Свёртки

Свёртки

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

minList [] = 0
minList (x:xs) = min x (minList xs)

concat [] = []
concat (xs:xss) = xs ++ (concat xss)
```

Свёртки

```
map f [] = []
map f (x:xs) = f x : (map f xs)

filter f [] = []
filter f (x:xs) =
  if (f x)
  then x : (filter f xs)
  else filter f xs
```

не забывайте об '

Свёртки

```
any f [] = False
any f (x:xs) = if (f x) then True else any f xs

all f [] = True
all f (x:xs) = if (f x) then all f xs else False



any f (x:xs) = f x || any f xs
all f (x:xs) = f x && all f xs
```

Правые свёртки

```
x1 # (x2 # (x3 # (... # u)))
```

	#	u
sum	+	0
minList	min	0 (-∞)
concat	++	[]
map	:	[]
filter	`(\x r -> if (f x) then x:r else r)`	[]
any	`(\x r -> f x || r)`	False
all	`(\x r -> f x && r)`	True

Правые свёртки

→ hash → h

```
foldr h u [] = u
foldr h u (x:xs) = h x (foldr h u xs)
```

```
sum list = foldr (+) 0 list

filter f list =
  foldr (\x r -> if (p x) then x:r else r) [] list
```

concat, any, all - самостоятельно

Функции с аккумулятором

```
sum list = sum' list 0
  where
  sum' [] acc = acc
  sum' (x:xs) acc = sum' xs (acc+x)


minList list = minList' list 0
  where
  minList' [] acc = acc
  minList' (x:xs) acc = minList' xs (min acc x)
```

Функции с аккумулятором

```
concat list = concat' list []
  where
  concat' [] acc = acc
  concat' (x:xs) acc = concat' xs (acc ++ x)


reverse list = reverse' list []
  where
  reverse' [] acc = acc
  reverse' (x:xs) acc = reverse' xs (x:acc)
```

Левые свёртки

```
(((u # x1) # x2) # .. ) # xn
```

	#	u
sum	+	0
minList	min	0 (-∞)
concat	++	[]
reverse	:	[]
any	`(\r x -> r || f x)`	False
all	`(\r x -> r && f x)`	True

Левые свёртки

```
foldl h u list = foldl' u list
  where
  foldl' u [] = u
  foldl' u (x:xs) = foldl' (h u x) xs

foldl h u []     =  u
foldl h u (x:xs) =  foldl f (f u x) xs

sum list = foldl (+) 0 list

reverse list = foldl (flip (:)) [] list

concat list = foldl (++) [] list
```

Свёртки

```
foldl (+) 0 [1..10] == 55
foldr (+) 0 [1..10] == 55
```

В чём подвох?

Какие типы у foldl и foldr?

Свёртки

```
foldl (+) 0 [1..10] == 55
foldr (+) 0 [1..10] == 55

В чём подвох?

Какие типы у foldl и foldr?

foldl (-) 0 [1..10] == -55
foldr (-) 0 [1..10] == -5
```

Свёртки

```
(((u # x1) # x2) # .. ) # xn
x1 # (x2 # (x3 # (... # u)))
```

Когда результаты левой и правой свертки совпадают?

Свёртки

список из 1 элемента: (u # x1) (x1 # u)

$\forall x$: u # x = x # u (1)

1) # оперирует над аргументами одного типа
2) u коммутирует с каждым элементом этого типа

не следует, что u - единица для #, но часто
u = 0,1,[],false,true...

Свёртки

пусть u – единичный элемент

∀a,b,c: ((u#a)#b)#c = a#(b#(c#u))
 → (a#b)#c = a#(b#c) (2)

- ассоциативная операция

(1) + (2) – результаты свёрток совпадают

Свёртки

```
foldl1, foldr1  :: (a -> a -> a) -> [a] -> a

foldl1 f (x:xs)  =  foldl f x xs

foldr1 f [x]     =  x
foldr1 f (x:xs)  =  f x (foldr1 f xs)   ?

sum list = foldl1 (+) list
sum list = foldr1 (+) list
```

Свёртки

```
:set +s

foldl1 (+) [1..10000000]
foldr1 (+) [1..10000000]

const :: a → b → a

foldl1 (const) [1..10000000]  ?
foldr1 (const) [1..10000000]  ?
```

Свёртки

```
:set +s

let a = map (show) [1..10000000]

foldl1 (++) a
foldr1 (++) a

foldr1 (flip const) [1..10000(00000000000)]
```

Префиксные суммы

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

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

вычсления последовательности промежуточных результатов свертки

Префиксные суммы

```
> scanl (+) 0 [1..10]
[0,1,3,6,10,15,21,28,36,45,55]

> scanr (+) 0 [1..10]
[55,54,52,49,45,40,34,27,19,10,0]

> scanl (-) 0 [1..10]
[0,-1,-3,-6,-10,-15,-21,-28,-36,-45,-55]

> scanr (-) 0 [1..10]
[-5,6,-4,7,-3,8,-2,9,-1,10,0]
```

попробуйте сами

Префиксные суммы

```
scanr h u []     = [u]
scanr h u (x:xs) = h x (head rest) : rest
  where rest = scanr h u xs

scanl h u list = scanl' [u] list
  where
  scanl' u [] = reverse u
  scanl' u (x:xs) = scanl' ((h (head u) x):u) xs
```

```
scanl h u xs  =  u : (case xs of
                        []   -> []
                        x:xs -> scanl h (h u x) xs)
```

Функции над деревьями

1) сумма элементов дерева
2) кол-во элементов дерева
3) из дерева в список

Свертки над деревьями

```
sumTree :: (Num a) => Tree a -> a
sumTree EmptyTree = 0
sumTree (Node a l r) = a + (sumTree l) + (sumTree r)

countTree :: Tree a -> Int
countTree EmptyTree = 0
countTree (Node a l r) = 1 + (countTree l) + (countTree r)

tree2list = см.л.10
```

свертки деревьев

- foldTree onEmpty _ EmptyTree = onEmpty
- foldTree onEmpty onNode (Node a l r) = onNode a (foldTree onEmpty onNode l) (foldTree onEmpty onNode r)

Свертки над деревьями

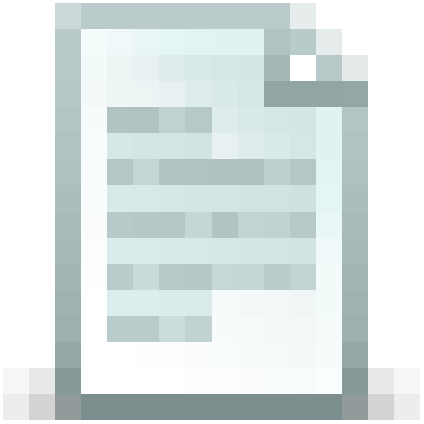
```
sumTree tree = foldTree 0 (\a l r -> a + l + r) tree

countTree = foldTree 0 (\a l r -> 1 + l + r)

tree2list = ?
```

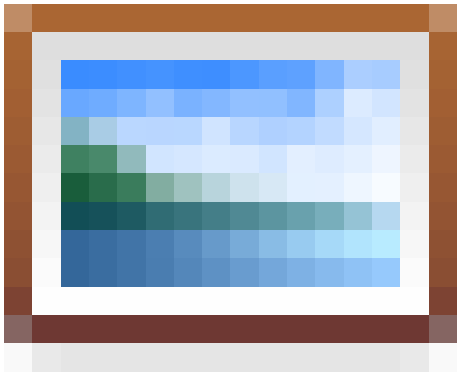
Свертки над деревьями

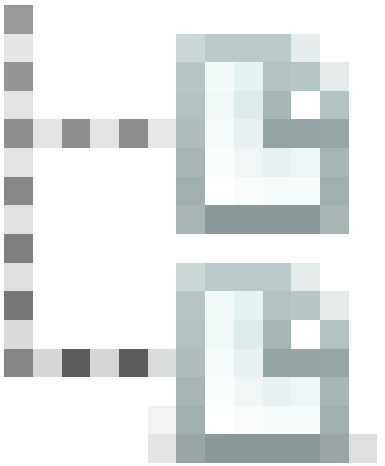
```
sumTree tree = foldTree 0 (\a l r -> a + l + r) tree

countTree = foldTree 0 (\a l r -> 1 + l + r)

tree2list = foldTree [] (\a l r -> l ++ [a] ++ r)
```

Синтетический пример

```
data Doc = Text String
   | Picture [Bool]
   | Composite [Doc]
   deriving (Show)
```







Синтетический пример

```
Text s - создает text-doc из строки
Picture img - создает picture-doc из массива бит
Composite docs - создает composite-doc из списка частей

isText (Text _) = True
isText _ = False – проверяет, является ли документ text-doc
isPicture (Picture _) = True
isPicture _ = False - является ли документ picture-doc
isComposite (Composite _) = True
isComposite _ = True – является ли документ composite-doc

Text d = doc - достает String из text-doc, в противном
случае ошибка
Picture d = doc - достает byte[] из picture-doc, в противном
случае ошибка
Composite d = doc - достает список частей из composite-doc,
в противном случае ошибка
```

Синтетический пример

Функция, составляющая список использованных в документе картинок в виде списка массивов байт:

Синтетический пример

Функция, составляющая список использованных в документе картинок в виде списка массивов байт:

```
findPictures (Text _) = error "documet is a text!"
findPictures (Picture pic) = pic
findPictures (Composite docs) =
  concat $ map findPictures $ filter (not . isText) docs
```

Синтетический пример

Функция, вычисляющая суммарную длину текста в документе

Синтетический пример

```
textLength (Text txt) = length txt
textLength (Picture _) = 0
textLength (Composite docs) = sum $ map textLength $ docs
```

Синтетический пример

Функция, заменяющую в документе все картинки на текст "<img />"

Синтетический пример

```
pic2tag  (Text txt) = Text txt
pic2tag  (Picture _) = Text "<img />"
pic2tag  (Composite docs) = Composite (map pic2tag docs)
```

Синтетический пример

Все такие функции тоже будут обладать общей структурой, и их тоже можно вычислять снизу вверх при помощи свертки

```
foldDoc t p c doc = case doc of
  Text txt -> t txt
  Picture pic -> p pic
  Composite docs -> c (map (foldDoc t p c) docs)


findPictures2 = foldDoc (\x -> []) (\x -> x) concat
```

В данном случае нельзя сказать, что функции получились короче или существенно читаемей. Однако, по крайней мере, теперь точно не будет ошибок в самой процедуре обхода - она написана и оттестирована всего 1 раз

Свертки rulez

В общем случае, операцию свертки можно аналогичным образом определить для любой древовидной структуры. Списки тоже являются частным случаем такой структуры