Existenciální datové typy a polymorfismus vyšších řádů

```
data Vypocet a b = V (a->b)
data Vypocty a b = Vs (a->c) (c->b)    Tohle nejde, c není definované
data Vypocty a b = forall c. Vs (a->c) (c->b)

class Player a where play::a->IO ()
instance Player Human where ...
instance Player AI where ...
data PLAYER = forall a. (Player a)=>PLAYER a

zpracuj::(a->d)->[b]->[c]->[d]
zpracuj f bs cs = map f bs ++ map f cs
zpracuj (const 5) "ahoj" [1,2,3]  by klidně mohlo fungovat, ale neotypuje se
zpracuj::(forall a. a->d)->[b]->[c]->[d]
```

Dynamické typování v Haskellu, modul Data.Typeable

```
class Typeable a where typeOf::a->TypeRep
data TypeRep = ... deriving (Eq, Show, Typeable)

cast  :: (Typeable a, Typeable b) =>   a -> Maybe    b
gcast :: (Typeable a, Typeable b) => c a -> Maybe (c b)
```

Standardní monolitická pole s konstantním přístupem a lineárními updaty

```
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]
accumArray::(Ix i)=>(e->e'->e)->e->(i,i)->[(i, e')]->Array i e
(!)       ::(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.
Vícerozměrná pole pomocí `array ((1,1),(100,100)) [((i,j),i+j)|i<-[1..100],j<-[1..100]]`

Rozdílová pole, module Data.Array.Diff

`DiffArray` se chová jako *`Array`*, ale umí dělat úpravy v O(1) a přistupovat k poslední verzi v O(1).
Prakticky jsou ale vždy pomalejší než stromy

Vyvažované stromy, moduly Data.Map, Data.IntMap, Data.Set, Data.IntSet

Varianty *`Int...`* mají stejné operace, ale klíče jsou vždy *`Int`*y. Jsou stejné, jenom rychlejší.

```
         Set                                 Map
null      ::Set a->Bool              null     ::Map k a->Bool
size      ::Set a->Int               size     ::Map k a->Int
member    ::Ord a=>a->Set a->Bool    member   ::Ord k=>k->Map k a->Bool
isSubsetOf::Ord a=>Set a->Set a->Bool (!)     ::Ord k=>Map k a->k->a
empty     ::Set a                    empty    ::Map k a
singleton ::a -> Set a               singleton::k->a->Map k a
insert    ::Ord a=>a->Set a->Set a   insert   ::Ord k=>k->a->Map k a->Map k a
delete    ::Ord a=>a->Set a->Set a   delete   ::Ord k=>k->Map k a->Map k a
```

Pole s konstantním update, odkazy

Existuje monáda *`Control.Monad.ST`* s typem `data ST s a`. Také existuje funkce
`runST :: (forall s. ST s a) -> a`, která provede výpočet. Všimněte si toho `forall`...
A v modulu *`Data.Array.ST`* jsou pole uvnitř *`ST`* monády:

```
runSTArray::Ix i=>(forall s. ST s (STArray s i e)) -> Array i e
newArray ::...=>(i, i)->e->m (a i e) newListArray::...=> (i, i)->[e]->m (a i e)
readArray::...=>a i e->i->m e        writeArray ::...=> a i e ->i->e-> m ()
getBounds::...=>a i e->m (i, i); getElems::...->m [e]; getAssocs::...->m [(i,e)]

count::[Int]->Array Int Int    Řekněme, že čísla jsou 0..9
count n = runSTArray $ do a<-newArray (0,9) 0
                          mapM_ (\i->readArray a i >>= writeArray a i . (+1)) n
                          return a
```

V modulu *Data.STRef* jsou reference uvnitř *ST* monády:

```
newSTRef :: a -> ST s (STRef s a)          readSTRef :: STRef s a->ST s a
writeSTRef :: STRef s a -> a -> ST s ()   modifySTRef::STRef s a->(a->a)->ST s ()

swap::STRef s a->STRef s a->ST s ()
swap a b = do x<-readSTRef a; y<-readSTRef b; writeSTRef a y; writeSTRef b x
```

Protože ve skutečnosti je *IO* monáda jenom *ST RealWord*, existují odkazy a pole i v monádě *IO*.
Odkazy jsou v modulu *Data.IORef* (nahraďte ve funkcích *ST* za *IO*), pole jsou v *Data.Array.IO*.
S polemi se zachází úplně stejně, jenom nejdou "vyndat" z monády. Ale `freeze` udělá kopii v lineárním čase.
Obě pole mají společnou třídu *MArray* z *Data.Array.MArray*, proto mají společné operace.

DFS na sto a jeden způsob

Klasické pole –– DFS trvá O(n^2); rozdílové pole –– DFS trvá O(n)

```
dnum::Array Int [Int]->Int->Array Int Int
dnum g s = d [s] (array (bounds g) [(i,-1)|i<-indices g]) 0 where
  d []     m _ = m
  d (s:ss) m n = if m!s==(-1) then d (g!s++ss) (m//[(s,n)]) (n+1) else d ss m n
```

Vyvážený strom, DFS trvá O(m+n log n)

```
dnum_imap::Array Int [Int]->Int->IntMap Int Int
dnum_imap g s = d [s] IntMap.empty 0 where
  d []     m _ = m
  d (s:ss) m n = if IntMap.notMember s m
                 then d ((g!s)++ss) (IntMap.insert s n m) (n+1) else d ss m n
```

Neboxované pole v *ST* monádě, O(m)

```
dnum_stu::Array Int [Int]->Int->UArray Int Int
dnum_stu g s = runSTUArray $ newArray (bounds g) (-1) >>= d [s] 0 where
  d []     _ m = return m
  d (s:ss) n m = do sn<-readArray m s
                    if sn==(-1) then writeArray m s n >> d (g!s++ss) (n+1) m
                                else d ss n m
```

Pole líné i v indexech, package LazyArray z Hackage

```
lArray     ::(Ix i)=>           (i,i)->[(i,e)]->Array i [e]
lArrayMap  ::(Ix i)=>([e]->e')->(i,i)->[(i,e)]->Array i e'
lArrayFirst::(Ix i)=>      e   ->(i,i)->[(i,e)]->Array i e

dnum_la::Array Int [Int]->Int->Array Int Int
dnum_la g s = marks where
  list = d [s] 0
  marks = lArrayFirst (-1) (bounds g) list
  d []     _ = []
  d (s:ss) n = (s,n) : if n == marks!s then d (g!s++ss) (n+1) else d ss n
```

```
                              Calc
                              ----
import Char
import Control.Monad
import Control.Monad.Instances

--Parser z minula
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)

--Basic parsers
char::MonadPlus m=>Parser m Char
char=Parser(\s->case s of []->mzero; (c:cs)->return (c,cs))
chars::MonadPlus m=>(Char->Bool)->Parser m [Char]
chars p = Parser(\s->return $ span p s)

--More high level combinators
spaces=chars isSpace >> return ()
symbol s=spaces >> char >>= \c->guard $ c==s
number=spaces >> chars isDigit >>= \n->if null n then mzero else return $ read n
multiple p op = p >>= \a->multiple' p op a where
  multiple' p op a = (do o<-op; b<-p; multiple' p op (a`o`b)) `mplus` return a

--Expression parsing
addop = (do symbol '+'; return (+)) `mplus` (do symbol '-'; return (-))
mulop = (do symbol '*'; return (*)) `mplus` (do symbol '/'; return div) `mplus`
        (do symbol '%'; return mod)
expop = do symbol '^'; return (^)

expr last=multiple (mulexp last) addop
mulexp last=multiple (expo last) mulop
expo last=multiple (term last) expop
term last=number `mplus`
          (do symbol '('; res<-expr last; symbol ')'; return res) `mplus`
          (do symbol 'l'; 'a'<-char; 's'<-char; 't'<-char; return last)

main = getContents >>= eval 0 . lines where
  eval _ []     = return ()
  eval l (e:es) = case parse (expr l) e of Just (a,[])->print a >> eval a es
                                           otherwise->putStrLn"Chyba">>eval l es

                              Demo
                              ----
import Array

--pomoci pole
demorse = map num2char . code2num 1 where
  code2num n []                        = []
  code2num n ('|':s)                   = n : code2num 1 s
  code2num n (c:s)    | c `elem` ".-" = code2num (2*n+if c=='-' then 1 else 0) s
                      | otherwise      = code2num n s
  morsenums = listArray (1,31) " etianmsurwdkgohvf~l~pjbxcyzq~~"
  num2char n = morsenums!n

--pomoci stromu
data MTree = Nic | Z Char MTree MTree

demorse2 = demo morsetree where
  demo _         []     = []
  demo Nic       str    = demo morsetree $ tail $ dropWhile (/='|') str
  demo (Z z l r) (c:s) | c=='|'         = z : demo morsetree s
                       | c `elem` ".-" = demo (if c=='.' then l else r) s
                       | otherwise      = demo (Z z l r) s
```

```
  morsetree= Z ' ' (Z 'e' (Z 'i' (Z 's' (Z 'h' Nic Nic) (Z 'v' Nic Nic))
                                 (Z 'u' (Z 'f' Nic Nic) Nic))
                          (Z 'a' (Z 'r' (Z 'l' Nic Nic) Nic)
                                 (Z 'w' (Z 'p' Nic Nic) (Z 'j' Nic Nic))))
                   (Z 't' (Z 'n' (Z 'd' (Z 'b' Nic Nic) (Z 'x' Nic Nic))
                                 (Z 'k' (Z 'y' Nic Nic) (Z 'c' Nic Nic)))
                          (Z 'm' (Z 'o' Nic           Nic)
                                 (Z 'g' (Z 'z' Nic Nic) (Z 'q' Nic Nic))))
```

Bala

```
import Array
--pomoci poli
bala xs = maximum [ max-min | (sum,max)<-psums, let min=mini!sum ] where
  n = length xs
  psums = zip (scanl (\x y->x+2*y-1) 0 xs) [0..]
  mini = accumArray min n (-n,n) psums

--bez poli, protoze soucty se meni jenom o jedna
bala2 xs = bal xs 1 (repeat n) (repeat n) 0 where
  n = length xs
  bal []     i (a:as) _      m = m `max` i-a
  bal (0:xs) i (a:as) b      m = bal xs (i+1) as    ((min a i):b) (m`max`i-a)
  bal (1:xs) i (a:as) (b:bs) m = bal xs (i+1) (b:(min a i):as) bs (m`max`i-a)
```

Usek

```
usek::[Int]->Int
usek xs = longest psums maxs 0 where
  n = length xs
  psums = zip [0..] $ scanl (+) 0 xs
  maxs = foldr(\x y->if snd x>snd(head y)then x:y else y) [(n+1,minBound)] psums
  longest []     _      b                  = b
  longest (x:xs) (m:ms) b | snd x<=snd m = longest (x:xs) ms$b`max`(fst m-fst x)
                          | otherwise    = longest xs (m:ms) b
```