НЕИЗМЕНЯЕМЫЕ СТРУКТУРЫ ДАННЫХ

(PERSISTENT DATA STRUCTURES)

Арсений Жижелев, Праймтолк / zhizhelev@primetalk.ru

ПЛАН

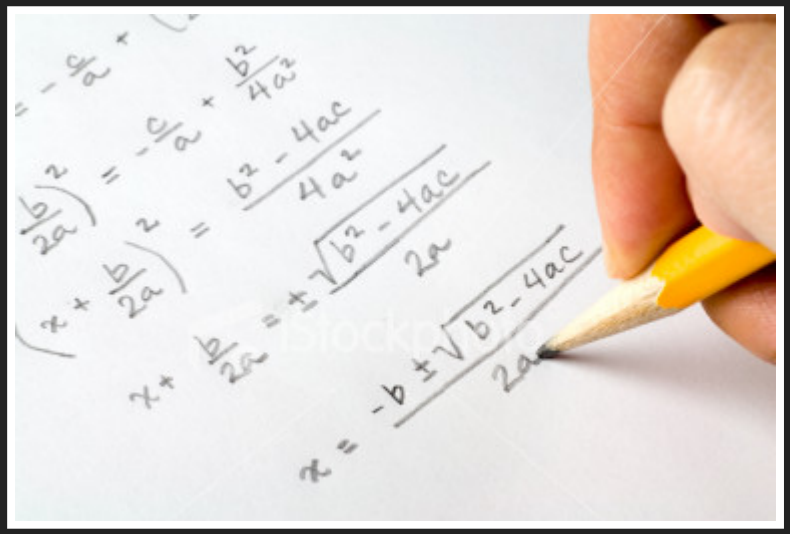
- Введение
- Аналогии из реального мира
- Конструирование новых данных
- Некоторые красивые структуры данных
- Структуры данных, представляющие вычисления
- Корректное программирование

Введение

- Времена изменились
- Высокоуровневые программы
- Многоядерность (с 2005 г.)
- Неизменяемые данные оказались **удобнее**

Аналогии

- Математические выкладки
- Построение рисунков добавлением элементов
- Сборка модели из деталей конструктора



Конструирование новых данных

1. Раскладывание по контексту имен (связывание имён)
2. Промежуточные вычисления
3. Сборка результата

Конструирование новых данных (пример 1, filter)

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

```scala
def filter[A](p: A => Boolean)(list: List[A]): List[A] =
  list match {
    case Nil                  => Nil
    case x :: xs if p(x)      => x :: filter(p)(xs)
    case _ :: xs              => filter(p)(xs)
  }
```

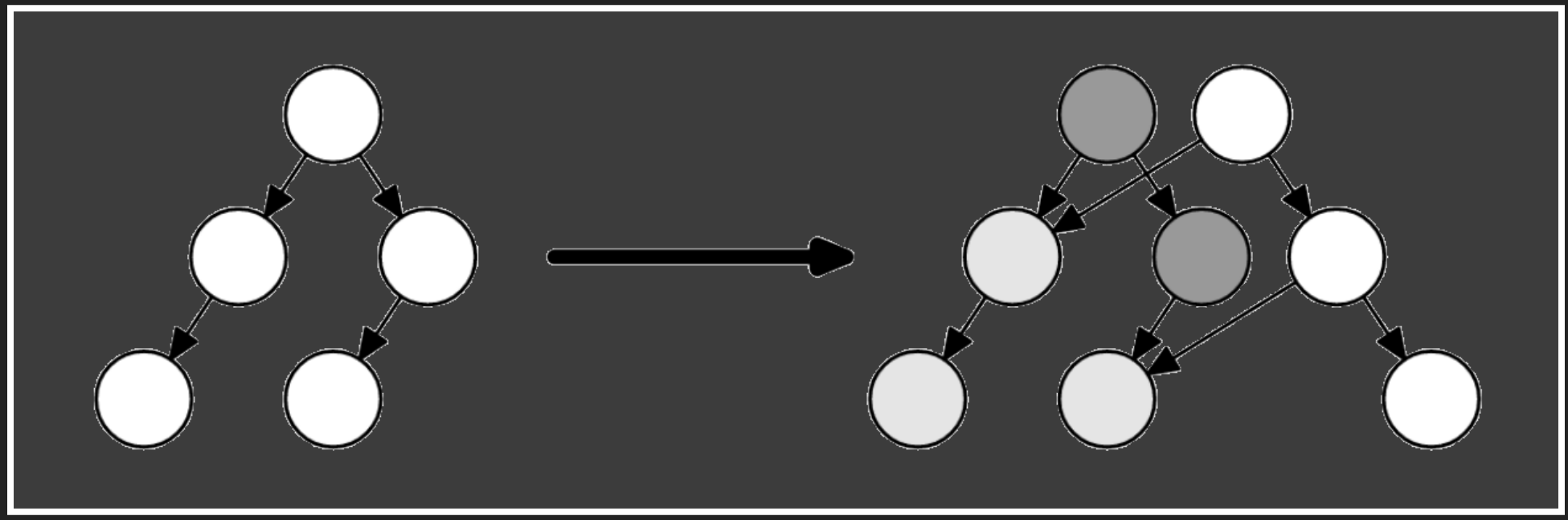
```scala
def filter[A](p: A => Boolean)(list: List[A]): List[A] =
  list match {
    ...
    case x :: xs =>
      val flag =  p(x)
      val filteredTail = filter(p)(xs)
      val result = if(flag) x :: filtered else filtered
      result
    ...
  }
```

Конструирование новых данных (пример 2 разбиение на строки)

```haskell
lines           :: String -> [String]
lines ""        =  []
lines s         =  let (l, s') = break (== '\n') s
                     in  l : case s' of
                                 []      -> []
                                 (_:s'') -> lines s''
```

```scala
def lines(s: String): List[String] =
  s match {
    case ""   => Nil
    case _    =>
      val (l, s1) = break(_ == '\n')(s)
      val tail = s1 match {
                                 case ""       => Nil
                                 case _ :: s2 => lines(s2)
                               }
      l :: tail
  }
```

Модификация неизмяемых структур



Красивые структуры данных (список)

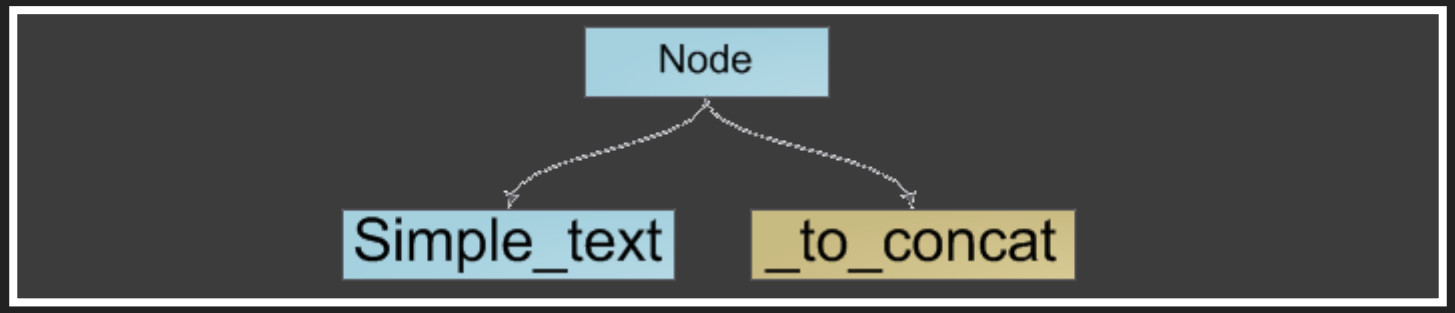
```
data List a = Nil | Cons a List a
```

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

```
sealed trait List[+T]
case object Nil extends List[Nothing]
final case class Cons[+T](head: T, tail: List[T])
  extends List[T]
```

```
sealed abstract class List[+A] extends AbstractSeq[A] with ..
case object Nil extends List[Nothing]
final case class ::[B](
  private var hd: B,
  private[scala] var tl: List[B]) extends List[B]
```

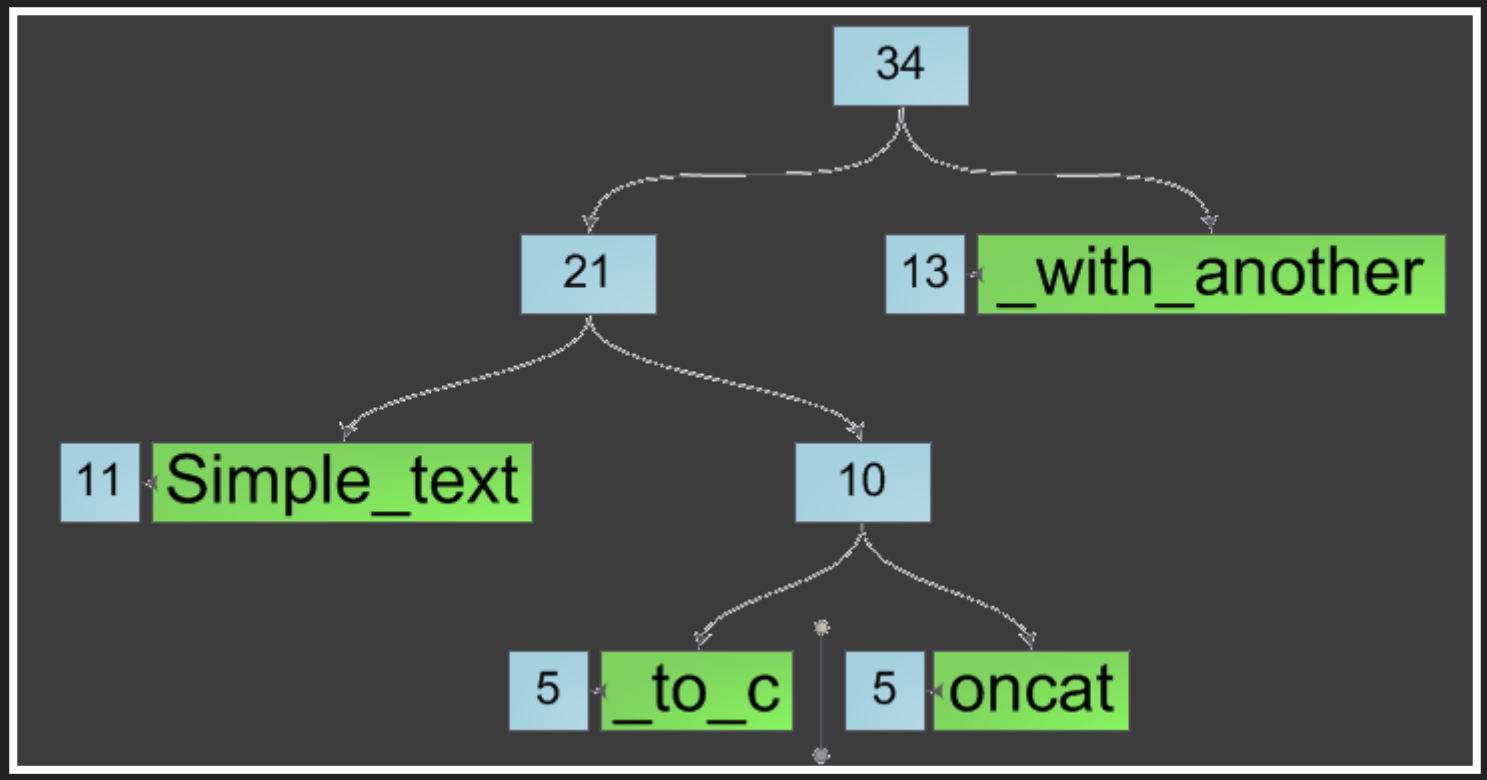
Красивые структуры данных (Rope/веревка)

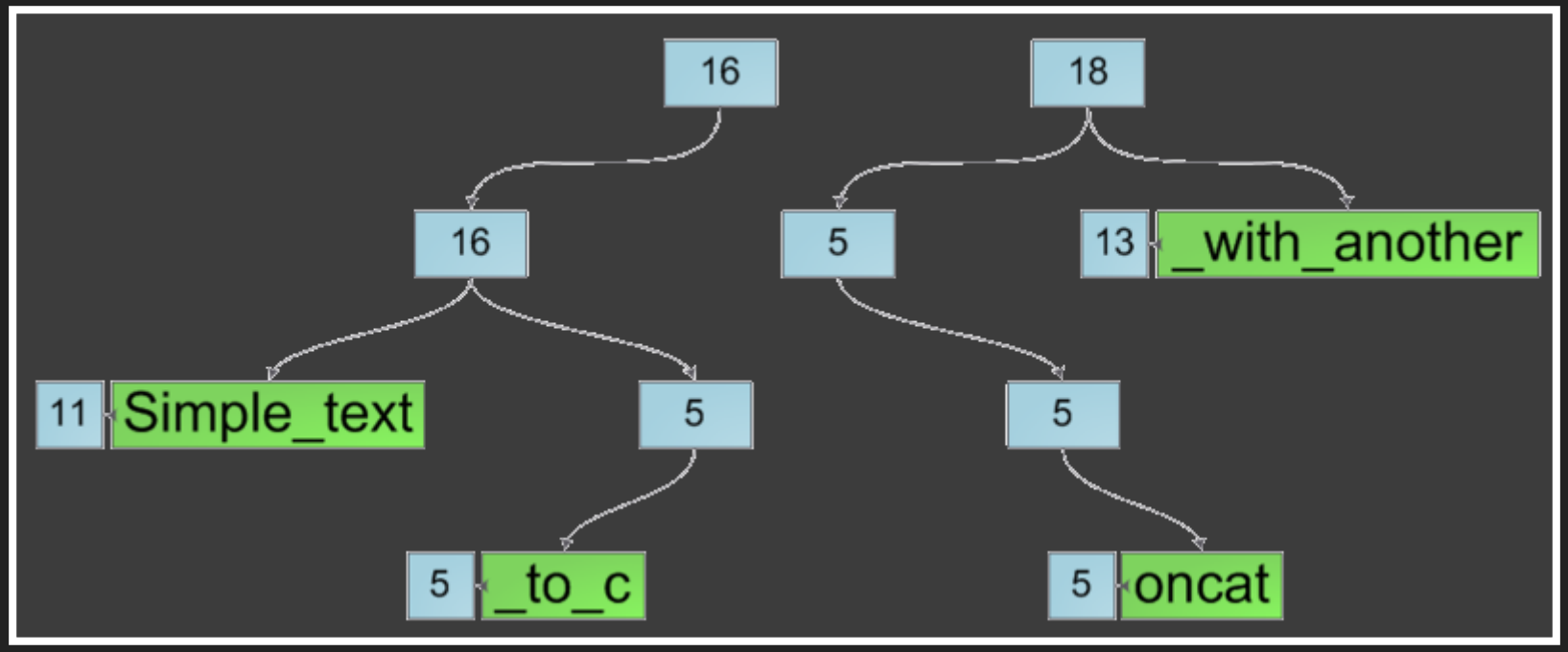


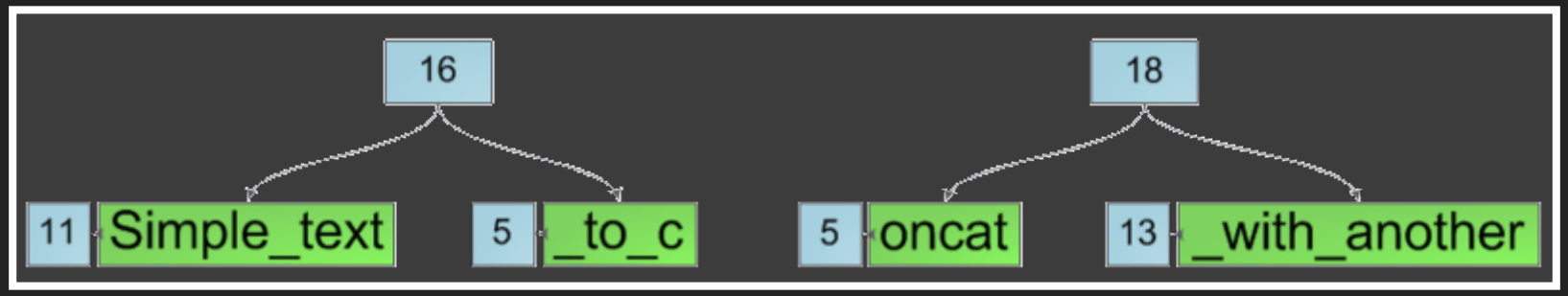
Производительность

- Index O(log n)
- Concat O(log n) (в худшем случае - O(n))
- Split O(log n)
- Insert O(log n) (в худшем случае - O(n))
- Delete O(log n)

Красивые структуры данных (Rope, разрезание)




16
18
16
5
13
_with_another
11
Simple_text
5
5
5
_to_c
5
oncat


16
18
11
Simple_text
5
_to_c
5
oncat
13
_with_another

Красно-чёрные деревья

- Обычные деревья бинарного поиска
- К узлам добавлен цвет - красный/чёрный
- И поддерживаются инварианты:
  - красные узлы не могут ссылаться на красные узлы
  - количество чёрных узлов от вершины до пустых узлов одинаково для всего дерева
- В результате: дерево полностью сбалансировано

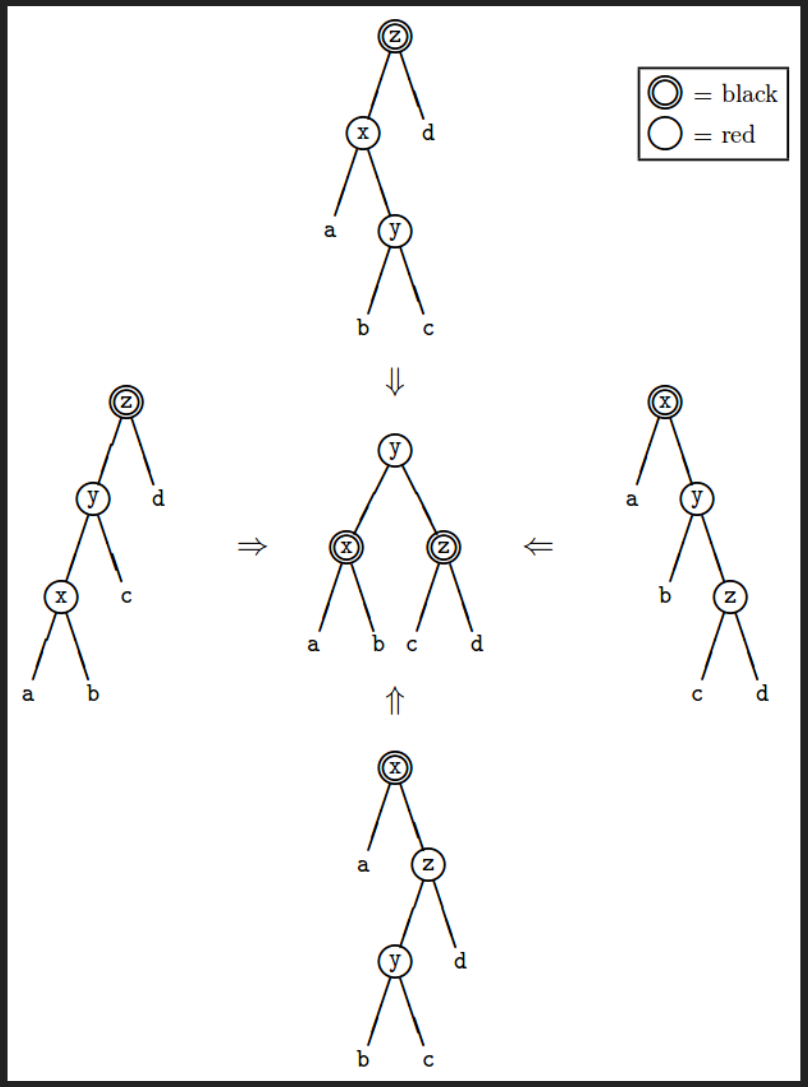
Красно-чёрные деревья (модель)

```
data Color    = R | B
data Tree elt = E | T Color (Tree elt) elt (Tree elt)
```

```
sealed trait Color
case object Red extends Color
case object Black extends Color

sealed trait Tree[+A]
case object E extends Tree[Nothing]
case class T[A](
  color: Color,
  left: Tree[A],
  x: A,
  right: Tree[A]
) extends Tree[A]
```

Красно-чёрные деревья (балансировка)



```
def balance[A: Ordering](n: T[A]): Tree[A] = n match {
  case T(Black, T(Red, T(Red, a, x, b), y, c), z, d) => ???
  case T(Black, T(Red, a, x, T(Red, b, y, c)), z, d) => ???
  case T(Black, a, x, T(Red, T(Red, b, y, c), z, d)) => ???
  case T(Black, a, x, T(Red, b, y, T(Red, c, z, d))) => ???
  case t                                             => t
}
```

```
def balance[A: Ordering](n: T[A]): Tree[A] = n match {
  case ... => T(Red, T(Black, a, x, b), y, T(Black, c, z, d))
  case ... => T(Red, T(Black, a, x, b), y, T(Black, c, z, d))
  case ... => T(Red, T(Black, a, x, b), y, T(Black, c, z, d))
  case ... => T(Red, T(Black, a, x, b), y, T(Black, c, z, d))
  case t   => t
}
```

Вычисления как данные

- lazy val
- by ref
- Task (fs2) (~ IO)
- Stream (fs2)

Stream (fs2) пример

```
def fahrenheitToCelsius(f: Double): Double =
  (f - 32.0) * (5.0/9.0)
def converter[F[_]: Sync]: F[Unit] =
  io.file.readAll[F](Paths.get("fahrenheit.txt"), 4096)
    .through(text.utf8Decode)
    .through(text.lines)
    .filter(s => !s.trim.isEmpty && !s.startsWith("//"))
    .map(line => fahrenheitToCelsius(line.toDouble).toString)
    .intersperse("\n")
    .through(text.utf8Encode)
    .through(io.file.writeAll(Paths.get("celsius.txt")))
    .run

val u: Unit = converter[IO].unsafeRunSync()
```

Корректное программирование

- Программа делает то, что от неё ожидают:
  - Не зависает, не "падает" с исключениями
  - Результаты правильные
  - Производительность соответствует ожиданиям
- Как убедиться, что программа отвечает нашим ожиданиям?

Корректное программирование (пример 1 merge sort JavaScript)

```
function merge(left, right, arr) {    // 1,2,3
  var a = 0;                          // 4

  while (left.length && right.length) {
    arr[a++] = (right[0] < left[0]) ? right.shift() // ~
                                    : left.shift();
  }
  while (left.length) {                             // ?
    arr[a++] = left.shift();
  }
  while (right.length) {                            // len
    arr[a++] = right.shift();                       // exc?
  }
}
```

```
function mergeSort(arr) {
  var len = arr.length;

  if (len === 1) { return; }

  var mid = Math.floor(len / 2),
      left = arr.slice(0, mid),
```

```
    right = arr.slice(mid);

  mergeSort(left);
  mergeSort(right);
  merge(left, right, arr);
}
```

Корректное программирование (merge sort Haskell)

```
merge []          ys                      = ys // triv 1
merge xs          []                      = xs // 2
merge xs@(x:xt) ys@(y:yt) | x <= y        = x : merge xt ys // 6
                          | otherwise     = y : merge xs yt
                // head
                // first = min
                // rec short
```

```
split (x:y:zs) = let (xs,ys) = split zs in (x:xs,y:ys)
split [x]      = ([x],[])
split []       = ([],[])

mergeSort []  = []
mergeSort [x] = [x]
mergeSort xs  = let (as,bs) = split xs
                in merge (mergeSort as) (mergeSort bs)
```

Корректное программирование - неизменяемые данные

- Все имена - константы в пределах области видимости
- Нет эфемерного состояния
- Нет неожиданных побочных эффектов

Производительность (память)

- Результаты вычислений - новые структуры данных
- Сборка мусора!
- Алгоритмы - специальные
- Просто работает

Заключение

- Неизменяемые данные - в арсенал программиста!
- В многопоточных программах - обязательно (heisenbug)
- В программах со сложной бизнес-логикой - обязательно

Вопросы?

Арсений Жижелев, Праймтолк / zhizhelev@primetalk.ru