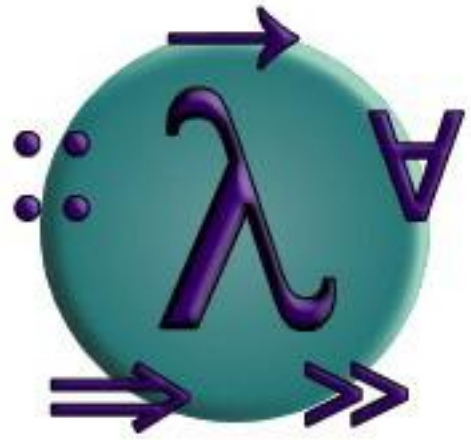
# PROGRAMMING IN HASKELL



Рекурсия и функции высших порядков

# Введение

Как известно, многие функции могут быть определены через другие функции

```
fac  :: Int → Int
fac n = product [1..n]
```

fac может быть определена через product (произведение) чисел от 1 до n.

Вычисления происходят поэтапно путем применения функции к её аргументам
Например:

```
  fac 4
=
  product [1..4]
=
  product [1,2,3,4]
=
  1*2*3*4
=
  2
  4
```

# Рекурсивные функции

В Хаскеле, как и в других языках программирования, функции, определенные через самих себя называются рекурсивными

```
fac 0 = 1
fac n = n * fac (n-1)
```

fac возвращает 1 от 0, для любого другого  целого факториал определяется как произведение текущего числа на  факториал предыдущего значения

Например:

```
fac 3
=
3 * fac 2
=
3 * (2 * fac 1)
=
3 * (2 * (1 * fac 0))
=
3 * (2 * (1 * 1))
=
3 * (2 * 1)
=
3 * 2
=
6
```

```
> fac (-1)

Exception: stack overflow
```

# Рекурсия в списках

Рекурсию можно использовать не только для чисел, но и для списков.

```
product       :: Num a ⇒ [a] → a
product []     = 1
product (n:ns) = n * product ns
```

product возвращает 1 для пустого списка, для непустого голова умножается на product от хвоста.

Например:

```
  product [2,3,4]
=
  2 * product [3,4]
=
  2 * (3 * product [4])
=
  2 * (3 * (4 * product []))
=
  2 * (3 * (4 * 1))
=
  24
```

Используя ту же схему, что и в product можем определить length (длину списка).

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

length для пустого списка возвращает 0, для непустого списка применяем ту же функцию к хвосту списка

Например:

```
  length [1,2,3]
=
  1 + length [2,3]
=
  1 + (1 + length [3])
=
  1 + (1 + (1 + length []))
=
  1 + (1 + (1 + 0))
=
  3
```

Используя аналогичный образец рекурсии мы можем определить функцию reverse в списках.

```
reverse        :: [a] → [a]
reverse []      = []
reverse (x:xs) = reverse xs ++ [x]
```

reverse возвращает пустой список для исходного пустого списка, для непустого формируем новый список : к голове списка добавляем revers для хвоста .

Например:

```
  reverse [1,2,3]
=
  reverse [2,3] ++ [1]
=
  (reverse [3] ++ [2]) ++ [1]
=
  ((reverse [] ++ [3]) ++ [2]) ++ [1]
=
  (([] ++ [3]) ++ [2]) ++ [1]
=
  [3,2,1]
```

# Несколько аргументов

Функции с более чем одним аргументом могут быть определены с помощью рекурсии. Например:

- Объединение элементов двух списков:

```
zip             :: [a] → [b] → [(a,b)]
zip []     _      = []
zip _      []     = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

- Удаление первых  n элементов из списка:

```
drop          :: Int → [a] → [a]
drop 0 xs       = xs
drop _ []       = []
drop n (_:xs) = drop (n-1) xs
```

- Объединение двух списков:

```
(++)          :: [a] → [a] → [a]
[]      ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

# Быстрая сортировка

- Пустой список считается отсортированным;
- В непустом списке формируется два подсписка для дальнейшей сортировки – в одном находятся элементы, меньше по значению, чем голова, в другом - элементы, большие , чем голова. Процедура сортировки повторяется для каждого из полученных списков.

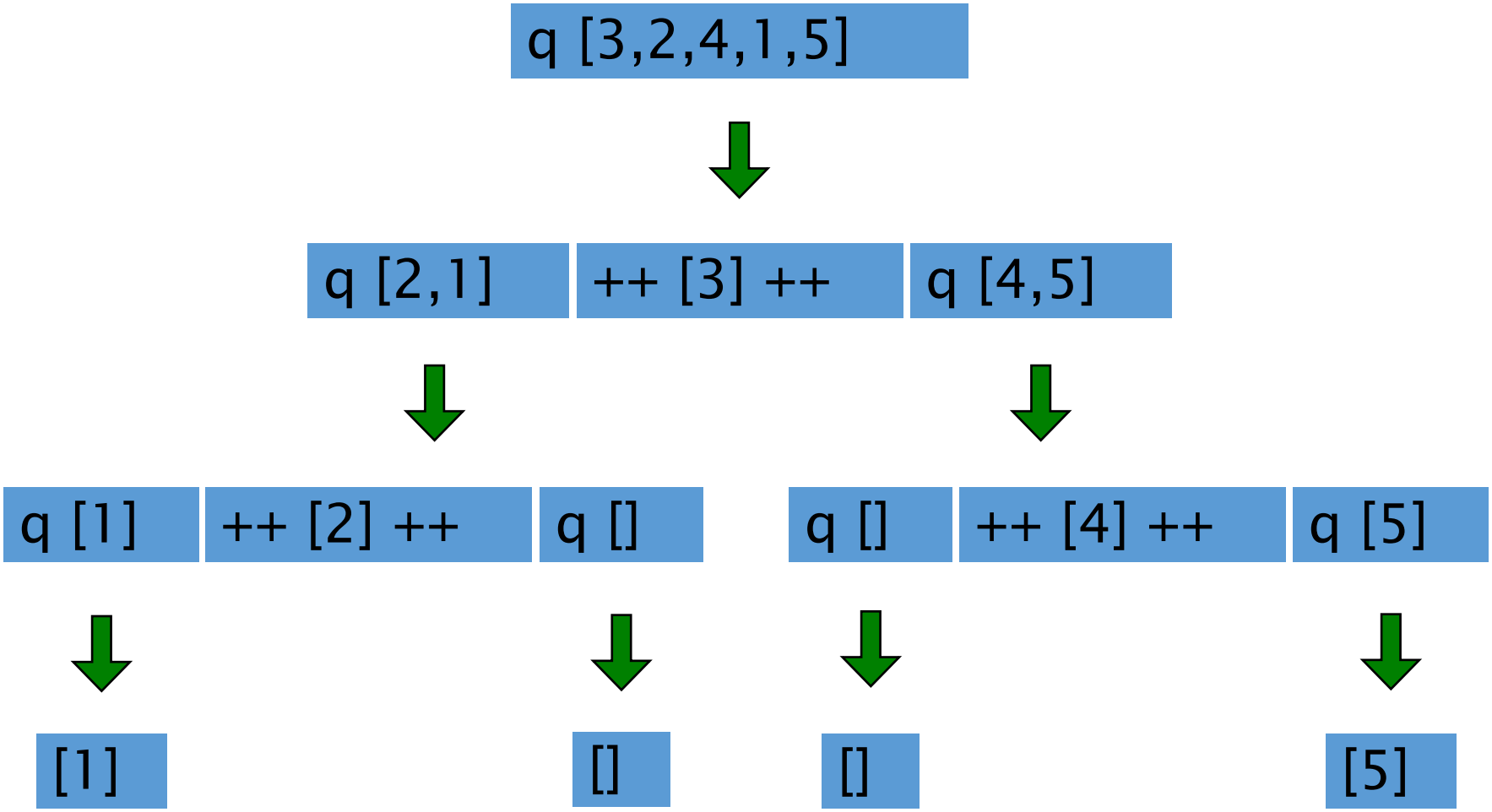
Используя рекурсию, опишем данный алгоритм:

```
qsort        :: Ord a ⇒ [a] → [a]
qsort []      = []
qsort (x:xs) =
   qsort smaller ++ [x] ++ qsort larger
   where
      smaller = [a | a ← xs, a ≤ x]
      larger  = [b | b ← xs, b > x]
```

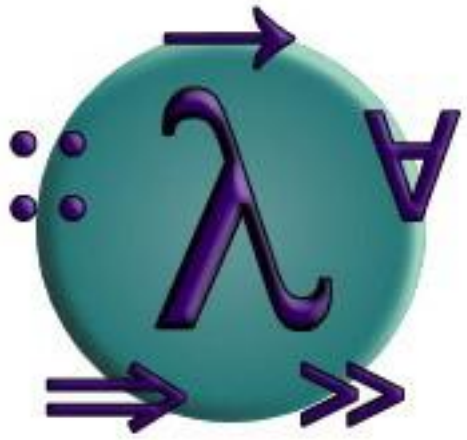
Note:

- Данный вариант является, пожалуй, самой простой реализацией сортировки

Сократим qsort как q



# PROGRAMMING IN HASKELL



Функции высших порядков

# Введение

Функции, которые принимают в качестве аргумента функцию или возвращают функцию в качестве результата называются функциями высших порядков.

```
twice    :: (a → a) → a → a
twice f x = f (f x)
```

twice принимает в качестве первого аргумента функцию.

# Функция map

Применяет заданную функцию к каждому элементу списка

```
map :: (a → b) → [a] → [b]
```

Например:

```
> map (+1) [1,3,5,7]

[2,4,6,8]
```

Функция map  может быть определена через списковые конструкции:

```
map f xs = [f x | x ← xs]
```

Либо через рекурсию:

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

# Функция filter

Функция filter выбирает каждый элемент списка, который удовлетворяет заданному предикату.

```
filter :: (a → Bool) → [a] → [a]
```

Например:

```
> filter even [1..10]

[2,4,6,8,10]
```

filter легко может быть определена через списки:

```
filter p xs = [x | x ← xs, p x]
```

А также рекурсию:

```
filter p []     = []
filter p (x:xs)
  | p x         = x : filter p xs
  | otherwise   = filter p xs
```

# функция foldr

Некоторые функции для обработки списков могут быть определены по следующей схеме рекурсии:

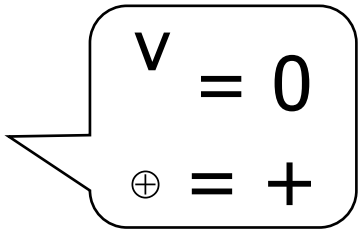
```
f []     = v
f (x:xs) = x ⊕ f xs
```

f отображает пустой список в некоторое значение v, а непустой список – в некоторые действия с хвостом и головой списка (некоторая функция ⊕ применяется к голове, и f применяется к хвосту.

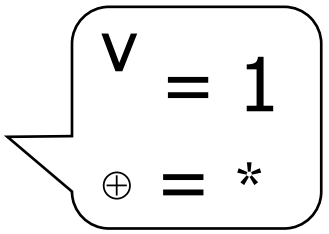
# Например:

```
f []     = v
f (x:xs) = x ⊕ f xs
```

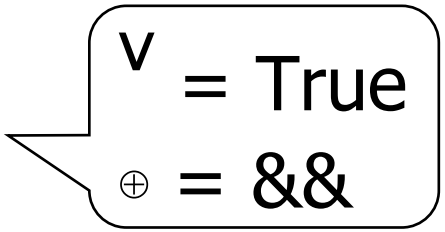
```
sum []     = 0
sum (x:xs) = x + sum xs
```




```
product []     = 1
product (x:xs) = x * product xs
```




```
and []     = True
and (x:xs) = x && and xs
```



Функция foldr (правая свертка) воплощает этот образец рекурсии, используя в качестве функции $\oplus$ и значение v в качестве аргументов.

Например:

```
sum     = foldr (+) 0

product = foldr (*) 1

or      = foldr (||) False

and     = foldr (&&) True
```

foldr может быть описана с помощью рекурсии:

```
foldr :: (a → b → b) → b → [a] → b

foldr f v []     = v

foldr f v (x:xs) = f x (foldr f v xs)
```

Хотя, можно воспринимать  foldr не рекурсивно, если заменять каждый (:) в списке на заданную функцию, а []  на заданное значение

Например:

```
    sum [1,2,3]
=
    foldr (+) 0 [1,2,3]
=
    foldr (+) 0 (1:(2:(3:[])))
=
    1+(2+(3+0))
=
    6
```

Заменяем каждый (:) на (+) и [] на 0.

Например:

```
  product [1,2,3]
=
  foldr (*) 1 [1,2,3]
=
  foldr (*) 1 (1:(2:(3:[])))
=
  1*(2*(3*1))
=
  6
```

Заменяем (:) на (*) и [] на 1.

# Другие примеры

Вспомним функцию length :

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

Например:

```
  length [1,2,3]
=
  length (1:(2:(3:[])))
=
  1+(1+(1+0))
=
  3
```

Заменим (:) на
λ_ n → 1+n
и [] на 0.

Таким образом :

```
length = foldr (λ_ n → 1+n) 0
```

Вспомним функцию reverse:

```
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Например:

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

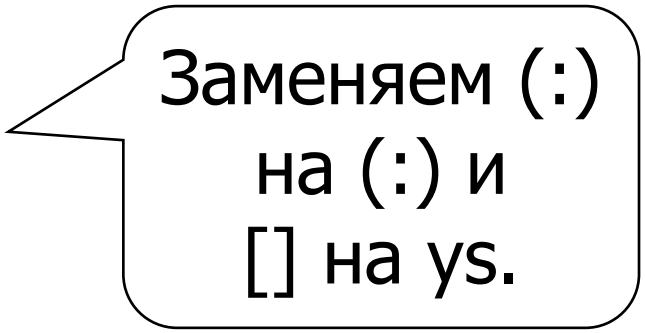
Заменяем (:)
на λx xs → xs ++ [x]
и [] на [].

получаем

```
reverse =
   foldr (λx xs → xs ++ [x]) []
```

Наконец, отметим, что функция (++) имеет более компактную реализацию через foldr:

```
(++ ys) = foldr (:) ys
```



# Другие функции

Функция (.) композиция

В математике композиция функций
когда результат работы одной функции полностью передается на вход другой функции.

```
(.)   :: (b → c) → (a → b) → (a → c)
f . g  = λx → f (g x)
```

Например:

композицией двух функций f и g называется функция , заключающаяся в последовательном применении к чему - то функции f , а затем функции g

```
odd :: Int → Bool
odd  = not . even
```

Функция all определяет, соответствует ли каждый элемент списка указанному предикату.

```
all     :: (a → Bool) → [a] → Bool
all p xs = and [p x | x ← xs]
```

Например:

```
> all even [2,4,6,8,10]

True
```

Функция any определяет, есть ли в указанном списке хотя бы один элемент, соответствующий предикату.

```
any     :: (a → Bool) → [a] → Bool
any p xs = or [p x | x ← xs]
```

Например:

```
> any (== ’ ’) "abc def"

True
```

Функция takeWhile выбирает элементы из списка, пока они соответствуют указанному предикату.

```
takeWhile :: (a → Bool) → [a] → [a]
takeWhile p []     = []
takeWhile p (x:xs)
   | p x           = x : takeWhile p xs
   | otherwise     = []
```

Например:

```
> takeWhile (/= ’ ’) "abc def"

"abc"
```

Обратная функция dropWhile удаляет из списка элементы, пока они соответствуют предикату.

```
dropWhile :: (a → Bool) → [a] → [a]
dropWhile p []     = []
dropWhile p (x:xs)
   | p x           = dropWhile p xs
   | otherwise     = x:xs
```

For example:

```
> dropWhile (== ' ') "       abc"

"abc"
```