Моноиды

Моноидом называется тройка (M, #, u), где # - ассоциативная бинарная операция на M, а u - ее единичный элемент.

- (int, +, 0)
- (int, *, 1)
- (bool, &&, true)
- (bool, ||, false)
- (M, min, Top), где M - вполне упорядоченное множество, а Top - его максимальный элемент
- (списки, append, пустой список)
- (множества, объединение, пустое множество)
- (сортированные списки, merge, пустой список)
- (матрицы, умножение, единичная матрица)
- (эндоморфизмы, композиция, тождественная функция), где эндоморфизмом на множестве M называется функция из M в M
- (блоки кода, последовательное выполнение, пустой блок кода)

Моноиды

• Если M1,M2 - моноиды, то M1 x M2 - моноид (декартово произведение множеств, покомпонентное применение #, пара из двух единиц)

• Если # - ассоциативная бинарная операция на M, но единичного элемента среди M нет, то можно его добавить: (M+{e}, #', e), где x #' e = e #' x = x, x#'y = x#y.

• Если (M, #, e) - моноид, то (M, @, e) - тоже моноид, где x @ y = y # x - т.н. "двойственный" моноид.

• Если (M, #, e) - моноид, то для любого P тотальные функции P->M образуют моноид

```
      (P->M, \f1 f2 -> \p -> f1 p # f2 p, \p -> e)
```

Моноиды

- Например, если M - моноид блоков кода, то можно получить моноид (процедуры с параметром P, последовательное выполнение, пустая процедура)

- Аналогично - частичные функции или словари - моноид по объединению

Гомоморфизм моноидов

Если (M, #, e) и (P, @, u) – моноиды,

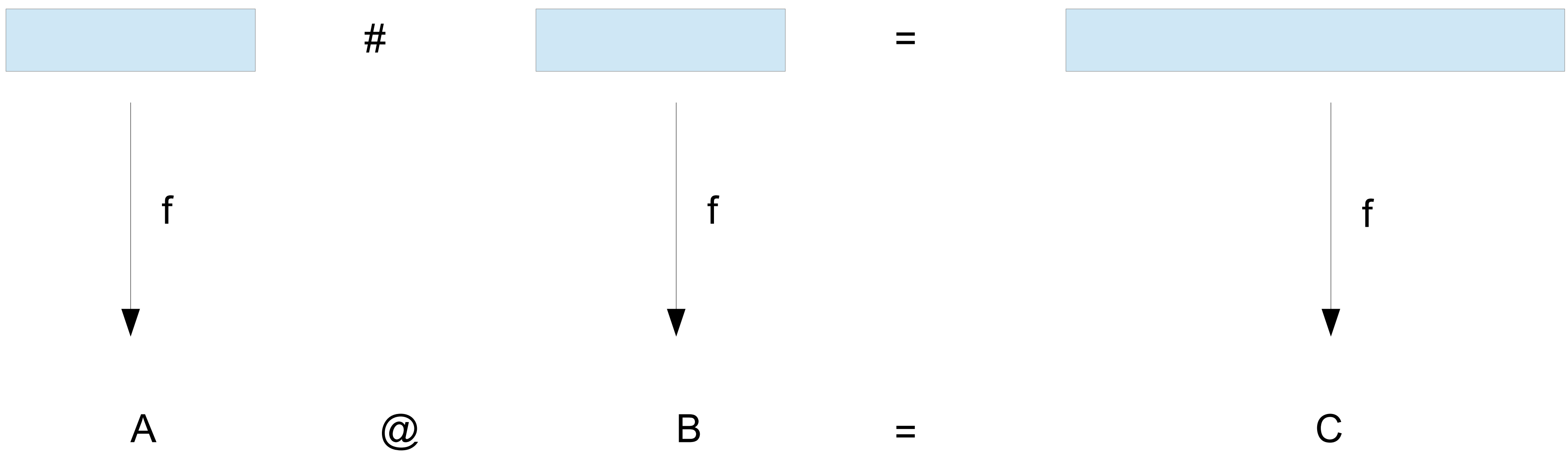
то функция f :: M -> P

называется гомоморфизмом между этими двумя моноидами,

если f (m1 # m2) = f m1 @ f m2 для всех m1,m2 из M.

чему равно u ?

Гомоморфизм моноидов



Списочный гомоморфизм

Списочным гомоморфизмом называется гомоморфизм из моноида (списки, append, пустой список) в какой-либо другой моноид.

То есть, функция f называется списочным гомоморфизмом, если существует оператор #, такой, что f (xs ++ ys) = f xs # f ys. Это свойство позволяет независимо вычислить результаты применения функции для подсписков, и собрать из них результат для всего списка при помощи #.

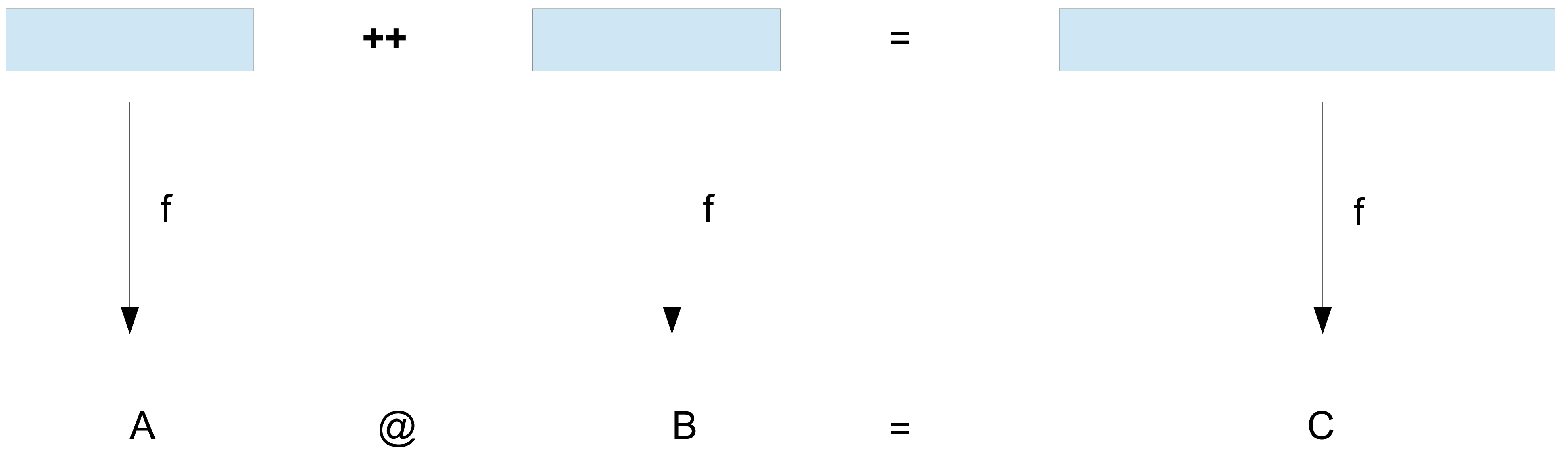
Списочный гомоморфизм

	#	u	m
sum	+	0	\x → x
length	+	0	\x → 1
filter p	++	[]	\x → if (p x) then [x] else []
map f	++	[]	\x → f x
sort	merge	[]	\x → [x]

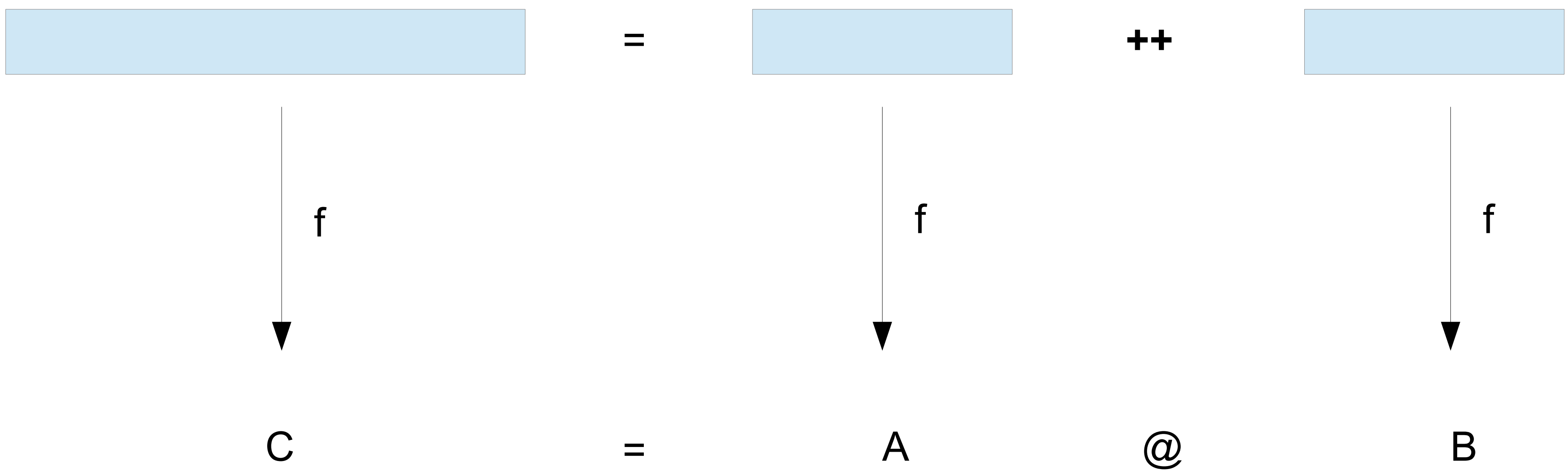
Списочный гомоморфизм

Чрезвычайно важно, что благодаря ассоциативности @, в выражении x1 @ x2 @ x3 @ ... @ xn можно расставлять скобки как угодно, вычисляя его в любом порядке (надо, однако, помнить, что @ не обязан быть коммутативным).

Фича гомоморфизма на списках



Фича гомоморфизма



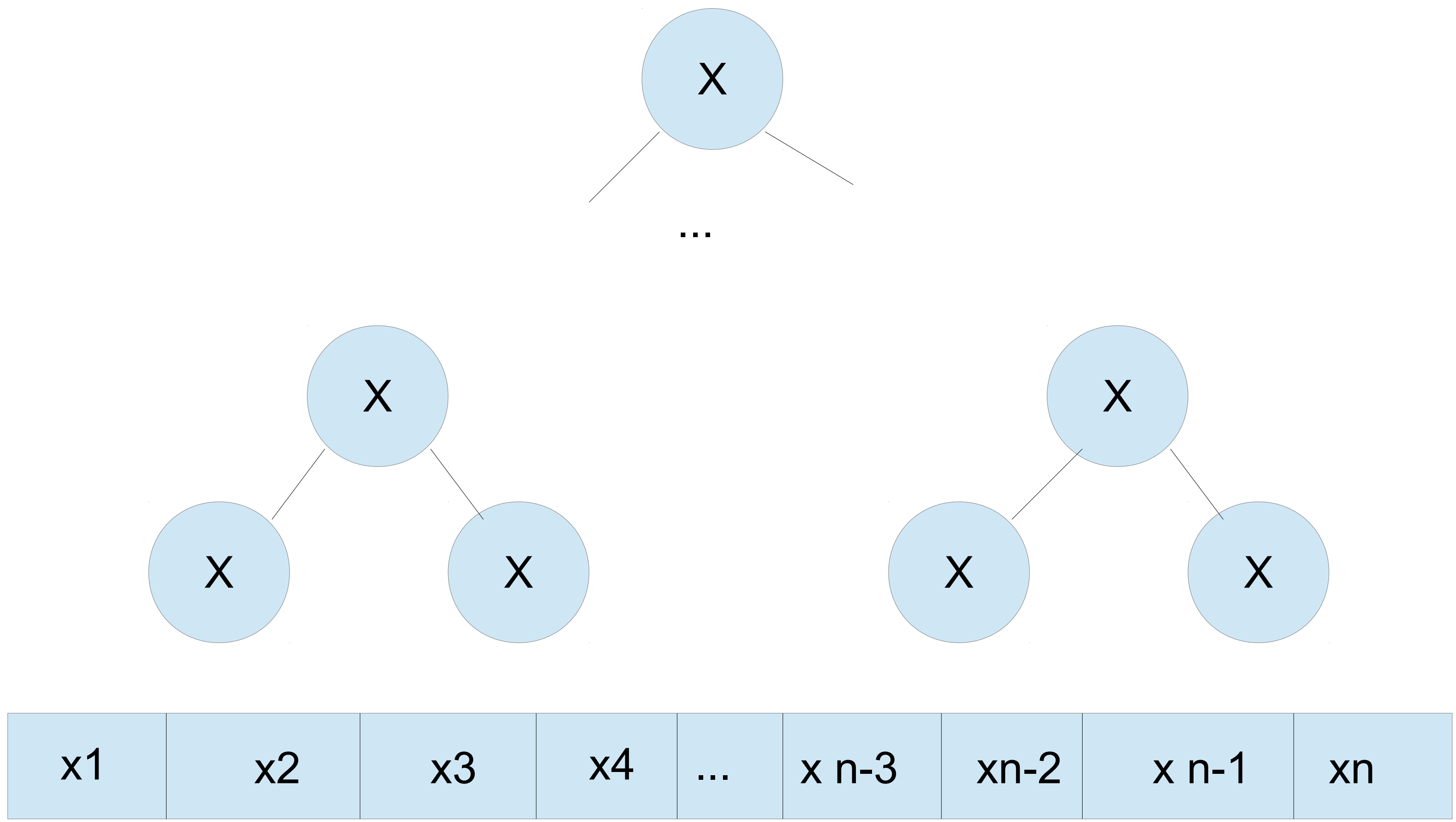
Гомоморфизм

```
f ~ (map, reduce)

f [x] = map x

reduce A::[] B::[] = A @ B
```

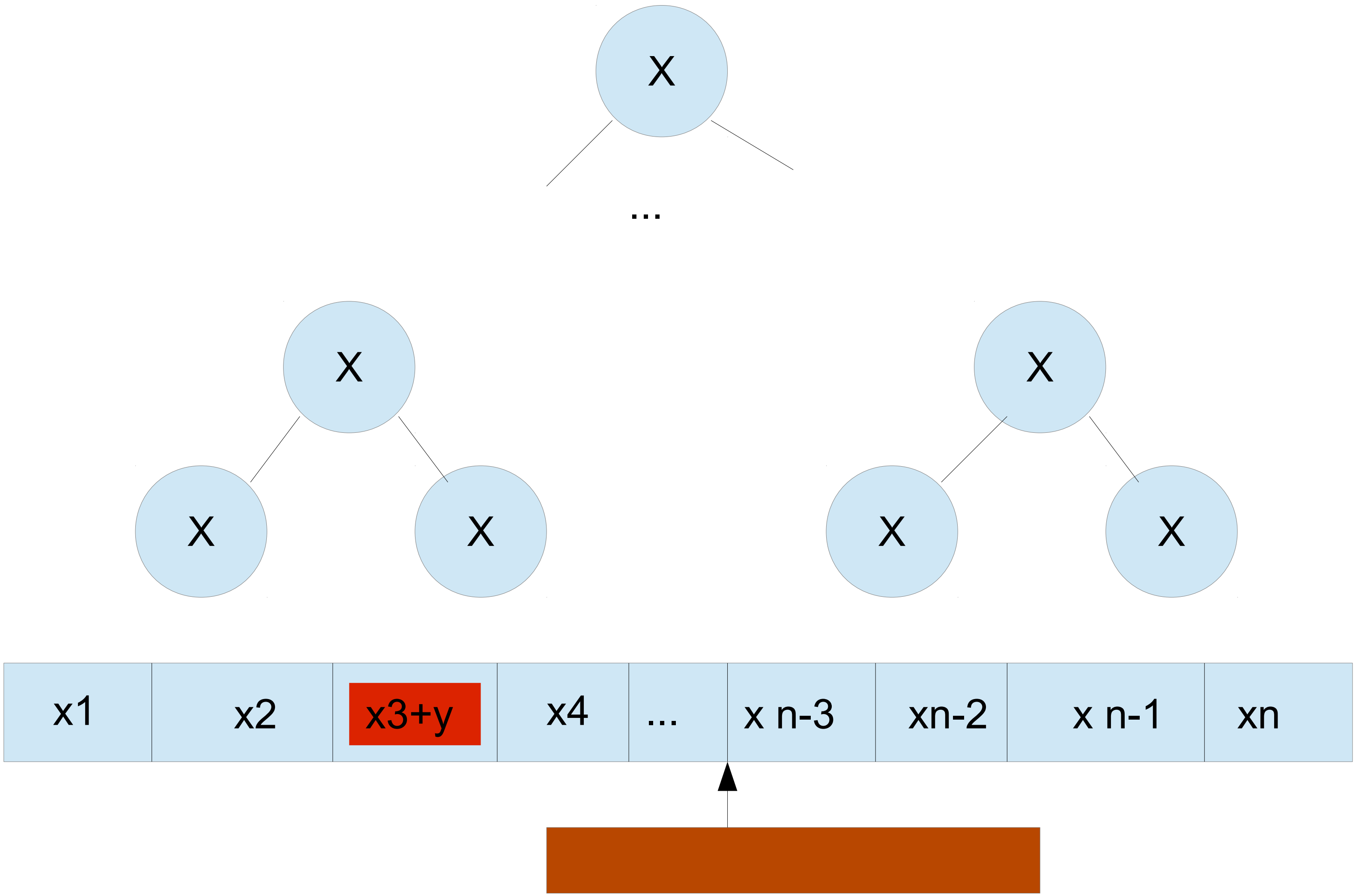
Гомоморфизм



Гомоморфизм

1) Позволяет вычислить ответ для N элементов за log N фаз (соответствующих уровням дерева), каждая из которых может быть распараллелена. На P процессорах можно ускорить программу в O(P/logP) раз.
2) Позволяет при изменении значения какого-нибудь элемента перевычислить ответ для всего списка за O(log N) операций, изменив только элементы по пути от измененного к корню.

Гомоморфизм



Списочный гомоморфизм

```
regEx = /^(a+ b* c)*$/

s0 --a--> s1
s1 --a--> s1
s1 --b--> s2
s1 --c--> s0 // bingo
s2 --b--> s2
s2 --c--> s0 // bingo
s? --?--> s3
```

Списочный гомоморфизм

```
regEx = /(a+ b* c)*/
```

	a	b	c
s0	s1	s3	s3
s1	s1	s2	s0
s2	s3	s2	s0
s3	s3	s3	s3

Списочный гомоморфизм

```
regEx = /(a+ b* c)*/
```

	a	b	c
s0	s1	s3	s3
s1	s1	s2	s0
s2	s3	s2	s0
s3	s3	s3	s3

`“aab”, “ab”, “ac” “c”`

эндоморфизм на S

Списочный гомоморфизм

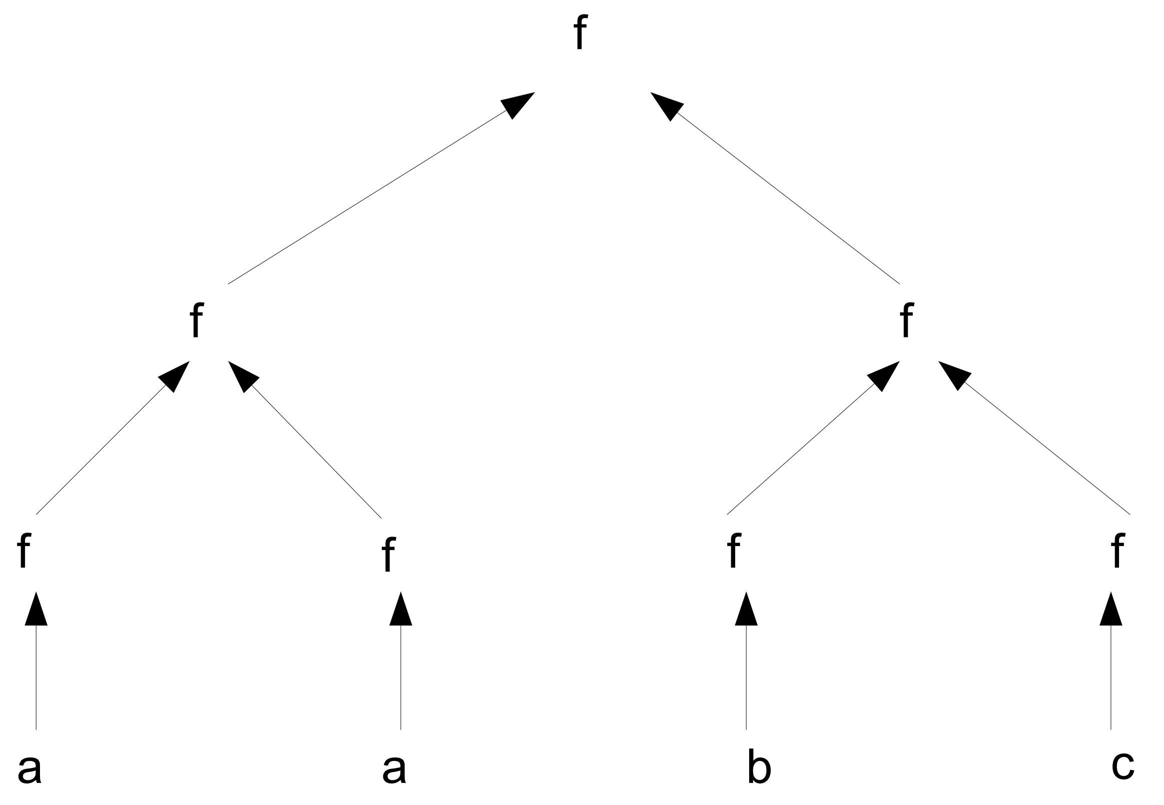
```
map x = \s-> fx s  -- fx – функция-столбец
reduce f g = f o g
```

	a	b	c
s0	s1	s3	s3
s1	s1	s2	s0
s2	s3	s2	s0
s3	s3	s3	s3

`“aab”, “ab”, “ac” “c”`

эндоморфизм на S

Списочный гомоморфизм



Списочный гомоморфизм

[Третья теорема о гомоморфизмах] Если функция f выражается и в виде левой, и в виде правой свертки с одинаковым начальным значением (но, возможно, разной операцией #), то она является списочным гомоморфизмом - и, следовательно, допускает параллельное и инкрементальное вычисление с помощью дерева.

Списочный гомоморфизм

Haskell, fold, в один поток...

Классы типов

Равно есть везде, но равны ли наши деревья?

let a = list2tree [1,5,3,2]

a == a?

полиморфизм

Классы типов

Классы типов

```
class Eq a where
  (==) :: a -> a -> Bool
```

Как это работает?

```
(==) :: (Eq a) => a -> a -> Bool

instance Eq Integer where
  x == y = x ‘integerEq‘ y
```

Классы типов

```
data Tree a = EmptyTree | Node a (Tree a) (Tree a)
  deriving (Show, Read)

instance (Eq a) => Eq (Tree a) where
  EmptyTree == EmptyTree = True
  (Node a l1 r1) == (Node b l2 r2) =
    (a==b) && (l1 == l2) && (r1 == r2)
  _ == _ = False
```

Деревья

```
singleton :: a -> Tree a
singleton x = Node x EmptyTree EmptyTree

treeInsert :: (Ord a) => a -> Tree a -> Tree a
treeInsert x EmptyTree = singleton x
treeInsert x (Node a left right)
  | x == a = Node x left right
  | x < a = Node a (treeInsert x left) right
  | x > a = Node a left (treeInsert x right)

list2tree :: (Ord a) => [a] -> Tree a
list2tree = l2t EmptyTree
  where
    l2t acc [] = acc
    l2t acc (head:tail) = l2t (treeInsert head acc)
tail
```

Классы типов

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  x /= y = not (x == y)
```

Классы типов

```
class (Eq a) => Ord a where
  (<), (<=), (>=), (>) :: a -> a -> Bool
  max, min :: a -> a -> a
```

Классы типов

```
#==# - списки равны или являются идентичными по
перевороту:

[1,2,3] #==# [1,2,3] -- True
[1,1,1] #==# [1,1,2] -- False
[1,2,3] #==# [3,2,1] -- True
```

Классы типов

```
class Itch a where
  (#==#) :: a -> a -> Bool

instance (Eq a) => Itch [a] where
  [] #==# [] = True
  x #==# y = (x == reverse y) || (x == y)
```

Классы типов

```
lol a b = a #==# b

> :t lol
lol :: Itch a => a -> a -> Bool
```

Классы типов

```
#==# - списки являются одинаковыми множествами (Set):

[1,2,3] #==# [1,2,3] -- True
[1,1,1] #==# [1,1,2] -- False
[1,2,3] #==# [3,2,1] -- True
[1,2,2] #==# [2,1,2] -- True
[1,2,2] #==# [1,2] -- True
[1,2,15] #==# [13] -- False
```

Класс моноидов

```
class  Monoid m  where
  mappend   :: m -> m -> m
  mempty :: m

  mconcat :: [m] -> m
```

Класс моноидов

```
mappend mempty x = x
mappend x mempty = x
mappend x (mappend y z) = mappend (mappend x y) z
mconcat = foldr mappend mempty
```

Класс моноидов

Data.Monoid

Списки

```
instance Monoid [a] where
    mappend = (++)
    Mempty = []
```

Числа

```
instance Monoid Integer where
    mappend = (+)
    Mempty = 0

instance Monoid Integer where
    mappend = (*)
    mempty = 1
```

Класс моноидов

Обёртки

```
Num a => Monoid (Sum a)

newtype Sum a = Sum { getSum :: a }
        deriving (Eq, Ord, Read, Show, Bounded)

Num a => Monoid (Product a)

newtype Product a = Product { getProduct :: a }
        deriving (Eq, Ord, Read, Show, Bounded)

mconcat [Sum1, Sum 2, Sum 3, Sum 4] = Sum 10
```

На самом деле, Sum {getSum = 10}, но это неважно! **newtype**

```
mconcat [Product 1, Product 2, Product 3, Product 4] =
[Product 24]
```

Класс моноидов (writer)

Writer (аккумулятор)

```
newtype Writer w a = Writer { runWriter :: (a, w) }
```

В чём отличие data от newtype?

```
instance (Monoid w) => Monad (Writer w) where
    return x = Writer (x, mempty)
    (Writer (x,v)) >>= f = let (Writer (y, v')) = f x in Writer
(y, v `mappend` v')
```

```
:t tell
tell :: MonadWriter w m => w -> m ()
```

Класс моноидов (writer)

```
import Control.Monad.Writer

fact :: Integer -> Writer String Integer
fact 0 = return 1
fact n = do
  let n' = n-1
  tell $ show n ++ " - 1 \n"
  m <- fact n'
  tell $ "fact " ++ show m ++ "\n"
  let r = n*m
  tell $ show n ++ " * " ++ show m ++ "\n"
  return r

:t runWriter
runWriter :: Writer w a -> (a, w)

putStr $ snd $ runWriter $ fact 10
```

Класс моноидов (writer)

```
import Control.Monad.Writer

fact2 :: Integer -> Writer (Sum Integer) Integer
fact2 0 = return 1
fact2 n = do
  let n' = n-1
  tell $ Sum 1
  m <- fact2 n'
  let r = n*m
  tell $ Sum 1
  return r
```

Первая лаба

```
putStr $ snd $ runWriter $ fact2 10
```

Класс моноидов (writer)

```
import Control.Monad.State
fact3 :: Integer -> State Integer Integer
fact3 0 = return 1
fact3 n = do
  let n' = n-1
  modify (+1)
  m <- fact3 n'
  let r = n*m
  modify (+1)
  return r


runState (fact3 10) 0
```

Writer понятливее

Класс моноидов (writer)

Ещё два моноида: Any, All

```
import Control.Monad.Writer

fact4 :: Integer -> Writer Any Integer
fact4 0 = return 1
fact4 n = do
  let n' = n-1
  m <- fact4 n'
  let r = n*m
  tell (Any (r==120))
  return r

> runWriter $ fact 2
> runWriter $ fact 10
```

Коммутативные, некоммутативные, двойные моноиды

(+), (++)
Законы моноидов

```
fact5 :: Integer -> Writer (Dual String) Integer
fact5 0 = return 1
fact5 n = do
  let n' = n-1
  tell $ Dual $ show n ++ " - 1\n"
  m <- fact5 n'
  tell $ Dual $ "fact " ++ show m ++ "\n"
  let r = n*m
  tell $ Dual $ show n ++ " * " ++ show m ++ "\n"
  return r

let Dual a = snd $ runWriter $ fact 10
putStrLn a
```

Умножение моноидов

```
{- instance (Monoid a,Monoid b) => Monoid (a,b) where
    mempty = (mempty,mempty)
    mappend (u,v) (w,x) = (u `mappend` w,v `mappend` x) -}

tellFst a = tell $ (a,mempty)
tellSnd b = tell $ (mempty,b)

fact6 :: Integer -> Writer (String,Sum Integer) Integer
fact6 0 = return 1
fact6 n = do
  let n' = n-1
  tellSnd (Sum 1)
  tellFst $ show n ++ " - 1 \n"
  m <- fact6 n'
  let r = n*m
  tellSnd (Sum 1)
  tellFst $ show n ++ " * " ++ show m ++ "\n"
  return r
```

Снова свёртки

```
import Data.Foldable

data Tree a = EmptyTree | Node a (Tree a) (Tree a)
  deriving (Show, Read)

instance Foldable Tree where
  foldMap f EmptyTree = mempty
  foldMap f (Node k l r) = foldMap f l `mappend` f k `mappend`
    foldMap f r

let tree = list2tree [1,4,6,8]

foldMap (Any . (== 1)) tree
foldMap (All . (> 5)) tree
foldMap (All . even) tree
```

Снова свёртки

Найти минимальный и максимальный элемент дерева, сконструировав моноид.

```
min, max
maxBound::Int, maxBound::Float, minBound::Bool
Eq, Ord, Read, Show, Bounded

list2tree ([3,1,5,87,4]) :: (Num a, Ord a) => Tree a
list2tree ([3,1,5,87,4]::[Int]) :: Tree Int

foldMap ( ??? ) (tree::[Int])
```

Снова свёртки

```
newtype Max a = Max { getMax :: a }
        deriving (Eq, Ord, Read, Show, Bounded)

instance (Num a, Ord a, Bounded a) => Monoid (Max a) where
        mempty = minBound
        Max x `mappend` Max y = Max $ max x y
```

Итого

Интерфейс моноидов нужен для реализации алгоритмов, включая распараллеливание и поиск по дереву