Ликбез: Языки программирования. Лекция 4. Пролетая над гнездом граблей

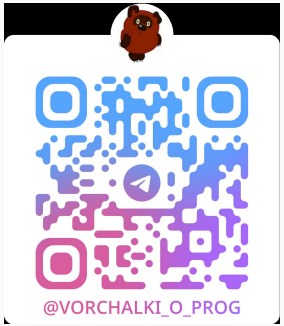
Алексей Недоря, февраль 2026

Профессия: Языки программирования и компиляторы

Краткая профессиональная биография:

- Первый компилятор: 1984
- Компиляторы для 9 языков программирования
- Участие в стандартизации языков программирования
  - Modula-2 ISO/IEC
  - Oberon-2 Oakwood Guidelines
- Системная архитектура
- Разработка 7 языков программирования

- t.me/vorchalki_o_prog
- Статьи в журнале Цифровая экономика
- Факультет Компилятороварения



Ликбез: Разработка языков программирования

Лекции:

- 26.11.2025: История. Зачем и почему. запись
- 17.12.2025. Как разрабатывать. Личная история. запись
- 21.01.2026. От личного к общему. Требования. запись

18.02.2026 Содержание:

- Грабли и требования
- Вредные конструкции
- Синтаксис: привычка свыше нам дана
- Kotlin: unified type system
- JavaScipt: дублирование
- Typescript, Kotlin: null safety
- Go: злое взаимодействие конструкций

Следующие лекции:

- 18.03 Вопросы и ответы
- ?

Языки:

- 04.02.2026: Артель запись

Дальше:

- 04.03.2026. Фикус
- 01.04.2026. Клаус
- ?

Грабли и требования

Что такое **грабли**?

- Не обязательно ошибки дизайна
- Решения, у которых есть неожиданные, часто неудобные (плохие) последствия

Почему сейчас про грабли? Два направления движения:

- От общего к деталям
- **От деталей к обобщениям** - сначала набрать материал

А как же **требования**?

- Используем общие требования: простота, удобство программирования, расширяемость (развиваемость), …

Вредные конструкции (anti-features): Austral, Pony, Argentum, Яс, Тривиль

Запрет	Austral	Pony	Argentum	Тривиль	Яс
No global state	√	√	√	x	?
No uninitialized variables	√	√	√	√	√
No variable shadowing	√		x	x	√
No macros	√	√	√	√	√
No arithmetic precedence	√	√	x	x	
No function overloading	√	√	√	√	√
No implicit type conversions	√			x	
No exceptions (and no surprise control flow)	√	√	√		√
No garbage collection	√	x	√	x	
No destructors	√	x	√	√	√
Нет явных указателей	√	√	√	√	√
Нет неявного импорта		√		√	√
Нет неквалифицированного импорта	x	x		√	√
Нет стирания типов (type erasure)		√	√	√	√
Нет неявного сужения типа (implicit smart cast)				√	√

Синтаксис: Простой, понятный, регулярный

- простой и понятный
- читаем слева направо
- опора на знаки препинания

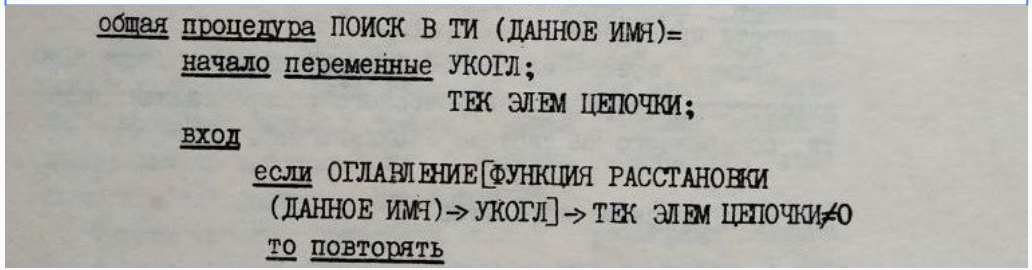
- регулярность (TS)
- отсутствие дублирования (TS)

C	`typedef int (*Operation)(int, int);`
Kotlin	`typealias Operation = (Int, Int) -> Int`
Go	`type Operation func(x, y int) int`
TS	`type Operation = (x: number, y: number) => number`
Rust	`type Operation = fn(x: int, y: int) -> int;`
Trivil	`тип Операция = фн(x: Цел64, y: Цел64): Цел64`

```
type Op = (x: number, y: number) => number
function add(x: number, y: number): number { return x+y }
let x = (x: number, y: number): number => { return x+y }
let x = function (x: number, y: number): number { return x+y }
```

Синтаксис: Привычка свыше нам дана… (или даешь энергосбережение)

ЯРМО 1980:



```
общая процедура ПОИСК В ТИ (ДАННОЕ ИМЯ)=
    начало переменные УКОГЛ;
                      ТЕК ЭЛЕМ ЦЕПОЧКИ;
    вход
        если ОГЛАВЛЕНИЕ[ФУНКЦИЯ РАССТАНОВКИ
         (ДАННОЕ ИМЯ)->УКОГЛ]->ТЕК ЭЛЕМ ЦЕПОЧКИ≠0
         то повторять
```

```
фн найти описание(область: асд.Область, имя: Строка): асд.Описание {
  пусть оп = найти в области(область, имя)
  если оп # пусто {
    вернуть оп^
  }
  основа.добавить ошибку("СЕМ-НЕ-НАЙДЕНО", имя)
}
```

найти описание	найтиОписание
найти_описание	НайтиОписание

Многие языки	Oberon, Тривиль
==	=
!=	#
!	~
&&, ||	&, |
=	:=

C, Kotlin, TS	`if (x == 1 && y) {}`
Go, Rust	`if x == 1 && y {}`
Тривиль	`если x = 1 & y {}`

We all get habit from above,
And it replaces good and love.

Большие проблемы маленького Ифа: всего лишь скобочки…

```
if ( условие ) тело else тело                    тело: { операторы } | оператор
```

C: {} можно опустить	`int x = 0;`
	`if (x > 0) if (x < 3) x = 1;`
	`else x = 2;`
		После выполнения: x равно 0
		
		Сложность понимания, синтаксис обманывает.

Kotlin	`if (true) else false`	Можно опустить тело, что странно

Go: () не нужны	`type S struct {b bool}`
	`x := S{}.b  // ok`
	`if S{}.b {} // unexpected . at end of statement`
		Потенциальные проблемы с конструкторами, в языках похожих на Go

Большие проблемы маленького Ифа: борьба со сложностью

Вложенные if - очень сложная для понимания конструкция, источник ошибок.

Дейкстра: Охраняемые команды. Нет else, всегда явные условия.	Вир: в условном операторе нет ‘иначе’	Оператор выбора: switch, match	Второй условный оператор Swift: `guard` Тривиль: `надо`

`if x >= y -> max := x;`
`  y >= x -> max := y;`
`fi`
			`надо № > 1 иначе вернуть 1`
			`вернуть факториал(№-1) * №`
			
			`надо x типа T иначе авария(“текст”)`
			
			`цикл мод среди список-модулей {`
			`  анализ и генерация(мод)`
			`  надо нет ошибок() иначе прервать`
			`}`

Kotlin: Unified type system: Суха теория, мой друг…

Унифицированная система типов: Есть единый **супертип** для любого типа типовой системы

- Nothing <: T <: Any?
- Any <: Any?
- Int <: Any (и так же Long, Double, …)

Массивы:

- `Array<Int>` - настройка обобщенного типа
- `IntArray` - *has the same methods and properties as* `Array(Int)` . However, they are **not** related by subtyping…

```
fun check(a: Array<Int>) {
    val ia: IntArray = a     // Type mismatch
    val ai: Array<Int> = ia // Type mismatch
}
```

Где хорошо

- class G<T> … настройка на любой тип
- Тип выражения:

```
fun check(cond: Boolean): Any {
    return if (cond) 1 else "a"
}
```

Note: the presence of such specialized types allows the compiler to perform additional array-related optimizations.

В чем разница?

- Array<Int> - это массив указателей
- IntArray - массив 32-х разрядных целых

Javascript: Семантические грабли: дублирование

link

JavaScript is unique to have two nullish values: `null` and `undefined`. Semantically, their difference is very minor: `undefined` represents the absence of a value, while `null` represents the absence of an *object*.

`null` like `undefined`:

- accessing any property on `null` throws a `TypeError` instead of returning `undefined` or searching prototype chains.
- `null` is treated as falsy for boolean operations, and nullish for nullish coalescing and optional chaining.

unlike `undefined`:

- The `typeof null` result is `"object"`. This is a bug in JavaScript that cannot be fixed due to backward compatibility.
- Unlike `undefined`, `JSON.stringify()` can represent `null` faithfully.

Typescript: undefined vs. null safety

`let a: string = null // compile-time error`
`let b: string | null = null // ok`
 	
`let c: string = undefined // compile-time error`
`let d: string | undefined = undefined // ok`	

`function foo(s: string) { console.log(s) }`
`function bar(x: string | undefined) { foo(x as string) }`
`bar(undefined)`
	`undefined`

`let a = new Array<string>(5)` `console.log(a.length, a[1])`	`5, undefined`
`let b = new Array<string>` `console.log(b.length, b[1])`	`0, undefined`

`class C {`
`  static a: string`
`  b!: string`
`}`
`console.log(C.a, new C().b)`
	`undefined, undefined`

Kotlin: null safety

Kotlin explicitly supports nullability as part of its type system, meaning you can explicitly declare which variables or properties are allowed to be null. Also, when you declare **non-null variables, the compiler enforces that these variables cannot hold a null value**, preventing an NPE.

Kotlin: null safety или немножко беременность

Kotlin explicitly supports nullability as part of its type system, meaning you can explicitly declare which variables or properties are allowed to be null. Also, when you declare **non-null variables, the compiler enforces that these variables cannot hold a null value**, preventing an NPE.

The only possible causes of an NPE in Kotlin are:

- An explicit call to throw NullPointerException()
- Usage of the not-null assertion operator !!
- Data inconsistency during initialization

```
// Kotlin
class A { val f = 1 }

class C {
    val a: A
    init {
        fn()
        a = A()
    }
    public fun fn() {
        println(a.f)
    }
}
fun main() {
    val c = C()
    c.fn()
}
```

run-time error

Cannot invoke "A.getF()" because "this.a" is null

Kotlin: null safety или чуть-чуть беременный

compile-time error

```swift
// Swift
class A { var f = 1 }

class C {
    var a: A
    init() {
        fn()
        a = A()
    }
    public func fn() {

    }
}

var c = C()
c.fn()
```

```kotlin
// Kotlin
class A { val f = 1 }

class C {
    val a: A
    init {
        fn()
        a = A()
    }
    public fun fn() {
        println(a.f)
    }
}
fun main() {
    val c = C()
    c.fn()
}
```

run-time error

Cannot invoke "A.getF()" because "this.a" is null

Kotlin: семантические грабли

```kotlin
// Kotlin
class A { val f = 1 }

class C {
    val a: A
    init {
        fn()
        a = A()
    }
    public fun fn() {
        println(a.f)
    }
}
fun main() {
    val c = C()
    c.fn()
}
```

run-time error

Cannot invoke "A.getF()" because "this.a" is null

```swift
// Swift
class A { var f = 1 }

class C {
    var a: A
    init() {
        fn()
        a = A()
    }
    public func fn() { }
}

var c = C()
c.fn()
```

compile-time error

Go: грабли взаимодействия конструкций (feature interaction)

```go
package main

import "fmt"

type IM interface {
   Method()
}
type SM struct {i int}

func (x *SM) Method() {
    x.i = 1
}
func check(im IM) {
   if im != nil {
     im.Method()
   }
   fmt.Println("checked")
}
func main() {
   check(nil) // 1

   var S_nil *SM = nil
   check(S_nil) // 2
}
```

```
checked
panic: runtime error: invalid memory address or nil pointer dereference
[signal SIGSEGV: segmentation violation code=0x1 addr=0x0 pc=0x497ac0]

goroutine 1 [running]:
main.(*SM).Method(0x4e31b8?)
        /tmp/sandbox2195040405/prog.go:11
main.check({0x4e3198?, 0x0?})
        /tmp/sandbox2195040405/prog.go:15 +0x26
main.main()
        /tmp/sandbox2195040405/prog.go:23 +0x25

Program exited.
```

Features

- No null safety
- Methods can be called for null reference (like static)
- Two nil's are different comparison

Советы разработчику языка

- Что написано …, не вырубить (из следующих версий) или семь раз отмерь, потом добавь
- Задублишь, людей насмешишь
- В одну телегу впрячь не можно коня и трепетную лань
- Бросая в воду камешки, смотри на круги, ими образуемые; иначе такое бросание будет пустою забавою

Вопросы

EOF