```
-- literate haskell .lhs
-- radky zacinaji >
{-
> lhsFunkce x = lhsTelo (x * x)
>   where
>    lhsTelo y = y `div` 2

-- z prelude
foldr :: (a->b->b) -> b -> [a] -> b
foldr f v []     = v
foldr f v (x:xs) = f x (foldr f v xs)

(.) :: (b->c) -> (a->b) -> (a->c)
(f . g) x = f (g x)
-- > test0 x =  faze3 ( faze2 ( faze1  x ) )
-- > test1 x = (faze3 . faze2 . faze1) x
-- > test2   =  faze3 . faze2 . faze1 -- stejne typy

infixr 0 $
($) :: (a->b) -> a -> b
f $ x = f x
-- > test3 x = faze3 $ faze2 $ faze1 x

infixr 0 $!
($!) :: (a->b) -> a -> b
f $! x = ... interni, pro striktni vyhodnocovani
-- vyhodnoti top konstruktor x
-}
```

```
-----------------------------------
-- 11.5.2009 stromy
-- (1) data len v listoch
data LTree a = LLeaf a
             | LBranch (LTree a) (LTree a)

-- (2) data interne, BVS
data BTree a = BLeaf
             | BBranch (BTree a) a (BTree a)
-- BVS = BTree (TKey,TValue) -- dvojice :-(

-- (3) data: ine v listoch, ine vnutri
data DTree a b = DLeaf a
               | DBranch (DTree a b) b (DTree a b)
binVyraz :: DTree Int String
binVyraz = DBranch (DLeaf 1)
                   "+"
                   (DBranch (DLeaf 2)"*"(DLeaf 3))
-- R-B trees ...
t1BTree = BBranch
            (BBranch BLeaf 1 BLeaf)
            2
            (BBranch
              (BBranch BLeaf 3BLeaf)
              4
              (BLeaf) )

{- typovy konstruktor BTree, LTree druhu(::) * -> *
   typ.k. DTree druhu (kind) *->*->*
-- datovy konstruktor BLeaf :: BTree a -- polymorfni
   dat.k. BBranch :: BTree a -> a -> BTree a -> BTree a
   dat.k. DLeaf :: a -> DTree a b
-}
```

```
foldBT :: b->(b->a->b->b)->BTree a-> b
  -- ala foldr pro seznamy (strukt. rek.)
foldBT fLeaf fBranch bt = f bt  where
  f  BLeaf           = fLeaf
  f (BBranch l x r) = fBranch (f l) x (f r)

copyBT t = foldBT BLeaf BBranch t -- "identita"
sizeBT t = foldBT 0 (\l x r->l+1+r) t
depthBT t = foldBT 1 (\l x r->1+max l r) t
listify t = foldBT [] (\l x r->l++(x:r)) t -- infixBT
postfix t = foldBT [] (\l x r->l++r++[x]) t
sumBT t  = foldBT 0 (\l x r->l+x+r) t
mapBT f t  = foldBT BLeaf (\l x r->BBranch l (f x) r) t
heapify t = foldBT BLeaf (\l x r->inHeap l x r) t
  -- where inHeap na dalsi strane
prumer t = fromInteger suma/fromInteger size
  -- obecne: potrebuji postspracovani
 where
  (suma,size) = foldBT (0,0)
                (\(ls,lv) x (rs,rv)->(ls+x+rs,lv+1+rv)) t
--bvs t = foldBT True (\l x r->l && r && locOrdered l x r) t
  -- where locOrdered l x r = ...
vaha :: BTree (Int,a) -> (Int,Int)
       --   (freq,key)-> (freq, vaha); vaha = sum(freq*depth)
vaha t = foldBT (0,0) (\(fl,vl)(fx,_k)(fr,vr)
            ->(fl+fx+fr,vl+fl+fx+vr+fr) ) t
-- ... a varianty fold:
 -- Obecne musim vracet i puvodni i spracovanou strukt. (tj. dvo
```

```
-- mkHeap :: BTree a -> BTree a
  -- jen preusporadani, ne stavba
mkHeap BLeaf = BLeaf
mkHeap (BBranch l x r) = inHeap (mkHeap l) x (mkHeap r)

inHeap BLeaf x BLeaf = BBranch BLeaf x BLeaf
inHeap l@(BBranch l1 xl l2) x BLeaf
  | x <= xl = BBranch l                 x  BLeaf
  | True    = BBranch (inHeap l1 x  l2) xl BLeaf
inHeap BLeaf x r@(BBranch r1 xr r2)
  | x <= xr = BBranch BLeaf x  r
  | True    = BBranch BLeaf xr (inHeap r1 x  r2)
inHeap l@(BBranch l1 xl l2) x r@(BBranch r1 xr r2)
  | x <= xl && x <= xr
            = BBranch l                x  r
  | xl<= xr = BBranch (inHeap l1 x l2) xl r
  | True    = BBranch l                xr (inHeap r1 x  r2)

-- aktivni konstruktory: buduji pozmenenou strukturu
t1Heap =
  bBranch
    (bBranch bLeaf 1 bLeaf)
  2
  (bBranch
    (bBranch bLeaf 3 bLeaf)
      4
      (bLeaf)
    )
  where
 bLeaf = BLeaf
 bBranch = inHeap
```

```
----------------
-- stromy n-arne (Rose Tree)
data RTree a = RT a [RTree a]

-- aplikace: XML...
data XML a = Elm a [XML a]
           | Txt String
-- a/String  -- jen tag
-- a/(String,[(String,String)])
-- show bez chyb, praca so zoznamami elementov

{- -- aplikace: univ.vyrazy, synt. stromy ...
data Expr a = Var String
            | Fn String [Expr a]
            ...
            | Con a
            | If (Expr Bool) (Expr a) (Expr a)
            | Inc (Expr Int)
-- vyrazy v FP, nejen PP
-- fantomove typy
-}
```

```
-- huffmanovo kodovani: optimalni, hladovy alg.
-- vzdaleny ;-) cil
{- test_HDE s = let t = mkHTree $ freq s
              in s == (dec t $ enc (hstrom2tab t) s)
-}

data HStrom a = HL a
              | HV (HStrom a) (HStrom a)
hstrom2tab :: HStrom (Char {-,Int-}) -> [(Char,String)]
hstrom2tab (HL(c)) = [(c,"")]
hstrom2tab (HV l r)   =
  map (\(c,kod)->(c,'0':kod))
    (hstrom2tab l)
   ++
  map (\(c,kod)->(c,'1':kod))
    (hstrom2tab r)

-- volani> hs2t "" strom -- akumulator
hs2t kod (HL(c,"")) = [(c,kod)]
hs2t kod (HV l r)   =
  hs2t (kod++"0") l ++ hs2t (kod++"1") r

enc :: (Eq a,Show a) => [(a,[b])] -> [a] -> [b]
enc tab []     =  []
enc tab (x:xs) =
  enc1 tab x ++ enc tab xs
   where
    enc1 []             x =
      error("enc: missing code for:"++show x)
    enc1 ((x1,kod):tab) x =
      if x==x1 then kod
                else enc1 tab x
```

```
enc2 tab xs =
    concat(map(\x -> snd(head(filter (f x) tab))) xs)
  where f x = \(x1,_kod) -> x==x1  -- x: lambda-lifting
        -- f x = (x==).fst -- varianta
-- = concat $ map (\x -> snd $ head $ filter f tab) xs

-- concat []     =  [] -- v Prelude
-- concat (x:xs) =  x ++ concat xs
-- anebo: concat xss = foldr (++) [] xss
```

```
dec :: HStrom a -> [Char] -> [a]
dec strom []    = []
dec strom inp  = c : dec strom inp1
  where
    (c, inp1)      = dec1 strom inp
    dec1 :: HStrom a -> [Char] -> (a,[Char])
    dec1 (HL c) inp           = (c,inp)
    dec1 (HV l r)   (’0’:inp) = dec1 l inp
    dec1 (HV l r)   (’1’:inp) = dec1 r inp
    dec1 (HV l r)   ( x :inp) = error ("dec: bad code:"++show x)
    dec1 (HV l r)   []        = error "dec: unexpected EOF"

dec2 :: HStrom a -> [Char] -> [a]
dec2 strom inp = case x of
                   Nothing       -> []
       -- interni chyba: nic se nedekoduje
                   Just (c,inp1) -> c : dec strom inp1
  where
    x = dec1 strom inp
    dec1 :: HStrom a -> [Char] -> Maybe (a,[Char])
    dec1 (HL c) inp              = Just (c,inp)
    dec1 (HV l r)      (’0’:inp) = dec1 l inp
    dec1 (HV l r)      (’1’:inp) = dec1 r inp
    dec1 (HV l r)      ( x :inp) = Nothing
    dec1 (HV l r)      []        = Nothing
-- pozn: odebirana cast je dana kontextem
{-
data Maybe a = Nothing
             | Just a
          deriving (Eq, Show)
-}
```

```
mkHTree :: [(Char,Int)] -> HStrom Char
mkHTree xs = faze5
  where
    cmp (c1,f1) (c2,f2) = f1<=f2
    -- cmp   = \(c1,f1) (c2,f2) -> f1<=f2
    faze1 = sort cmp xs
    faze2 = map (\(c,f) -> (HL c, f)) faze1
    faze3 = iterate spoj faze2
    faze4 = dropWhile (\x -> length x > 1) faze3
    faze5 = fst $ head $ head faze4
    spoj [x]                  = [x]
    spoj ((c1,f1):(c2,f2):xs) = insert cmp (HV c1 c2,f1+f2) xs
    sort cc xs = foldr (insert cc) [] xs
    insert cc x [] = [x]
    insert cc x vs@(y:ys)
      | x`cc`y = (x:vs)
      | True = y:insert cc x ys
```

```
freq :: String -> [(Char,Int)] -- histogram
freq xs = foldr ordBagUnion [] $ map (\x->[(x,1)]) xs
  -- O(n^2) :-(

ordBagUnion [] ys = ys
ordBagUnion xs [] = xs
ordBagUnion xx@((x,fx):xs) yy@((y,fy):ys)
  | x <  y = (x,fx)   :ordBagUnion xs yy
  | x == y = (x,fx+fy):ordBagUnion xs ys  -- memory leak
  | x >  y = (y,fy)   :ordBagUnion xx ys

foldDvoj f e [] = []
foldDvoj f e xs =
  (head.head.dropWhile((1<).length).iterate(spojDvoj f))xs
  where
   spojDvoj f (x1:x2:xs) = f x1 x2 : spojDvoj f xs
   spojDvoj f  xs        = xs
freq2 xs = foldDvoj ordBagUnion [] $ map (\x->[(x,1)]) xs
testFreq k c = [s|s<-varOpak1 k ['a'..c],
                  let s1=s,
                  freq2 s /= freq s ]
-- testFreq 7 'f' : 1'23, length 279936 testov
{- prelude
iterate :: (a->a) -> a -> [a]
iterate f x = x : iterate f (f x)

dropWhile :: (a->Bool) -> [a] -> [a]
dropWhile _ []     = []
dropWhile p (x:xs)
  | p x            = dropWhile p xs
  | otherwise      = x : xs
-- DC: ukoncenie podla vztahu dvoch prvkov
-}
```

```
varOpak :: [a] -> [b] -> [[(a,b)]]
varOpak []     _  = [[]] -- vhodne "dodefinovano"
varOpak (x:xs) r  =
  [(x,y):vs| y<-r, vs <- varOpak xs r]

varOpak1 :: Int -> [b] -> [[b]]
varOpak1 k r =
  map             -- pro všechny variace
      (map        -- pro všechny prvky ve var.
           snd)   -- zahazuju dom., vracím hodnotu
      (varOpak    -- volání puv. fce dostane:
           [1..k] -- domain jako seznam
           r)     -- range bez zmen

komb :: Int -> [a] -> [[a]]
komb 0     _      = [[]] -- vhodne "dodef."
komb (n+1) []     = []
komb (n+1) (x:xs) =
  [x:ko | ko <- komb n xs] ++ komb (n+1) xs

kombOpak :: Int -> [a] -> [[a]]
kombOpak 0     _      = [[]] -- vhodne "dodef."
kombOpak (n+1) []     = []
kombOpak (n+1) (xs) =
  [head xs:ko | ko <- kombOpak n xs] ++ kombOpak (n+1) (tail xs)
```

```
-- testy: QuickCheck 2000, SmallCheck 2008, ...
test_HDE s = let t=mkHTree $ freq s
              in s == (dec t $ enc (hstrom2tab t) s)
prop_HDE k c = [s|s<-kombOpak k ['a'..c], not $ test_HDE s]
                 --  varOpak1

-- > prop_HDE 3 'b'
-- >> ["aaa","bbb"]

-- s kombOpak: 15 'g' : 35'', 54264 testov
-- s varOpak1:  7 'f'  : okolo 2', 279936 testov
{-
-- Zde ukonceno na Prednasce 12.05.2009,
-- nektere drobnosti preskocene
Obsahlejsi popis napr. Huffmanova kodovani v textu "Haskell v pr
--------------------------------------------- -}
```

```
-----------
stableSort cmp xs =
  map fst $ -- odstraneni poradi (A)
    sort1 cmp2 $ -- trideni se zmenou porovnavaci fce (B)
      zip xs [1..] --pridani poradi (C)
  where cmp2 (x,i) (y,j) = x `cmp` y && not (y `cmp` x)
                         || x `cmp` y && y `cmp` x && i <= j
sort1 cmp []     = []
sort1 cmp (p:ys) =
    sort1 cmp ([y|y<-ys,y`cmp`p]++(p:[y|y<-ys,not$y`cmp`p]))

{- -- add
F# (.NET), Scala (nad JVM)
lift: na Maybe, Either
na zoznamy (vektory), matice
na funkcie - casova zavislost
DSEL - Domain Specific (Embedded) Language
fantomove typy
  Calling hell from heaven and heaven from hell, 1999
kontext
Blub: hypoteticky prog. jazyk, Paul Graham

-}
```

```
--------------
-- (tvorba indexu: reverzny zoznam)

type Word = String
splitWords :: String -> [Word]
splitWords st = split (dropSpace st)
split :: String -> [Word] - dostává ř. bez úvodních mezer
split [] = []
split st = (getWord st):split(dropSpace(dropWord st))

dropSpace st = dropWhile (\x->elem x whitespace) st
dropWord st = dropWhile (\x->not(elem x whitespace)) st
dropWhile :: (t -> Bool) -> [t] -> [t] - obecná výkonná procedur
dropWhile p [] = [] -- zahodí poč. prvky, které nesplňují podm.
dropWhile p (a:x) -- v prelude
  | p a   = dropWhile p x
  | otherwise = (a:x)
getWord st = takeWhile (\x-> not(elem x whitespace)) st
-- DC: takeWhile p x =  -- v prelude

whitespace = [ , \t, \n]
```

Aritmetika s testováním chyb (pomocí Maybe)

```
lift2M :: (a->b->c)->Maybe a -> Maybe b -> Maybe c
lift2M op Nothing _         = Nothing  chyba v 1. arg.
lift2M op _       Nothing   = Nothing  chyba v 2. arg.
lift2M op (Just x) (Just y) = Just (x op y)  bez chyb

minusM :: Maybe Float -> Maybe Float -> Maybe Float
minusM x y = lift2M (-) x y

-- Vyvolání chyby
delenoM x y = if y==Just 0 then Nothing --test a vyvolání chyby
       else lift2M (/) x y
? delenoM(Just 3)(minusM(Just 2)(Just 2)) -- 3/(2-2)
? delenoM(Just 3)(lift2M(-)(Just 2)(Just 2)) -- dtto Nothing

data AV a = AV a :-: AV a  | AV a :/: AV a | Con a |

eval :: AV Integer  -> Maybe Integer
eval (av1 :/: av2) = delenoM (eval av1) (eval av2)
eval (av1 :-: av2) = lift2M (-) (eval av1) (eval av2)
eval (Con x)       = Just x -- cena za Maybe: zabalení výsledku

-- Odchycení chyby:
? catch (eval (Con 3 :/: (Con 2 :-: Con 2)) ) 1
catch Nothing  oprData = opravnaFce oprData
catch (Just x) _       = x
```

map pro jiné d.s.

pro binární stromy

rozbor podle konstruktoru

```
mapTree :: (a->b) -> Tree a -> Tree b
mapTree f (Leaf a)     = Leaf (f a)
mapTree f (Branch l r) = Branch (mapTree f l)
                                (mapTree f r)
```

n-ární stromy

```
data NTree a = Tr a [NTree a]
mapNT :: (a->b) -> NTree a -> NTree b
mapNT f (Tr x trees) = Tr (f x) (map (mapNT f) trees)
```

typ Maybe, Union - nerekurzivní

```
mapM :: (a->b) -> Maybe a -> Maybe b
mapM f Nothing = Nothing
mapM f (Just x)= Just (f x)

mapU :: (a->c)->(b->d) -> Union a b -> Union c d
mapU f g (Left x)  = Left (f x)
mapU f g (Right y) = Right(g y)
```

Stavové programování

(Návrhový vzor iterátor)

N.vzory ve FP lze (často) napsat jako kod s funkc. parametry vs. (často) pseudokód v OOP

```
iterator::s->(s->s)->(s->Bool)->s
  -- vrací celý (interní) stav, zde není výstupní projekce
iterator init next done =
  head( dropWhile (not.done) --mezivýsl. se průběžně
          ( iterate next init ) ) -- zahazují
```

DC: fixpoint :: (s->s)->s->s :vrací v=f^n(x) pro min. n:v=f(v)

Počítání s programy

Typicky: dokazování vlastností programů
(Částečná) správnost vzhledem k specifikaci
Transformace programů pro optimalizaci
př.: asociativita (++) append

```
[] ++ ys  = ys
(x:xs) ++ ys = x: (xs++ys)
```

Tvrzení: x++(y++z) = (x++y)++z pro konečné x, y, z.

```
x=[] : LS = []++(y++z) = y++z
PS = ([]++y)++z = y++z
x=a:v : LS = (a:v)++(y++z) =  / def. ++
a:(v++(y++z)) =  / ind. předp. ++
a:((v++y)++z)
PS = ((a:v)++y)++z =   / def. ++
((a:(v++y)) ++ z) = / def. ++
a:((v++y)++z) QED ?
```