Odvozování instancí

```
class T a where...
instance Eq a  => T a          Obě najednou
instance Num a => T a          jsou nekorektní
```

Užitečné monadické funkce

```
sequence :: Monad m => [m a] -> m [a]
sequence = foldr mcons (return []) where mcons p q = p>>=\x->q>>=\y->return(x:y)
sequence_ :: Monad m => [m a] -> m ()
sequence_ = foldr (>>) (return ())
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f as = sequence (map f as)
mapM_:: Monad m => (a -> m b) -> [a] -> m ()
mapM_ f as = sequence_ (map f as)

readLines n = sequence $ replicate n getLine
```

Další příklad monády – parser

```
newtype Parser a = Parser {parse::String->[(a,String)]}

instance Monad Parser where
  return a = Parser (\s -> [(a,s)])
  fail _ = Parser (\s -> [])
  p >>= f = Parser (\s->concat [parse (f a) s' | (a,s')<-parse p s])

char = Parser (\s -> case s of []     -> []
                               (c:cs) -> [(c,cs)])
trichar = do a<-char; b<-char; c<-char; return (a,b,c)
```

Někdy chceme spojovat persery paralelně a ne sériově. K tomu se dá použít třída `MonadPlus`. Ta zavádí funkce

```
class Monad m=>MonadPlus m where
  mplus::m a->m a->m a
  mzero::m a
```

Interpretace `mplus` dle monády, například
1) pokud první větev selže, zkus druhou (`Maybe`)
2) vyzkoušej obě větve (`List`)

Funkce `mzero` musí vracet neutrální prvek pro `mplus` tak, aby platilo
♣ `mzero 'mplus' m  ==  m 'mplus' mzero  ==  m`
Někdy je požadavků ještě více, hlavně `mzero >>= f  ==  v >> mzero  ==  mzero`.

```
instance MonadPlus Parser where
  mzero = Parser (\s->[])
  a 'mplus' b = Parser (\s->parse a s ++ parse b s)

sat::(Char->Bool)->Parser Char
sat p = do {c<-char; if p c then return c else mzero}

space=sat isSpace; digit=sat isDigit; letter=sat isAlpha

many p = do {a<-p; as<-many p; return (a:as)} 'mplus' return []
```

Pak `parse (many digit) "09a"` vrátí `[("09","a"),("0","9a"),("","09a")]`.

```
spaces=many space; digits=many digit; word=many letter

addop::Parser (Int->Int->Int)
addop = do {sat (=='+'); return (+)} 'mplus' do {sat (=='-'); return (-)}
number::Parser Int
number = digits >>= return . read
aplusb = do {a<-number; op<-addop; b<-number; return $ a 'op' b}
```

Pak `head $ parse aplusb "1+34"` vrátí `(35,"")`.

`parse word "a"` vrátí `[("a",""),("","a")]`. Co vrátí `null $ parse word ("a"++undefined)`?

Parser nad libovolnou monádou

Aktuální parser vrací vždy všechny výsledky. Někdy bychom ale chtěli vracet jenom jeden nebo hlásit chyby. Potřebovali bychom měnit strategii, jak zacházet s naparsovanými výsledky. Strategie ... to zní jako monáda.

```
newtype Parser m a = Parser {parse::String->m (a,String)}

instance Monad m => Monad (Parser m) where
  return a = Parser (\s->return (a,s))
  fail a = Parser (\_->fail a)
  p >>= f = Parser (\s->parse p s >>= \(a,s')->parse (f a) s')

instance MonadPlus m => MonadPlus (Parser m) where
  mzero = Parser (\_->mzero)
  a `mplus` b = Parser (\s->parse a s `mplus` parse b s)

type BacktrackingParser = Parser []
type MaybeParser = Parser Maybe
type ErrorReportingParser = Parser (Either String)
```

Ale `Either String` není `MonadPlus`

Modul Monad

```
class Monad m=>MonadPlus m where mplus::m a->m a->m a
                                 mzero::m a
guard :: MonadPlus m => Bool -> m ()
msum :: MonadPlus m => [m a] -> m a
join              :: Monad m => m (m a) -> m a
when, unless      :: Monad m => Bool -> m () -> m ()
ap                :: Monad m => m (a -> b) -> m a -> m b
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
filterM           :: Monad m => (a -> m Bool) -> [a] -> m [a]
liftM             :: Monad m => (a -> b) -> (m a -> m b)
liftM2            :: Monad m => (a -> b -> c) -> (m a -> m b -> m c)
liftM3, liftM4, liftM5 :: Monad m => ...
```

Pole a odkazy

Existuje monáda `Control.Monad.ST` s typem `data ST s a`. Také existuje funkce
`runST :: (forall s . ST s a) -> a`, která provede výpočet. Toho `forall` si zatím nevšímejte:)
V modulu `Data.STRef` jsou reference uvnitř `ST` monády:

```
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 ()
```

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

```
runSTArray::Ix i=>(forall s . ST s (STArray s i e)) -> Array i e
newArray ::...=>(i, i)->e->m (a i e) newListArray::...=> (i, i)->[e]->m (a i e)
readArray::...=>a i e->i->m e        writeArray  ::...=> a i e ->i->e-> m ()
getBounds::...=>a i e->m (i, i); getElems::...->m [e]; getAssocs::...->m [(i,e)]

swap::STRef s a->STRef s a->ST s ()
swap a b = do x<-readSTRef a; y<-readSTRef b; writeSTRef a y; writeSTRef b x

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
```

Protože ve skutečnosti je `IO` monáda jenom `ST RealWord`, existují odkazy a pole i v monádě `IO`.
Odkazy jsou v modulu `Data.IORef` (nahraďte ve funkcích `ST` za `IO`), pole jsou v `Data.Array.IO`.
S polemi se zachází úplně stejně, jenom nejdou "vyndat" z monády. Ale `freeze` udělá kopii v lineárním čase.

Domácí úkoly

Budou na webu...