Факультет
компиляторoварения

Язык Фикус

ML и обработка массивов

Обо мне

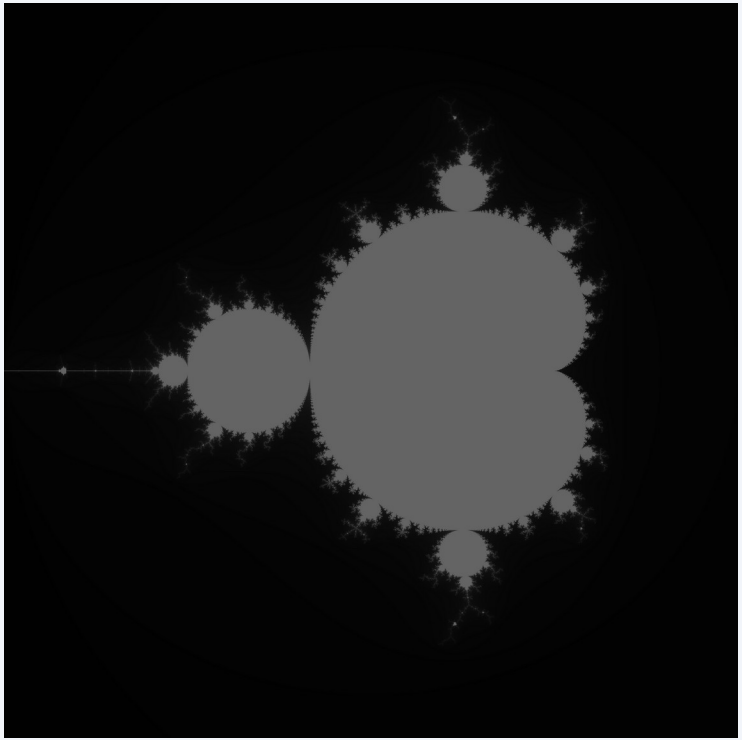
Дмитрий Соломенников

dmitry.solomennikov@axiomjdk.ru

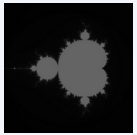
- 10+ лет опыта в языках и компиляторах
- Соавтор языков
  - ArkTS[1]
  - Тривиль[2]
  - Аккорд (не опубликовано)
- Контрибьютор компиляторов
  - Все предыдущие
  - Flow9[3]
  - Embedded Common Lisp[4]
  - KPHP[5]
  - Ficus[6]

1. https://gitee.com/igelhaus/arkcompiler_runtime_core/releases/download/arkts-spec-release-2025-01-15/ArkTSSpecification.pdf
2. https://gitflic.ru/project/alekseinedoria/trivil-0
3. https://github.com/area9innovation/flow9
4. https://ecl.common-lisp.dev/
5. https://github.com/VKCOM/kphp
6. https://sourcecraft.dev/compiler-potion-faculty/ficus

Вычислительная задача



- Задачи представлены в виде чисел
- Много расчетов
- Правильного выбора нет

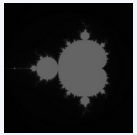


Python[1]: привычный выбор

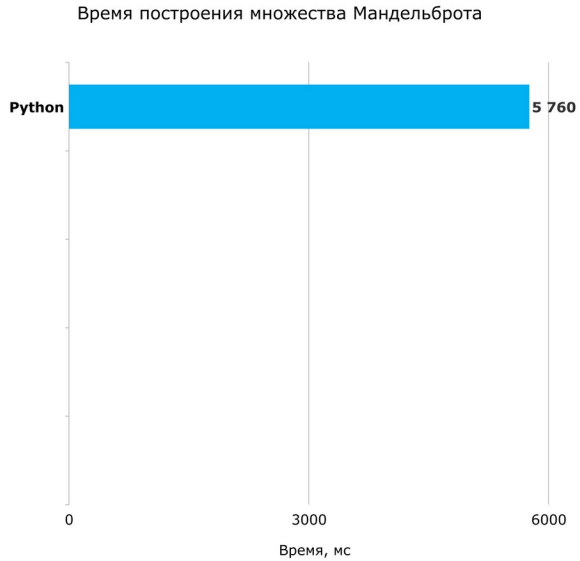
```python
def mandelbrot_python(height, width, max_iter):
    # Создаем пустую матрицу
    output = [[0 for _ in range(width)] for _ in range(height)]

    for i in range(height):
        # Преобразуем координаты пикселя в координаты на плоскости
        y0 = -1.5 + (i * 3.0) / height
        for j in range(width):
            x0 = -2.0 + (j * 3.0) / width
            x, y = 0.0, 0.0
            iteration = 0

            # Тот самый "тяжелый" цикл
            while (x * x + y * y <= 4.0) and (iteration < max_iter):
                xtemp = x * x - y * y + x0
                y = 2.0 * x * y + y0
                x = xtemp
                iteration += 1

            output[i][j] = iteration
    return output
```

1. https://habr.com/ru/articles/975052/

Python[1]: привычный выбор

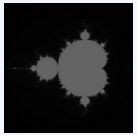


```python
def mandelbrot_python(height, width, max_iter):
    # Создаем пустую матрицу
    output = [[0 for _ in range(width)] for _ in range(height)]

    for i in range(height):
        # Преобразуем координаты пикселя в координаты на плоскости
        y0 = -1.5 + (i * 3.0) / height
        for j in range(width):
            x0 = -2.0 + (j * 3.0) / width
            x, y = 0.0, 0.0
            iteration = 0

            # Тот самый "тяжелый" цикл
            while (x * x + y * y <= 4.0) and (iteration < max_iter):
                xtemp = x * x - y * y + x0
                y = 2.0 * x * y + y0
                x = xtemp
                iteration += 1

            output[i][j] = iteration
    return output
```

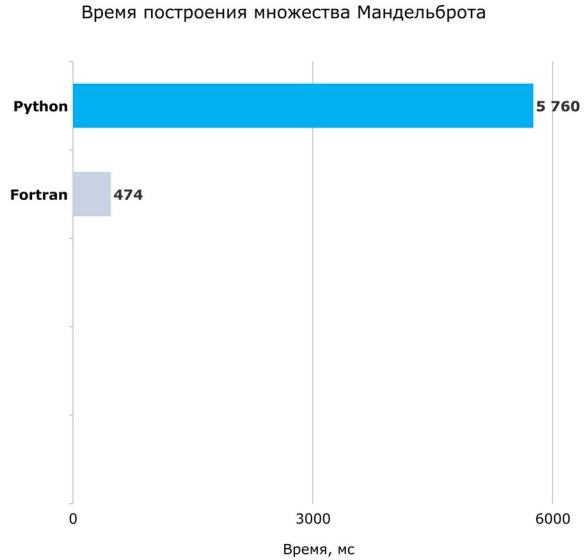


1. https://habr.com/ru/articles/975052/

Fortran[1]: классика численных расчетов

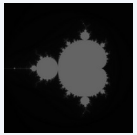


```fortran
subroutine compute_mandelbrot(image, width, height,
  max_iter, min_x, max_x, min_y, max_y)
    integer, intent(in) :: width, height, max_iter
    real(8), intent(in) :: min_x, max_x, min_y, max_y
    integer, intent(out) :: image(height, width)
    integer :: i, j, iter
    real(8) :: x0, y0, x, y, xtemp, dx, dy
    dx = (max_x - min_x) / (width - 1)
    dy = (max_y - min_y) / (height - 1)
    do i = 1, height
        y0 = min_y + (i - 1) * dy
        do j = 1, width
            x0 = min_x + (j - 1) * dx
            x = 0.0d0
            y = 0.0d0
            iter = 0
            do while ((x * x + y * y <= 4.0d0) .and. (iter < max_iter))
                xtemp = x * x - y * y + x0
                y = 2.0d0 * x * y + y0
                x = xtemp
                iter = iter + 1
            end do
            image(i, j) = iter
        end do
    end do
end subroutine compute_mandelbrot
```

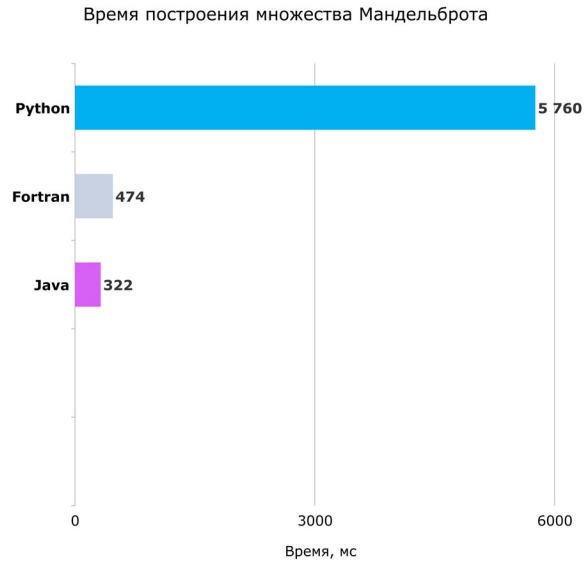


1. https://habr.com/ru/articles/975052/

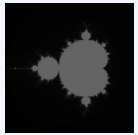
Java: добротное бизнес-решение



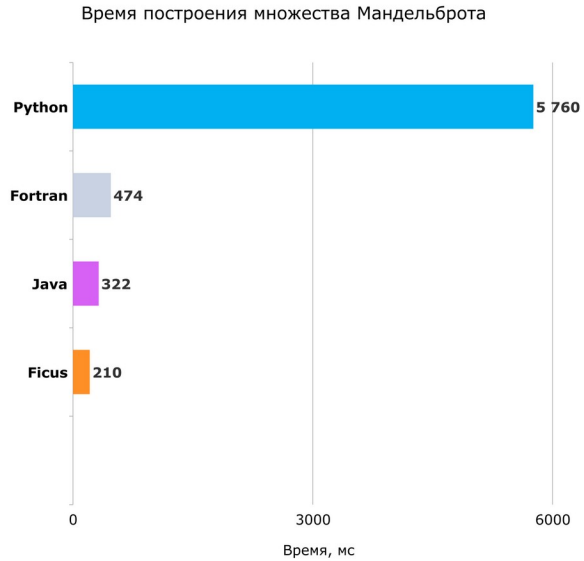
```java
public static byte[][] mandelbrot_java(
  int height, int width, byte max_iter) {
    byte[][] output = new byte[height][width];

    for (int i = 0; i < height; i++) {
        // Преобразуем координаты пикселя в координаты на плоскости
        float y0 = -1.5f + (i * 3.0f) / height;

        for (int j = 0; j < width; j++) {
            float x0 = -2.0f + (j * 3.0f) / width;
            float x = 0.0f;
            float y = 0.0f;
            byte iteration = (byte)0;
            // Тот самый "тяжелый" цикл
            while ((x * x + y * y <= 4.0f) && (iteration < max_iter)) {
                iteration += (byte)1;
                float xtemp = x * x - y * y + x0;
                y = 2.0f * x * y + y0;
                x = xtemp;
            }
            output[i][j] = iteration;
        }
    }
    return output;
}
```



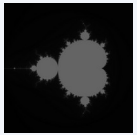
Ficus: ML-язык для обработки массивов



```
fun mandelbrot_ficus(height: int, width: int, max_iter: uint8): uint8[,] {
    val output = array((height, width), 0u8)

    for i ← 0:height {
        // Преобразуем координаты пикселя в координаты на плоскости
        val y0 = -1.5f + ((i :> float) * 3.0f) / (height :> float)

        for j ← 0:width {
            val x0 = -2.0f + ((j :> float) * 3.0f) / (width :> float)
            var x = 0.0f
            var y = 0.0f
            var iteration = 0u8
            // Тот самый "тяжелый" цикл
            while (x * x + y * y ⩽ 4.0f) && (iteration < max_iter) {
                iteration = ((iteration + 1u8) :> uint8)
                val xtemp = x * x - y * y + x0
                y = 2.0f * x * y + y0
                x = xtemp
            }
            output[i, j] = iteration
        }
    }
    return output
}
```

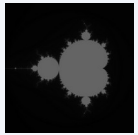


Ficus: ML-язык для обработки массивов



```
fun mandelbrot_ficus(height: int, width: int, max_iter: uint8): uint8[,] {
    val output = array((height, width), 0u8)
    @parallel
    for i ← 0:height {
        // Преобразуем координаты пикселя в координаты на плоскости
        val y0 = -1.5f + ((i :> float) * 3.0f) / (height :> float)
        @parallel
        for j ← 0:width {
            val x0 = -2.0f + ((j :> float) * 3.0f) / (width :> float)
            var x = 0.0f
            var y = 0.0f
            var iteration = 0u8
            // Тот самый "тяжелый" цикл
            while (x * x + y * y ⩽ 4.0f) && (iteration < max_iter) {
                iteration = ((iteration + 1u8) :> uint8)
                val xtemp = x * x - y * y + x0
                y = 2.0f * x * y + y0
                x = xtemp
            }
            output[i, j] = iteration
        }
    }
    return output
}
```

Ficus: ML-язык для обработки массивов



```
fun mandelbrot_ficus(height: int, width: int, max_iter: uint8): uint8[,] {
    val output = array((height, width), 0u8)
    @parallel
    for i ← 0:height {
        // Преобразуем координаты пикселя в координаты на плоскости
        val y0 = -1.5f + ((i :> float) * 3.0f) / (height :> float)
        @parallel
        for j ← 0:width {
            val x0 = -2.0f + ((j :> float) * 3.0f) / (width :> float)
            var x = 0.0f
            var y = 0.0f
            var iteration = 0u8
            // Тот самый "тяжелый" цикл
            while (x * x + y * y ⩽ 4.0f) && (iteration < max_iter) {
                iteration = ((iteration + 1u8) :> uint8)
                val xtemp = x * x - y * y + x0
                y = 2.0f * x * y + y0
                x = xtemp
            }
            output[i, j] = iteration
        }
    }
    return output
}
```

Примитивы

Значения

```
1 val a = 1
2 println(a)
```

- Значения не изменяются

Значения

```
1 val a = 1
2 val a = 2
3 println(a)
```

- Значения не изменяются
- Имя значения – трюк

Значения

```
1  val a = 1
2  val a = a + 1
3  println(a)
4
5  // Ошибка
6  // a = a + 1
```

- Значения не изменяются
- Имя значения – трюк

Значения

```
1 val a0 = 1
2 val a1 = a0 + 1
3 println(a1)
```

- Значения не изменяются
- Имя значения – трюк
- Удобно переиспользовать имя

Переменные

```
1 var a = 1
2 a = a + 1
3 println(a)
```

- Переменные **изменяются**

Численные типы

```
1 val a: int = 1
2 val b: int32 = 1i32
3 val c: int32 = (1 :> int32)
4 val d = 1.0 // double
5 val e = 1.0f
```

- Целые числа – `int`
- Дробные числа – `double`
- Другие требуют типа

Численные типы

```
// Численные типы:
// int - машинное слово
// int8, int16, int32, int64
// uint8, uint16, uint32, uint64
// float, double
// # long, half
```

- Целые числа – `int`
- Дробные числа – `double`
- Другие требуют типа

Символы

```
1 val a = 'a'
2 val b = '\u0000'
3 val c = '\U00000000'
4 val e = ''
5 val e = '33'
```

- Символы Юникода
- Можно числа (\u, \U, \x, \ooo)

Строки

```
1 val s = ""
2 val u = "string"
3 val f = f"Интерполяция {u}"
4 val r = r"\ df\"
5 val m = f"
6 multiline"
```

- Символы Юникода
- Можно числа (\u, \U, \x, \ooo)

Составные типы

Списки

```
1  val lst = 1 :: 2 :: 3 :: 4 :: []
2
3  println(lst)
4
5  // [1, 2, 3, 4]
```

- Списки имутабельны

Списки

```
1  val lst = [:: for i ← 0:10 {i}]
2
3  println(lst)
4
5  // [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- Списки имутабельны
- Возможно построение списков

Кортежи: создание

```
1  type tup = (int, char, double)
2  type tup2 = (int * 3)
3
4  val a: tup = (1, 'a', 1.0)
5  val b: tup2 = (0, 1, 2)
6  val c = (b.2, 5, a.0)
7
8  println(c) // (2, 5, 1)
```

- Кортежи иммутабельны
- Доступ через `.n`

Кортежи: разбор

```
1  val a = (1, 'Ж', 1.0)
2  val (_, c, _) = a
3
4  println(c) // Ж
```

- Разбор на части
- Имена можно опускать через _

Кортежи: поэлементные операции

```
1  val a = (1, 2, 3)
2  val b = a .+ a
3
4  println(b) // (2, 4, 6)
```

- Для равных кортежей
- Операция применяется поэлементно

Союзные типы (варианты)

```
1  type sum = V: int | Z:{x: int} | W: (int, double)
2
3  fun string(v: sum) {
4    | V(i) => f"V({i})"
5    | Z{x=x} => f"Z{{x: {x}}}"
6    | W(i, d) => f"W({i}, {d})"
7  }
8
9  val (a, b) = (W(1, 1.), V(0))
10
11 println(a)
12 println(b)
```

Optional

```
1  var o: int? = None
2  o = Some(1)
3
4  println(o) // Some(1)
```

- Используется символ ?
- Есть набор функций для работы

Записи: инициализация

```
1 type rec = {x: int; y: char; z: double}
2 val r = rec { x=1, y='a', z=1.0 }
3
4 println(r) // {x=1, y='a', z=1.0}
```

- Записи требуют имён
- Доступ через `.имя`

Записи: разбор

```
1  type rec = {x: int; y: char; z: double}
2  val r = rec { x=1, y='a', z=1.0 }
3
4  val {x=xn, y} = r
5
6  println(xn) // 1
7  println(y)  // a
```

- Разбор допускает подмену

Записи: сокращенная замена

```
1  type rec = {x: int; y: char; z: double}
2  val r = rec { x=1, y='a', z=1.0 }
3
4  val q = r.{y='b'}
5
6  println(q.y)  // b
```

- Копирование с заменой поля

Записи: изменяемые поля

```
1  type rec = {x: int; var y: char; z: double}
2  val r = rec { x=1, y='a', z=1.0 }
3
4  r.y = 'b'
5
6  println(r.y)  // b
```

- Поля записи можно править

Факультет
компилятороварения

Массивы

Массивы: создание

```
1  val a = [1, 2, 3, 4]
2  val b = [for e@i ← a {e * i}]
3
4  println(b)
```

- Массивы иммутабельны
- Данные мутабельны

Массивы: размерности

```
1  val a = [1, 2; 3, 4]
2  a[0, 0] = 0
3
4  println(a)
5  /*
6  [0, 2;
7   3, 4]
8  */
```

- Массивы иммутабельны
- Данные мутабельны
- Доступ через `a[i]`

Массивы: развертка массива

```
1 val a = [1, 2]
2 val b = [3, 4]
3 val c = [\a, \b]
4
5 println(c) // [1, 2, 3, 4]
```

- Раскрывается \a
- Можно использовать многократно

Массивы: развертка массива с размерностями

```
1  val a = [1, 2]
2  val b = [3, 4]
3  val c = [\a; \b]
4
5  println(c)
6  /*[1, 2;
7     3, 4]*/
```

- Раскрывается \a
- Можно использовать многократно
- Можно задавать размерности

Массивы: подмассивы

```
1  val a = [11, 12, 13; 21, 22, 23; 31, 32, 33]
2  val b = a[0:2, 0:2]
3
4  println(b)
5  /*[11, 12;
6     21, 22]*/
```

- Задается диапазон в размерности

Массивы: подмассивы

```
1  val a = [11, 12, 13; 21, 22, 23; 31, 32, 33]
2  val b = a[0:2, :]
3
4  println(b)
5  /*[11, 12, 13;
6     21, 22, 23]*/
```

- Задается диапазон в размерности
- Можно указать всю размерность :

Массивы: модификация подмассивов

```
1  val a = [11, 12, 13; 21, 22, 23; 31, 32, 33]
2  val b = a[0:2, :]
3
4  b[0, 0] = 88
5
6  println(b)
7  /*[88, 12, 13;
8     21, 22, 23]*/
```

- Задается диапазон в размерности
- Можно указать всю размерность :
- Подмассив меняет данные

Массивы: модификация подмассивов

```
1  val a = [11, 12, 13; 21, 22, 23; 31, 32, 33]
2  val b = a[0:2, :]
3
4  b[0, 0] = 88
5
6  println(a)
7  /*[88, 12, 13;
8     21, 22, 23;
9     31, 32, 33]*/
```

- Задается диапазон в размерности
- Можно указать всю размерность :
- Подмассив меняет данные

Массивы: выравнивание массива

```
1 val a = [11, 12, 13; 21, 22, 23; 31, 32, 33]
2 val b = a[:]
3
4 println(b)
5 // [11, 12, 13, 21, 22, 23, 31, 32, 33]
```

- Массив превращается в 1D

Массивы: выравнивание массива

```
1  val a = [11, 12, 13; 21, 22, 23; 31, 32, 33]
2  val b = a[:]
3
4  b[0] = 88
5
6  println(a)
7  /*[88, 12, 13;
8     21, 22, 23;
9     31, 32, 33]*/
```

- Массив превращается в 1D
- Но продолжает быть связан

Массивы: счет с конца

```
1  val a = [11, 12, 13; 21, 22, 23; 31, 32, 33]
2  val b = a[:, .-1]
3
4  println(b)
5  /*[13;
6     23;
7     33]*/
```

- `.-` считает с конца

Массивы: счет с конца

```
1 val a = [11, 12, 13; 21, 22, 23; 31, 32, 33]
2 val b = a[.-1, .-1]
3
4 println(b) // 33
```

- `.-` считает с конца
- `[.-, .-]` - правый нижний угол

Массивы: каждый второй ряд

```
1  val a = [
2  11, 12, 13, 14;
3  21, 22, 23, 24;
4  31, 32, 33, 34;
5  41, 42, 43, 44
6  ]
7  val b = a[::2, :]
8
9  println(b)
10 /*[11, 12, 13, 14;
11    31, 32, 33, 34]*/
```

Массивы: в обратном порядке

```
1  val a = [
2  11, 12, 13, 14;
3  21, 22, 23, 24;
4  31, 32, 33, 34;
5  41, 42, 43, 44
6  ]
7  val b = a[::-1, :]
8
9  println(b)
10 /*[41, 42, 43, 44;
11    31, 32, 33, 34;
12    21, 22, 23, 24;
13    11, 12, 13, 14]*/
```

Массивы: перевернуть ряды и строки

```
1  val a = [
2  11, 12, 13, 14;
3  21, 22, 23, 24;
4  31, 32, 33, 34;
5  41, 42, 43, 44
6  ]
7  val b = a[::-1, ::-1]
8
9  println(b)
10 /*[44, 43, 42, 41;
11    34, 33, 32, 31;
12    24, 23, 22, 21;
13    14, 13, 12, 11]*/
```

Массивы: транспонирование матрицы

```
1  val a = [
2  11, 12, 13, 14;
3  21, 22, 23, 24;
4  31, 32, 33, 34;
5  41, 42, 43, 44
6  ]
7  val b = a'
8
9  println(b)
10 /*[11, 21, 31, 41;
11    12, 22, 32, 42;
12    13, 23, 33, 43;
13    14, 24, 34, 44]*/
```

Массивы: инвертирование матрицы

```
1  val a = [
2  1., 2., 4.;
3  3., 3., 1.;
4  4., 1., 3.
5  ]
6  val b = a\1 // Инвертирование матрицы QR разложением
7
8  println(b)
9  /*
10 [-0.2105263157894736, 0.05263157894736837, 0.2631578947368421;
11  0.131578947368421, 0.3421052631578947, -0.2894736842105263;
12  0.2368421052631578, -0.1842105263157895, 0.07894736842105267]
13 */
```

Массивы: решение линейного уравнения

```
1  val A = [1., 1.; 1., -1.] // x + y = 2
2  val b = [2., 0.]          // x - y = 0
3
4  val c = A\b // Решение системы
5
6  println(c)
7  // [0.9999999999999999, 0.9999999999999999]
```

Массивы: перемножение матриц

```
1  val a = [1, 2; 3, 4]
2  val b = a * a
3
4  println(b)
5  /*[7, 10;
6     15, 22]*/
```

Массивы: свёртка

```
1  val a = [1, 2; 3, 4]
2
3  val b = fold sum=0 for x ← a {sum + x}
4
5  println(b) // 10
```

Массивы: zero

```
1 val a = array((2,2), (1, 1, 1))
2
3 println(a.zero[2, 2]) // (0, 0, 0)
```

- Значения за границами равны 0
- Ноль зависит от типа элементов

Массивы: wrap

```
1  val a = [1, 2; 3, 4]
2  val b = a.wrap[2, 2]
3
4  println(b) // 1
```

- Индексы по модулю

Массивы: clip

```
1  val a = [1, 2; 3, 4]
2  val b = a.clip[2, 2]
3
4  println(b) // 4
```

- Индексы прижимаются к границам

Массивы: поэлементные операции

```
1 val a = [(1, 1), (2, 2), (3, 3)]
2 val b = a .+ (1, 1)
3
4 println(b) // [(2, 2), (3, 3), (4, 4)]
```

- Операция применяется к элементам

Сопоставление с образцом

Сопоставление с образцом

```
1 val a = 2
2 match a {
3     | 2 => println("2")
4     | _ => println("не 2")
5 }
```

- Примитивы

Сопоставление с образцом

```
1  val a = "2"
2  match a {
3      | "2" => println("2")
4      | _   => println("не 2")
5  }
```

- Примитивы

Сопоставление с образцом

```
1  val a = 2 :: 3 :: []
2  match a {
3      | 2 :: 3 :: _ => println("совпало")
4      | h :: t => println(h)
5      | _   => println("не совпало")
6  }
```

- Списки

Сопоставление с образцом

```
1 val a = None :: Some(1) :: []
2 match a {
3     | None :: Some(x) :: _ => println(f"{x}")
4     | h :: t => println(h)
5     | _    => println("не совпало")
6 }
```

- Списки сложных типов

Сопоставление с образцом

```
1 val a = (1, 2)
2 match a {
3     | (1, x) => println(x)
4     | (z, _) => println("пусто")
5 }
```

- Кортежи

Репозиторий