Modul Expr

```
module Expr where
data Expr = Plus Expr Expr | Minus Expr Expr |
            Mul Expr Expr |   Div Expr Expr |
            Mod Expr Expr |   Num Integer   |
            Var Variable  |                    {-pro Eval3.hs-}
            Assign Variable Expr |             {-pro Eval4.hs-}
            Output Expr   |                    {-pro Eval5.hs-}
            Try Expr Expr                      {-pro Eval6.hs-}
          deriving (Show)

type Variable = String
type Values = [(Variable, Integer)]
```

Modul Eval1

```
import Expr
-- vyhodnocení výrazu
eval::Expr->Integer
eval (Plus e1 e2) = eval e1 + eval e2
eval (Minus e1 e2) = eval e1 - eval e2
eval (Mul e1 e2) = eval e1 * eval e2
eval (Div e1 e2) = eval e1 `div` eval e2
eval (Mod e1 e2) = eval e1 `mod` eval e2
eval (Num n) = n
```

Jak přidat ošetřování chyb (dělení nulou) a ohodnocení proměnnýc a nezbláznit se z toho?

```
data Result x = Chyba String | Hodnota x deriving (Show)

bind :: Result a -> (a->Result b) -> Result b
bind (Chyba s) _ = Chyba s
bind (Hodnota a) f = f a
ret :: x -> Result x
ret x = Hodnota x
err :: String -> Result x
err ch = Chyba ch

eval1::Expr->Result Integer
eval1 (Plus e1 e2) = eval1 e1 `bind` \r1 ->
                     eval1 e2 `bind` \r2 ->
                     ret (r1 + r2)
eval1 (Minus e1 e2) = eval1 e1 `bind` \r1 ->
                      eval1 e2 `bind` \r2 ->
                      ret (r1 - r2)
eval1 (Mul e1 e2) =  eval1 e1 `bind` \r1 ->
                     eval1 e2 `bind` \r2 ->
                     ret (r1 * r2)
eval1 (Div e1 e2) =  eval1 e1 `bind` \r1 ->
                     eval1 e2 `bind` \r2 ->
                     if r2 == 0 then err "Deleni nulou" else ret (r1 `div` r2)
eval1 (Mod e1 e2) =  eval1 e1 `bind` \r1 ->
                     eval1 e2 `bind` \r2 ->
                     if r2 == 0 then err "Deleni nulou" else ret (r1 `mod` r2)
eval1 (Num n) = ret n
```

Modul Eval2

```
import Expr
```

Haskell má speciální třídu pro monády

```
class Monad m where              {-m::*->*-}
  (>>=) ::m a->(a->m b)->m b     {-bind-}
  return::a->m a                 {-ret-}
  fail  ::String->m a            {-err-}
  (>>)  ::m a->m b->m b
  f >> g = f >>= \_ -> g
```

Aby něco bylo monáda, musí platit tři axiomy

- `(return x) >>= f  == f x`
- `m >>= return  == m`
- `(m >>= f) >>= g  == m >>= (\x -> f x >>= g)`

Haskell má navíc speciální notaci pro monády
- ♣ **do** {x} je ekvivalentní x
- ♣ **do** {x;y} je ekvivalentní x >> **do** y
- ♣ **do** {v <- x;y} je ekvivalentní x >>= \v**->** **do** y
- ♣ **do** {**let** x;y} je ekvivalentní **let** x **in** **do** y

```
data Result x = Chyba String | Hodnota x deriving (Show)
instance Monad Result where
  Chyba s >>=  _ = Chyba s
  Hodnota a >>= f = f a
  return x = Hodnota x
  fail s = Chyba s

eval::Monad m => Expr->m Integer
eval (Plus e1 e2) = do r1 <- eval e1
                       r2 <- eval e2
                       return (r1 + r2)
eval (Minus e1 e2) = do r1 <- eval e1
                        r2 <- eval e2
                        return (r1 - r2)
eval (Mul e1 e2) = do r1 <- eval e1
                      r2 <- eval e2
                      return (r1 * r2)
eval (Div e1 e2) = do r1 <- eval e1
                      r2 <- eval e2
                      if r2==0 then fail "Deleni nulou" else return(r1 `div` r2)
eval (Mod e1 e2) = do r1 <- eval e1
                      r2 <- eval e2
                      if r2==0 then fail "Deleni nulou" else return(r1 `mod` r2)
eval (Num n) = return n
```

Modul Eval3

```
import Expr
```

Přidáme ohodnocení proměnných

```
data Result x = Chyba String | Hodnota x deriving (Show)
instance Monad Result where
  Chyba s >>=  _ = Chyba s
  Hodnota a >>= f = f a
  return x = Hodnota x
  fail s = Chyba s
```

Budeme obecní. Zavedeme typ *Vypocet*, který provádí výpočet v monádě m pomocí stavu s.

```
data Vypocet m s x = V {unV::s->m x}
instance Monad m => Monad (Vypocet m s) where
  V vyp1 >>= f = V (\s->vyp1 s >>= \x->unV (f x) s)
  return x = V (\_ -> return x)
  fail ch  = V (\_ -> fail ch)

class               MonadRead m s | m -> s    where get :: m s
instance Monad m => MonadRead (Vypocet m s) s where get = V (\s -> return s)

runVypocet :: Vypocet m s x -> s -> m x
runVypocet (V f) state = f state

eval::(Monad m, MonadRead m Values) => Expr->m Integer
eval (Plus e1 e2) = do  r1 <- eval e1;  r2 <- eval e2;   return (r1 + r2)
eval (Minus e1 e2) = do r1 <- eval e1;  r2 <- eval e2;   return (r1 - r2)
eval (Mul e1 e2) = do   r1 <- eval e1;  r2 <- eval e2;   return (r1 * r2)
eval (Div e1 e2) = do   r1 <- eval e1;  r2 <- eval e2;
                        if r2==0 then fail"Deleni nulou"else return(r1 `div` r2)
eval (Mod e1 e2) = do   r1 <- eval e1;  r2 <- eval e2
                        if r2==0 then fail"Deleni nulou"else return(r1 `mod` r2)
eval (Num n) = return n
eval (Var s) = do ohodnoceni <- get
                  case lookup s ohodnoceni of
                    Just x -> return x
                    Nothing -> fail ("Neznama promenna " ++ s)
```

Modul Eval4

```
import Expr
```

Přidáme změnu proměnných

```
data Result x = Chyba String | Hodnota x deriving (Show)
instance Monad Result where
  Chyba s >>=  _ = Chyba s
  Hodnota a >>= f = f a
  return x = Hodnota x
  fail s = Chyba s

data Vypocet s x = V {unV::s->(s, Result x)}
instance Monad (Vypocet s) where
  V vyp1 >>= f = V (\s->let (s', val) = vyp1 s in case val of
                                                    Chyba ch -> (s', Chyba ch)
                                                    Hodnota x->unV (f x) s')
  return x = V (\s -> (s,return x))
  fail ch =  V (\s -> (s,fail ch))

class    MonadRead m s | m -> s   where get :: m s
instance MonadRead (Vypocet s) s  where get = V (\s -> (s,return s))

class    MonadState m s | m -> s  where put :: s -> m ()
instance MonadState (Vypocet s) s where put s = V (\_ -> (s,return ()))

runVypocet :: Vypocet s x -> s -> (s, Result x)
runVypocet (V f) state = f state

eval::(Monad m, MonadRead m Values, MonadState m Values) => Expr->m Integer
eval (Plus e1 e2) = do  r1 <- eval e1;  r2 <- eval e2;   return (r1 + r2)
eval (Minus e1 e2) = do r1 <- eval e1;  r2 <- eval e2;   return (r1 - r2)
eval (Mul e1 e2) = do   r1 <- eval e1;  r2 <- eval e2;   return (r1 * r2)
eval (Div e1 e2) = do   r1 <- eval e1;  r2 <- eval e2;
                        if r2==0 then fail"Deleni nulou"else return(r1 `div` r2)
eval (Mod e1 e2) = do   r1 <- eval e1;  r2 <- eval e2
                        if r2==0 then fail"Deleni nulou"else return(r1 `mod` r2)
eval (Num n) = return n
eval (Var s) = do ohodnoceni <- get
                  case lookup s ohodnoceni of
                    Just x -> return x
                    Nothing -> fail ("Neznama promenna " ++ s)
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
```

Modul Eval5

```
import Expr
```

Přidáme logování postupu

```
data Result x = Chyba String | Hodnota x deriving (Show)
instance Monad Result where
  Chyba s >>=  _ = Chyba s
  Hodnota a >>= f = f a
  return x = Hodnota x
  fail s = Chyba s
type Stream w = [w]->[w]
data Vypocet w s x = V {unV::((s,Stream w)->(s, Stream w, Result x))}
instance Monad (Vypocet w s) where
  V vyp1 >>= f = V (\s->let (s', w', val) = vyp1 s in case val of
                                Chyba ch -> (s', w', Chyba ch)
                                Hodnota x -> unV (f x) (s', w'))
  return x = V (\(s,w) -> (s,w,return x))
  fail ch =  V (\(s,w) -> (s,w,fail ch))
```

```
class     MonadRead m s | m -> s     where get :: m s
instance MonadRead (Vypocet w s) s  where get = V (\(s,w) -> (s,w,return s))
class     MonadState m s | m -> s    where put :: s -> m ()
instance MonadState (Vypocet w s) s where put s = V (\(_,w) -> (s,w,return ()))
class     MonadWrite m w | m -> w    where write :: w -> m ()
instance MonadWrite (Vypocet w s) w where write w = V (\(s, ws)->
                                                    (s, ws.(w:), return ()))

runVypocet :: Vypocet w s x -> s -> (s, [w], Result x)
runVypocet (V f) state = let (s, ws, r)=f (state, id) in (s, ws [], r)

eval::(Monad m, MonadRead m Values, MonadState m Values, MonadWrite m Integer)
    => Expr->m Integer
{-...-}
eval (Output e) = do r <- eval e
                     write r
                     return r
```

Modul Eval6

```
import Expr
```

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`.

```
instance MonadPlus (Vypocet w s) where
  V f1 'mplus' V f2 = V (\(s,w)->let (s', w', r') = f1 (s, w) in case r' of
                           Chyba _ -> f2 (s', w')
                           _       -> (s', w', r'))
  mzero = V (\(s,w)->(s,w,fail "mzero"))

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

Nebo `f2 (s,w)`, rozmyslete si

Pro *MonadPlus* existují užitečné funkce jako

```
msum :: MonadPlus m => [m a] -> m a
msum = foldr mplus mzero

guard :: MonadPlus m => Bool -> m ()
guard True  = return ()
guard False = mzero
```

Domácí úkoly

♣ Napište příkaz `ls`, který dostane na příkazové řádce několik adresářů a každý adresář rekurzivně projde a vypíše.
♣ Napište příkaz `diff`, který dostane na příkazové řádce dva soubory (pokud jeden chybí, tak je to standardní vstup) a najde diff, tj. nejmenší počet řádek, které je třeba odstranit či přidat, aby se z prvního souboru stal druhý. Na výstup (jako `diff -u`) vypište nezměněné řádky s mezerou na začátku, odstraněné s − na začátku a přidané s + na začátku.
♣ Rozšiřte monádu z *Eval6.hs* tak, aby podporovala funkci `stop`. Tato funkce přeruší výpočet v monádě, vrátí aktuální stav, aktuální výpis a zbytek výpočtu. Uživatel může zbytek výpočtu kdykoliv spustit. Proveďte to tak, že přidáte třídu *MonadStop*, která obsahuje funkci **stop::*m ()***. Samozřejmě je třeba upravit typ *V* a monádové instance.
♣ Upravte monádu z *Eval6.hs* tak, aby **'mplus'** vyhodnocoval obě větve výpočtu paralelně (po 1 bindu na každé straně). Samozřejmě opět můžete upravit typ *V* a jeho monádové instance.
♣ Vytvořte funkci **wrapper::*f->IO ()***, která dostane libovolnou funkci. Všechny potřebné parametry funkce načte ze standardního vstupu (každý je na jedné řádce) a její výsledek vypíše na standardní výstup. Načítat a vypisovat musíte umět *Int*, **[** *Int* **]** (na jedné řádce oddělené mezerami) a *String* (celá řádka i s mezerami).
♣ A možná další, viz web.