Standardní pole

```
array     ::(Ix a)=>(a,a)->[(a,b)]->Array a b   array (1,10) [(i,i)|i<-[1..10]]
listArray ::(Ix a)=>(a,a)->[b]->Array a b       listArray (1,10) [1..10]
(!)       ::(Ix a)=>Array a b->a->b             a!1
bounds    ::(Ix a)=>Array a b -> (a,a)  indices::(Ix a)=>Array a b->[a]
elems     ::(Ix a)=>Array a b -> [b]    assocs ::(Ix a)=>Array a b->[(a,b)]
(//)      ::(Ix a)=>Array a b->[(a,b)]->Array a b a//[(1,2), (3,4)], dělá celou kopii
```

Pole jsou líná v hodnotách − nevyhodnocují, dokud nemusí, tj. funguje

```
prefixSumsTo n = result
  where result = array (0, n) $ (0, 0) : [i + result ! (i-1) | i <- [1..n]]
```

Vícerozměrná pole pomocí `array ((1,1),(100,100)) [((i,j),i+j)|i<-[1..100],j<-[1..100]]`

Modul Expr

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

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

Modul Eval1

```
-- vyhodnocení výrazu
eval::Expr->Integer
eval (Plus e1 e2) = eval e1 + eval e2     eval (Div e1 e2) = eval e1‘div‘eval e2
eval (Minus e1 e2) = eval e1 - eval e2    eval (Mod e1 e2) = eval e1‘mod‘eval e2
eval (Mul e1 e2) = eval e1 * 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           ret :: x -> Result x
bind (Chyba s) _ = Chyba s                              ret x = Hodnota x
bind (Hodnota a) f = f a                                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

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                 return x = Hodnota x
  Hodnota a >>= f = f a                     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
```