Modul Eval3

Přidáme ošetření chyb. Dá se k tomu 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 (náš případ, *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`.

```
data Result = Chyba (Maybe String) | Hodnota x
instance MonadPlus Result where
  Hodnota x `mplus` _     = Hodnota x
  Chyba _   `mplus` druha = druha
  mzero = Chyba Nothing

eval :: (Monad m,MonadPlus m) => Expr -> m Integer
{-...-}
eval (Try e1 e2) = eval e1 `mplus` eval e2
```

Modul Eval4

Přidáme ohodnocení proměnných

```
data Vypocet x = V {unV :: Values -> Result x}
instance Monad Vypocet where
  V v >>= f = V (\o -> v o >>= \x -> unV (f x) o)
  return x = V (\_ -> return x)
  fail ch  = V (\_ -> fail ch)

get :: Vypocet Values
get = V (\vals -> return vals)

runVypocet :: Vypocet x -> Values -> Result x
runVypocet (V vyp) vals = vyp vals

eval::Expr->Vypocet Integer
{-...-}
eval (Var s) = do ohodnoceni <- get
                  case lookup s ohodnoceni of
                    Just x -> return x
                    Nothing -> fail ("Neznama promenna " ++ s)
```

Modul Eval5

Přidáme změnu proměnných

```
data Vypocet s x = V {unV :: Values -> Result (x, Values)}
instance Monad (Vypocet s) where
  V v >>= f = V (\o1 -> v o1 >>= \(b, o2) -> unV (f b) o2)
  return x = V (\o -> return (x, o)))
  fail ch =  V (\_ -> fail ch)

get :: Vypocet o o
get = V (\o -> return (o, o))

put :: Vypocet o ()
put o = V (\_ -> return ((), o))

eval :: => Expr -> Vypocet Integer
{-...-}
eval (Assign s e) = do r <- eval e
                       ohodnoceni <- get
                       put (update ohodnoceni s r)
                       return r

update :: Values -> Variable -> Integer -> Values
update []         s v             = [(s,v)]
update ((s1,v1):t) s v | s == s1   = (s,v) : t
                       | otherwise = (s1, v1) : update t s v
```

Control.Monad

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Funktor by měl splňovat

- ♣ `fmap id == id`
- ♣ `fmap (f . g) == fmap f . fmap g`

```
class Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b
  return :: a -> m a
  fail   :: String -> m a

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

instance  MonadPlus []  where
    mzero =  []
    mplus = (++)

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 ()
```