Функциональное программирование
Лекция 7. Свертки и развертки

Денис Николаевич Москвин

СПбГУ, факультет МКН,
бакалавриат «Современное программирование», 2 курс

23.10.2025

План лекции

1. Свертки списков
2. Развертки и оптимизации
3. Полугруппы и моноиды
4. Класс типов `Foldable`

План лекции

Наблюдение

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

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

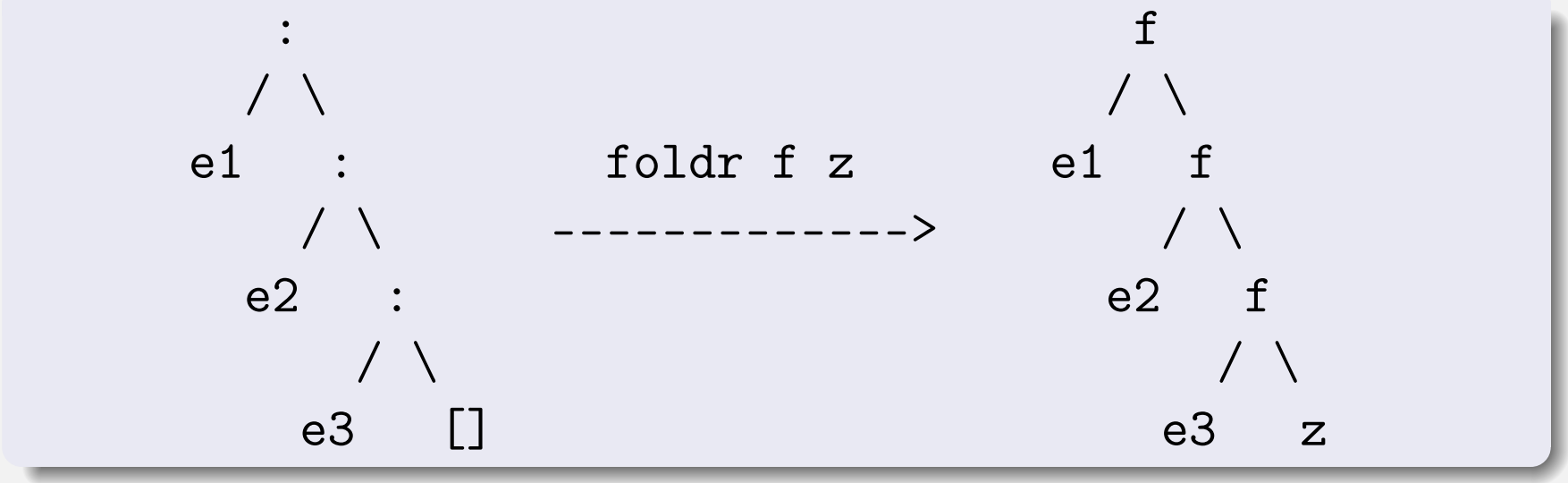
```
allOdd :: [Integer] -> Bool
allOdd []     = ???
allOdd (x:xs) = odd x && allOdd xs
```

Видна общая схема рекурсии.

Правая свертка

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

```
e1 : (e2 : (e3 : []))  -->  e1 `f` (e2 `f` (e3 `f` z))
```



Конкретные свертки через foldr

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

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

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

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

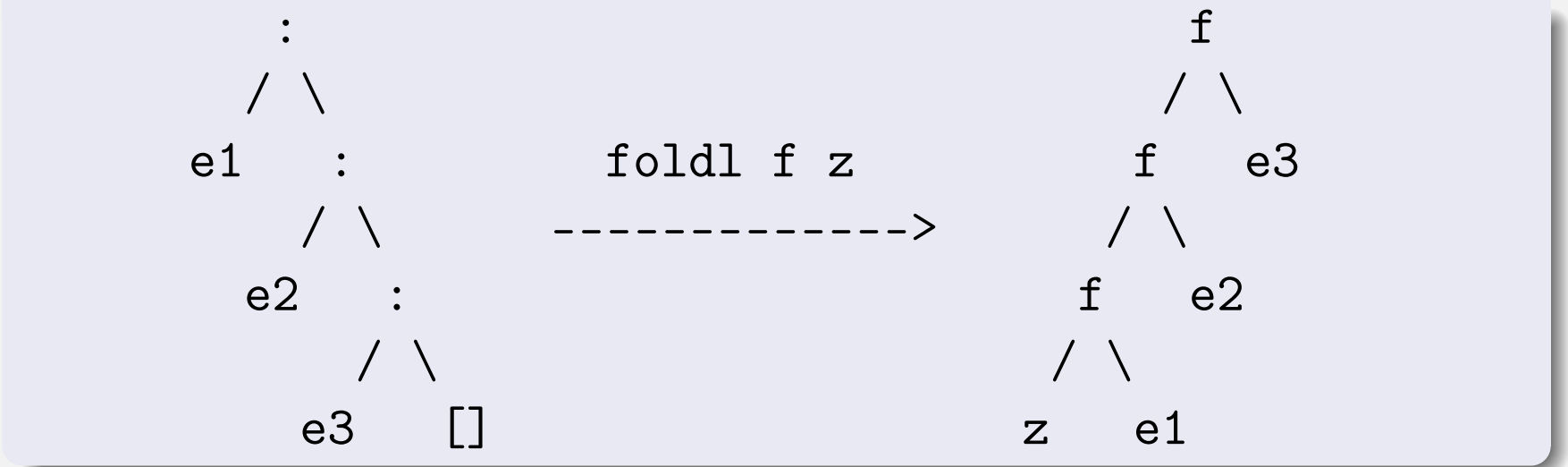
А что получится в результате такой свертки?

```
foldr (:) []
```

Левая свертка

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

```
e1 : (e2 : (e3 : []))  -->  ((z `f` e1) `f` e2) `f` e3
```



Рекурсия хвостовая — оптимизируется. Однако thunk из цепочки вызовов `f` нарастает.

Строгая версия левой свертки

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f z []     =  z
foldl f z (x:xs) =  foldl f acc xs
  where acc = f z x
```

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

- Теперь thunk из цепочки вызовов `f` не нарастает — вычисление `acc` форсируется на каждом шаге.
- Это самая эффективная из сверток, но все левые свертки не умеют работать с бесконечными списками.

«Продуктивность» правой свертки

Правая свертка дает поработать сворачивающей функции

```
all :: (a -> Bool) -> [a] -> Bool
all p = foldr (\x b -> p x && b) True
(&&) :: Bool -> Bool -> Bool
True  && x  =  x       -- (1)
False && _  =  False   -- (2)
```

```
all (<2) [1..]                                          -- def all
⇝foldr (\x b -> x<2 && b) True [1..]                    -- foldr PM
⇝foldr (\x b -> x<2 && b) True (1 : [2..])              -- foldr(2)
⇝(\x b -> x<2 && b) 1 (foldr (\x b -> x<2 && b) True [2..])
⇝(1<2) && (foldr (\x b -> x<2 && b) True [2..])         -- (&&)PM
⇝True  && (foldr (\x b -> x<2 && b) True [2..])         -- (&&)(1)
⇝foldr (\x b -> x<2 && b) True (2 : [3..])              -- foldr(2)
⇝(\x b -> x<2 && b) 2 (foldr (\x b -> x<2 && b) True [4..])
⇝(2<2) && (foldr (\x b -> x<2 && b) True [3..])         -- (&&)PM
⇝False && (foldr (\x b -> x<2 && b) True [3..])         -- (&&)(2)
⇝False
```

Версии сверток без начального значения

Для непустых списков можно обойтись без инициализатора

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

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

Аналогично реализована строгая версия `foldl1'`.

Сканы

Представляют собой списки последовательных шагов свертки.

```
scanl (#) z [a, b, ...] ≡ [z, z # a, (z # a) # b, ...]
```

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

```
GHCi> scanl (++) "Result: " ["A","B","C"]
["Result: ","Result: A","Result: AB","Result: ABC"]
GHCi> scanl (*) 1 [1..] !! 5
120
```

Можно и с бесконечными списками (в отличие от `foldl`).

Правый скан

Правый скан накапливает результаты справа налево.

```
scanr :: (a -> b -> b) -> b -> [a] -> [b]
scanr _ z []         = [z]
scanr (#) z (x:xs) = (x # q) : qs
  where qs@(q:_) = scanr (#) z xs
```

```
GHCi> scanr (+) 0 [1,2,3]
[6,5,3,0]
GHCi> scanr (++) " obtained" ["A","B","C"]
["ABC obtained","BC obtained","C obtained"," obtained"]
```

Для сканов выполняются следующие тождества

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

План лекции

Развертка

Развертка — операция двойственная к свертке.

```
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> h b n = if n > b then Nothing else Just (n,succ n)
GHCi> enumFT a b = unfoldr (h b) a
GHCi> enumFT 3 10
[3,4,5,6,7,8,9,10]
```

Еще пример: возможное определение `iterate`

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

Функция build

Функция build — другой способ организовать развертку

```
build :: (forall b. (a -> b -> b) -> b -> b) -> [a]
build g = g (:) []
```

Полиморфизм второго ранга в ее типе гарантирует полиморфную по b реализацию g.

```
GHCi> import GHC.Exts (build)
GHCi> g c n = c 'H' (c 'i' (c '!' n))
GHCi> build g
"Hi!"
GHCi> g' c n = go 'A' where go x = x `c` go (succ x)
GHCi> take 10 $ build g'
"ABCDEFGHIJ"
```

Этот механизм развертки используют для высокоуровневых оптимизаций.

Правило foldr/build (short cut fusion)

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr c n []     =  n
foldr c n (x:xs) =  x `c` (foldr c n xs)

build :: (forall b. (a -> b -> b) -> b -> b) -> [a]
build g = g (:) []
```

Если в программе отсутствуют вызовы seq, то имеет место следующая эквивалентность

```
foldr f z (build g)  ≡  g f z
```

Большинство библиотечных функции над списками рассматриваются как продюсеры или консьюмеры и перевыражаются через build или foldr.

Высокоуровневые оптимизации через foldr/build

```
iterate :: (a -> a) -> a -> [a]
iterate f x =  x : iterate f (f x)

iterateFB :: (a -> b -> b) -> (a -> a) -> a -> b
iterateFB c f y = go y where go x = x `c` go (f x)

{-# RULES
"iterate"    [~1] forall f x. iterate f x =
                       build (\c _ -> iterateFB c f x)
"fold/build" forall k z g. foldr k z (build g) = g k z
"iterateFB" [1] iterateFB (:) = iterate
#-}
{-# INLINE [1] build #-}
```

Фазы симплификатора (4,3,2,1,0), тильда – до, без – после.
Подробнее: текст (eng), мое видео (рус).

План лекции

Определение полугруппы

Полугруппа — это множество с ассоциативной бинарной операцией над ним.

```
infixr 6 <>
class Semigroup a where
  (<>) :: a -> a -> a
```

Для любой полугруппы должен выполняться закон:

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

Список — полугруппа относительно конкатенации `(++)`.

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

Полное определение полугруппы: stimes

```
infixr 6 <>
class Semigroup a where
  (<>) :: a -> a -> a

  sconcat :: NonEmpty a -> a

  stimes :: Integral b => b -> a -> a
  stimes = stimesDefault
```

```
GHCi> stimes 5 "Ab"
"AbAbAbAbAb"
```

Какова сложность stimes?

Полное определение полугруппы: sconcat

```haskell
infixr 5 :|
data NonEmpty a = a :| [a]
class Semigroup a where
  {-# MINIMAL (<>) | sconcat #-}
  (<>) :: a -> a -> a
  a <> b = sconcat (a :| [b])
  sconcat :: NonEmpty a -> a
  sconcat (a :| as) = foldr (<>) a as
  stimes :: Integral b => b -> a -> a
```

```
GHCi> import Data.List.NonEmpty (NonEmpty(..), fromList)
GHCi> sconcat $ "AB" :| ["CDE","FG"]
"ABCDEFG"
GHCi> sconcat $ fromList $ ["AB","CDE","FG"]
"ABCDEFG"
```

Определение моноида

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

```haskell
class Semigroup a => Monoid a where
  {-# MINIMAL mempty | mconcat #-}
  mempty  :: a
  mempty = mconcat []

  -- In a future GHC release will be removed
  mappend :: a -> a -> a
  mappend = (<>)

  mconcat :: [a] -> a
  mconcat = foldr mappend mempty
```

Законы моноида

```
class Semigroup a => Monoid a where
  mempty  :: a
  mappend :: a -> a -> a
  mconcat :: [a] -> a
```

Для любого моноида должны выполняться законы

```
mempty <> x  ≡  x

x <> mempty  ≡  x

mconcat ≡ foldr (<>) mempty
```

Конечно же `mappend` должен быть ассоциативным.

Реализация представителей моноида

Список — моноид относительно конкатенации (++), нейтральный элемент — это пустой список.

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

Реализация представителей моноида

Список — моноид относительно конкатенации (++), нейтральный элемент — это пустой список.

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

А Bool — моноид?

Реализация представителей моноида

Список — моноид относительно конкатенации (++), нейтральный элемент — это пустой список.

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

А Bool — моноид?

- Да, причем дважды: относительно конъюнкции (&&) и дизъюнкции (||).
- Чтобы реализовать разные интерфейсы для одного типа, упакуем его в обертки newtype.

Реализация представителей моноида: Bool

```
newtype All = All { getAll :: Bool }
  deriving (Eq, Ord, Read, Show, Bounded)
instance Semigroup All where
  All x <> All y = All (x && y)
  stimes = stimesIdempotentMonoid
```

```
newtype Any = Any { getAny :: Bool }
  deriving (Eq, Ord, Read, Show, Bounded)
instance Semigroup Any where
  Any x <> Any y = Any (x || y)
  stimes = stimesIdempotentMonoid
```

Каковы должны быть реализации моноида? (`mempty = ?`)

```
GHCi> getAny . mconcat . map Any $ [False,False,True]
True
```

Реализация представителей моноида: числа

А числа — моноид?

Реализация представителей моноида: числа

А числа — моноид?

Да, причем четырежды (собственно числа, то есть представители `Num` — дважды) :

- относительно сложения (нейтральный элемент это 0);
- относительно умножения (нейтральный элемент это ?);
- относительно `min` (нейтральный элемент это ?);
- относительно `max` (нейтральный элемент это ?).

Из-за полиморфизма чисел упаковки в `newtype` реализованы вокруг произвольного параметра.

Числа как моноид относительно сложения

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

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}
```

Что такое `mconcat` для `Sum a`? `stimes` для `Sum a`?

```
GHCi> Sum 2 * Sum 3 - Sum 5
Sum {getSum = 1}
```

Числа как моноид относительно умножения

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

instance Num a => Semigroup (Product a) where
  (<>) = coerce ((*) :: a -> a -> a)  -- Data.Coerce

instance Num a => Monoid (Product a) where
  mempty = Product 1
```

Для использования coerce из Data.Coerce нужно подключить расширение ScopedTypeVariables.

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

Что такое mconcat для Product a? stimes для Product a?

Моноид относительно min

Моноид относительно min формируют не только числа:

```haskell
newtype Min a = Min { getMin :: a }
  deriving (Eq, Ord, Read, Show, Bounded)

instance Ord a => Semigroup (Min a) where
  (<>) = coerce (min :: a -> a -> a)
  stimes = stimesIdempotent

instance (Ord a, Bounded a) => Monoid (Min a) where
  mempty = maxBound
```

```
GHCi> Min "Hello" <> Min "Hi"
Min {getMin = "Hello"}
GHCi> mempty :: Min Int
Min {getMin = 9223372036854775807}
```

Что такое mconcat для Min a? stimes для Min a?

Моноид и полугруппа Min

```
GHCi> (getMin . mconcat . map Min) [7,3,2,12] :: Int
2
GHCi> (getMin . mconcat . map Min) [] :: Int
9223372036854775807
```

Некоторые типы данных не формируют моноида `Min`, оставаясь полугруппой:

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

Это можно исправить перейдя от моноидальной `mconcat` к полугрупповой `sconcat` (и от списка к `NoEmpty`)

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

План лекции

1. Свертки списков
2. Развертки и оптимизации
3. Полугруппы и моноиды
4. Класс типов `Foldable`

Класс Foldable

```haskell
class Foldable t where
  foldr, foldr' :: (a -> b -> b) -> b -> t a -> b
  foldr f z t = appEndo (foldMap (Endo . f) t) z

  foldl, foldl' :: (a -> b -> a) -> 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

  ...
```

Класс Foldable (продолжение)

```
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

  {-# MINIMAL foldMap | foldr #-}
  ...
```

Представители класса Foldable

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

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

    foldl _ z Nothing = z
    foldl f z (Just x) = f z x
```

А также `Set` из `Data.Set`, `Map k` из `Data.Map`, `Seq` из `Data.Sequence`, `Tree` из `Data.Tree` и т.п.

Полное определение класса Foldable

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

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

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

  elem :: Eq a => a -> t a -> Bool

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

Законы Foldable

```
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
```

Если контейнер `t` не только `Foldable`, но и `Functor`, то

```
foldMap f ≡ fold . fmap f
foldMap f . fmap g ≡ foldMap (f . g)
```

Второе следует из первого благодаря закону `Functor`.

Обобщенные специальные свертки

Многие функции, исторически реализованные для списков, были обобщены до `Foldable`:

```
concat :: Foldable t => t [a] -> [a]
concatMap :: Foldable t => (a -> [b]) -> t a -> [b]

and,or :: Foldable t => t Bool -> Bool

any,all :: Foldable t => (a -> Bool) -> t a -> Bool

maximumBy,minimumBy
    :: Foldable t => (a -> a -> Ordering) -> t a -> a

notElem :: (Foldable t, Eq a) => a -> t a -> Bool

find :: Foldable t => (a -> Bool) -> t a -> Maybe a
```

Either как Foldable

```haskell
instance Foldable (Either e) where
  foldMap _ (Left _)  = mempty
  foldMap f (Right y) = f y

  foldr _ z (Left _)  = z
  foldr f z (Right y) = f y z

  length (Left _)  = 0
  length (Right _) = 1

  null             = isLeft
```

```
GHCi> maximum (Right 37)
37
GHCi> maximum (Left 37)
*** Exception: maximum: empty structure
```

Пара как Foldable

```
instance Foldable ((,) s) where
  foldMap f (_,y) = f y

  foldr f z (_,y) = f y z

  length _  = 1

  null _    = False
```

```
GHCi> foldr (+) 5 ("Answer",37)
42
GHCi> maximum (100,42)
???
```