Билеты к экзамену по функциональному программированию. Haskell

Тамарин Вячеслав

January 26, 2021

1 Основы программирования на Haskell. Связывание. Рекурсия. Базовые конструкции языка.

2 Основные встроенные типы языка Haskell. Система модулей. Частичное применение, каррирование.

3 Операторы и их сечения в Haskell. Бесточечный тип.

4 Ошибки. Основание. Строгие и нестрогие функции. Ленивое и энергичное исполнение.

4.1 Ошибки

Специальное обозначение ошибки исполнения: $\perp$ — основание, дно, bottom. Библиотечная константа `undefined` — пример реализации $\perp$. Другая реализация:

```haskell
bot = 1 + bot
fortyTwos = 42 : fortyTwos
```

Это пример продуктивной расходимости — `take 5 fortyTwos`.

Теперь мы можем написать более „аккуратную” версию факториала

```haskell
factorial n =
  if n < 0
  then error "factorial: negative argument"
  else if n > 1
    then n * factorial (n-1)
    else 1
```

Использованная здесь библиотечная функция `error` — это гибкая версия `undefined` с настраеваемым сообщением об ошибке:

```
GHCi> factorial (-3)
*** Exception: factorial: negative argument
```

Тип `Bool` будет иметь три значения: `True`, `False`, `undefined`, так как вычисление могло не завершится.

```haskell
\bot :: forall {a}. a
```

4.2 Строгость функций

Haskell гарантирует вызов по необходимости, если мы запустим функцию, игнорирующую аргумент даже с расходимостью, то получим ответ. Функции, игнорирующие значение своего аргумента, называются *нестрогими* по этому аргументу.

Для *строгих* функций всегда результатом расходимости будет расходимость.

Небольшой неэффективностью ленивого вычисления является удержание всех незавершенных вычислений через указатели. Компилятор борется с этим с помощью анализатора строгости – если вычисления гарантировано строгие, он снимает нестрогость.

Другая неэффективность — нарастание отложенных вычислений (thunk). Чтобы форсировать вычисление можно использовать следующую функцию:

```
seq :: a -> b -> b
seq :: \bot b =  \bot
seq a b = b, если a != \bot
```

Форсирование происходит до слабой головной нормальной формы, то есть до барьера распространения $\perp$: конструктор данных, лямбда-абстрацкия, частично примененная функция.

```
GHCi> seq (undefined, undefined) 42
42
GHCi> seq (\x -> undefined) 42
42
GHCi> seq ((+) undefined) 42
42
```

Через seq определяется энергичная аппликация (с вызовом-по-значению):

```
infixr 0 $!
($!) :: (a -> b) -> a -> b
f $! x = x `seq` f x
```

Форсирование приводит к „худшей определенности”

```
GHCi> ignore undefined
42
GHCi> ignore $! undefined
*** Exception: Prelude.undefined
```

Пример использования seq: вспомним факториал с аккумулирующим параметром

```
factorial n = helper 1 n where
  helper acc k | k > 1 = helper (acc * k) (k - 1)
               | otherwise = acc
```

Из-за ленивости acc будет содержать thunk вида

$$(\ldots((1 * n) * (n - 1)) * (n - 2) * \ldots * 2)$$

Оптимизатор GHC обычно справляется, имея встроенный анализатор строгости. Но можно, не полагаясь на него, написать

```
factorial n = helper 1 n where
  helper acc k | k > 1 = (helper $! acc * k) (k - 1)
               | otherwise = acc
```

5 Алгебраические типы данных. Сопоставление с образцом, его семантика.

Из базовых кирпичиков типов можно строить новые с помощью суммы, произведения и возведения в степень.

5.1 Сопоставление с образцом

Пусть мы хотим поменять местами два элемента в паре:

```
swap :: (a, b) -> (b, a)
swap :: (x, y) = (y, x)
```

Здесь конструкция `(x, y)` представляет собой **образец**.

```
GHCi> swap (5 + 2, True)
(True, 7)
```

При вызове функции происходит **сопоставление с образцом**:

- проверяется, что конструктор `( , )` подходящий
- переменные `x` и `y` связываются с выражениями `5 + 2` и `True`
- осуществляется подстановка вместо переменных в теле функции `swap`

5.2 Тип суммы

Перечисление — тип с 0-арными конструкторами данных:

```
data CardinalDirection = North | East | South | West deriving Show
```

Конструкторы данных имеют тип `CardinalDirection`:

```
GHCi> dir = North
GHCi> :t dir
dir :: CardinalDirection
```

Сопоставление с образцом происходит сверху вниз

```
hasPole :: CardinalDirection -> Bool
hasPole North = True
hasPole South = True
hasPole _ = False
```

Подчеркивание (или переменная) задают неопровержимый образец.

```
GHCi> hasPole North
True
GHCi> hasPole West
False
```

Встроенные типы данных (`Char`, `Int`, `Integer`) ведут себя так, как будто определены как перечисления, поэтому можно использовать литералы как образцы.

5.3 Декартово произведение

Тип-произведение с одним конструктором данных

```
data PointDouble = PtD Double Double
deriving Show

GHCi> :type PtD
PtD :: Double -> Double -> PointDouble

midPointDouble :: PointDouble -> PointDouble
-> PointDouble
midPointDouble (PtD x1 y1) (PtD x2 y2) =
PtD ((x1 + x2) / 2) ((y1 + y2) / 2)

GHCi> midPointDouble (PtD 3.0 5.0) (PtD 9.0 8.0)
PtD 6.0 6.5
```

Можно параметризовывать типовым параметром:

```
data Point a = Pt a a
deriving Show

GHCi> :type Pt
Pt :: a -> a -> Point a
```

`Point` — оператор над типами, конкретный тип получается его аппликацией к некоторому типу, например, `Int`.

```
GHCi> :kind Point
Point :: * -> *

GHCi> :kind Point Int
Int :: *
```

Кайнды — система типов над системой типов Haskell.

5.4 Экспоненциальные типы

Экспоненциальный тип — это тип функции.

```
data Endom a = Endom (a -> a)
appEndom :: Endom a -> a -> a
appEndom (Endom f) = f

GHCi> e = Endom (\n -> 2 * n + 3)
GHCi> :t e
e :: Num a => Endom a
GHCi> :t appEndom e
appEndom e :: Num a => a -> a
GHCi> e `appEndom` 5
13
```

5.5 Рекурсивные типы

При объявлении типов можно использовать рекурсию. Это значит, что допустимо указывать объявляемый тип в качестве аргумента конструктора данных:

```
GHCi> data Nat = Zero | Suc Nat deriving Show
```

Конструктор `Suc` при этом оказывается эндоморфизмом над типом `Nat`

```
GHCi> :t Zero
Zero :: Nat
GHCi> :t Suc
Suc :: Nat -> Nat
```

Это позволяет строить неограниченное число обитателей данного типа:

```
GHCi> one = Suc Zero
GHCi> two = Suc (Suc Zero)
GHCi> three = Suc two
GHCi> four = Suc three
```

Это так называемые числа Пеано. Их можно рассматривать как способ кодирования натуральных чисел в унарной системе исчисления. Поскольку тип `Nat` это тип суммы, тотальные функции над ним, использующие опровержимые образцы, определяются несколькими равенствами

```
GHCi> {pred (Suc n) = n; pred Zero = Zero}
GHCi> pred two
Suc Zero
```

5.6 Стандартные алгебраические типы

- Тип `Maybe a` позволяет задать „необязательное” значение

```
data Maybe a = Nothing | Just a
maybe :: b -> (a -> b) -> Maybe a -> b
find :: (a -> Bool) -> [a] -> Maybe a
```

- Тип `Either a b` описывает одно значение из двух

```
data Either a b = Left a | Right b
either :: (a -> c) -> (b -> c) -> Either a b -> c

head' :: [a] -> Either String a
head' (x:_) = Right x
head' [] = Left "head': empty list"
```

5.7 Сопоставление с образцом

Сопоставление происходит сверху вниз, затем слева направо. Сопоставление бывает

- успешным (succeed);
- неудачным (fail);
- расходящимся (diverge).

```
bar (1, 2) = 3
bar (0, _) = 5
```

- (0, 7) — неудача в первом, успех во втором;
- (2, 1) — две неудачи и, как следствие, расходимость;
- (1, 5-3) — успех, так как для сопоставления нужно приходится форсировать
- (1, undefined) — расходимость, так как форсируем undefined
- (0, undefined) — неудача на первом, успех на втором

6 Объявления `type`, `newtype`. Метки полей.

7 Списки, стандартные функции работы с ними. Генерация (выделение) списков.

Списки встроены, но мы можем определить их сами:

```
data [] a = [] | a : ([] a)
infixr 5 :
```

Для удобства введён синтаксический сахар

```
[1,2,3] = 1:(2:(3:[])) = 1:2:3:[]
```

7.1 Стандартные функции из Data.List

```
head :: [a] -> a
head (x:_) = x
head [] = error "Prelude.head: empty list"

tail :: [a] -> [a]
tail (x:xs) = xs
tail [] = error "Prelude.tail: empty list"
```

Это частичная функция, в современном Haskell использовать их не рекомендуется.

Еще одним важнейшим оператором для списков служит оператор бинарной конкатенации: он делает из двух списков один, присоединяя первый к началу второго.

```
infixr 5 ++
(++) :: [a] -> [a] -> [a]
[] ++ ys = ys
(x:xs) ++ ys = x : xs ++ ys
```

Из реализации оператора видно, что его сложность (число рекурсивных вызовов до наступления терминирующего условия) линейно зависит от размера первого списка и не зависит от второго.

```
length :: [a] -> Int

concat :: [[a]] -> [a]
concat [] = []
concat (xs:xss) = xs ++ concat xss

infix 4 `elem`
elem :: (Eq a) => a -> [a] -> Bool
elem _ [] = False
elem x (y:ys) = x == y || elem x ys
```

Функция, осуществляющая поиск значения с заданным ключом в ассоциативном списке (то есть списке пар ключ-значение)

```
lookup :: Eq a => a -> [(a,b)] -> Maybe b
lookup _ [] = Nothing
lookup key ((k,v):kvs)
  | key == k = Just v
  | otherwise = lookup key kvs
```

Оператор, возвращающий элемент списка с заданным индексом

```
infixl 9 !!
(!!) :: [a] -> Int -> a
xs     !! n | n < 0 = error "Prelude.!!: negative index"
[]     !! _         = error "Prelude.!!: index too large"
(x:_)  !! 0         = x
(_:xs) !! n         = xs !! (n-1)
```

Левая ассоциативность позволяет удобно обслуживать вложенные списки:

```
GHCi> ["Hello","world"] !! 0 !! 1
'e'
GHCi> ["Hello","world"] !! 1 !! 2
'r'
```

7.1.1 Подсписки

Функция take получает целое число n и список и возвращает первые n элементов списка. Если элементов меньше, чем n, возвращается сколько есть. Если n не положительно, возвращается пустой список.

```haskell
take :: Int -> [a] -> [a]
take n _ | n <= 0 = []
take _ []         = []
take n (x:xs)     = x : take (n - 1) xs
```

```haskell
drop :: Int -> [a] -> [a]
drop n xs | n <= 0 = xs
drop _ []          = []
drop n (_:xs)      = drop (n - 1) xs
```

7.1.2 Функции высших порядков над списками

В библиотеке `Data.List` много функций высших порядков. У следующих двух функций первый аргумент — функция типа `a -> Bool`, то есть унарный предикат.

```haskell
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs)
  | p x = x : filter p xs
  | otherwise = filter p xs

takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile _ [] = []
takeWhile p (x:xs)
  | p x = x : takeWhile p xs
  | otherwise = []
```

У функции высшего порядка map функциональный аргумент — произвольная функция:

```haskell
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

Функция map обрабатывает каждый элемент списка переданной функцией-обработчиком, формируя список результатов той же длины, но, возможно, другого типа.

7.1.3 Семейства `zip` и `zipWith`

```haskell
zip :: [a] -> [b] -> [(a,b)]
zip [] _ = []
zip _ [] = []
zip (a:as) (b:bs) = (a,b) : zip as bs

zip3 :: [a] -> [b] -> [c] -> [(a,b,c)]
zip4 :: [a] -> [b] -> [c] -> [d] -> [(a,b,c,d)]
...

unzip :: [(a,b)] -> ([a],[b])
unzip3 :: [(a,b,c)] -> ([a],[b],[c])
...

zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith _ [] _ = []
zipWith _ _ [] = []
zipWith f (a:as) (b:bs) = f a b : zipWith f as bs

zipWith3 :: (a -> b -> c -> d) -> [a] -> [b] -> [c] -> [d]
...
```

Все перечисленные семейства содержат функции вплоть до 7 аргументов.

7.1.4 Способы генерации списков

```
GHCi> take 2 ones
[1,1]
GHCi> take 4 (numsFrom 3)
[3,4,5,6]

take 2 ones → -- (1)
take 2 (1 : ones) → -- (2)
1 : take (2-1) ones) → -- (3)
1 : take 1 (1 : ones) → -- (4)
1 : 1 : take (1-1) ones) → -- (5)
1 : 1 : take 0 ones) → -- (6)
1 : 1 : []
```

- (1) сопоставление с образцом форсирует вычисление ones до $WHNF$;
- (2) подходит последнее уравнение в определении `take`, используем его;
- (3) сопоставление с образцом форсирует вычисление до $WHNF$ обоих аргументов `take`: первого, чтобы отвергнуть первое из уравнений в определении `take`, второго — чтобы отвергнуть второе;
- (4) используем последнее уравнение в определении `take`;
- (5) сопоставление с образцом форсирует вычисление до $WHNF$ первого аргумента `take`;
- (6) подходит первое уравнение в определении `take`, используем его.

```
GHCi> squares = map (^2) (numsFrom 0)
GHCi> takeWhile (<=100) squares
[0,1,4,9,16,25,36,49,64,81,100]

GHCi> fibs = 0 : 1 : zipWith (+) fibs (drop 1 fibs)
GHCi> take 10 fibs
[0,1,1,2,3,5,8,13,21,34]
```

Для формирования „нелинейных” последовательностей имеется другая техника, носящая название выделение списка (**list comprehension**)

```
GHCi> digits = [0..9]
GHCi> [ x^2 | x <- digits ]
[0,1,4,9,16,25,36,49,64,81]
```

Часть справа от вертикальной черты носит название генератора: элементы, связываемые с переменной `x` пробегают по всему списку `digits`. Слева от вертикальной черты находится выражение, в котором можно использовать `x`.

При наличии нескольких генераторов чаще обновляется тот, что правее:

```
GHCi> [ [x,y] | x <- "ABC", y <- "de" ]
["Ad","Ae","Bd","Be","Cd","Ce"]
```

Генераторы могут ссылаться на значения из предыдущих генераторов; кроме того можно использовать предикаты над этими значениями для фильтрации результатов.

7.1.5 Последовательности

```
GHCi> [1..10]
[1,2,3,4,5,6,7,8,9,10]
GHCi> [1,3..17]
[1,3,5,7,9,11,13,15,17]

ones = 1 : ones
numsFrom n = n : numsFrom (n + 1)
```

8 Специальный полиморфизм. Классы типов. Объявление представителей. Классы типов `Eq`, `Ord`, `Enum`, `Bound`.

8.1 Параметрический полиморфизм

Когда в определение функции входит тип как параметр, и мы вместо него можем подставить любой тип. Например,

```
id :: a -> a
id x = x

id True     :: Bool
id "Badger" :: [Char]
id id       :: a -> a
```

Этот код универсален: можем использовать любой тип параметра и реализация не зависит от какой-то специфики типа.

Изучим функцию, определяющую принадлежность списку:

```
elem :: a -> [a] -> Bool
elem _ [] = False
elem x (y:ys) = x == y || elem x ys
```

Очевидно, для элементов нужно отношение равенства. Если для встроенных типов сравнение определено, для пользовательских типов тоже можно определить отношение равенства, то для функций все сложнее:

```
suc :: (forall a. (a -> a) -> a -> a) -> (a -> a) -> a -> a
suc = \n s z -> n s (s z)

suc' :: (forall a. (a -> a) -> a -> a) -> (a -> a) -> a -> a
suc' = \n s z -> s (n s z)

suc ?= suc'
```

С точки зрения β-эквивалентности это две разные функции в β-NF. За счет использования `forall` передать можно только числа Черча.

Эти функции поточечно равны, но не равны *интенсинонально* (совпали по β-NF). Такие называются *экстенсионально* равными. Чтобы доказать, что функции равны придется для всех различные рассуждения использовать, поэтому в хаскеле функции несравнимы.

8.2 Специальный полиморфизм

Это вид полиморфизма, противоположный параметрическому: интерфейс полностью общий (полиморфный), но реализация специализирована для конкретных типов. Например, так устроено сложение чисел.

8.3 Классы типов

Класс типов — именованный набор имен функций с сигнатурами, параметризованными общим типовыми параметром:

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

Имя класса типов задает ограничение, называемое *контекстом.*

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

elem :: Eq a => a -> [a] -> Bool
elem _ [] = False
elem x (y:ys) = x == y || elem x ys
```

Чтобы сделать тип *представителем класса*, нужно реализовать требуемые функции класса:

```haskell
instance Eq Bool where
  True  == True  = True
  False == False = True
  _     == _     = False
  x /= y = not (x == y)
```

Тип-представитель может быть полиморфным:

```haskell
instance Eq a => Eq [a] where
  [] == [] = True
  (x:xs) == (y:ys) = x == y && xs == ys
  _      == _      = False
```

Можно не реализовывать неравенство, оно определено по умолчанию. Для механической реализации можно использовать `deriving Eq`.

8.4 Расширение класса

Класс `Ord` наследует все методы `Eq` и содержит новые методы:

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

  compate x y = if x == y then EQ
                else if x <= y then LT
                else GT

  x < y = case compare x y of {LT -> True; _ -> False}
  x <= y = case compare x y of {GT -> False; _ -> True}
  x > y = case compare x y of {GT -> True; _ -> False}
  x >= y = case compare x y of {LT -> False; _ -> True}
  max x y = if x <= y then y else x
  min x y = if x <= y then x else y

sort :: Ord a => [a] -> [a]
```

Допустимо множественное наследование:

```haskell
class (Eq a, Show a) => MyClass a where
  ...
```

8.5 Enum, Bounded

```haskell
class Enum a where
  succ, pred     :: a -> a
  toEnum         :: Int -> a
  fromEnum       :: a -> Int

  enumFrom       :: a -> [a]              -- [n..]
  enumFromThen   :: a -> a -> [a]         -- [n, n'..]
  enumFromTo     :: a -> a -> [a]         -- [n..m]
  enumFromThenTo :: a -> a -> a -> [a]   -- [n, n'..m]

class Bounded a where
  minBound, maxBound :: a
```

Класс Integer потенциально бесконечен в обе стороны, поэтому он является Enum, но не является Bounded.

9 Внутренняя реализация классов типов.

Организована через механизм передачи словарей. Словарь для класса — запись его методов.

```haskell
data Eq' a = MkEq { eq, ne :: a -> a -> Bool}
```

Здесь хранится две *функции-селекторы*, они выбирают методы равенства и неравенства из данного словаря.

```haskell
GHCi> :t eq
eq :: Eq' a -> a -> a -> Bool
GHCi> :i ne
eq :: Eq' a -> a -> a -> Bool
```

Теперь вместо того, что называлось контекстом, теперь просто дополнительный параметр.

Объявления представителей транслируются в функции, возвращающие словарь или в функции, принимающие словарь и возвращающие более сложный словарь:

```haskell
dEqInt :: Eq' Int
dEqInt = MkEq {
  eq = eqInt, -- это низкоуровневый способ сравнивать Intы
  ne = \x y -> not $ eqInt x y
}

dEqList :: Eq' a -> Eq' [a]
dEqList (MkEq e _) = MkEq el (\x y -> not $ el x y) -- e - словарь для сравнения элементов списка
  where el [] [] = True
        el (x:xs) (y:ys) = x `e` y && xs `el` ys
        el _ _ = False
```

Теперь можем написать enum': он требует, чтобы ему дали словарь для сравнения элементов типа a первым аргументом, а остальное аналогично.

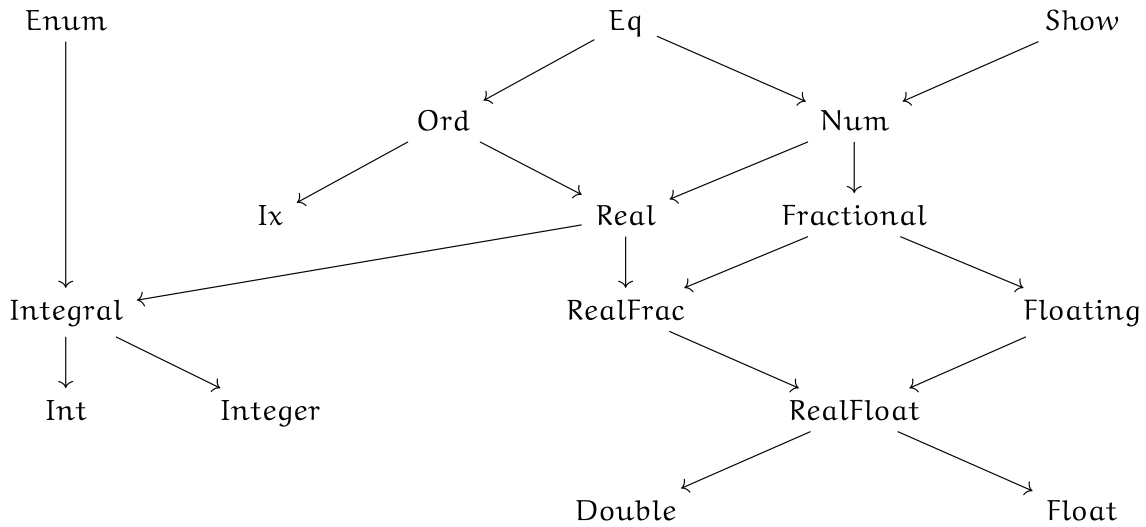
```haskell
elem' :: Eq' a -> a -> [a] -> Bool
elem' _ _ [] = False
elem' d x (y:ys) = eq d x y || elem' d x ys

GHCi> elem' dEqInt 2 [3, 5, 2]
True
GHCi> elem (dEqList dEqInt) [3, 5] [[4], [1, 2, 3], [3, 5]]
True
```

10 Стандартные классы типов: Num и его наследники, Show и Read.

```
class (Eq a, Show a) => Num a where
  (+), (-), (*) :: a -> a -> a
  negate        :: a -> a
  abs, signum   :: a -> a
  fromInteger   :: Integer -> a

  x - y = x + negate y
  negate x = 0 - x
```



Автоматического приведения типов в Хаскеле нет. Всегда нужно использовать приведение.

11 Полугруппы и моноиды. Представители класса типов Semigroup и Monoid.

11.1 Полугруппы

Полугруппа — множество с ассоциативной бинарной операцией над ним.

```
infixr 6 <>
class Semigroup a where
  (<>) :: a -> a -> a
  sconcat :: NonEmpty a -> a   -- конкатенация, все : заменяем на <>
  stimes :: Integral b => b -> a -> a   -- повторяем b раз и вставляем между копиями <>

infixr 5 :|
data NonEmpty a = a :| [a]  -- произведение a и [a]
```

За счет ассоциативности операции, можно достичь логарифмической трудности операций. Для любой полугруппы должен выполняться закон:

```
(x <> y) <> z = x <> (y <> z)
```

Список — полугруппа относительно конкатенации:

```
class Semigroup [a] where
  (<>) = (++)
```

Можем использовать в функциях, которые не работают на пустых списках:

```
GHCi> import Data.List.NonEmpty
GHCi> sconcat $ "AB" :| ["CDE", "FG"]
"ABCDEFG"
GHCi> stimes 5 "Ab"
"AbAbAbAbAb"
```

11.2 Моноиды

Моноид — множество с ассоциативной бинарной операцией над ним и нейтральным элементом для этой операции.

```
class Semigroup a => Monoid a where
  mempty :: a  -- нейтральный элемент
  mappend :: a -> a -> a  -- операция
  mappend = (<>)
  mconcat :: [a] -> a
  mconcat = foldr mappend mempty
```

Законы:

```
mempty <> x = x
x <> mempty = x
(x `mappend` y) `mappend` z = x `mappend` (y `mappend` z)
```

Список — моноид относительно (++), нейтральный — пустой список:

```
instance Semigroup [a] where
  (<>) = (++)
instance Monoid [a] where
  mempty = []
  mconcat = concat
```

Числа тоже моноид, причем четырежды: сложение – нуль, умножение – единица, минимум – `maxBound`, максимум – `minBound`. Для разделения этих вариантов при реализации можем обернуть числа в `newtype`-коробочку:

```
newtype Sum a = Sum { getSum :: a} deriving (Eq, Ord, Read, Show, Bounded)
instance Num a => Semigroup (Sum a) where
  Sum x <> Sum y = Sum (x + y)
instance Num a => Monoid (Sum a) where
  mempty = Sum 0

GHCi> Sum 3 <> Sum 2
Sum {getSum = 5}
```

`mconcat` — сумма нескольких чисел, `stimes` — умножение.

Теперь напишем для умножения, используя `coerce` – безопасное приведение, если мы имеем два одинаковых рантаймовых представления, то можем привести одно к другому автоматически:

```
newtype Product a = Product { getProduct :: a} deriving (Eq, Ord, Read, Show, Bounded)
instance Num a => Semigroup (Product a) where
  (<>) = coerce ((*) :: a -> a -> a)  -- Data.Coerce
instance Num a => Monoid (Product a) where
  mempty = Product 1

GHCi> Product 3 <> Product 2
Product {getProduct = 6}
```

Для использования `coerce`, нужно подключить расширение `ScopedTypeVariables`, чтобы расширить область видимости `a` из типа.

`Integer` не имеет максимума, поэтому, хотя это и полугруппа, моноидом быть не может. Также не существует моноида для строк.

Для решения такой проблемы можем перейти к полугрупповой операции `sconcat` и от списка к `NoEmpty`:

```
GHCi> (getMin . sconcat . fromList . fmap Min) ["Hello", "Hi"]
"Hello"
-- вместо
```

```
GHCi> (getMin . mconcat . fmap Min) ["Hello", "Hi"]
<interactive>: error: No instance for (Bounded [Char])
```

Булев тип тоже является моноидом относительно конъюнкции и дизъюнкции.

```
newtype All = All { getAll :: Bool} deriving (Eq, Ord, Read, ShowBounded)
instance Semigroup All where
  (<>) = coerce (&&)
instance Monoid All where
  mempty = All True

newtype Any = Any { getAny :: Bool} deriving (Eq, Ord, Read, ShowBounded)
instance Semigroup Any where
  (<>) = coerce (||)
instance Monoid Any where
  mempty = Any False
```

12 Свертки списков. Правая и левая свертки. Энергичные версии. Развертки.

12.1 Правая свертка

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f ini [] = ini
foldr f ini (x:xs) = x `f` (foldr f ini xs)

p : q : r : []  ======>  p `f` (q `f` (r `f` ini))
```

Примеры использования:

```
sum :: [Integer] -> Integer
sum = foldr (+) 0

concat :: [[a]] -> [a]
concat = foldr (++) []

allOdd :: [Integer] -> Bool
allOdd = foldr (\n b -> odd n && b) True

id = foldr (:) []
```

Правые свертки позволяют работать с бесконечными списками.

12.2 Левая свертка

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f ini [] = ini
foldl f ini (x:xs) = foldl f (ini `f` x) xs

p : q : r : []  ======>  ((ini `f` p) `f` q) `f` r
```

Здесь хвостовая рекурсия оптимизируется, но нарастает thunk из цепочки вызовов: запустятся вычисления только, когда список кончится, так как форсирования нет (нет `seq` или сопоставления с образцом).

```
foldl' :: (b -> a -> b) -> b -> [a] -> b
foldl' f ini [] = ini
foldl' f ini (x:xs) = arg `seq` foldl' f arg xs
                         where arg = f ini x
```

Здесь thunk не нарастает, вычисление `arg` форсируется на каждом шаге, это самая эффективная из сверток, но, как и любая левая свертка, не работает с бесконечным циклом.

12.3 Версии без начального значения

```haskell
foldr1 :: (a -> a -> a) -> [a] -> a
foldr1 _ [x] = x
foldr1 f (x:xs) = f x (foldr1 f xs)
foldr1 _ [] = error "foldr1: EmptyList"
```

```haskell
foldl1 :: (a -> a -> a) -> [a] -> a
foldl1 f (x:xs) = foldl f x xs
foldl1 _ [] = error "foldl1: EmptyList"
```

Аналогично можно сделать строгую версию `foldl1'`.

12.4 Сканы

Это списки последовательных шагов свертки:

```haskell
scanl :: (b -> a -> b) -> b -> [a] -> [b]
scanl _ z [] = [z]
scanl (#) z (x:xs) = z : scanl (#) (z # x) xs

GHCi> scanl (++) "!" ["a", "b", "c"]
["!", "!a", "!ab", "!abc"]
GHCi> scanl (*) 1 [1..] !! 5 -- факториал
120
```

В отличие от `foldl` может работать и с бесконечными списками.
Правый скан накапливает результаты в обратном порядке:

```haskell
scanr :: (a -> b -> b) -> b -> [a] -> [b]
scanr _ z [] = [z]
scanr f z (x:xs) = f x q : qs
                    where qs@(q:_) = scanr f z xs   -- здесь используем псевдоним
```

Тождества для сканов:

```haskell
head (scanr f z xs) = foldr f z xs
last (scanl f z xs) = foldl f z xs
```

12.5 Развертка

Двойственная свертке операция.

```haskell
unfoldr :: (b -> Maybe (a, b)) -> b -> [a]
unfoldr g ini
  | Nothing     <- next = []
  | Just (a, b) <- next = a : unfoldr g b
  where next = g ini

GHCi> helper x = if x == 0 then Nothing else Just (x, x - 1)
GHCi> unfoldr helper 10
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Пример использования — возможное определение `iterate`

```haskell
iterate f = unfoldr (\x -> Just (x, f x))
```

13 Класс типов Foldable и его представители.

Идея: обобщить свертки на более общий тип, то есть заменить скобки на букву.

```
class Foldable t where
  fold :: Monoid m => t m -> m
  fold = foldMap id

  foldMap :: Monoid m => (a -> m) -> t a -> m
  foldMap f = foldr (mappend . f) mempty

  foldr, foldr' :: (a -> b -> b) -> b -> t a -> b
  foldr f z t = appEndo (foldMap (Endo . f) t) z

  foldl, foldl' :: (b -> a -> b) -> a -> t b -> a
  foldl f z t = appEndo (getDual (foldMap (Dual . Endo . flip f) t)) z

  foldr1, foldl1 :: (a -> a -> a) -> t a -> a
```

Минимальное определение foldMap или foldr. Изучим полезные функции:

```
class Foldable t where
  toList :: t a -> [a]

  null :: t a -> Bool
  null = foldr (\_ _ -> False) True

  lenght :: t a -> Int
  length = foldl' (\c _ -> c + 1) 0

  elem :: Eq a => a -> t a -> Bool
  elem x = foldr f False
             where f result elem = result || x == elem

  maximum, minimum :: Ord a => t a -> a

  sum, product :: Num a => t a -> a
  sum = getSum . foldMap Sum
```

13.1 Представители

```
instance Foldable [] where
  foldr = Prelude.foldr
  foldl = Prelude.foldl
  foldr1 = Prelude.foldr1
  foldl1 = Prelude.foldl1

instance Foldable Maybe where
  foldr _ z Nothing = z
  foldr f z (Just x) = f x z
```

Еще много контейнеров: **Set, Map, Tree, Seq**. Также Either, Pair. Чтобы объявить представителя двухпараметрического типа, первый параметр нужно связать.

```
GHCi> foldr (+) 5 (Right 37)
42
GHCi> foldr (+) 5 (Left 37)
5
GHCi> foldr (+) 5 ("Answer", 37)
42

GHCi> maximum (Right 37)
```

```
37
GHCi> maximum (Left 37)
*** Exception: maximum: empty structure
GHCi> maximum (100, 42)
42
```

Реализация:

```haskell
foldr f z t = appEndo (foldMap (Endo . f) t ) z
foldl f z t = appEndo
  (getDual (foldMap (Dual . Endo . flip f) t)) z
fold         = foldMap id
length       = getSum . foldMap (Sum . const 1)
sum          = getSum . foldMap Sum
product      = getProduct . foldMap Product
minimum      = getMin . foldMap Min
maximum      = getMax . foldMap Max
foldr f z    = foldr f z . toList
foldl f z    = foldl f z . toList
```