Standardní monády

```
instance  Monad Maybe  where
    (Just x) >>= k   =  k x
    Nothing  >>= k   =  Nothing
    return           =  Just
    fail s           =  Nothing
instance  Monad []  where
    m >>= k          = concat (map k m)
    return x         = [x]
    fail s           = []


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

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 (-)}
mulop :: Parser (Int->Int->Int)
mulop = do {sat (=='*'); return (*)} `mplus` do {sat (=='/'); return (/)}
```

```
chain :: Parser Int -> Parser (Int -> Int -> Int) -> Parser Int
chain p op = do n <- p
                rest n
 where rest n = do o <- op
                   m <- p
                   rest (n `o` m)
                `mplus` return n
expr, term, factor, number :: Parser Int
expr = term `chain` addop
term = factor `chain` mulop
factor = digit `mplus` do { sat (=='('); n <- expr; sat (==')'); return n }
number = digits >>= return . read

                         Parser nad libovolnou monádou
                         -----------------------------
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
```