Ликбез: Языки программирования. Лекция 5. Как древо жизни зеленеет…

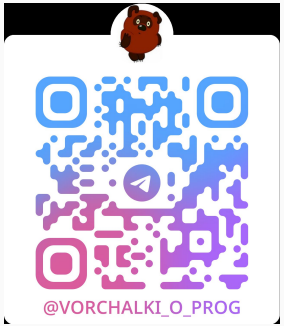
Алексей Недоря, март 2026

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

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



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. Пролетая над гнездом граблей. запись

18.03.2026 Содержание:

- Самое сложное в разработке языка?
- Этапы разработки
- Почему языки развиваются?
- Тривиль: дорабатываем лексику
- Тривиль: попутная типизация

Следующие лекции:

- ?

Языки:

- 04.02.2026: Артель запись
- 04.03.2026. Фикус запись

Дальше:

- 01.04.2026. Клаус
- ?

Хитрые ответы на хорошие вопросы

- Что в разработке языка самое сложное?
- Кто-то начинает с прагматики, чтобы потом придумать семантику, а потом синтаксис? Как всё происходит? На какой этап надо потратить больше всего времени/усилий? Почему?
- Почему большинство (живых) языков продолжают изменяться? Неизменный язык — мёртвый язык?

Что в разработке языка самое сложное?

Короткий ответ:

- Взаимодействие с человеками
- Взаимодействие конструкций языка

3 основных вида разработки:

- Производственный заказ, производственная разработка (ArkTS, Swift, C#, Go?)
- Идея - идейная команда (Артель, K1, Pony, Rust?, Go?)
- Авторский язык (Тривиль, Клаус, Фикус, Argentum, Austral?)

Что в разработке языка самое сложное?

	Сложности в начале работы	Сложности по ходу работы
Для всех вариантов	• Определить задачи, требования и приоритеты	• Изменение требований и приоритетов - не должно быть неявным • Работа с противоречивыми требования • Взаимодействие конструкций
Производственный заказ	• Зафиксировать основные требования и приоритеты в документах нужного уровня управления	• Отработка обратной связи • Удерживание команды в рамках • Удерживание языка в рамках • Сохранение баланса
Идейная команда	• Определение правил обсуждения и выработки решений • Выбор диктатура/демократия	• поиск консенсуса
Авторский язык	• Поставить себе рамки	• Самоцензура • С кем обсуждать?

Этапы разработки. Как все происходит?

- Кто-то начинает с прагматики, чтобы потом придумать семантику, а потом синтаксис? Как всё происходит? На какой этап надо потратить больше всего времени/усилий? Почему?

Этапы разработки

1. Ядро языка - то, что очевидно надо: числа, операции, операторы. На мой вкус:
   ```
   фн  факториал(№: Цел64): Цел64 {}
   ```
2. Надо (полезно) делать компилятор. И делать его целиком (насквозь).

Этапы разработки: Ядро языка - пример Тривиля

Типы

- Байт, Цел64, Слово64
- Вещ64
- Лог
- Символ
- Строка, Строка8
- вектор: []T
- класс: **класс** (база) {}
- может быть: **мб** T

Описания

- **тип** T = *тип*
- **конст** к (: T)? = *знач*
  конст к = 1
  конст к: Байт = 1
- **пусть** п(: T)? (= | :=) *знач*
  пусть № = 1 // *val*
  пусть №: Байт := 1 // *var*
- функции и методы

Операторы

- :=, ++, --
- **если** *усл* {} **иначе** {}
- **надо** *усл* **иначе** (завер | {})
- **выбор** *по выраж или типу*
- **пока** *усл* {}
- **цикл** *перем* **среди** *век* {}
- **прервать**, **вернуть** *знач*?
- **авария**(“описание”)

Функции, методы

из библиотеки “вывод”:
фн ф*(формат: Строка, список: ...*) {}
^ вариативный, полиморфный
из библиотеки “строки”:
тип Сборщик* = класс ...
фн (сб: Сборщик) добавить строку*(ст: Строка) {}

Этапы разработки: добавление сложных конструкций

Этапы разработки

1. Ядро языка - то, что очевидно надо: числа, операции, операторы. На мой вкус: функция вычисления факториала
2. Надо (полезно) сделать компилятор. И делать его целиком (насквозь).
3. Добавление сложных конструкций
   - Обоснование
   - Сравнительный анализ
   - Синтаксис, семантика
   - Прототипирование
4. Начать спецификацию языка. Обычно приводит к необходимости переделать все, начиная с пункта 1.

Сложные конструкции - то, что не устоялось в языках программирования, территория поиска:

- Вариантные типы, объединения, опциональные типы
- Интерфейсы (interface, protocol, trait, duck typing)
- Лямбды, замыкания
- Обобщенные типы или модули
- Concurrency, parallelism
- Раздельная компиляция

Этапы разработки

Этапы разработки

1. Ядро языка - то, что очевидно надо: числа, операции, операторы. На мой вкус: функция вычисления факториала
2. Надо (полезно) сделать компилятор. И делать его целиком (насквозь).
3. Добавление сложных конструкций
4. Спецификация языка (начало)
5. Bootstrapping
6. Проверка разных вариантов, например: две кодогенерации…
7. Спецификация языка и тесты соответствия (compliance test suite)

На какой этап надо потратить больше всего времени/усилий?

Ответ: 3-7

Почему?

Ответ: Потому что все взаимосвязано, а *на колу мочало*. Приходится возвращаться к подпунктам пункта 3 и переделывать.

Rob Pike: разработка цикла **for** заняла год.

Почему большинство (живых) языков продолжают изменяться?

Почему развивается Тривиль?

Версия	**Добавлено**	**Зачем**
v0.9.2	• многострочные литералы	Арс
v0.9.3	• функциональные типы и значения • тип протокол	Арвиль
v0.9.4	• преобразования класс ⇔ протокол	Арвиль
v0.9.5	• анонимные типы и конструкторы векторов	неудобно без
v0.9.*		

Тривиль как прототип. Задача:

- Проверить то, что будет полезно в следующих языкам (прототипирование)
- Доработать то, что было сделано недостаточно хорошо

Задачи на ближайшую версию:

- Пересмотреть лексику: идентификатор, литералы, битовые операции
- Сделать попутную типизацию

Следующие задачи:

- Улучшить: обобщенные модули
- Сделать: Итераторы
- Доработать: опциональные типы

Исследование: строковые и символьные литералы

Язык	**строковый литерал** **single line string**	**многострочный литерал** **multiline string**	**символ** **char, unicode code point**
Тривиль (до)	`"привет\n"`	`` `привет,` `мир` `` -- *raw string*	`'а'  '\u0404'`

Артель	`"привет"`
	`\привет↵`
		`"привет`
		`мир"`
		`\привет↵`
		`\мир↵`
			`"а"`

Клаус	`"123абв"`
	`"123""абв"`
	`"123"#20"абв"№0A`
		n/a	`'X'`
			`№2A`

Pony	*`all strings are multiline`*	`"привет`
		`мир"`
		`"""привет, мир"""`
			`'A'`
			`'\x41'`

Swift	`"hello"`	`let x = """`
		`  hello`
		`  world`
		`  """`
			single-character string

Отступление: Семантика и синтаксис в эпоху ANI и HNI

- Синтаксис должен быть очевидно читаемым.
- Точнее: семантика должна быть (оче-)видна сквозь синтаксис.

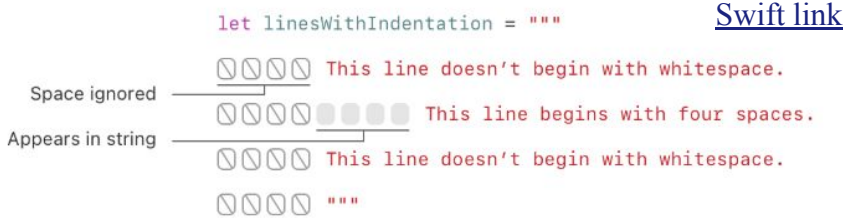
Тривиль версии 0.9.6 и 0.9.7

Версия 0.9.6

- Идентификатор:
  - Было: `это буква?`
    `если тип-вектор?(т) {}`
  - Добавлено: `т' а''`
    `пусть т' = т(:ТипВектора)`
  - Восклицательный знак нельзя использовать в именах: `буква!`
    Он может быть использован для других операций
- Логические литералы: `да` `нет`
- Битовые операции и сдвиги изменены:
  `|&` `||` `|^` `|~` `|>` `|<`
- Попутная типизация: `=>`

Версия 0.9.7

- Многострочный литерал
  - Было: `` `привет, мир` `` (raw)
  - Стало:
```
пусть привет = """
               привет,
                мир
               """
```
- Обратная кавычка оставлена для шаблонов (string interpolation)



Swift link

Попутная типизация. TypeScript smart cast

```
class C {  foo() {} }
class D extends C {
  bar() {}
}

function f1(x: C) {
  if (x instanceof D) {
    x.bar()
  }
}
function f2(x: C | null) {
  if (x != null) {
    x.foo()
  } // x!.foo()
}
```

```
function f3(x: C | D) {
  if (x instanceof C) {
    x.foo()
  } else {
    x.bar() // вот здесь уже весело
  }
}
function f4(cond: boolean, c: C, d: D) {
  let x = cond ? c : d // + вывод типов
  if (x instanceof C) {
    x.foo()
  } else {
    x.bar() // еще веселее (хрупкий код)
  }
}
```

Ограничения:

- локальные переменные и параметры

Делается для конструкций:

- условия в if, while содержащие:
- instanceof
- == nil, undefined
- != nil, undefined
- typeof

Распространяется на

- then часть (if, while)
- else часть (if)

Попутная типизация. Swift optional chaining

```swift
var name: String? = "Luna"

if let unwrapped = name {
    print("Hello, \(unwrapped)!")
}

let firstName: String? = "Luna"
let lastName: String? = "Valley"

if let first = firstName, let last = lastName {
    print("Full name: \(first) \(last)")
}
```

```swift
let score: Int? = 85
if let value = score, value > 80 {
    print("High score: \(value)")
}

let info: [String: Any] = ["age": 30]
if let age = info["age"] as? Int {
    print("User is \(age) years old")
}

guard let unwrapped = name else { return }
print("Hello, \(unwrapped)")
```

Ограничения:

- нет ограничений на вид операнда (новая переменная)
- только для опциональных типов

Делается для конструкций:

- if let, guard let, case let
- if var, guard var, case var

Распространяется на

- только на then часть (if, guard)

Попутная типизация. Тривиль (Go) переменная варианта

```
выбор пусть тек: тип оп {  // switch cur := op.(type) {}
когда асд.ОператорПрисвоить:
   им.выражение(тек.Л)
   им.выражение(тек.П)
когда асд.ОператорНадо:
   им.условие условного оператора(тек, тек.условие)
   им.оператор(пусто, тек.если-нет)
когда асд.ОператорАвария:
   им.выражение(тек.В)
другое
   авария(строки.ф("неизвестный оператор 2: $тип;", тек))
}


выбор пусть цель: тип асд.основа(тек.цель)^ {...}
```

Ограничения:

- нет ограничений на вид операнда (новая переменная)

Делается для конструкций:

- выбор по типу

Распространяется на

- вариант

Явная попутная типизация в Тривиле. Постановка задачи

Постановка задачи:

- Убрать лишние подтверждения типа (700 в компиляторе) и преобразования типа (220 в компиляторе)
- Упростить код, оставляя его явным (очевидным)
- Решение должно быть простым в реализации

```
надо м # пусто иначе вернуть ""
цикл а среди м^.атрибуты {}
```

```
пусть тт = основа(т)
надо тт # пусто иначе вернуть ложь
пусть от = тт^
вернуть от = тЦел64 | от = тСлово64
```

```
если оп типа ОписаниеТипа {
    пусть оп-типа = оп(:ОписаниеТипа)
    если оп-типа.Т типа ТипКласс {
        пусть кл = оп-типа.Т(:ТипКласс)
        цикл поле среди кл.поля {...}
    }
}
```

Явная попутная типизация. Поиск решения

Недостатки Swift решения:

- Неестественный синтаксис с точки зрения чтения слева направо
- Особый синтаксис вместо использования логического И (&), нельзя: `if cond && let x = y`

Так же как в выборе по типу	**если пусть** т': т # **пусто** { т'.поле } **если пусть** т': т **тип** ТипКласс {}	• неестественный синтаксис - надо запомнить
Переставим **пусть**	**если** т # **пусто пусть** т' { т'.поле } **если** т **типа** ТипКласс **пусть** т' {}	• более естественный синтаксис • надо менять синтаксис выбора
Изменим выбор по типу	Вместо: **выбор пусть** т': **тип** т {} **выбор тип** т **пусть** т' {}	• операция не тоже самое, что синтаксис оператора • хуже читается
Новая операция, читается: тогда	**если** т # **пусто** => т' { т'.поле } **если** т **типа** ТипКласс => т' {} **если** т => т' { т'.поле } **если** делать & т => т' { }	• естественно читается • разное по разному: в выборе и в условии • есть сокращенная запись • можно использовать не только в начале

Явная попутная типизация. Что получилось

надо м # **пусто** **иначе** **вернуть** ""
цикл а **среди** м^.атрибуты {}
	надо м => м' **иначе** **вернуть** ""
	цикл а **среди** м'.атрибуты {
	
	// тоже можно:
	надо м # **пусто** => м' **иначе** **вернуть** ""

пусть тт = основа(т) **надо** тт # **пусто** **иначе** **вернуть** **ложь** **пусть** от = тт^ **вернуть** от = тЦел64 | от = тСлово64	**надо** основа(т) => от **иначе** **вернуть** **ложь** **вернуть** от = тЦел64 | от = тБайт | от = тСлово64

если оп **типа** ОписаниеТипа {
  пусть оп-типа = оп(:ОписаниеТипа)
  если оп-типа.Т **типа** ТипКласс {
  пусть кл = оп-типа.Т(:ТипКласс)
  цикл поле **среди** кл.поля {...}
  }
}
	если оп **типа** ОписаниеТипа => оп' & оп'.Т **типа** ТипКласс => кл {
	  цикл поле **среди** кл.поля {...}
	}

Вопросы

EOF