

План лекций

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:

`and :: Bool -> Bool -> Bool`

`and False _ = False`

`and True x = x`

Кортежи (a,b). `fst`, `snd`.

Списки

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

`[]`

`1:2:[]`

`[1,2]`

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

`[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`

`(++) :: [a] -> [a] -> [a]`

`(++) [] ys = ys`

`(++) (x:xs) ys = x : (xs ++ ys)`

`(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)
```

```
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
```

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 _ _ _ = []
```

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

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

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

```
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

```

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

```

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

```