Další používané třídy

```
module Data.Ord
class Ord where ...
data Ordering = LT | EQ | GT
comparing :: Ord a => (b -> a) -> b -> b -> Ordering
comparing p x y = compare (p x) (p y)
```

Užitečné pro funkce z `Data.List`:

```
  sortBy :: (a -> a -> Ordering) -> [a] -> [a]
  insertBy :: (a -> a -> Ordering) -> a -> [a] -> [a]
  maximumBy :: (a -> a -> Ordering) -> [a] -> a
  minimumBy :: (a -> a -> Ordering) -> [a] -> a
```

```
module Data.Function
id, const, (.), flip, ($)
fix :: (a -> a) -> a                          fix f = let x = f x in x
on :: (b -> b -> c) -> (a -> b) -> a -> a -> c    infixl 0 `on`
(.*.) `on` f = \x y -> f x .*. f y
comparing p = compare `on` p
equal p = (==) `on` p
```

Užitečné i pro daší funkce z `Data.List`:

```
  nubBy :: (a -> a -> Bool) -> [a] -> [a]
  deleteBy :: (a -> a -> Bool) -> a -> [a] -> [a]
  unionBy :: (a -> a -> Bool) -> [a] -> [a] -> [a]
  intersectBy :: (a -> a -> Bool) -> [a] -> [a] -> [a]
  groupBy :: (a -> a -> Bool) -> [a] -> [[a]]
```

```
class Monoid a where                    instances:
  mempty  :: a                            Monoid (), Monoid [a]
  mappend :: a -> a -> a                  Monoid b => Monoid (a -> b)
  mconcat :: [a] -> a                     Monoid a, Monoid b => Monoid (a, b)
  mconcat = foldr mappend mempty

newtype Dual a = Dual { getDual :: a } deriving (Eq, Ord, Read, Show, Bounded)
instance Monoid a => Monoid (Dual a) where
  mempty = Dual mempty
  Dual x `mappend` Dual y = Dual (y `mappend` x)

newtype Endo a = Endo { appEndo :: a -> a }
instance Monoid (Endo a) where
  mempty = Endo id
  Endo f `mappend` Endo g = Endo (f . g)
```

monoidy `All`, `Any`, `Sum a`, `Prod a`, `First a`, `Last a`

```
class Foldable t where
  fold :: Monoid m => t m -> m
  foldMap :: Monoid m => (a -> m) -> t a -> m
  foldr :: (a->b->b) -> b -> t a -> b        foldl :: (a->b->a) -> a -> t b -> a
  foldr1 :: (a -> a -> a) -> t a -> a        foldl1 :: (a -> a -> a) -> t a -> a

foldMap f = foldr (mappend . f) mempty
foldr f z t = appEndo (foldMap (Endo . f) t) z
foldl f z t = appEndo (getDual (foldMap (Dual . Endo . flip f) t)) z
foldr' :: Foldable t => (a -> b -> b) -> b -> t a -> b
foldr' f z0 xs = foldl f' id xs z0
  where f' k x z = k $! f x z

module Data.Bits
class Num a => Bits a where
  (.&.) :: a -> a -> a      (.|.) :: a -> a -> a          xor :: a -> a -> a
  bit :: Int -> a           setBit :: a -> Int -> a       clearBit::a -> Int -> a
  complement :: a -> a      complementBit :: a -> Int -> a
  shift :: a -> Int -> a    shiftL :: a -> Int -> a       shiftR :: a -> Int -> a
  rotate:: a -> Int -> a    rotateL:: a -> Int -> a       rotateR:: a -> Int -> a
```

Rychlé Stringy

String má velký overhead −− je to (líný) seznam odkazů na 32bitové chary
Existuje proto typ *Data.ByteString*, nad kterým jsou definovány klasické operace pro *Word8*
`empty singleton pack unpack cons snoc append head uncons last tail init null length map reverse intersperse intercalate transpose foldl foldl1 foldr foldr1 concat concatMap any all maximum minimum scanl scanl1 scanr scanr1 mapAccumL mapAccumR mapIndexed replicate unfoldr unfoldrN take drop splitAt takeWhile dropWhile span spanEnd break breakEnd group groupBy inits tails split splitWith isPrefixOf isSuffixOf isInfixOf isSubstringOf findSubstring findSubstrings elem notElem find filter partition index elemIndex elemIndices elemIndexEnd findIndex findIndices count zip zipWith unzip sort copy packCString packCStringLen useAsCString useAsCStringLen getLine getContents putStr putStrLn interact readFile writeFile appendFile hGetLine hGetContents hGet hGetNonBlocking hPut hPutStr hPutStrLn`
Typ *Data.ByteString.Char8* má metody s *Char*em místo *Word8*, z *Char*ů se používá jenom 8 bitů.
Oba tyto typy mají celý string v jednom kusu paměti −− existují i líné varianty
Data.ByteString.Lazy a *Data.ByteString.Lazy.Char8*, které pracují s chunky po 64k.

Implementace *Data.ByteString* je viditelná v *Data.ByteString.Internal*:
```
data ByteString = PS !(ForeignPtr Word8) !Int !Int
fromForeignPtr :: ForeignPtr Word8 -> Int -> Int -> ByteString
toForeignPtr :: ByteString -> (ForeignPtr Word8, Int, Int)
```

♣ Pomocí rozšíření *-XOverloadedStrings* lze používat přetížené řetězcové literály:
```
class IsString a where fromString :: String -> a
```

Existuje instance
```
instance IsString [Char] where fromString = id
```
a každý *ByteString* definuje svojí vlastní.

Applicative a Alternative

```
class Functor f where
  fmap, (<$>) :: (a -> b) -> f a -> f b
  (<$) :: a -> f b -> f a
  (<$) = fmap . const

class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
  (*>) :: f a -> f b -> f b
  (<*) :: f a -> f b -> f a
instance Applicative [], IO, Maybe, STM, ZipList, ((->) a), (Either e)
instance Monoid m => Applicative ((,) m), (Const m)

class Applicative f => Alternative f where
  empty :: f a
  (<|>) :: f a -> f a -> f a

liftA :: Applicative f => (a -> b) -> f a -> f b
liftA f a = f <$> a
liftA2 :: Applicative f => (a -> b -> c) -> f a -> f b -> f c
liftA2 f a b = f <$> a <*> b
liftA3 :: Applicative f => (a -> b -> c -> d) -> f a -> f b -> f c -> f d
liftA3 f a b c = f <$> a <*> b <*> c
optional :: Alternative f => f a -> f (Maybe a)
optional v = Just <$> v <|> pure Nothing

sequence :: [IO a] -> IO [a]
sequence [] = return []
sequence (c:cs) = do x <- c
                     xs <- sequence cs
                     return (x : xs)

sequence [] = pure []
sequence (c:cs) =
  (:) <$> c <*> sequence cs
  liftA2 (:) c (sequence cs)

eval :: Exp v -> Env v -> Int
eval (Var x) = fetch x
eval (Val i) = return i
eval (Add p q) = do pv <- eval p
                    qv <- eval q
                    return (pv + qv)

eval (Var x) = fetch x
eval (Val i) = pure i
eval (Add p q) = (+) <$> eval p <*> eval q
               = liftA2 (+) (eval p) (eval q)

newtype Const a b = Const { getConst :: a}
instance Monoid m => Applicative (Const m) where
  pure _ = Const mempty
  Const f <*> Const v = Const (f `mappend` v)

miffy :: Monad m => m Bool -> m a -> m a -> m a
miffy mb mt me = do b <- mb
                    if b then mt else me
iffy :: Applicative f => f bool -> f a -> f a -> f a
iffy fb ft fe = cond <$> ft <*> fe
  where cond b t e = if b then t else e
```

Text.Parsec

```
data Parsec stream userstate result = ...
data ParsecT stream userstate monad result = ...

runParser, runP :: Stream s Identity t =>
                   Parsec s u a -> u -> SourceName -> s -> Either ParseError a
```

Parsec a *ParsecT* jsou instancemi *Monad*, *MonadPlus*, *Applicative*, *Alternative*
char, upper, lower, digit, hexDigit, space, spaces, tab, newline, oneOf, noneOf, anyChar, string

many, many1, manyTill, optional, sepBy, sepBy1, endBy, endBy1

```
parse_query :: Parsec s u [(String, Maybe String)]
parse_query = pair `sepBy` char '&'
  where pair = liftA2 (,) (many1 safe)
                          (optional (char '=' *> many safe))
        safe = oneOf urlBaseChars
           <|> char '%' *> liftA2 diddle hexDigit hexDigit
           <|> ' ' <$ char '+'
        diddle a b = toEnum . fst . head . readHex $ [a, b]
        urlBaseChars = ['a'..'z']++['A'..'Z']++['0'..'9']++"$-_.!*'(),"

parse_headers :: Parsec s u [(String, String)]
parse_headers = header `manyTill` crlf
  where header = (,) <$> fieldName <*> (char ':' *> spaces *> contents)
        fieldName = (:) <$> letter <*> many fieldChar
        fieldChar = letter <|> digit <|> oneOf "-_"
        contents = (++) <$> (many notEOL <* crlf)
                        <*> (continuation <|> pure [])
        continuation = (:) <$> (' ' <$ many1 (oneOf " \t")) <*> contents

        crlf = optional (char '\r') *> newline
        notEOL = noneOf "\r\n"
```

Skladani applicative

```
newtype Comp f g a = Comp { getComp :: f (g a) }
instance (Applicative f, Applicative g) => Applicative (Comp f g) where
  pure x = Comp (pure (pure x))
  Comp fs <*> Comp xs = Comp ((<*>) <$> fs <*> xs)
```

Skladani monad

```
getPassword :: IO (Maybe String)
getPassword =
  do s <- getLine
     if isValid s then return $ Just s
                  else return Nothing
```

```
askPassword :: IO ()
askPassword =
  do putStrLn "Insert new password:"
     maybe_value <- getPassword
     if isJust maybe_value
         then do putStrLn "Storing..."
         -- other stuff, incl. 'else'
```

```
newtype (Monad m) => MaybeT m a = MaybeT { runMaybeT :: m (Maybe a) }
instance Monad m => Monad (MaybeT m) where
  return  = MaybeT . return . Just
  x >>= f = MaybeT $ do maybe_value <- runMaybeT x
                        case maybe_value of Nothing    -> return Nothing
                                            Just value -> runMaybeT $ f value
instance Monad m => MonadPlus (MaybeT m) where...
instance MonadTrans MaybeT where lift = MaybeT . (liftM Just)
```

```
getValidPassword :: MaybeT IO String
getValidPassword =
  do s <- lift getLine
     guard (isValid s)
     return s
```

```
askPassword :: MaybeT IO ()
askPassword =
  do lift $ putStrLn "Insert password:"
     value <- getValidPassword
  -- value <- msum $ repeat getValidPassword
     lift $ putStrLn "Storing..."
```

```
Error    ErrorT    Either e a      m (Either e a)
State    StateT    s -> (a,s)      s -> m (a,s)
Reader   ReaderT   r -> a          r -> m a
Writer   WriterT   (a,w)           m (a,w)
Cont     ContT     (a -> r) -> r   (a -> m r) -> m r
```