)
count :: [Int] -> Array Int Int    Řekněme, že čísla jsou 0..9
count n = runSTArray $ do a <- newArray (0,9) 0
                          mapM_ (\i->readArray a i >>= writeArray a i . (+1)) n
                          return a
array bnds assocs = runSTArray $ do a <- newArray_ bnds
                                    mapM_ (uncurry $ writeArray a) assocs
                                    return a

runSTArray :: Ix i=>(forall s. ST s (STArray s i e)) -> Array i e
runSTUArray :: Ix i => (forall s. ST s (STUArray s i e)) -> UArray i e

freeze :: (Ix i, MArray a e m, IArray b e) => a i e -> m (b i e)
thaw :: (Ix i, IArray a e, MArray b e m) => a i e -> m (b i e)
unsafeFreeze :: (Ix i, MArray a e m, IArray b e) => a i e -> m (b i e)
unsafeThaw :: (Ix i, IArray a e, MArray b e m) => a i e -> m (b i e)
```

Tyto funkce fungují efektivně pro {*ST*, *IO*}*Array*<->*Array*, {*ST*, *IO*}*UArray*<->*UArray*

```
runSTArray starr = runST (starr >>= unsafeFreeze)
```