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 e
                      if r2==0 then fail "Deleni nulou" else return(r1 `mod` r2)
eval (Num n) = return n
```

PreludeIO

```
data IO
type FilePath = String

putChar    :: Char -> IO ()
putStr     :: String -> IO ()
putStrLn   :: String -> IO ()
print      :: Show a => a -> IO ()
getChar    :: IO Char
getLine    :: IO String
getContents:: IO String
interact   :: (String -> String) -> IO ()
readFile   :: FilePath -> IO String
writeFile  :: FilePath -> String -> IO ()
appendFile :: FilePath -> String -> IO ()
readIO   :: Read a => String -> IO a
readIO s =  case [x | (x,t) <- reads s, ("","") <- lex t] of
             [x] -> return x
             []  -> ioError (userError "Prelude.readIO: no parse")
             _   -> ioError (userError "Prelude.readIO: ambiguous parse")
readLn :: Read a => IO a
readLn =  do l <- getLine
             r <- readIO l
             return r

-- IO exceptions
data IOError
ioError    :: IOError -> IO a
userError  :: String -> IOError
catch      :: IO a -> (IOError -> IO a) -> IO a
```

Domácí úkoly

♣ Napište funkci **balanced::*[Int]->Int***, která dostane seznam nul a jedniček a vrátí délku nejdelší souvislé podposloupnosti, ve které je stejně nul a jedniček. Body za O(N^2) i O(N).
♣ Napište funkci **usek::*[Int]->Int***, která dostane posloupnost a vrátí délku její nejdelší souvislé podposloupnosti, která má nezáporný součet svých členů. Body za O(N^2) i O(N).
♣ Napište funkce **demorse::*String->String***, která dekóduje vstup v morseovce. Tečka je tečka, čárka je pomlčka, oddělovač znaků je svislítko a oddělovač slov jsou dvě svislítka. Ostatní znaky ignorujte.

♣ Rozšiřte monádu výpočet z *Eval3*.hs tak, aby podporovala funkci **stop :: *Vypocet ()***. Této funkci předáte string. Funkce přeruší výpočet v monádě a vrátí (uvnitř té monády samozřejmě) onen string a zbytek výpočtu. Uživatel může zbytek výpočtukdykoliv spustit.