

# План лекций

## Списки

### Введение

Компилятор ghc, ghci, Haskell Platform.

Haskell – чисто функциональный, типизированный язык программирования.

Чистые функции.

Типы Int, Integer, Float, Double, Bool = True | False, Char.

Арифметические операции.

`+`, `-`, `*`, `/`, `div`, `mod`

Тип функции:

`not :: Bool -> Bool`

`not False = True`

`not True = False`

`plus :: Int -> Int -> Int`

`plus x y = x + y`

`plus3 :: Int -> Int`

`plus3 = plus 3`

Кортежи (a,b).

`fst (x,y) = x`

`snd (x,y) = y`

`('a', True)`

Списки

`[a] = [] | a : [a]`

`[]`

`1:2:[]`

`[1,2]`

`[1..3] = [1,2,3]`

`[0,2..8] = [0,2,4,6,8]`

`[1,1.5..3] = [1.0,1.5,2.0,2.5,3.0]`

Конструктор списков (list comprehensions)

`[x | x <- [1..3]] = [1,2,3]`

`[(x,y) | x <- [1,2], y <- [1,2]] = [(1,1), (1,2), (2,1), (2,2)]`

`[(x,y) | x <- [1..3], y <- [1..4], x == y] = [(1,1), (2,2), (3,3)]`

## Базовые функции со списками

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

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

```
length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + length xs
```

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

```
(!!) :: [a] -> Int -> a
(x:_) !! 0 = x
(_:xs) !! n = xs !! (n-1)
```

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

```
reverse l = rev l [] where
    rev [] a = a
    rev (x:xs) a = rev xs (x:a)
```

```
[1,2,3] []
[2,3] 1:[]
[3] 2:1:[]
[] 3:2:1:[]
[] [3,2,1]
```

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

**drop**

## Бесконечные списки

```
[1..]
```

```
[2,4..]
```

```
take 5 [1..]
[1,2,3,4,5]
```

```
repeat :: a -> [a]
repeat x = x : repeat x
```

```
take 2 (repeat 3)
[3,3]
```

```

take 2 (3 : repeat 3)
3 : take 1 (repeat 3)
3 : take 1 (3 : repeat 3)
3 : 3 : take 0 (repeat 3)
3 : 3 : take 0 (3 : repeat 3)
3 : 3 : [] = [3,3]

```

```

($) :: (a -> b) -> a -> b
f $ x = f x

```

```

replicate :: Int -> a -> [a]
replicate n x = take n $ repeat x

```

```

cycle :: [a] -> [a]
cycle xs = xs ++ cycle xs

```

```

take 5 $ cycle [1,2]
[1,2,1,2,1]

```

```

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

```

Линейный генератор

```

f x = mod (5*x + 3) 11
take 5 $ iterate f 1
[1,8,10,9,4]

```

## Функции высших порядков

```

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

```

### dropWhile

```

filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs) = if p x then x : filter xs else filter xs

```

```

filter even [1..5]
[2,4]

```

```

filter (not . even) [1..5]
[1,3,5]

```

```

(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \x -> f (g x)

```

Решето Эратосфена

```

sieve :: [Integer] -> [Integer]
sieve (x:xs) = x : sieve (filter (\y -> y `mod` x /= 0) xs)

```

```

primes = sieve [2..]

```

## Map и zipWith

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

```
map (^2) [1..5]
[1,4,9,16,25]
```

```
map (2^) [1..5]
[2,4,8,16,32]
```

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith _ _ _ = []
```

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

```
fibs = 0:1:zipWith (+) fibs (tail fibs)
```

Вычисление в стратегии сверху вниз:

```
fib n = fibs !! n
```

```
fib 3
2
```

```
fibs !! 3
(0:1:zipWith (+) fibs (tail fibs)) !! 3
(1:zipWith (+) fibs (tail fibs)) !! 2
(zipWith (+) fibs (tail fibs)) !! 1
(0 + 1 : zipWith (+)
  (1:zipWith (+) fibs (tail fibs))
  (zipWith (+) fibs (tail fibs))) !! 1
(zipWith (+)
  (1:zipWith (+) fibs (tail fibs))
  (zipWith (+) fibs (tail fibs))) !! 0
(zipWith (+)
  (1:zipWith (+) fibs (tail fibs))
  (zipWith (+)
    (0:1:zipWith (+) fibs (tail fibs))
    (1:zipWith (+) fibs (tail fibs)))) !! 0
(zipWith (+)
  (1:zipWith (+) fibs (tail fibs))
  (0 + 1 : zipWith (+)
    (1:zipWith (+) fibs (tail fibs))
    (zipWith (+) fibs (tail fibs)))) !! 0
(1 + 1 : zipWith (+)
  (zipWith (+) fibs (tail fibs))
  (zipWith (+)
    (1:zipWith (+) fibs (tail fibs))
    (zipWith (+) fibs (tail fibs)))) !! 0
2
```

В Haskell применяется измененная версия этой стратегии. Создается ссылка на fibs и fibs будет вычисляться только один раз, на каждом шаге добавляя новые элементы.

## Свёртка

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

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

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

```
sum = foldr (+) 0
concat = foldr (++) []
```

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

```
reverse = foldl (flip (·)) []
```

```
flip :: (a -> b -> c) -> b -> a -> c
flip f x y = f y x
```

```
foldr1 :: (a -> a -> a) -> [a] -> a
foldr1 f [x] = x
foldr1 f (x:xs) = f x foldr1 f xs
```

```
maximum = foldr1 max
```

```
filter p = foldr (\x xs -> if p x then x:xs else xs) []
map f = foldr ((·) . f) []
length = foldr (\_ n -> 1 + n) 0
```

## Data.List и сортировки

```
transpose [] = []
transpose ([] : xss) = transpose xss
transpose ((x:xs) : xss) =
  (x : [h | (h:_) <- xss]) : transpose (xs : [ t | (_:t) <- xss])
```

```
qsort :: Ord a => [a] -> [a]
qsort [] = []
qsort (x:xs) = qsort (filter (<=x) xs) ++ [x] ++ qsort (filter (>x) xs)
```

```
isort :: Ord a => [a] -> [a]
isort [] = []
isort (x:xs) = insert x (isort xs) where
  insert x [] = [x]
  insert x ys@(y:ys') | x > y = y : insert x ys'
  | otherwise = x : ys
```

```

msort :: Ord a => [a] -> [a]
msort = mergeAll . sequences
  where
    sequences (a:b:xs)
      | a > b      = descending b [a] xs
      | otherwise = ascending b (a:) xs
    sequences xs = [xs]

    descending a as bs@(b:bs')
      | a > b      = descending b (a:as) bs'
    descending a as bs = (a:as): sequences bs

    ascending a as bs@(b:bs')
      | a <= b      = ascending b (\ys -> as (a:ys)) bs'
    ascending a as bs = as [a]: sequences bs

    mergeAll [x] = x
    mergeAll xs = mergeAll (mergePairs xs)

    mergePairs (a:b:xs) = merge a b: mergePairs xs
    mergePairs xs       = xs

    merge as@(a:as') bs@(b:bs')
      | a > b      = b:merge as bs'
      | otherwise = a:merge as' bs
    merge [] bs = bs
    merge as [] = as

```

## Монады

### Maybe

```
data Maybe a = Nothing | Just a
```

```

safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead (x:_) = Just x

```

```

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

```

```

lookup 2 [(1,"one"), (2,"two"), (3,"three")]
Just "two"

```

```

lookup 4 [(1,"one"), (2,"two"), (3,"three")]
Nothing

```

```

class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b

```

```
instance Monad Maybe where
  return x = Just x
  Nothing >>= f = Nothing
  (Just x) >>= f = f x
```

Последовательный поиск в нескольких списках (аналогичный подход при поиске в БД).

```
empDep      = [( "Mike", "It"), ( "Jan", "Sales")]
depCountry  = [( "It", "Japan"), ( "Sales", "USA")]
countryCurrency = [( "Japan", "JPY"), ( "USA", "USD")]
currencyRate = [( "JPY", 112), ( "USD", 1)]

f :: String -> Maybe Int
f emp = case lookup emp empDep of
  Nothing -> Nothing
  Just dep -> case lookup dep depCountry of
    Nothing -> Nothing
    Just country -> case lookup country countryCurrency of
      Nothing -> Nothing
      Just curr -> lookup curr currencyRate

fB emp = lookup' empDep emp >>= lookup' depCountry >>= lookup' countryCurrency
  >>= lookup' currencyRate where
    lookup' ps k = lookup k ps

fD emp = do dep <- lookup emp empDep
  country <- lookup dep depCountry
  currency <- lookup country countryCurrency
  lookup currency currencyRate
```

## List

```
instance Monad [a] where
  return x = [x]
  xs >>= f = concat (map f xs)
```

Построение грамматик.

```
f 'a' = "ca"
f 'b' = "bb"
f 'c' = "a"
```

```
take 3 $ iterate (>>= f) "abc"
["abc", "cabba", "acabbbbca"]
```

## IO

```
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
  (>>) :: m a -> m b -> m b
  m >> k = m >>= \x -> k
```

```
putChar :: Char -> IO ()
```

```
putStrLn :: [Char] -> IO ()
putStrLn xs = foldr (>>) (return ()) (map putChar xs) >> putChar '\n'
```

```
getChar :: IO Char
```

```
getLine :: IO [Char]
```

```
getLine = do x <- getChar
             if x == '\n'
             then return ""
             else do xs <- getLine
                    return (x:xs)
```

Структура программы.

Создайте файл `hello.hs`, со следующим содержимым и скомпилируйте: `ghc hello.hs`.

```
module Main where
```

```
main :: IO ()
```

```
main = putStrLn "Hello ,_World!"
```