

Практика 7. Свёртки

осень 2026

Андрей Стоян, Илья Колегов

осень 2026

Летучка 7

Летучка 7 · 7 минут · без конспектов · Ваше ФИ: _____

1. Кайнд конструктора типа `Maybe`: _____
2. Один шаг β -редукции: $(\lambda x. x) a \rightsquigarrow$ _____
3. Результат `foldr (-) 0 [1, 2]`: _____
4. Нейтральный элемент для `(++)` на списках: _____
5. Дополните кодирование списка Чёрча `[1, 2]`: $\lambda s z.$ _____

Полугруппы и моноиды

Список: полугруппа и моноид

✎ Напишите полные инстансы `Semigroup` и `Monoid` для списка в обёртке `Batch`. Объединение сохраняет порядок всех элементов.

```
newtype Batch a = Batch [a]
```

⚡Решение.

```
instance Semigroup (Batch a) where
    Batch xs <> Batch ys = Batch (xs ++ ys)

instance Monoid (Batch a) where
    mempty = Batch []
```

Статистика тестов: полугруппа и моноид

✎ Напишите полные инстансы `Semigroup` и `Monoid` для статистики тестов. При объединении блоков складываются число ошибок и общее число тестов.

```
data Counts = Counts { errors :: Int, total :: Int }
```

⚡Решение.

```
instance Semigroup Counts where
  Counts e n <> Counts e' n' = Counts (e + e') (n + n')

instance Monoid Counts where
  mempty = Counts 0 0
```

Maybe: полугруппа и моноид

✎ Напишите полные инстансы `Semigroup` и `Monoid` для своего варианта `Maybe`. `Missing` означает отсутствие результата; два `Present` объединяют содержащиеся значения. Выберите необходимые ограничения на `a`.

```
data Optional a = Missing | Present a
```

⚡Решение.

```
instance Semigroup a => Semigroup (Optional a) where
    Missing <> y = y
    x <> Missing = x
    Present x <> Present y = Present (x <> y)

instance Semigroup a => Monoid (Optional a) where
    mempty = Missing
```

Применение: статистика результатов

✎ Напишите `stats :: [Bool] -> Counts`, используя готовый инстанс: `True` — успешный тест, `False` — ошибка. Для `[True, False, True]` получите `Counts 1 3`, для пустого списка — `Counts 0 0`. Обойдитесь без явной рекурсии; пригодятся `map` и `mconcat`.

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

⚡Решение.

```
stats :: [Bool] -> Counts
stats = mconcat . map result
  where
    result ok = Counts (if ok then 0 else 1) 1
```

Свёртки списка

Один список, две свёртки

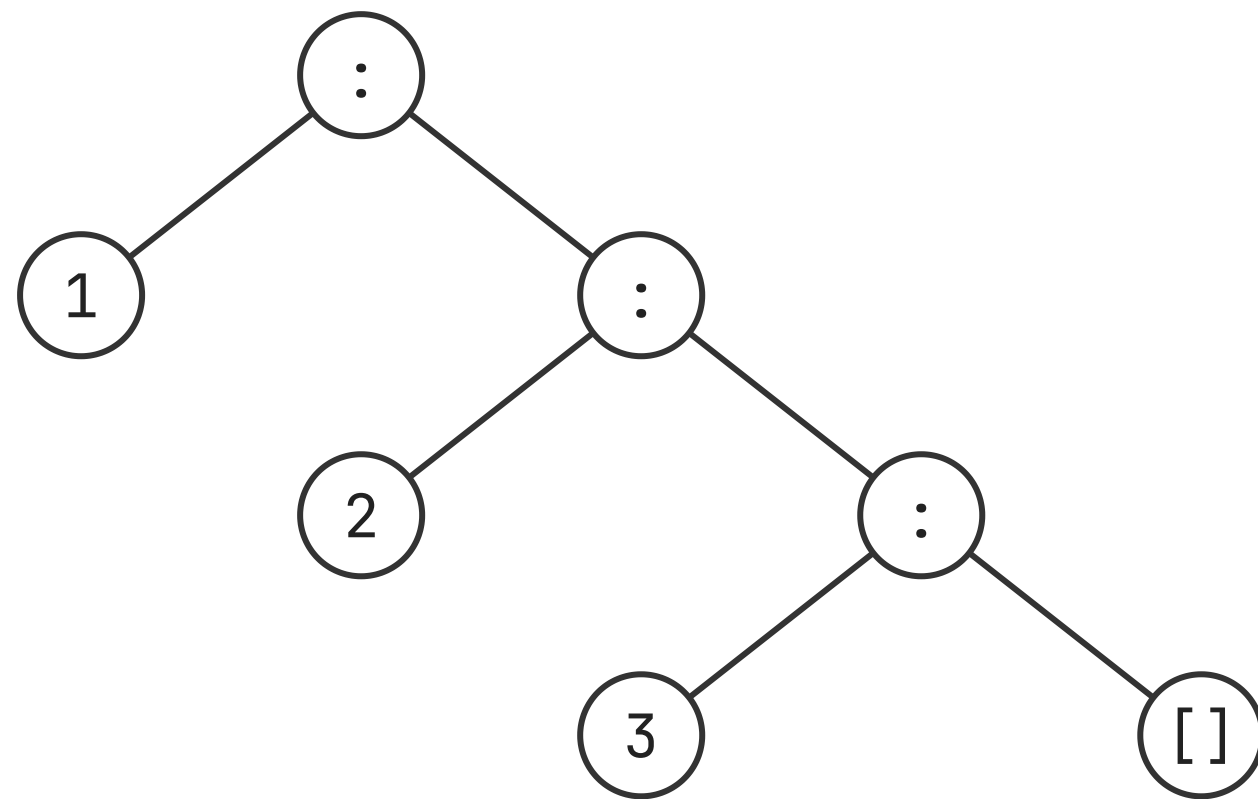
```
foldr :: (a -> b -> b) -> b -> [a] -> b  
foldl :: (b -> a -> b) -> b -> [a] -> b
```

 Расставьте скобки и вычислите `foldr (-) 0 [1,2,3]` и `foldl (-) 0 [1,2,3]`.

foldr: заменяем конструкторы

Список `[1,2,3]` — это `1 : (2 : (3 : []))`.

```
foldr (-) 0 [1,2,3]
```



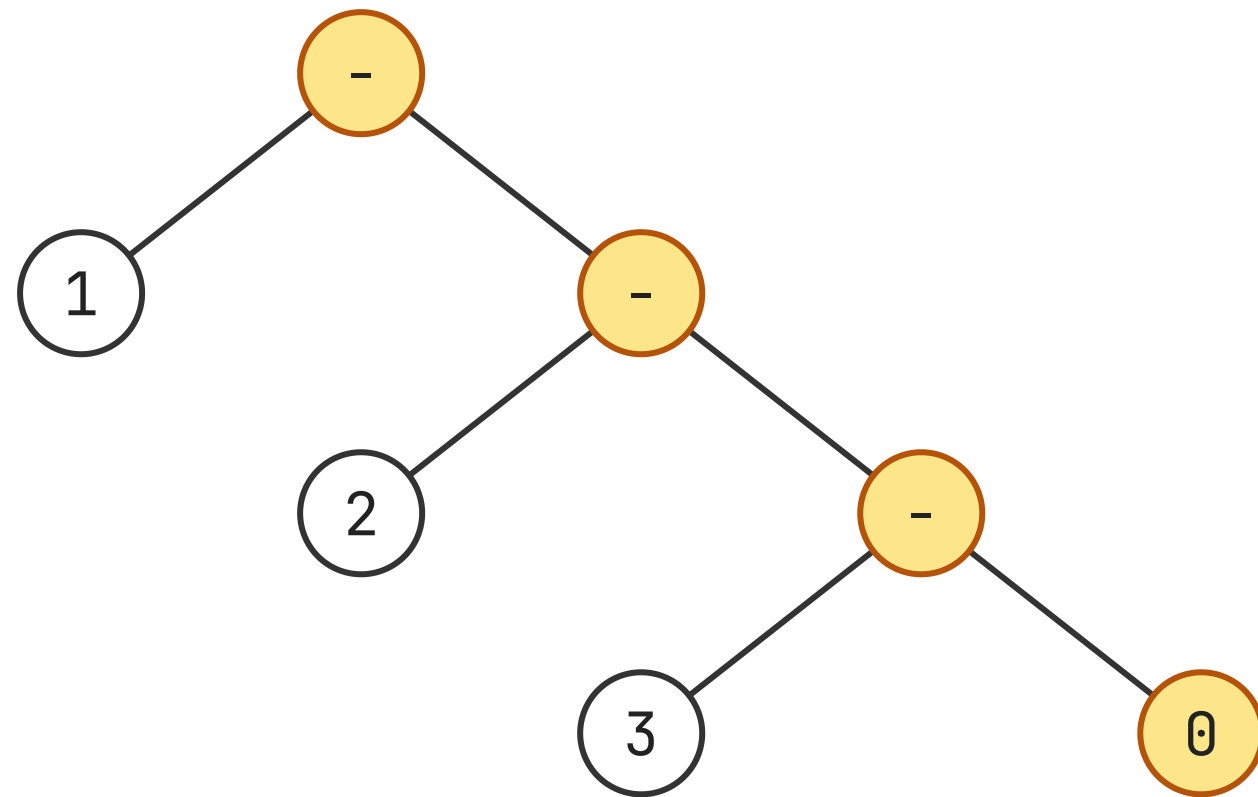
СКОБКИ

В каждом `(:)` два аргумента: элемент и хвост.

foldr: заменяем конструкторы

Вместо $(:)$ ставим $(-)$, вместо $[]$ — 0 .

$1 - (2 - (3 - 0))$



СКОБКИ

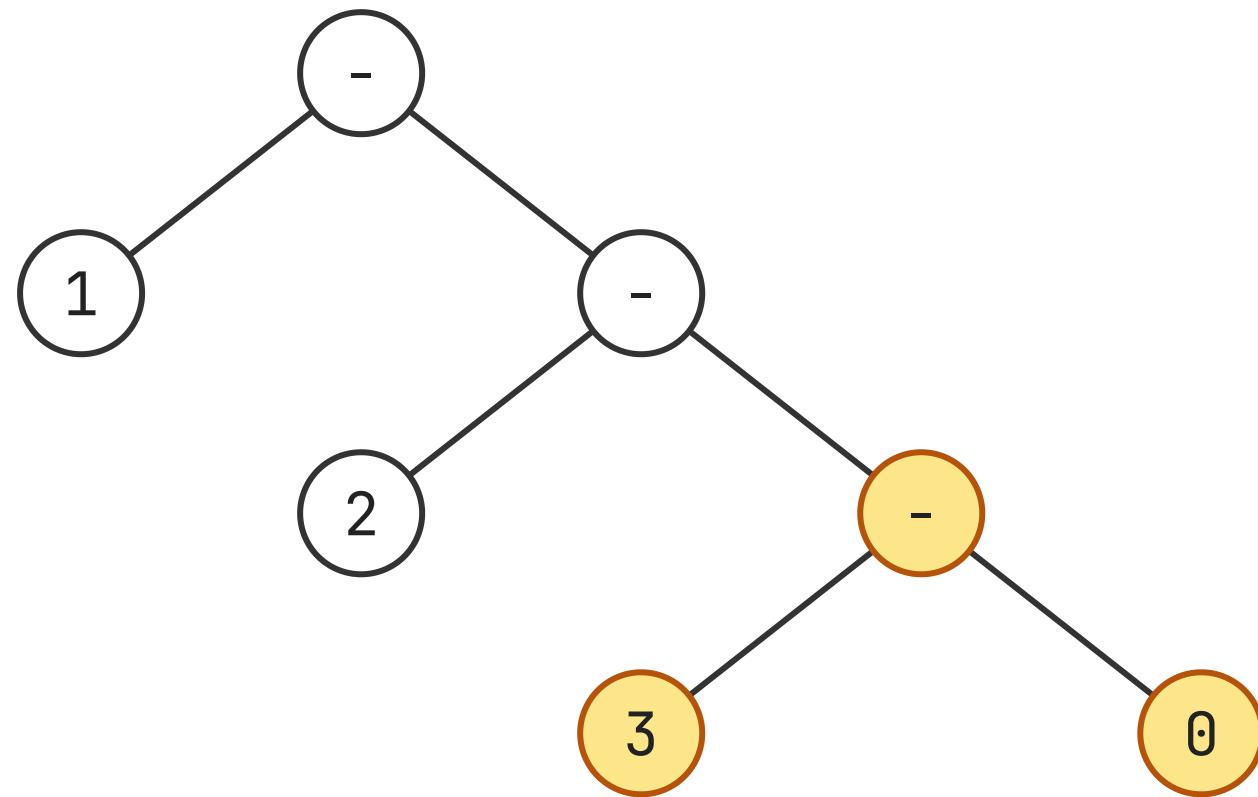
В каждом $(:)$ два аргумента: элемент и хвост.

Шаг получает элемент и результат свёртки хвоста.

foldr: заменяем конструкторы

Для этого выражения арифметика требует результат хвоста.

1 - (2 - (3 - 0))



СКОБКИ

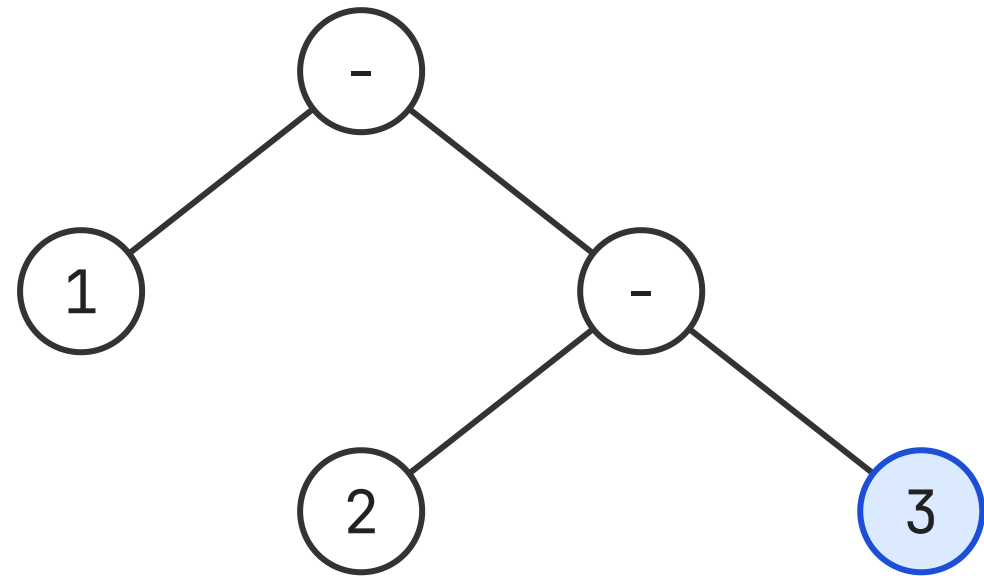
В каждом `(:)` два аргумента: элемент и хвост.

Шаг получает элемент и результат свёртки хвоста.

foldr: заменяем конструкторы

Заменяем только выделенное поддерево его результатом.

1 - (2 - 3)



СКОБКИ

В каждом $(:)$ два аргумента: элемент и хвост.

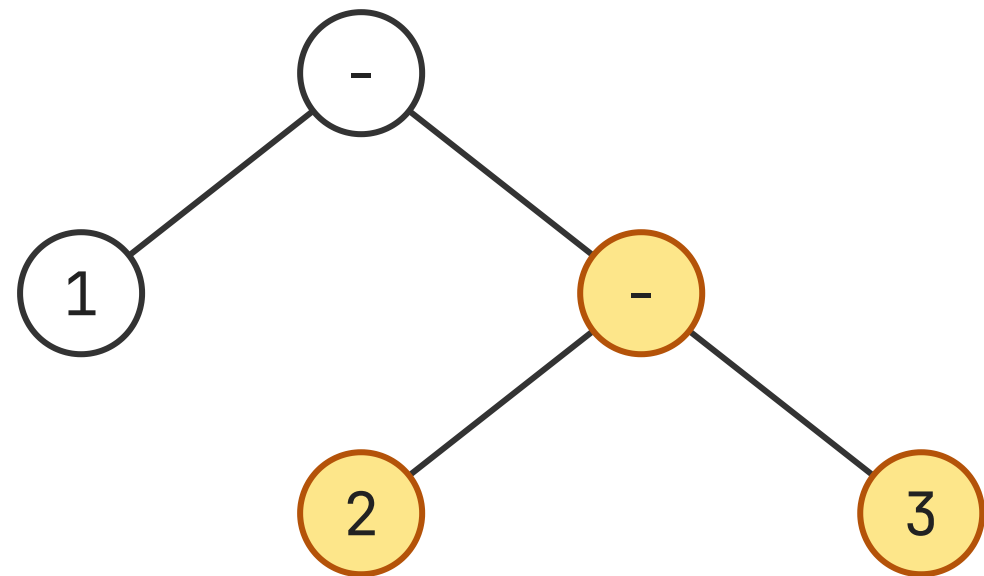
Шаг получает элемент и результат свёртки хвоста.

3 - 0 = 3.

foldr: заменяем конструкторы

Следующая операция — $2 - 3$.

$1 - (2 - 3)$



СКОБКИ

В каждом $(:)$ два аргумента: элемент и хвост.

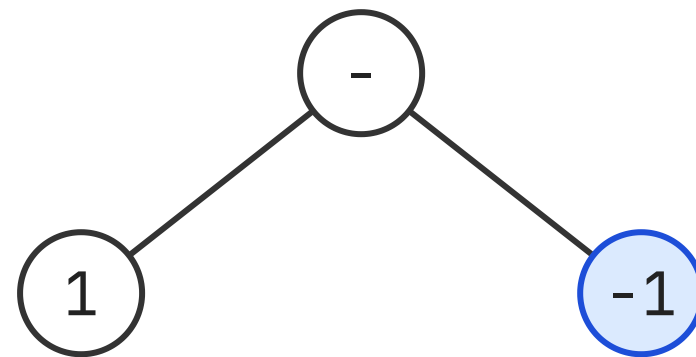
Шаг получает элемент и результат свёртки хвоста.

$3 - 0 = 3$.

foldr: заменяем конструкторы

Полученное -1 становится правым аргументом внешнего вычитания.

$1 - (-1)$



СКОБКИ

В каждом $(:)$ два аргумента: элемент и хвост.

Шаг получает элемент и результат свёртки хвоста.

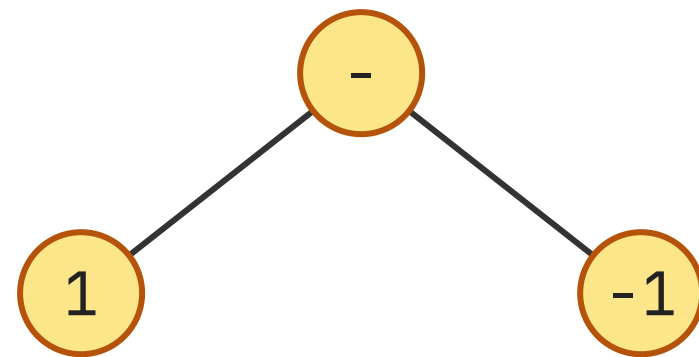
$3 - 0 = 3.$

$2 - 3 = -1.$

foldr: заменяем конструкторы

Осталось $1 - (-1)$.

$1 - (-1)$



СКОБКИ

В каждом $(:)$ два аргумента: элемент и хвост.

Шаг получает элемент и результат свёртки хвоста.

$3 - 0 = 3$.

$2 - 3 = -1$.

foldr: заменяем конструкторы

Возвращаем результат всей свёртки.

2

2

СКОБКИ

В каждом $(:)$ два аргумента: элемент и хвост.

Шаг получает элемент и результат свёртки хвоста.

$$3 - 0 = 3.$$

$$2 - 3 = -1.$$

$$1 - (-1) = 2.$$

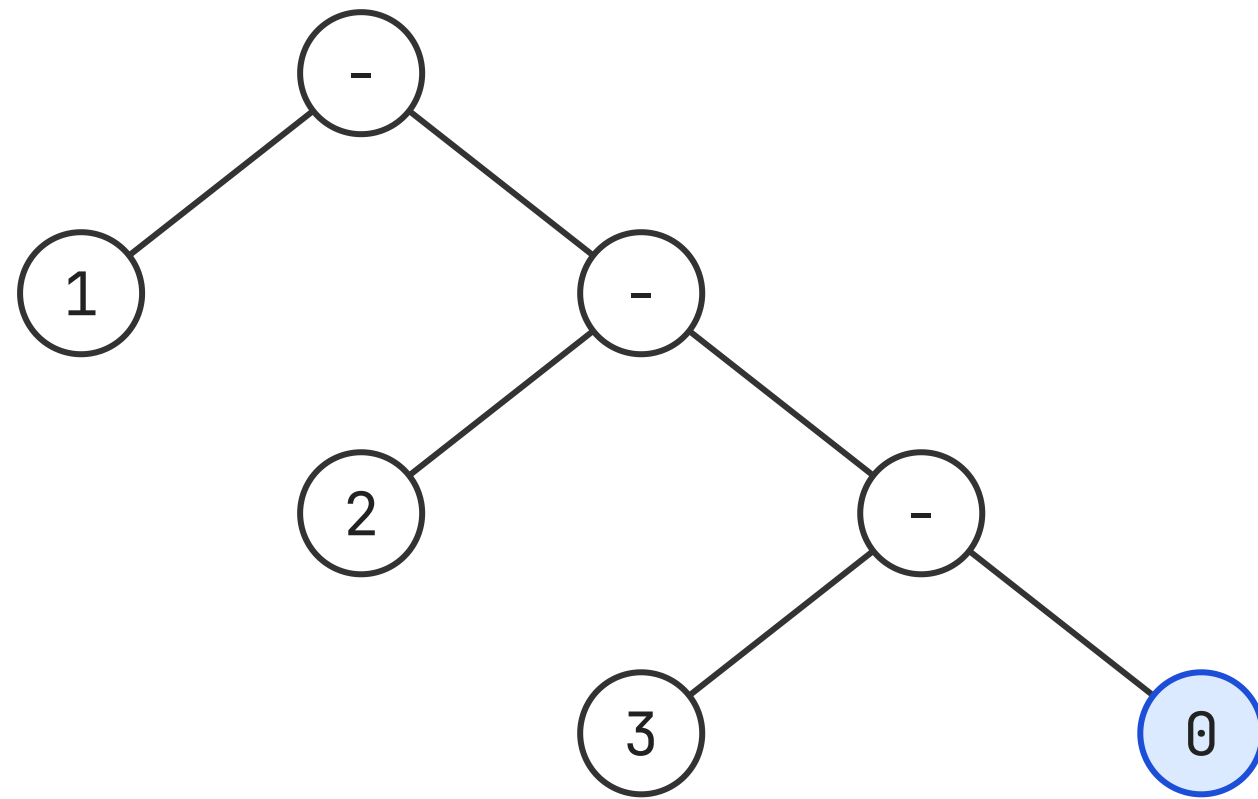
От foldr к foldl: дерево скобок

Правая свёртка: элемент слева, свёрнутый хвост справа.

```
foldr (-) 0 [1,2,3] = 1 - (2 - (3 - 0))
```

СКОБКИ

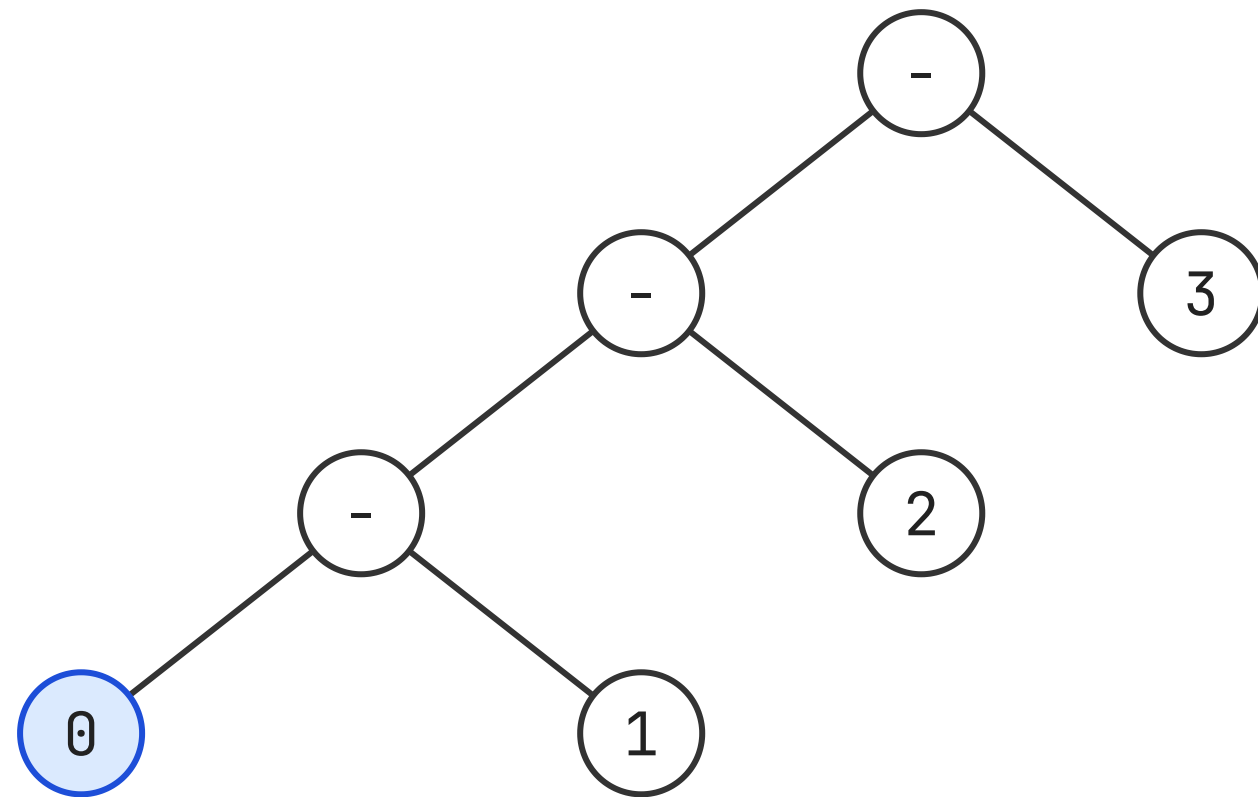
$1 - (2 - (3 - 0)) = 2.$



От foldr к foldl: дерево скобок

Левая свёртка: накопленный результат слева, следующий элемент справа.

```
foldl (-) 0 [1,2,3] = ((0 - 1) - 2) - 3
```



СКОБКИ

$$1 - (2 - (3 - 0)) = 2.$$
$$((0 - 1) - 2) - 3 = -6.$$

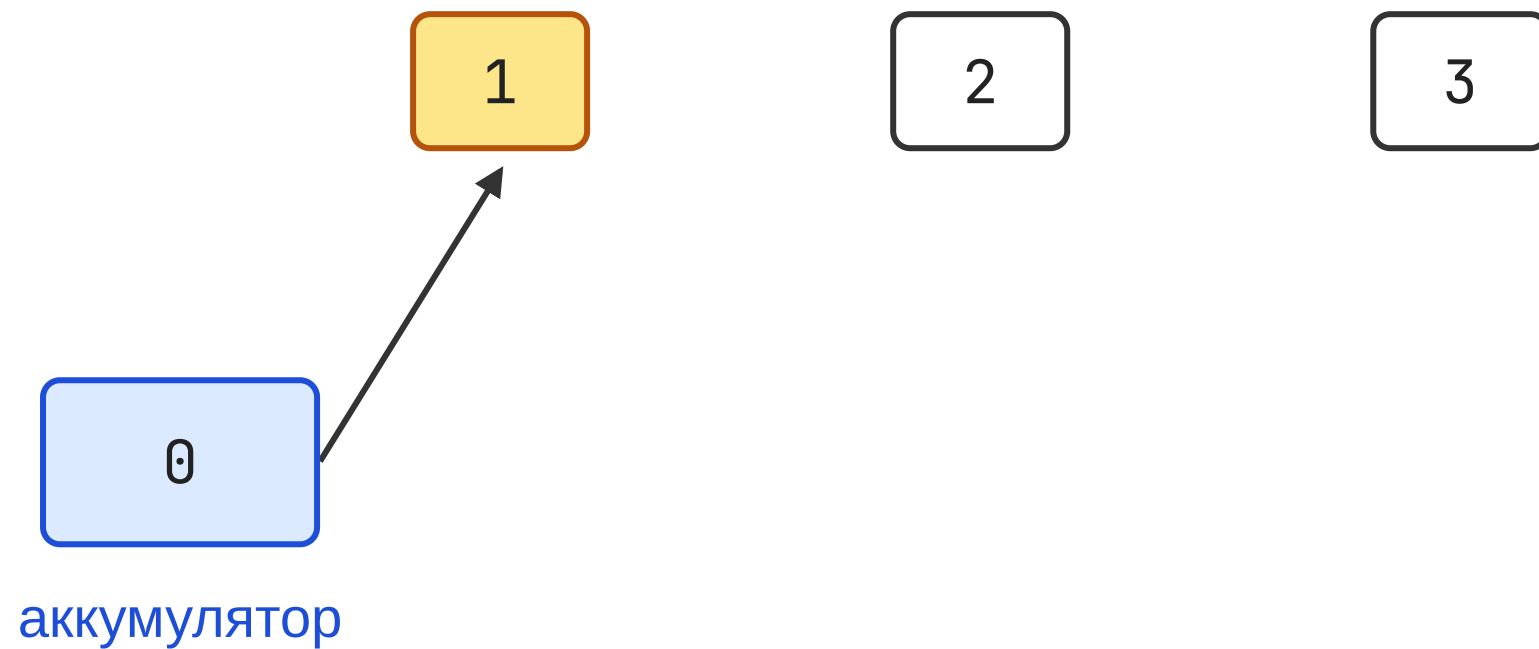
foldl: передаём аккумулятор

Каждый шаг получает накопленный результат и следующий элемент.

```
foldl (-) 0 [1,2,3]
```

СОСТОЯНИЯ

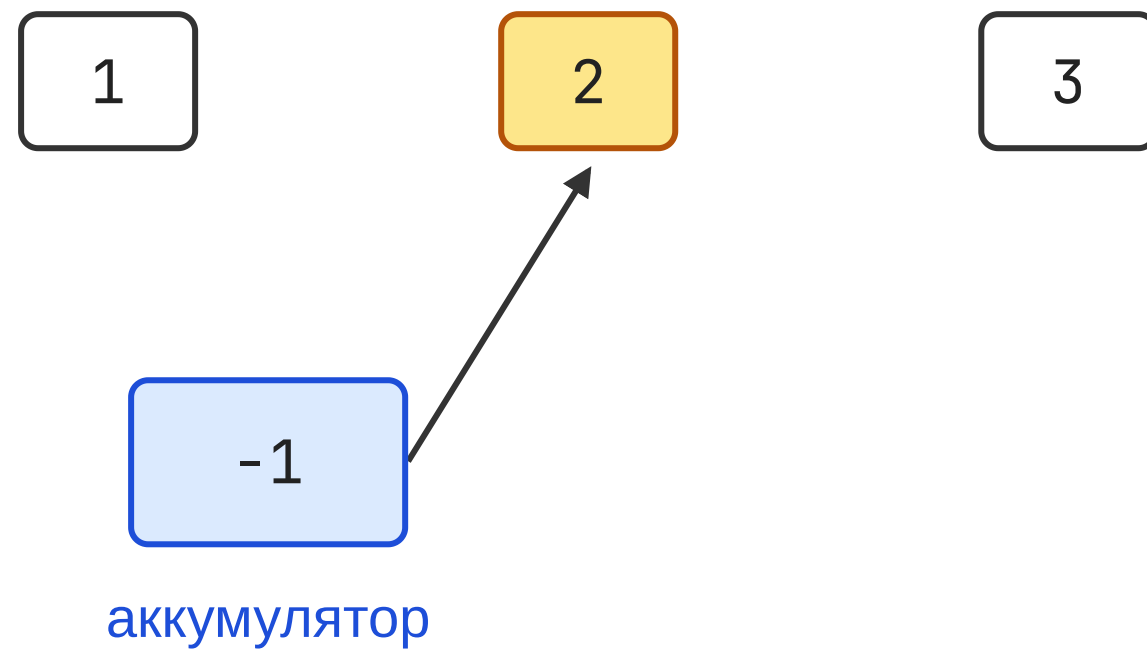
Начинаем с 0.



foldl: передаём аккумулятор

Результат шага становится аккумулятором следующего.

```
foldl (-) (0 - 1) [2,3]
```



СОСТОЯНИЯ

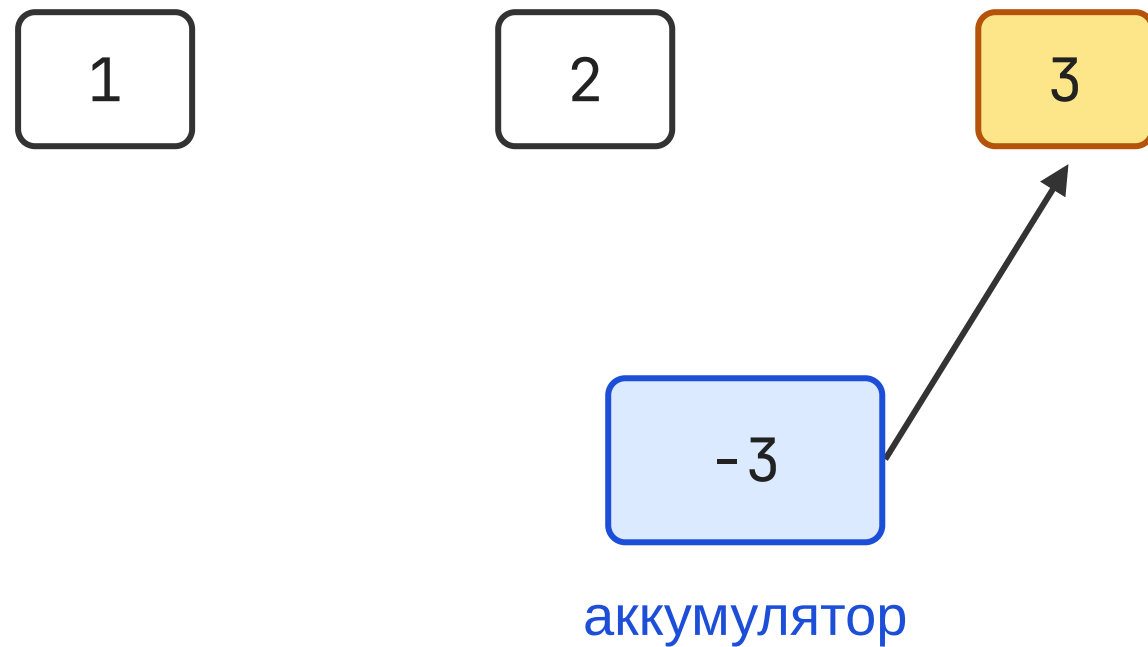
Начинаем с `0`.

`0 - 1 = -1.`

foldl: передаём аккумулятор

Скобки накапливаются слева.

```
foldl (-) ((0 - 1) - 2) [3]
```



СОСТОЯНИЯ

Начинаем с 0.

$0 - 1 = -1$.

$(-1) - 2 = -3$.

foldl: передаём аккумулятор

Дошли до `[]`: возвращаем аккумулятор.

$$((0 - 1) - 2) - 3 = -6$$

1

2

3

-6

результат

СОСТОЯНИЯ

Начинаем с 0.

$$0 - 1 = -1.$$

$$(-1) - 2 = -3.$$

$$(-3) - 3 = -6.$$

Скалярное произведение

Например, `prod [1,2,3] [4,5,6] == 32`.

✎ Реализуйте скалярное произведение через `zipWith` и `foldr`. При разной длине списков используйте общий префикс.

```
prod :: Num a => [a] -> [a] -> a
```

⚡ **Решение.**

```
prod :: Num a => [a] -> [a] -> a
prod xs ys = foldr (+) 0 (zipWith (*) xs ys)
-- 4 + (10 + (18 + 0)) == 32
```

Свёртка разных коллекций

Одна свёртка для разных коллекций

Как с `map` в прошлой практике: операция одна, формы коллекций разные.

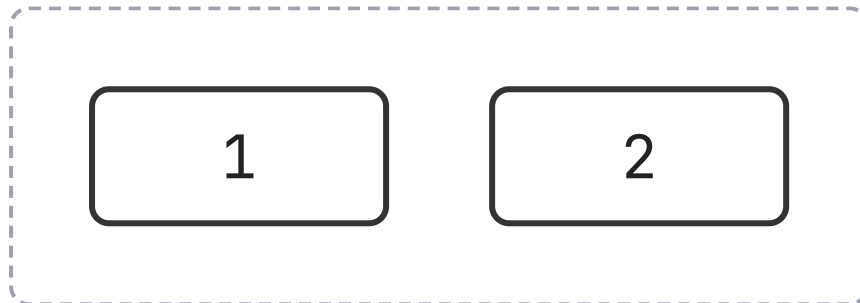
```
data Pair a = Pair a a
```

```
foldList  :: (a -> b -> b) -> b -> [a] -> b
```

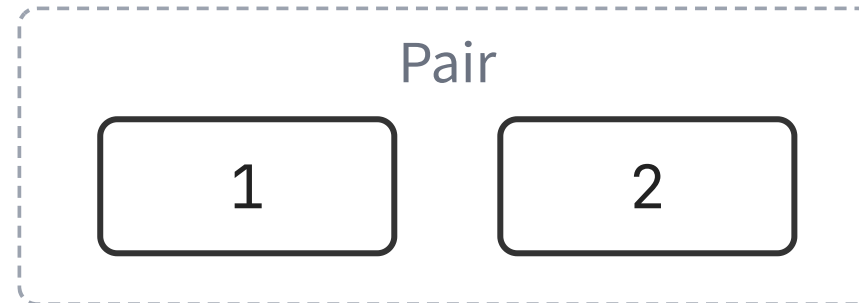
```
foldPair  :: (a -> b -> b) -> b -> Pair a -> b
```

```
foldMaybe :: (a -> b -> b) -> b -> Maybe a -> b
```

список



пара



Maybe



Одна свёртка для разных коллекций

Хотим одно имя `foldr`; тип коллекции выбирает реализацию.

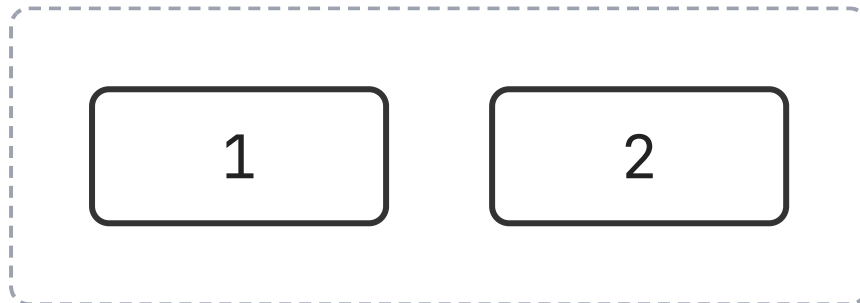
```
data Pair a = Pair a a
```

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

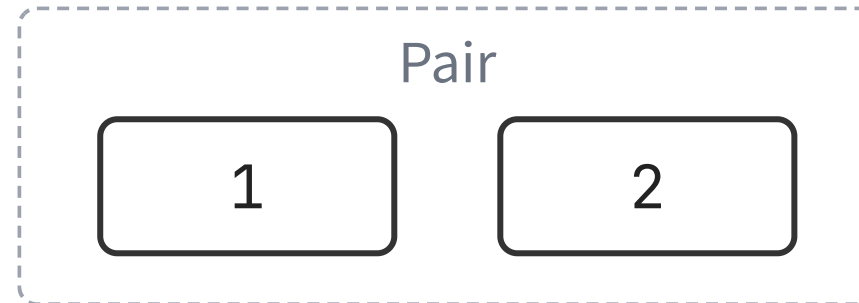
```
foldr :: (a -> b -> b) -> b -> Pair a -> b
```

```
foldr :: (a -> b -> b) -> b -> Maybe a -> b
```

список



пара



Maybe

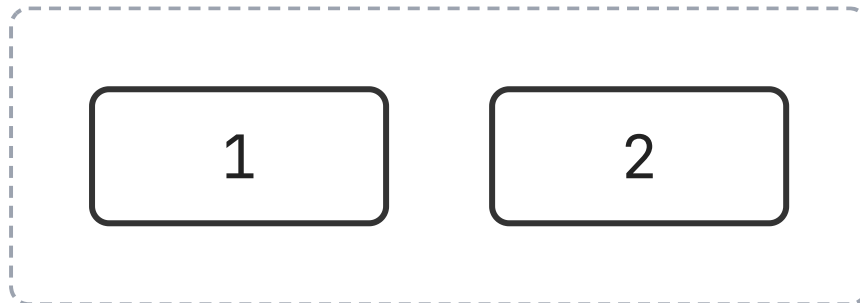


Одна свёртка для разных коллекций

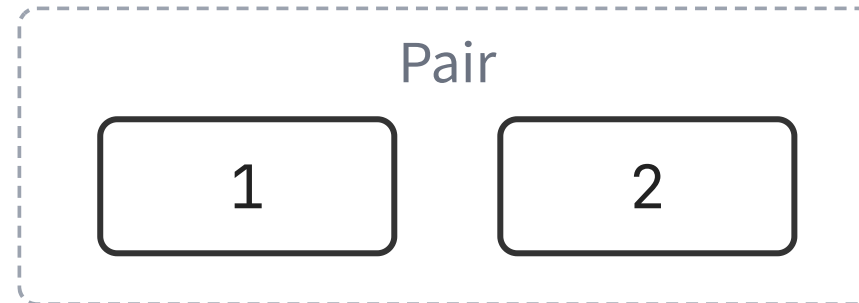
`Foldable` перегружает свёртку по конструктору типа `t`.

```
class Foldable t where  
  foldr :: (a -> b -> b) -> b -> t a -> b  
  foldMap :: Monoid m => (a -> m) -> t a -> m
```

`t = []`



`t = Pair`



`t = Maybe`



foldMap: преобразуем и объединяем

```
foldMap :: (Foldable t, Monoid m) => (a -> m) -> t a -> m
```

Применяем `f` к каждому элементу и объединяем результаты через `(<>)`. Для пустой коллекции получаем `mempty`.

```
data Pair a = Pair a a

instance Foldable Pair where
  foldMap f (Pair x y) = f x <> f y
```

 **Устно:** Что вернёт `foldMap (\x -> [x,x]) (Pair 1 2)`?

 **Решение.**

`[1,1] <> [2,2] == [1,1,2,2]`.

Реализуем foldMap для списка

✎ Завершите `foldMapList` через `foldr`. Шаг объединяет `f x` с результатом свёртки хвоста.

```
foldMapList :: Monoid m => (a -> m) -> [a] -> m
foldMapList f = foldr (\x rest -> _____) _____
```

⚡ **Решение.**

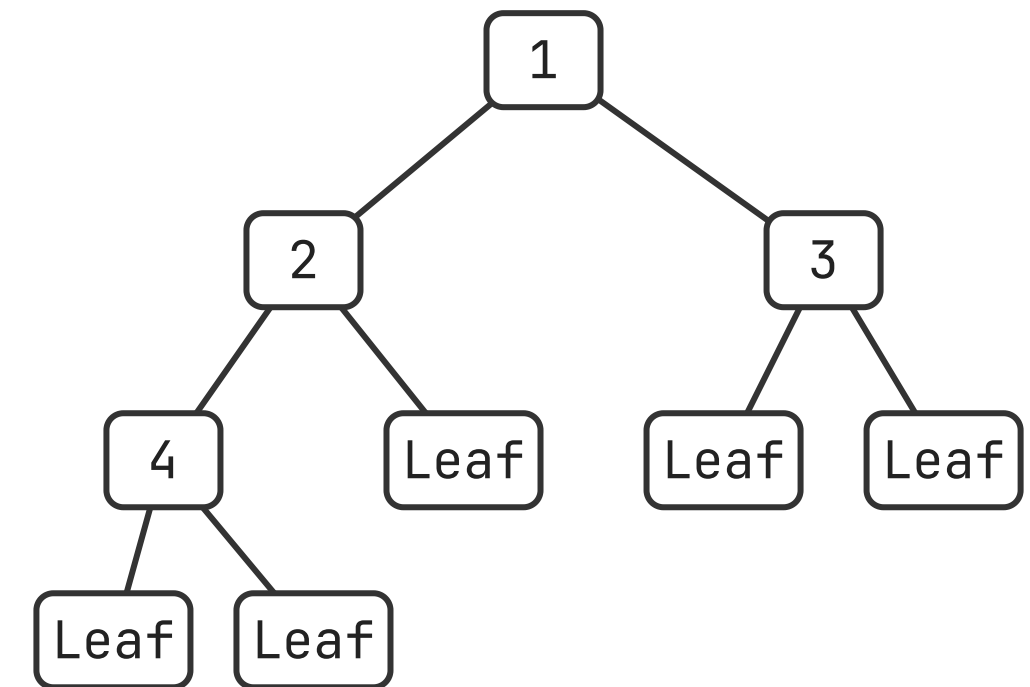
```
foldMapList f = foldr (\x rest -> f x <> rest) mempty
```

Дерево — ещё одна коллекция

```
data Tree a = Leaf | Node a (Tree a) (Tree a)
```

```
sampleTree :: Tree Int  
sampleTree =  
  Node 1  
    (Node 2 (Node 4 Leaf Leaf) Leaf)  
    (Node 3 Leaf Leaf)
```

В каждом **Node** один элемент. **Leaf** — пустое дерево.



Как с помощью **foldMap** получить список элементов дерева?

Порядок выберем такой: корень, левое поддерево целиком, затем правое.

Самостоятельно: полный инстанс

✎ Напишите полный инстанс `Foldable Tree`: заголовков и метод `foldMap`. Порядок — корень, левое поддерево целиком, затем правое; `Leaf` пуст. Не используйте `deriving Foldable`.

```
data Tree a = Leaf | Node a (Tree a) (Tree a)
```

⚡Решение.

```
instance Foldable Tree where
  foldMap _ Leaf = mempty
  foldMap f (Node x left right) =
    f x <> foldMap f left <> foldMap f right
```

Соединяем результаты для корня, левого и правого поддеревьев в выбранном порядке.

```
foldMap (\x -> [x]) sampleTree      -- [1,2,4,3]
getSum (foldMap Sum sampleTree)      -- 10
```

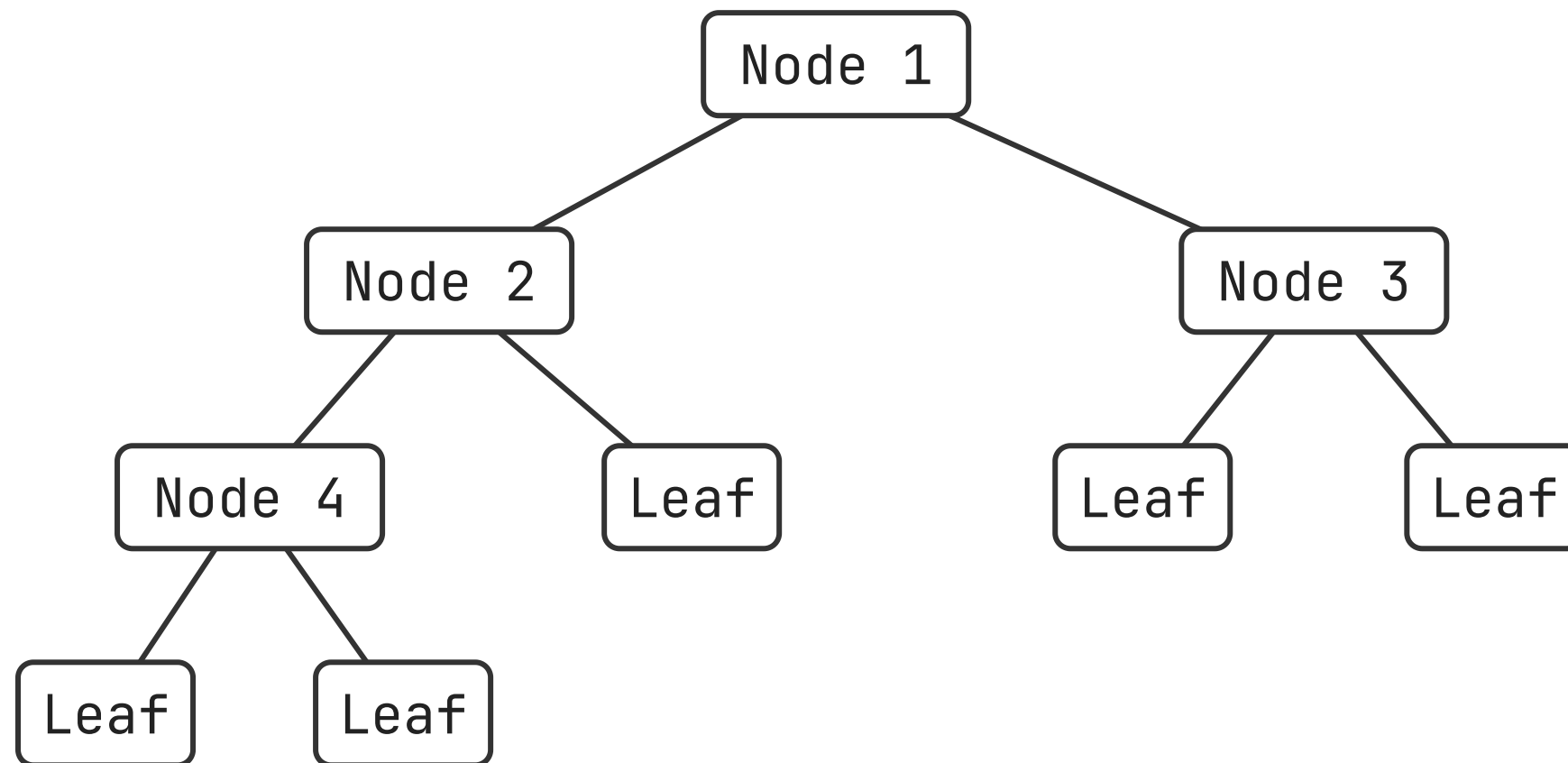
foldMap на дереве: преобразуем и объединяем

Применим написанный инстанс к нашему дереву.

```
foldMap (\x -> [x]) sampleTree
```

ПО НАПИСАННОМУ ИНСТАНСУ

`f x = [x]`: **результатом будет список.**



foldMap на дереве: преобразуем и объединяем

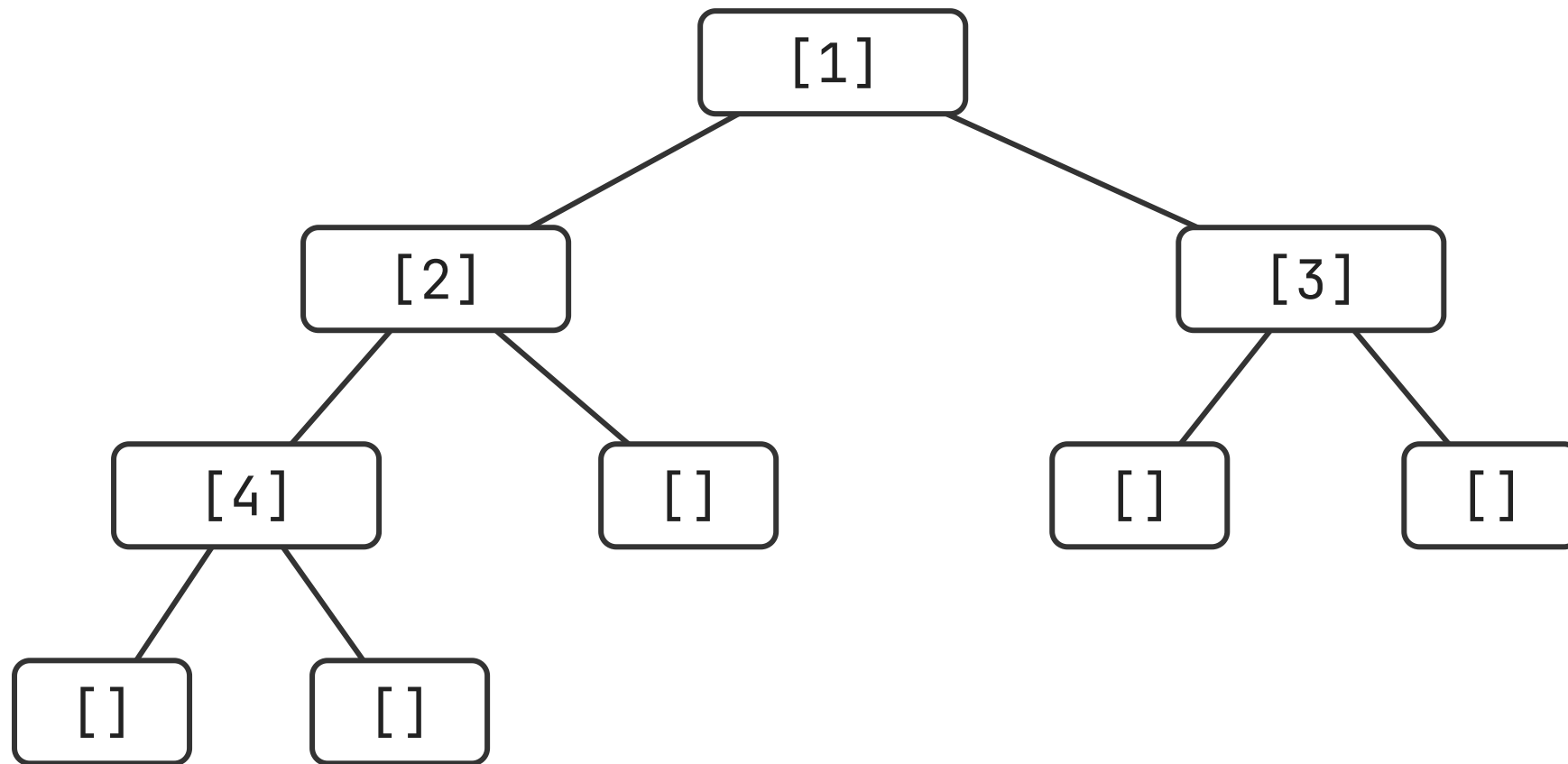
Каждый элемент даёт одноэлементный список; пустое дерево даёт `[]`.

```
f x = [x]; mempty = []
```

ПО НАПИСАННОМУ ИНСТАНСУ

`f x = [x]`: результатом будет список.

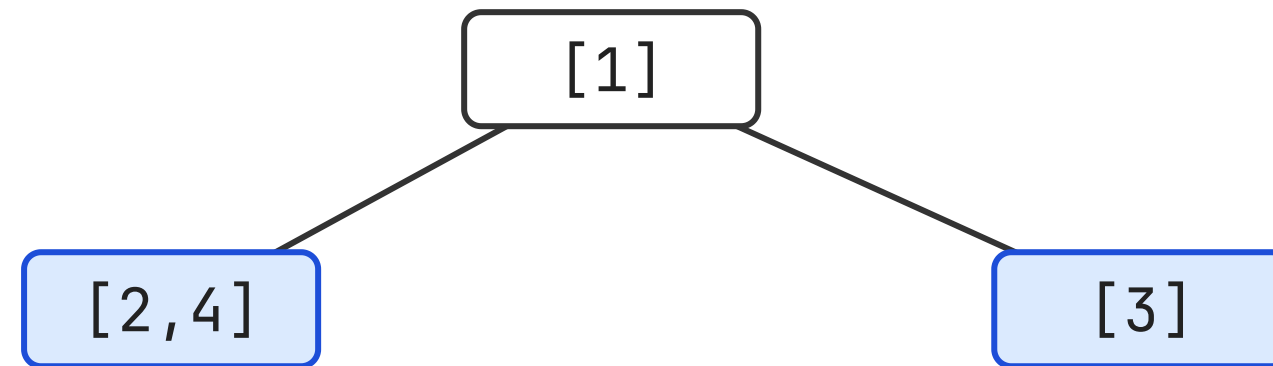
В узле: `f x <> foldMap f left <> foldMap f right`.



foldMap на дереве: преобразуем и объединяем

Объединяем результаты внутри левого и правого поддеревьев.

```
[1] <> ([2] <> [4] <> []) <> [3]
```



ПО НАПИСАННОМУ ИНСТАНСУ

`f x = [x]`: результатом будет список.

В узле: `f x <> foldMap f left <> foldMap f right`.

Слева — [2,4], справа — [3].

foldMap на дереве: преобразуем и объединяем

В корне соединяем его список с результатами обоих поддеревьев.

```
[1] <> [2,4] <> [3] = [1,2,4,3]
```

[1,2,4,3]

ПО НАПИСАННОМУ ИНСТАНСУ

`f x = [x]`: результатом будет список.

В узле: `f x <> foldMap f left <> foldMap f right`.

Слева — `[2,4]`, справа — `[3]`.

Для списков (<>) — это (++).

Работа с коллекциями

Собираем программу из универсальных кирпичиков

Преобразовать элементы → отобрать нужные → собрать результат.

```
import Data.Function ((&))

data Student = Student String [Double]

average :: [Double] -> Double
awarded :: [Student] -> Map String String
```

$x \ \& \ f = f \ x$: передаём данные следующей функции.

`average` дана: среднее оценок, для пустого списка — 0.

Работа с коллекциями: map → filter → foldMap

Вход — список студентов и их оценок.

```
1 awarded students = students
2   & map \(Student name marks) ->
3     (name, average marks))
4   & filter ((> 3) . snd)
5   & foldMap \(name, s) ->
6     Map.singleton name
7     ("clever cooky: " ++ show s))
```

Вход · [Student]

Ada: [2,4] · Bob: [0]

Cat: [4,6]

map · [(String, Double)]

filter · [(String, Double)]

foldMap · Map String String

Работа с коллекциями: map → filter → foldMap

`map` преобразует каждого студента в пару имени и средней оценки.

```
1 awarded students = students
2   & map \(Student name marks) ->
3     (name, average marks))
4   & filter ((> 3) . snd)
5   & foldMap \(name, s) ->
6     Map.singleton name
7     ("clever cooky: " ++ show s))
```

Вход · `[Student]`

Ada: [2,4] · Bob: [0]

Cat: [4,6]

`map · [(String, Double)]`

[("Ada",3.0),
("Bob",0.0), ("Cat",5.0)]

`filter · [(String, Double)]`

`foldMap · Map String String`

Работа с коллекциями: map → filter → foldMap

`filter` оставляет пары со средней оценкой строго больше 3.

```
1 awarded students = students
2   & map \(Student name marks) ->
3     (name, average marks))
4   & filter ((> 3) . snd)
5   & foldMap \(name, s) ->
6     Map.singleton name
7     ("clever cooky: " ++ show s))
```

Вход · `[Student]`

Ada: [2,4] · Bob: [0]
Cat: [4,6]

map · `[(String, Double)]`

[("Ada",3.0),
("Bob",0.0), ("Cat",5.0)]

filter · `[(String, Double)]`

[("Cat",5.0)]

foldMap · `Map String String`

Работа с коллекциями: map → filter → foldMap

`foldMap` собирает словарь наград из одноэлементных словарей.

```
1 awarded students = students
2   & map \(Student name marks) ->
3     (name, average marks))
4   & filter ((> 3) . snd)
5   & foldMap \(name, s) ->
6     Map.singleton name
7     ("clever cooky: " ++ show s))
```

Вход · `[Student]`

Ada: [2,4] · Bob: [0]
Cat: [4,6]

map · `[(String, Double)]`

[("Ada",3.0),
("Bob",0.0), ("Cat",5.0)]

filter · `[(String, Double)]`

[("Cat",5.0)]

foldMap · `Map String String`

Cat ↦ clever cooky: 5.0

Ленивость и выбор свёртки

Ленивость: вычисляем по потребности

`xs = [1..] :: [Integer]` задаёт бесконечный список; пока никто не запросил его элементы.

`xs`

ЧТО ПОТРЕБОВАЛОСЬ

Вычисление списка можно отложить.

`[1..]`

отложенное выражение

Ленивость: вычисляем по потребности

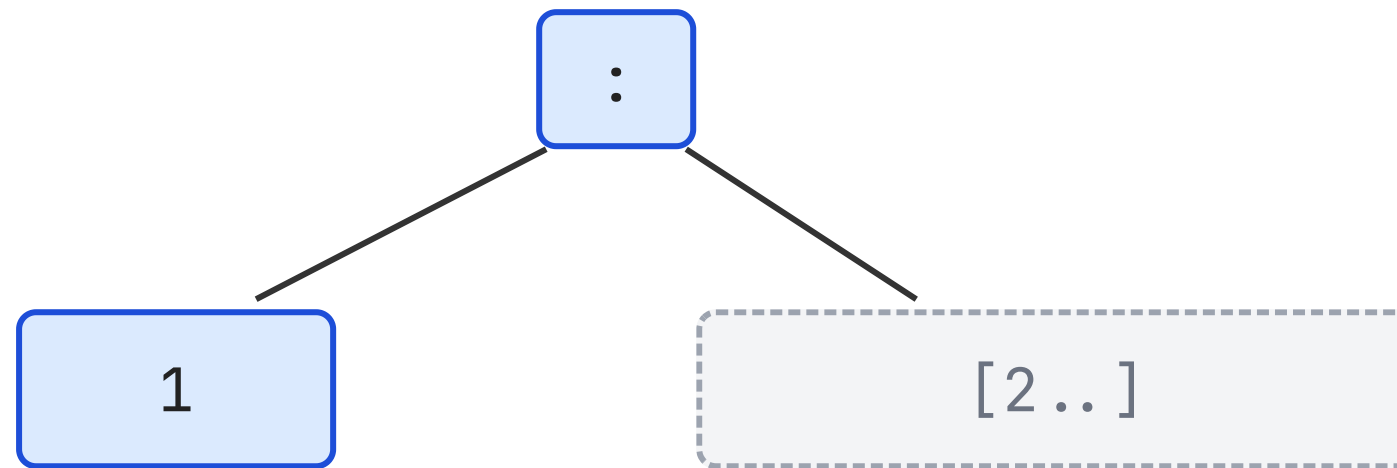
Чтобы напечатать `take 1 xs`, нужны первый `(:)` и его элемент. Хвост не нужен.

```
take 1 xs = [1]
```

ЧТО ПОТРЕБОВАЛОСЬ

Вычисление списка можно отложить.

Получили `[1]`; остальной список не потребован.



потребованное значение

хвост не запрашивали

Что нужно функции length?

Учебное определение длины списка:

```
length []      = 0
length (_:xs) = 1 + length xs

ys = [repeat 1, repeat 2] :: [[Int]]
```

 **Устно:** Что вернёт `length ys`? Нужно ли вычислять элементы `ys`, каждый из которых — бесконечный список?

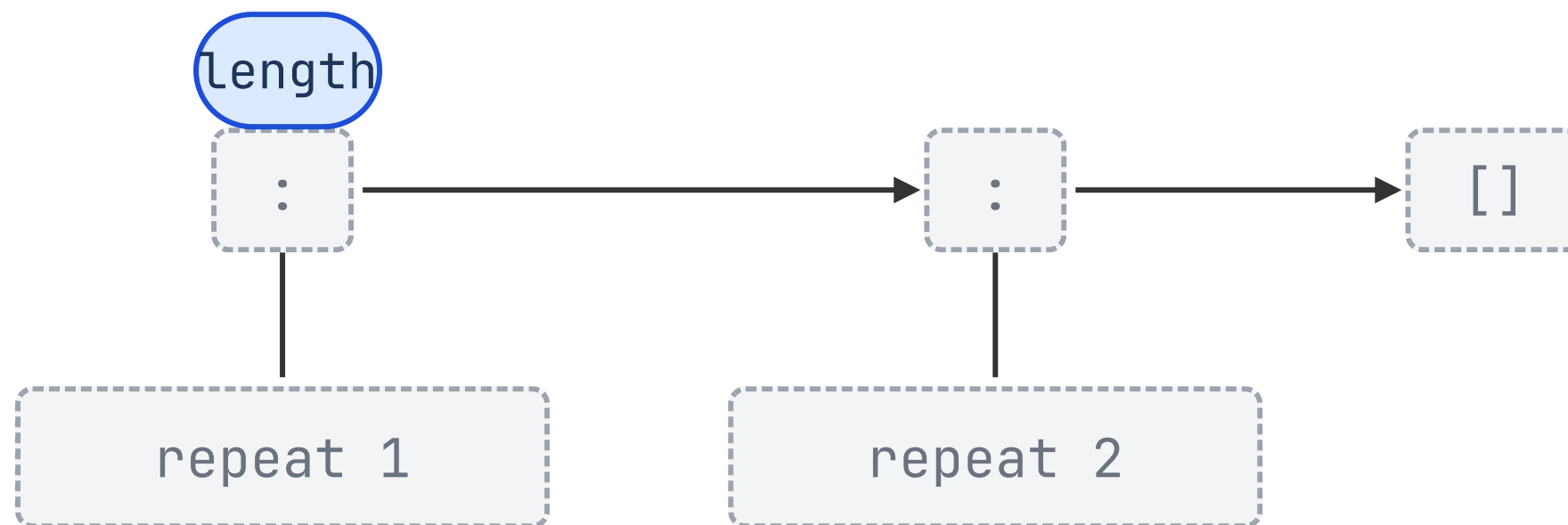
length: требуется форма внешнего списка

`ys = [repeat 1, repeat 2]` — конечный список двух бесконечных списков.

`length ys`

ЧТО ПОТРЕБОВАЛОСЬ

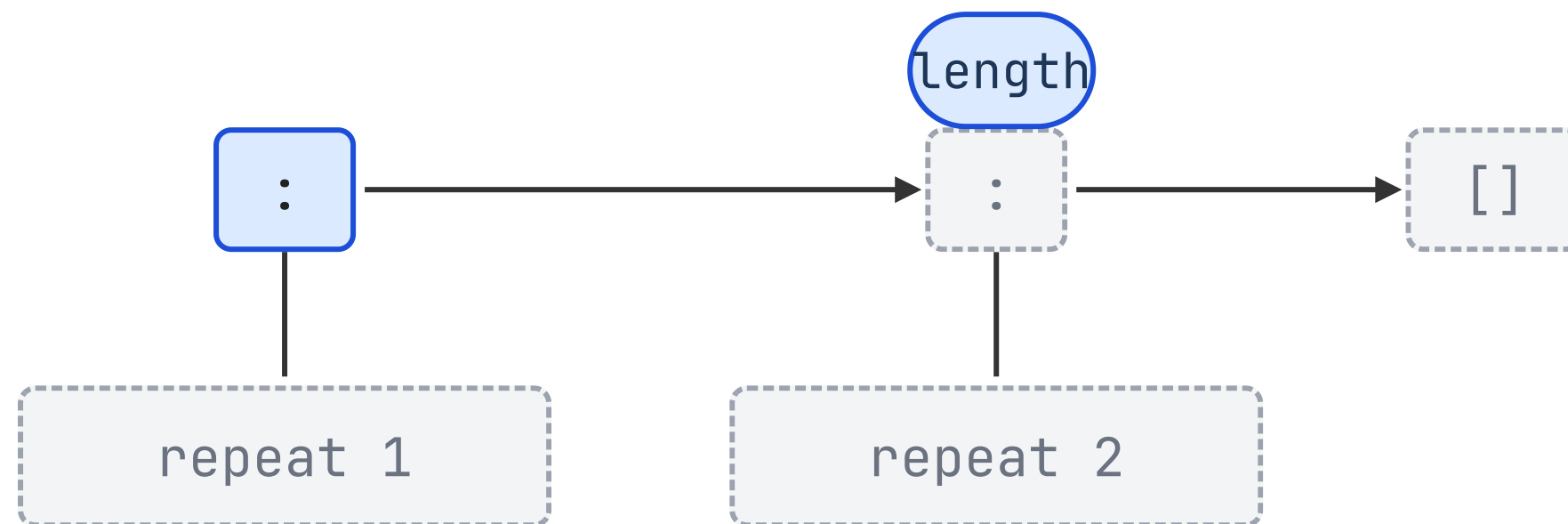
Запрашиваем первый конструктор.



length: требуется форма внешнего списка

Первый `(:)` добавляет единицу. Элемент `repeat 1` не потребован.

`1 + length [repeat 2]`



ЧТО ПОТРЕБОВАЛОСЬ

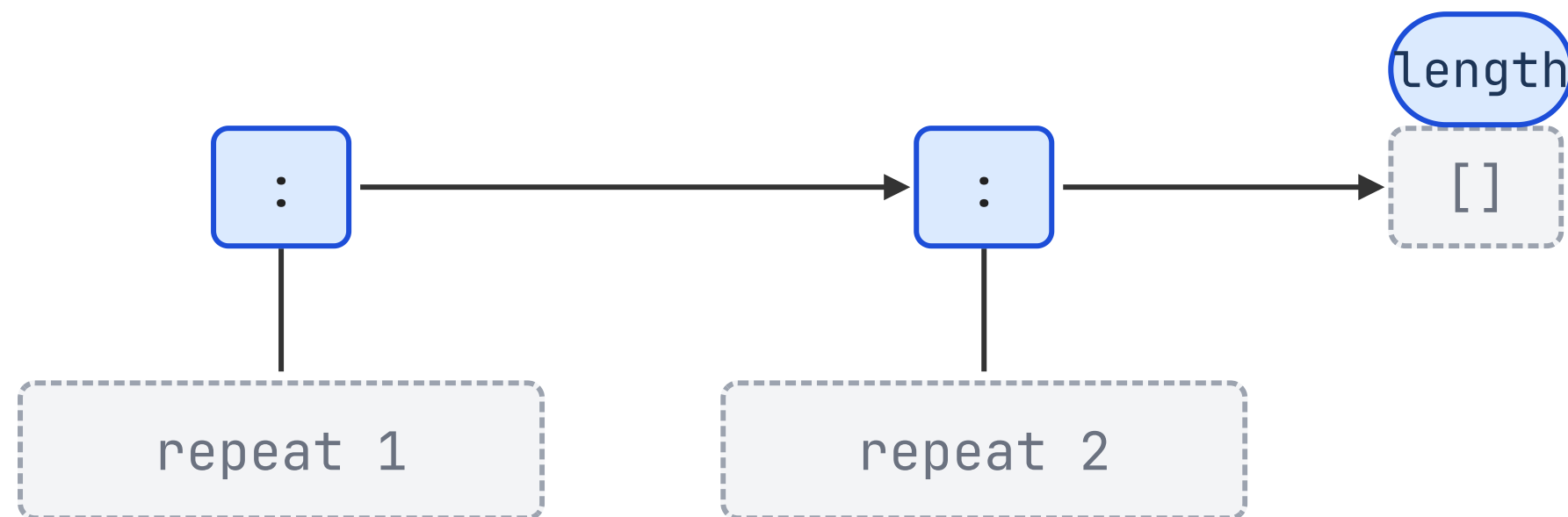
Запрашиваем первый конструктор.

Первый `(:)`; продолжаем по хвосту.

length: требуется форма внешнего списка

Второй `(:)` добавляет ещё единицу. Элемент `repeat 2` тоже не потребован.

`1 + (1 + length [])`



ЧТО ПОТРЕБОВАЛОСЬ

Запрашиваем первый конструктор.

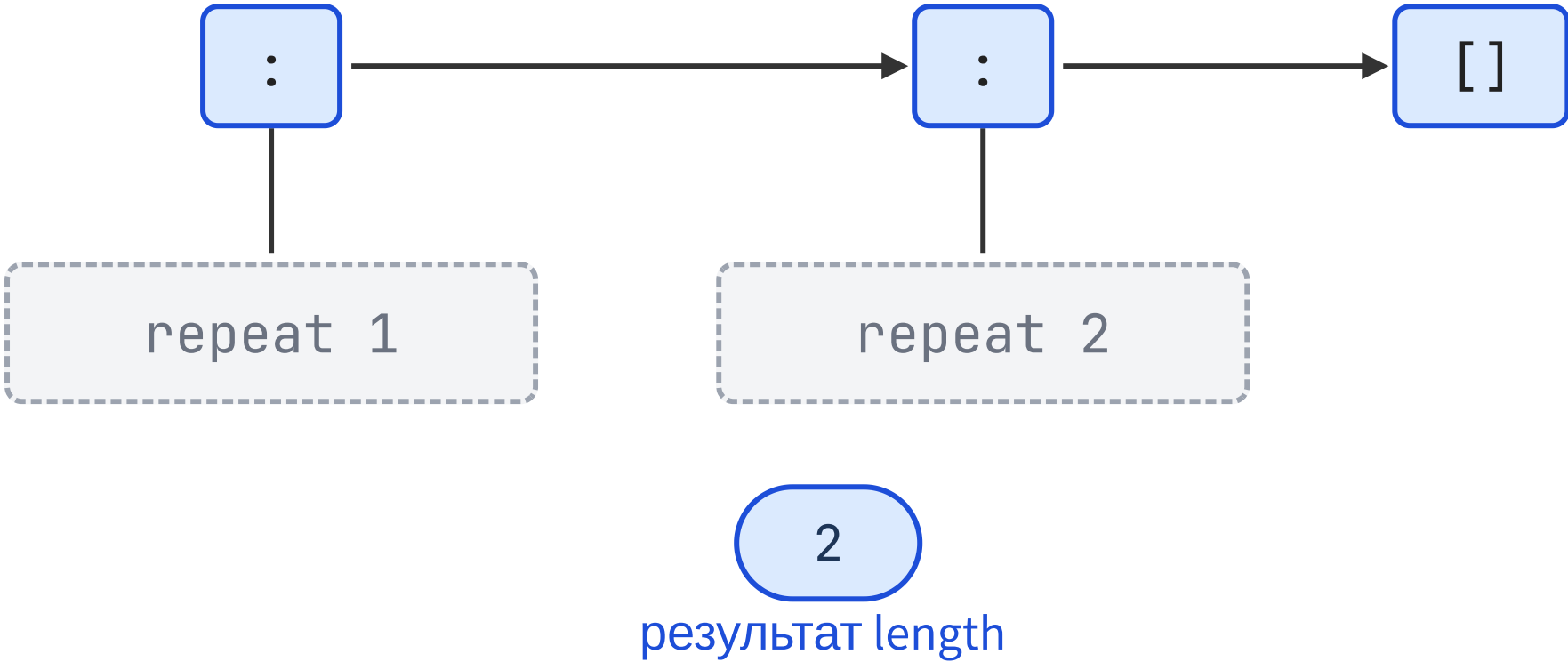
Первый `(:)`; продолжаем по хвосту.

Второй `(:)`; запрашиваем конец списка.

length: требуется форма внешнего списка

Дошли до `[]`: результат `2`. Внутренние бесконечные списки не вычислялись.

`1 + (1 + 0) = 2`



ЧТО ПОТРЕБОВАЛОСЬ

Запрашиваем первый конструктор.
Первый `(:)`; продолжаем по хвосту.
Второй `(:)`; запрашиваем конец списка.
`[]`: **останавливаемся, длина равна 2.**

Длина требует конца списка

```
length [undefined, undefined]      -- 2
length (1 : undefined)             -- ошибка
length ([1..] :: [Integer])        -- не завершится
```

`length` требует все внешние `(:)` и конечный `[]`, но не значения элементов.

Бесконечный список: календарь рассылок

Планируем пять ближайших рассылок по будням, начиная с 23 октября 2026 года.

Haskell: ленивый список

```
dates = iterate (addDays 1) start
weekdays = filter isWeekday dates
mailings = take 5 weekdays
```

Даны `start :: Day` и `isWeekday`, исключающий субботу и воскресенье. `addDays` — из `Data.Time`.

Python: итераторы

```
from datetime import date, timedelta
from itertools import count, islice

start = date(2026, 10, 23)
dates = (start + timedelta(days=n)
         for n in count())
weekdays = (d for d in dates
             if d.weekday() < 5)
mailings = list(islice(weekdays, 5))
```

Получим 23, 26, 27, 28 и 29 октября. Проверим семь дат; дальнейший календарь не потребуется.

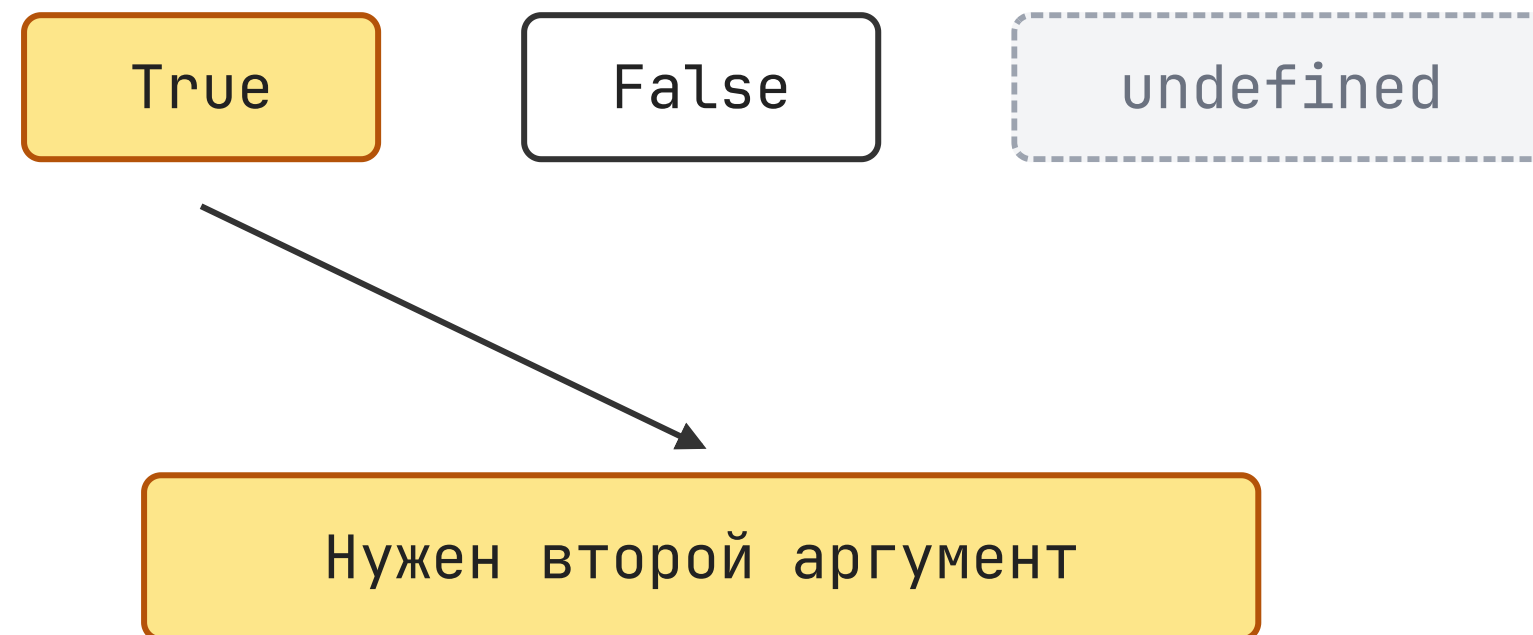
foldr: хвост можно не потребовать

```
foldr (&&) True (True : False : undefined)
```

```
True && (False && foldr (&&) True undefined)
```

ЧТО ТРЕБУЕТСЯ

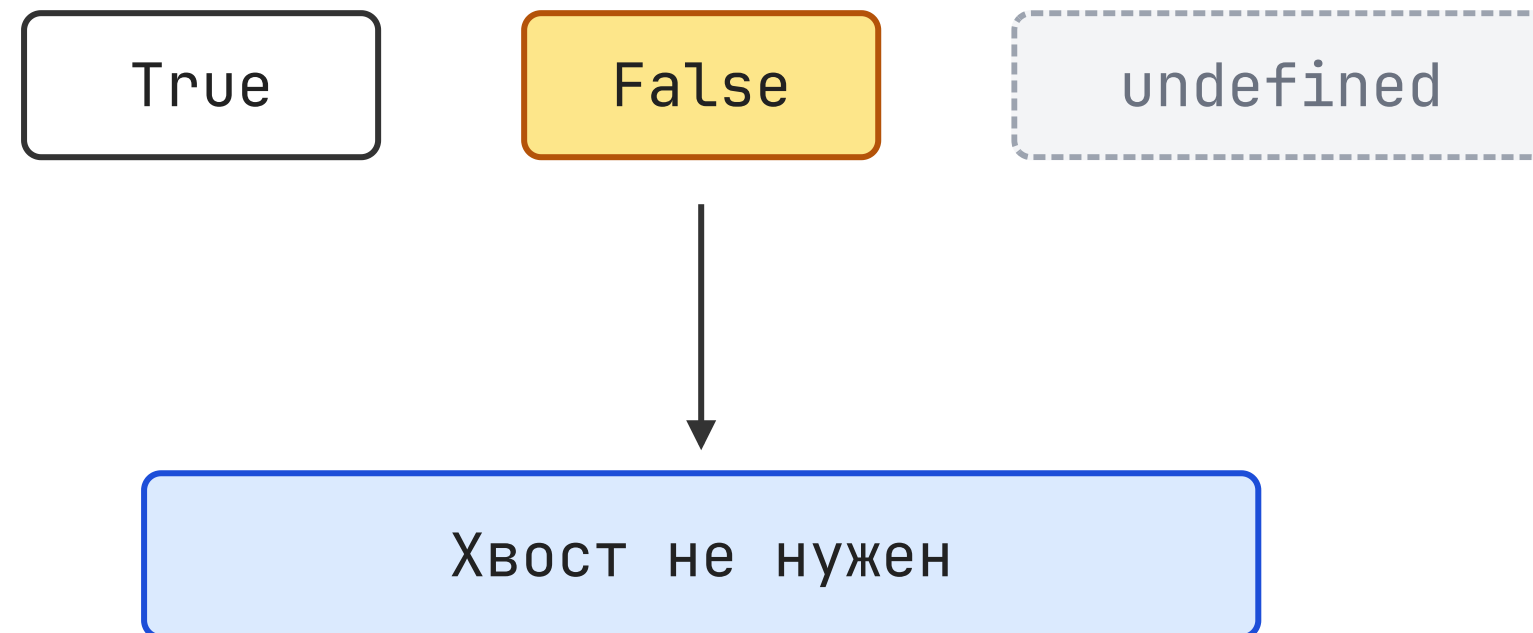
True && rest **требует** rest.



foldr: хвост можно не потребовать

`False && rest` возвращает `False`, не вычисляя `rest`.

`True && False`



ЧТО ТРЕБУЕТСЯ

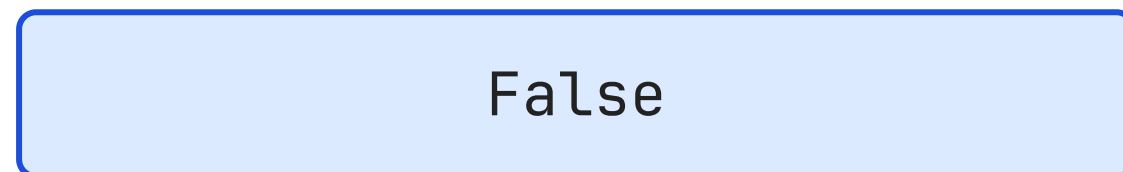
`True && rest` требует `rest`.

Первый аргумент `False` определяет результат.

foldr: хвост можно не потребовать

Результат — `False`. Ошибочный хвост остался невостребованным.

`False`



ЧТО ТРЕБУЕТСЯ

`True && rest` требует `rest`.

Первый аргумент `False` определяет результат.

`foldr` **завершился на конечном префиксе.**

foldl и foldl': что хранит аккумулятор?

Суммируем $[1, 2, 3]$: оба результата равны 6.

`foldl (+) 0 [1,2,3] / foldl' (+) 0 [1,2,3]`

foldl



выражение можно отложить

foldl'



новый аккумулятор до WHNF

РАЗЛИЧИЕ

Хвостовая рекурсия сама по себе не форсирует аккумулятор.

foldl и foldl': что хранит аккумулятор?

Без принуждения выражение может расти вместе со списком.

`foldl (+) 0 [1,2,3] / foldl' (+) 0 [1,2,3]`

foldl

$((0 + 1) + 2) + 3$

отложенные сложения

foldl'

6

число уже вычислено

РАЗЛИЧИЕ

Хвостовая рекурсия сама по себе не форсирует аккумулятор.

Для конечного числового накопления обычно берём `foldl'`.

WHNF не означает вычислить все поля вложенной структуры.

Какую свёртку выбрать?

Задача	Выбор	Что требуется
Найти ответ в конечном префиксе бесконечного списка	<code>foldr</code> с шагом, который может не потребовать <code>rest</code>	Только нужный префикс
Построить ленивый выходной список по частям	<code>foldr</code> с шагом, который выдаёт очередной конструктор	Следующая часть входа по потребности
Накопить числовой результат по всему конечному списку	<code>foldl'</code> из <code>Data.List</code>	Весь список; новый аккумулятор принуждается

Обе левые свёртки — `foldl` и `foldl'` — возвращают итоговый аккумулятор только после `[]`.

Прогноз: завершится ли вычисление?

✎ Укажите результат, «ошибка» или «не завершится». `xs = True : False : undefined, foldl' — из Data.List.`

```
foldr (&&) True xs
foldl' (&&) True xs
foldr (\x rest -> x == 4 || rest) False [1..]
```

⚡ Решение.

```
foldr (&&) True xs           -- False
foldl' (&&) True xs           -- ошибка
foldr (\x rest -> x == 4 || rest) False [1..] -- True
```

Что вы теперь умеете

- Писать инстансы `Semigroup` и `Monoid` и использовать их для объединения результатов
- Выбирать шаг и начальное значение свёртки списка
- Писать `Foldable` для своей коллекции
- Собирать обработку коллекций из `map`, `filter` и `foldMap`
- Определять, какая часть ленивого списка потребуется вычислению
- Выбирать между `foldr`, `foldl` и `foldl'` по задаче

Материалы

Строгий шаг целиком

```
foldlStrict :: (b -> a -> b) -> b -> [a] -> b
foldlStrict _ acc [] = acc
foldlStrict f acc (x:xs) =
    let acc' = f acc x
    in acc' `seq` foldlStrict f acc' xs
```

В программе используем `Data.List.foldl'`.

Функциональная стрелка: полугруппа и моноид

✎ **Факультативно:** Напишите полные инстансы `Semigroup` и `Monoid` для `Func`: функции получают один аргумент и объединяют результаты. Выберите ограничения на `a` и `b`.

```
newtype Func a b = Func { runFunc :: a -> b }
```

⚡Решение.

```
instance Semigroup b => Semigroup (Func a b) where
    Func f <> Func g = Func (\x -> f x <> g x)

instance Monoid b => Monoid (Func a b) where
    mempty = Func (const mempty)
```

Специализированная свёртка

```
data Nat = Z | S Nat

foldNat :: r -> (r -> r) -> Nat -> r
foldNat z _ Z = z
foldNat z s (S n) = s (foldNat z s n)
```

На место каждого конструктора ставим обработчик: для **Z** — значение, для **S** — функцию.

```
toInt :: Nat -> Int
toInt = foldNat 0 (+1)
```

Ленивость в GHCi

```
ghci> let xs = [1..] :: [Integer]
ghci> :sprint xs
xs = _
ghci> take 1 xs
[1]
ghci> :sprint xs
xs = 1 : _
ghci> let ys = [repeat 1, repeat 2] :: [[Int]]
ghci> length ys
2
ghci> :sprint ys
ys = [_,_]
```

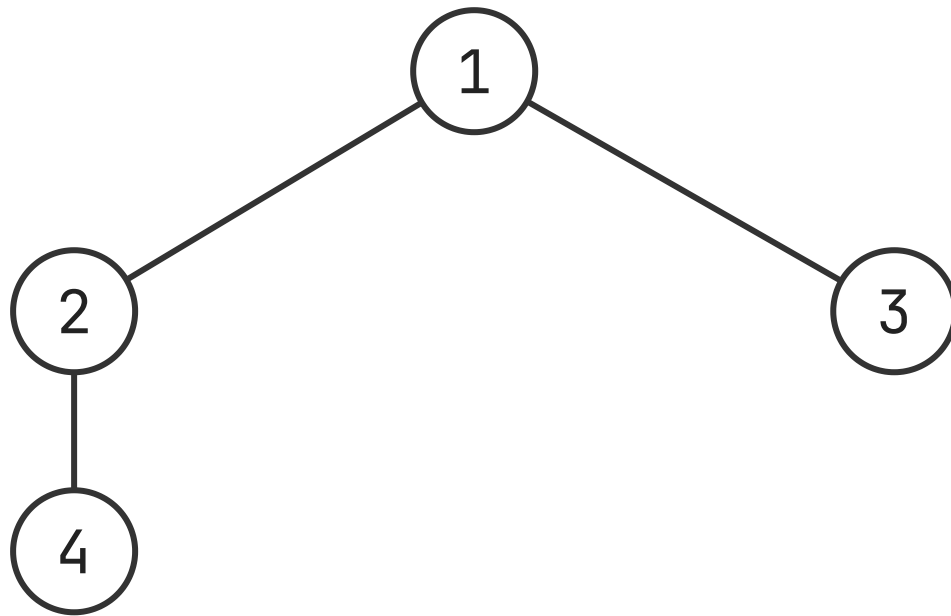
WHNF у составного аккумулятора

```
snd (foldl' (\_ x -> (undefined, x)) (0,0) [1..3])  
-- 3: конструктор пары вычислен, первое поле не потребовано
```

Строгость до WHNF не делает все поля структуры строгими.

DFS и BFS: разные порядки

Одно дерево: DFS [1, 2, 4, 3], BFS [1, 2, 3, 4].



Сумма та же; порядок элементов различается.

Ссылки

- Лекция 7: «Свёртки и развёртки», определения и законы.
- [Классы свёрток в Haskell.](#)
- [foldr, foldl и foldl'.](#)
- [Свёртки как список Чёрча.](#)
- [Конвейеры и модели накопления.](#)