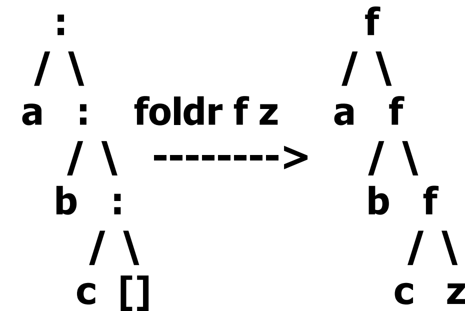
Triedenie s banánmi a šošovkami

možno to až k triedeniu (dnes) nepríde...

- Lex Augusteijn: Sorting Morphisms, LNCS 1608, 3rd Int.School of Advanced Functional Programming, 1998
- Erik Meijer, Maarten Fokkinga, Ross Paterson:
  Functional Programming with *Bananas, Lenses, Envelopes* and Barbed Wire,
  http://research.microsoft.com/en-us/um/people/emeijer/Papers/fpca91.pdf
- Erik Meijer: Merging Monads and Folds for Functional Programming, LNCS 925, 1st Int.School of Advanced Functional Programming, 1996
- Schémy rekurzie:
  - catamorphism
  - anamorphism

Schémy, ktoré sme už videli

```
foldr              :: (x -> u -> u) -> u -> [x] -> u
foldr f z []        = z
foldr f z (y:xs) = f y (foldr f z xs)

a : b : c : [] -> f a (f b (f c z))
```




```
-- g je vnorená lokálna funkcia

foldr :: (x -> u -> u) -> u -> [x] -> u
foldr f z = g
  where    g []      = z
           g (y:xs) = f y (g xs)
```

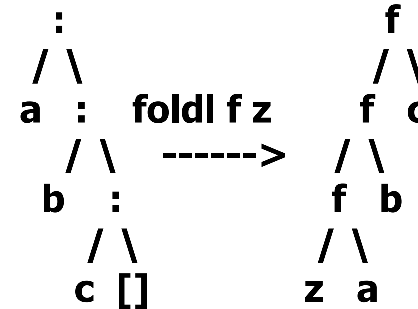
```
Main> foldr (+) 0 [1..100]
5050
```

```
Main> foldr (\x y->10*y+x) 0 [1,2,3,4]
4321
```

nepomenovaná analógia fcie: pom x y = 10*y+x

Schémy, ktoré sme už videli

```
foldl            :: (x -> u -> x) -> x -> [u] -> x
foldl f z []       =  z
foldl f z (y:xs) =  foldl f (f z y) xs

a : b : c : [] -> f (f (f z a) b) c
```

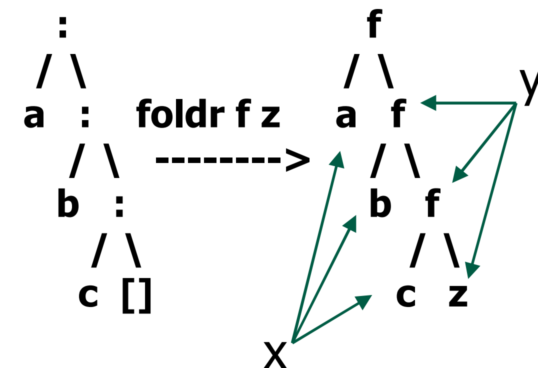



```
Main> foldl (+) 0 [1..100]
5050

Main> foldl (\x y->10*x+y) 0 [1,2,3,4]
1234
```

reverse s foldr a foldl

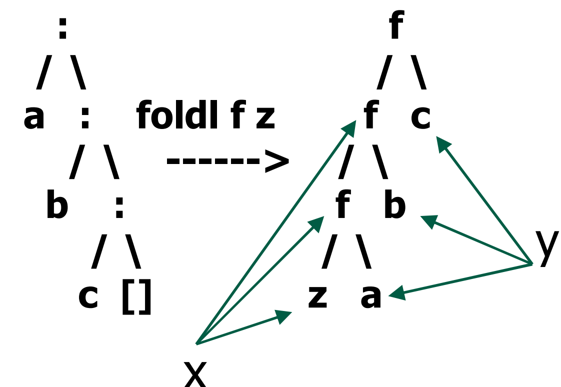
- myReverse xs = foldr (\x -> \y -> y ++ [x]) [] xs
  myReverse = foldr (\x y -> y ++ [x]) []

  prvok zoznamu výsledok rek.volania



- myReverse' xs = foldl (\x -> \y -> y:x) [] xs
  myReverse' = foldl (\x y -> y:x) []

  stará hodnota akumulátora prvok zoznamu

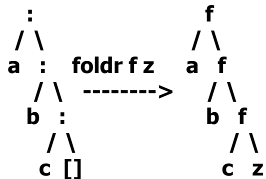


foldr a foldl pre pokročilejších

foldr (:) [] xs = xs

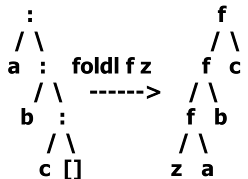
foldr (:) ys xs = xs++ys

definujte foldl pomocou foldr, alebo naopak:




```
myfoldl f z xs = foldr (\x -> \y -> (f y x)) z (myReverse xs)
myfoldr f z xs = foldl (\x -> \y -> (f y x)) z (myReverse xs)
```

(flip f)



- odstránime myReverse

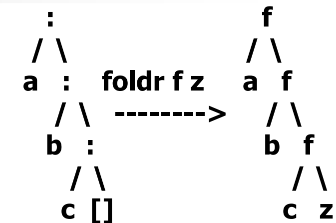
```
myReverse xs = foldr (\x -> \y -> (y ++ [x])) [] xs
myfoldl' f z xs = foldr (\x -> \y -> (f y x)) z
                        (foldr (\x -> \y -> (y ++ [x])) [] xs)
```

- odstránime ++

```
xs ++ ys = foldr (:) ys xs
myfoldl'' f z xs = foldr (\x -> \y -> (f y x)) z
                         (foldr (\x -> \y -> (foldr (:) [x] y)) [] xs)
```

hmmm..., teoreticky (možno) zaujímavé, prakticky nepoužiteľné ...

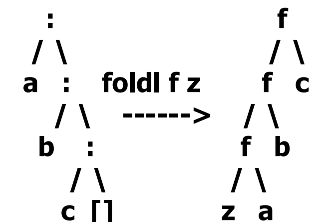
foldr a foldl posledný krát



Zamyslime sa, ako z foldr urobíme foldl:

induktívne predpokladajme, že rekurzívne volanie foldr nám vráti výsledok, t.j. hodnotu y, ktorá zodpovedá foldl:

- y = myfoldl f [b,c] = \z -> f (f z b) c



nech x je ďalší prvok zoznamu, t.j.

- x = a

ako musí vyzerať funkcia ??, ktorou "fold-r-ujeme", aby sme dostali myfoldl f [a,b,c] = \z' -> f (f (f z' a) b) c = ?? x y = ?? a (\z-> f (f z b) c)

- ?? = (\x y z' -> y (f z' x))

dosaďme:

- (\z' -> (\z -> f (f z b) c) (f z' a)) =
- (\z' -> f (f (f z' a) b) c) =
- \z' -> f (f (f z' a) b) c

Pre tých, čo neveria, fakt posledný krát

?? = (\x y z′ -> y (f z′ x))

- myfoldl''' f xs = foldr (\x y z -> y (f z x)) id xs
- myfoldl''' f [] = id
- myfoldl''' f [c] = (\x y z -> y (f z x)) c id = \z -> f z c
- myfoldl''' f [b,c] = (\x y z -> y (f z x)) b (\w -> f w c) =
\z -> (\w -> f w c) (f z b) =
\z -> f (f z b) c
- myfoldl''' f [a,b,c] = (\x y z -> y (f z x)) a (\w -> f (f w b) c) =
\z -> (\w -> f (f w b) c) (f z a) =
\z -> f (f (f z a) b) c
- myfoldl '''' f z xs = foldr (\x y z -> y (f x z)) id xs z

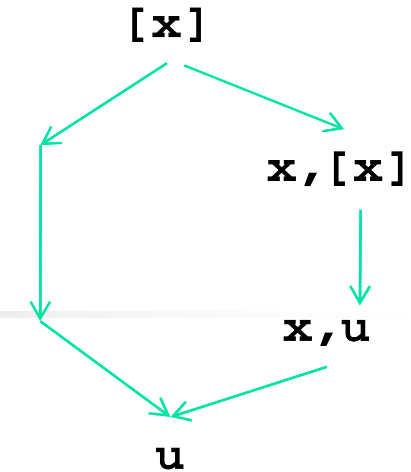
Domáca úloha 1: skúste vyjadriť foldr pomocou foldl ...

Schémy, čo sme ešte nevideli

- cata-morfizmus
- ana-morfizmus
- hylo-morfizmus
- para-morfizmus
- ...

- insertsort
- bublesort
- mergesort
- quicksort

Catamorphism



cata (*downwards as catastrophe*)
catamorphism alias bananas (| |), foldr

`data [x] = [] | x:[x]`

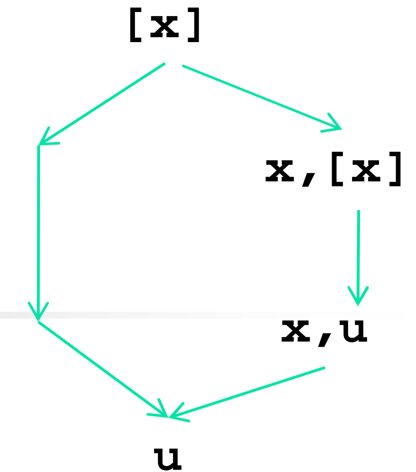
Cata typu T je funkcia T -> U:

- deštruuj vstup typu T,
- rekurzívne volaj cata na všetky podštruktúry typu T,
- výsledky skombinuj do výsledku typu U.

list-catamorphism [x] -> U:

- deštruuj [x] na a::x, as::[x],
- ak sa to nepodarí (lebo []), výsledok musí byť typu U,
- rekurzívne volaj list-cata na as::[x],
- rekurzívny výsledok ::u skombinuj s a::x a do výsledku typu U.

List-catamorphism




```
list_cata1            :: (x->u->u) -> u ->[x] -> u

list_cata1 f z []      = z
list_cata1 f z (a:as) = f a (list_cata1 f z as)
```

Príklad:

```
length1  = list_cata1 g 0
           where g x y = y+1

length1'  = list_cata1 (\x y -> y+1) 0
length1 = (| (\x y -> y+1), 0 |) – bananas ☺
length1'' = foldr (\x y -> y+1) 0
```

```
length1 [1..10]
10
length1' [1..10]
10
length1'' [1..10]
10
```

Iný príklad:

```
filter1 p  = list_cata1 g []
           where g a as    | p a            = (a:as)
                           | otherwise      = as

filter1' p = list_cata1 (\a as -> if (p a) then (a:as) else as) []
```

```
filter1 (even) [1..10]
[2,4,6,8,10]
filter1' (odd) [1..10]
[1,3,5,7,9]
```

List-catamorphism – príklady

```
fact1 n = list_cata1 (*) 1 [1..n]

map1 f   = list_cata1 g []
           where g a as = (f a:as)
```

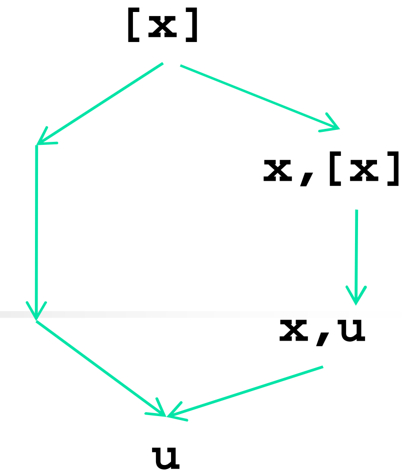
```
fact1 10
3628800

map1 (*2) [1..10]
[2,4,6,8,10,12,14,16,18,20]
```

Domáca úloha-2:
Definujte funkciu unzip :: [(a,b)] -> ([a],[b]) použitím list_cata1, aby unzip $[(a_1,b_1),(a_2,b_2),\ldots,(a_n,b_n)] = ([a_1,a_2, \ldots, a_n], [b_1,b_2, \ldots, b_n])$.

Domáca úloha-3:
Definujte funkciu average :: [Float] -> Float použitím list_cata1, aby average $[a_1,a_2, \ldots, a_n] = (a_1+a_2+ \ldots + a_n)/n$.

záver: list_cata1 je foldr tak, ako ho poznáme...

List-algebra




```
type List_alg x u   = (u, x->u->u)
list_cata2          :: List_alg x u ->[x] -> u

list_cata2 (z,f) = cata where
     cata []           = z
     cata (a:as)       = f a (cata as)

length2  = list_cata2 (0, g)
                         where g x y = y+1

filter2 p  = list_cata2 ([], g)
                         where g a as | p a          = (a:as)
                                      | otherwise = as

fact2 n = list_cata2 (1, (*)) [1..n]
```

Domáca úloha-4: Definujte reverse pomocou list_cata1, resp. list_cata2.

Identity Law

list_cata2 ([], (:)) = id

inými slovami:

foldr (:) [] = id

Fussion Law

f . list_cata1 (z1, g1) = list_cata1 (z2, g2)

inými slovami:

f . foldr g1 z1 = foldr g2 z2

ak platí

f z1 = z2 && f (g1 a b) = g2 a (f b)

Domáca úloha-5: dôkaz indukciou vzhľadom na dĺžku zoznamu...

Príklad na použitie Fussion Law:

(n*). list_cata2 (0,(+)) = list_cata2 (0, (+).(n*))

inými slovami:

(n*). foldr (+) 0 = foldr (+).(n*) 0

Dôkaz (pomocou Fussion Law): overíme predpoklady

(n*) 0 = 0

(n*) (a+b) = (n*a + n*b) = (+).(n*) a ((n*) b)

Acid Rain (fold/build theorem)

(deforestation)

list_cata2 (z,f) . g (:) [] = g f z

inými slovami
foldr f z . g (:) [] = g f z

Intuícia: Keď máme vytvoriť zoznam pomocou funkcie g zo zoznamových konštruktorov (:) [], na ktorý následne pustíme foldr, ktorý nahradí (:) za f a [] za z, namiesto toho môžeme konštruovať priamo výsledný zoznam pomocou g f z.

Otypujme si to:

Ak z :: u, potom f :: x->u->u, list_cata2 (z,f) :: [x]->u

Ľavá strana: ([x]->u).(t->[x]) … t->u

Pravá strana: g :: (x -> u -> u) -> u -> (t ->u) … t->u

length . map _ = length

foldr f z . g (:) [] = g f z

```
map :: (a -> b)  -> [a] -> [b]
map h = foldr ((:) . h) []                  -- (:).h a as = (:)(h a as) = h a: as
      = (\f z -> foldr (f . h) z) (:) []

length    :: [a] -> Int
length    = foldr (\ _ n -> n+1) 0
```

length . map h = length

```
(foldr (\ _ n -> n+1) 0) . (foldr ((:) . h) [])  =
```

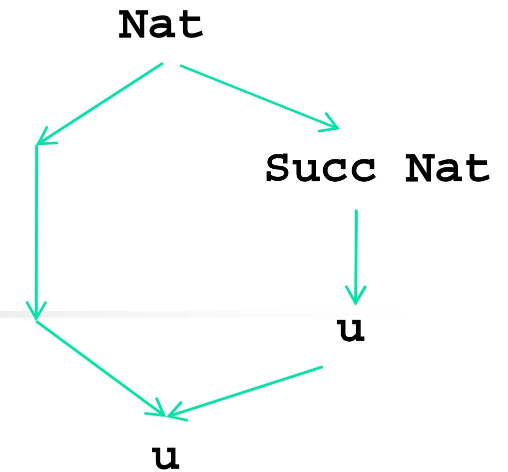
= podľa Acid Rain theoremy (čo je g ?)… g f z = (foldr (f . h) z)

```
foldr ((\ _ n -> n+1) . h) 0 = foldr ((\ _ n -> n+1) ) 0 = length
```

lebo

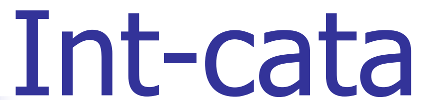
```
((\ _ n -> n+1) . h) x y = (\ _ n -> n+1) (h x) y = (\n -> n+1) y = y+1
```

Nat-cata




```
data Nat = Zero | Succ Nat
                      deriving(Show, Read, Eq)

nat_cata1 z f       = cata where
    cata n          = case n of
           Zero     -> z
           Succ x   -> f (cata x)

msucc :: Nat -> Nat
msucc x = Succ x

plus m = nat_cata1 m msucc

plus' m = nat_cata1 m (\x -> Succ x)
```

```
plus (Succ Zero) (Succ Zero)
Succ (Succ Zero)
```



Int-cata

prepíšeme inak:

```
int_cata1 z f      = cata where
    cata n         = case n of
           0       -> z
           n+1     -> f (cata n)

int_plus m = int_cata1 m (+1)
```

Domáca úloha 6: pomocou nat/int-cata definujte

- int_mult,
- odd, resp. even,
- 2^n.

Domáca úloha 7: skúste formulovať Fussion, Acid Rain pre Nat-cata, nájdite príklad použitia

take

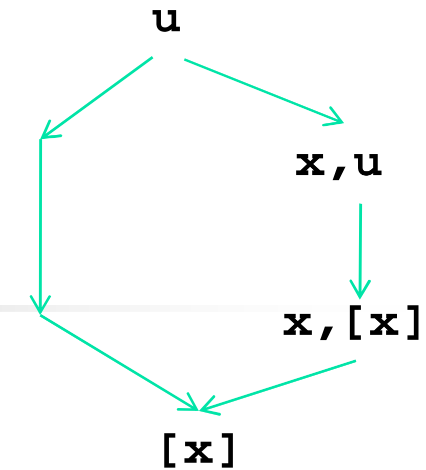
Výsledkom foldr f z je funkcia, do ktorej keď dosadíme, vypočíta výsledok

```
take' n xs = foldr (\a h -> \n -> case n of
                                 0 -> []
                                 (n+1) -> a:(h n) )
                   (\_ -> [])
                  xs
                  n
```

Cvičenie-8: Definujte xs!!n ako catamorphism.

Cvičenie-9: Definujte drop n ako catamorphism.

List-anamorphism



ana (upwards as anabolismus)
anamorphism alias lenses [(   )], unfold

data [x] = [] | x:[x]

list-anamorphism je funkcia typu U -> [x]:

- deštruuj U,
- komponenty typu U rekurzívne konvertuj do [x],
- skombinuj výsledky typu x a [x] do výsledku typu [x].

```
list_ana1:: (u->Bool) -> (u -> (x,u)) -> u -> [x]

list_ana1 p g b     | p b       = []       -- ak p b, tak končí rekuzia (terminátor)
                    | otherwise = a:list_ana1 p g b'
                    where (a,b') = g b             -- g je nextor
                    --     a je prvok do výsledného zoznamu
                    --          b' je argument na rekurzívne volanie list_ana1
```

list_ana1:: (u->Bool) -> (u -> (w,u)) -> u -> [w]

Zip

Definujme zip1 :: ([x],[y]) -> [(x,y)] pomocou list_ana1:

ujasnime si typy...

- u = ([x],[y])
- w = (x,y)
- p :: ([x],[y]) -> Bool
- g :: ([x],[y]) -> ((x,y) , ([x],[y]))

zip1 ([1,2,3],[11,22,33])
[(1,11),(2,22),(3,33)]

```
zip1        :: ([x],[y]) -> [(x,y)]
zip1        = list_ana1 p g    where
            p ([],[])           = True            -- True, ak končí rekuzia
            p ([],_)            = True
            p (_,[])            = True
            p (_,_)             = False
            g (a:as, b:bs)      = ((a,b), (as,bs))
                       -- (a,b) je prvok do výsledneho zoznamu
                       --  as,bs je argument na rekurzívne volanie list_ana1
```

list_ana1:: (u->Bool) -> (u -> (w,u)) -> u -> [w]

iterate, map

Definujte iterate1 :: (x -> x) -> x -> [x] pomocou list_ana1, aby
iterate1 f a = [a, f a, f f a, f f f a, ...]

ujasnime si typy:

- u = w = x
- p :: (x -> Bool) - terminátor
- g :: x -> (x,x) - nextor

take 10 (iterate1 (+1) 5)
[5,6,7,8,9,10,11,12,13,14]

```
iterate1 f = list_ana1 p g
          where
          p _        = False              -- never ending...
          g a        = (a,f a)            --    a je prvok do výsledneho zoznamu
                     --    (f a) je argument na rekurzívne volanie list_ana1
map2 f = list_ana1 p g
          where
                     p []        = True
                     p _         = False
                     g (a:as)  = (f a, as)  -- f a je prvok do výsledneho zozn.
                     --    as je argument na rekurzívne volanie list_ana1
```

List-coalgebra

Either je súčet typov (union type) s tagmi Left a Right:

```
data Either x y      = Left x | Right y

type List_coalg b a = b -> Either () (a, b)  -- dosadíme Left () | Right (a, b)
```

List_coalg je p (terminátor) aj g (nextor) dokopy:

- ak list_coalg b = Left _, tak p b = True,
- ak list_coalg b = Right(a, b'), tak (a, b') = g b

```
list_ana2:: List_coalg b a -> b -> [a]
list_ana2 lca         = ana where
    ana u = case lca u of
            Left _         -> []               -- vtedy končíme, lca u = Left _
            Right (a,b') -> a : ana b'    -- lca u = Right (a,b’), t.j. g b = (a,b’)
```

list_ana2:: List_coalg b a -> b -> [a]

List-coalgebra

```
destruct_count    0          = Left ()                  – kedy končíme
destruct_count    n          = Right (n,n-1)            – n ide do výsledku
                                                        – na n-1 ide rekurzia


count = list_ana2 destruct_count                        – downfrom
    where
        destruct_count    0          = Left ()
        destruct_count    n          = Right (n,n-1)
```

count 5
[5,4,3,2,1]

List-coalgebra’

```
type List_coalg b a = b -> Either () (a, b) ==== b -> Left () | Right (a,b)

data Either' x        = True' | False' x – predefinujem náš vlastný Either

type List_coalg' b a = b -> Either' (a, b)
type List_coalg' b a = b -> Either' (a, b) = b -> True' | False' (a,b)

list_ana2'            :: List_coalg' b a -> b -> [a]
list_ana2' lca        = ana where
    ana u = case lca u of
            True'          -> []
            False' (a,b') -> a : ana b'

count' = list_ana2' destruct_count
    where
            destruct_count    0           = True'
            destruct_count    n           = False' (n,n-1)
```

prime

Definujte prime ako anamorphismus

```
prime = sieve [2..] where
    sieve = list_ana2 destruct_prime where
            destruct_prime (x:xs) = Right(x, [y | y <-xs, y `mod` x > 0])

prime' = sieve [2..] where
    sieve = list_ana2 ' destruct_prime where
            destruct_prime (x:xs) = False '(x, [y | y <-xs, y `mod` x > 0])
```

take 100 prime
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97,101,103,107,109,113,127,131,137,139,149,151,157,163,167,173,179,181,191,...]

Cvičenie-8: Definujte [1,2,4,8,...,2^n, ...] ako anamorphism.
Cvičenie-9: Definujte [1,3,7,15,..., 2^n-1, ...] ako anamorphism.
Cvičenie-10: Definujte [1,2, 6, 24,..., n!, ...] ako anamorphism.

insert-sort

Definujte insert-sort ako list-catamorphismus (foldr)

```
insert_sort x        = list_cata2 ([], insert) x
    where
    insert a []      = [a]
    insert a (x:xs) | a < x    = a:x:xs
                    | otherwise = x : insert a xs

insert    :: Int -> [Int] -> [Int]

insert_sort'        :: [Int] -> [Int]
insert_sort'        = foldr insert []
```

```
insert_sort [2,2,1,34,5,2,4,23,2,4]
[1,2,2,2,2,4,4,5,23,34]
```

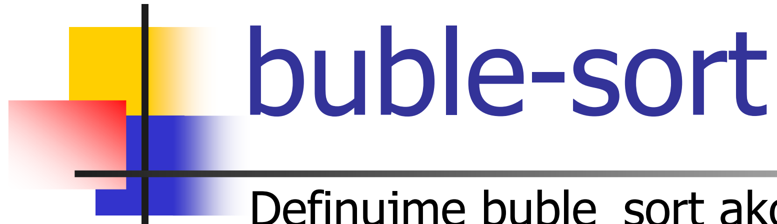
min-sort

Definujte min-sort ako list-anamorphismus

```
selection_sort extract = list_ana2 select
    where
    select []          = Left ()
    select x           = Right (extract x)

extra (x:xs)           = (x,xs)

min_sort xs = selection_sort extract_min xs
    where
    extract_min ls = (m, remove m ls)
        where
        m = minimum ls
        remove x []          = []
        remove x (y:ls)      | x == y = ls
                             | otherwise = y : remove x ls
```

```
selection_sort extra [4,3,2,4,2,1,3]
[4,3,2,4,2,1,3]
```

```
min_sort [4,3,2,4,2,1,3]
[1,2,2,3,3,4,4]
```



buble-sort

Definujme buble_sort ako list-catamorphismus

```
buble_sort ls          = selection_sort extract_buble ls
     where
     extract_buble [x]          = (x, [])
     extract_buble (x:ls)       =  if x < y then (x,y:m) else (y,x:m)
                                   where
                                   (y,m) = extract_buble ls
```

buble_sort [4,3,2,4,2,1,3]
[1,2,2,3,3,4,4]

```
buble_sort' ls         = selection_sort extract_buble ls
     where
     extract_buble (x:ls)       = list_cata2 ((x,[]), bub) ls
     bub x (y,ls)      = if x < y then (x,y:ls) else (y,x:ls)
```

buble_sort' [4,3,2,4,2,1,3]
[1,2,2,3,3,4,4]