385.71
Рейтинг

KTS

Создаем цифровые продукты для бизнеса

Подписаться

MAX1993M 8 мая 2024 в 16:39

Как новый компилятор K2 ускоряет компиляцию Kotlin на 94%

Средний 10 мин 15K

Блог компании KTS, Разработка мобильных приложений*, Разработка под Android*, Компиляторы*, Kotlin*

Обзор Перевод

Автор оригинала: Amanda Hinchman-Dominguez

Привет, меня зовут Мялкин Максим, я занимаюсь мобильной разработкой в KTS.

Не за горами выпуск новой версии Kotlin 2.0, основной частью которого является изменение компилятора на K2.

По замерам JB, K2 ускоряет компиляцию на 94%. Также он позволит ускорить разработку новых языковых фич и унифицировать все платформы, предоставляя улучшенную архитектуру для мультиплатформенных проектов.

Но мало кто погружался в то, как работает K2, и чем он отличается от K1.

Эта статья более освещает нюансы работы компилятора, которые будут полезны разработчикам для понимания, что же JB улучшают под капотом, и как это работает.

Контент статьи основывается на выступлении и овервью, но с добавлением дополнительной информации.

Оглавление

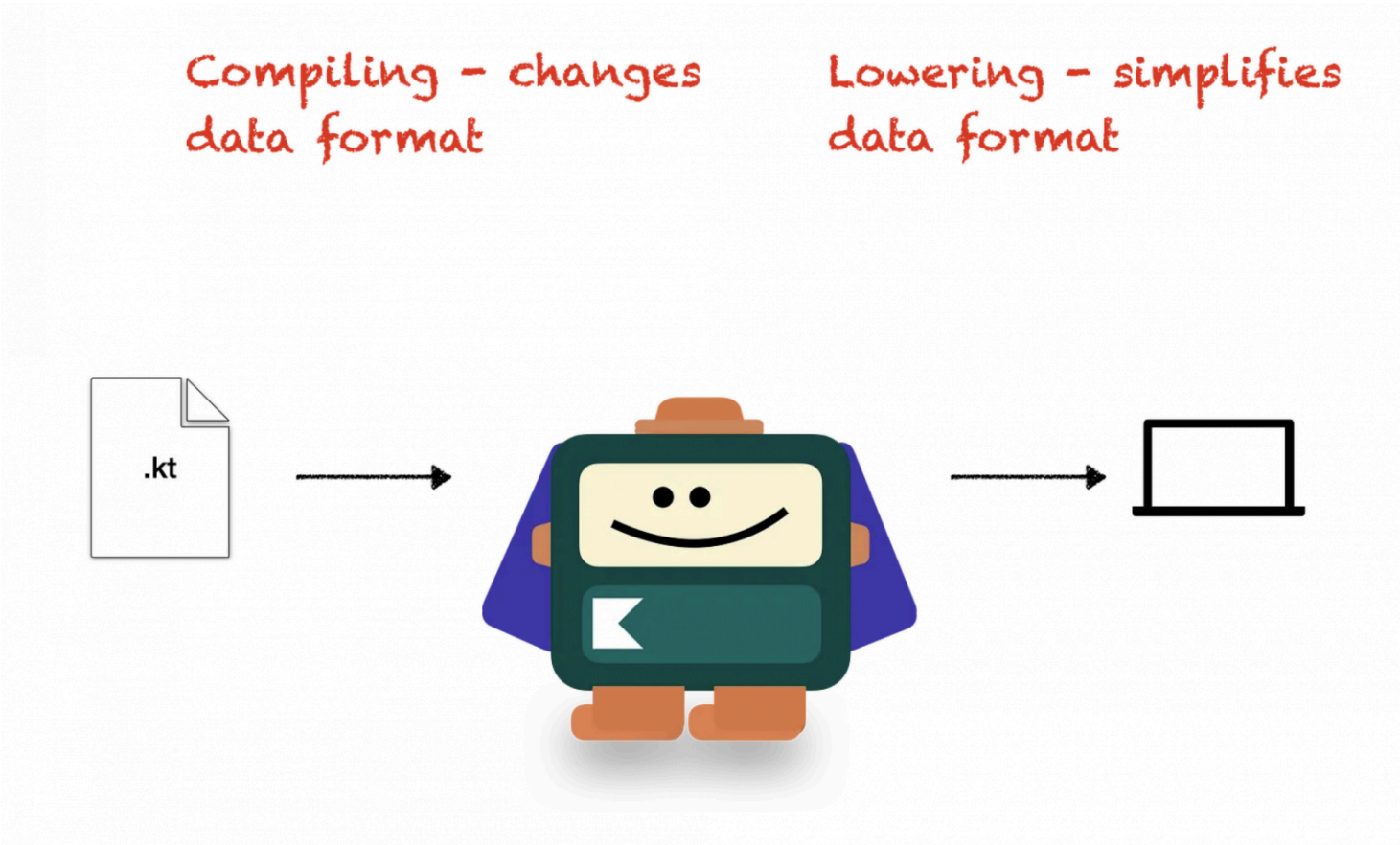
Как работает компилятор

Для начала поговорим о компиляторах в целом. По сути, **компилятор** — это программа. Она принимает код на языке программирования и превращает его в машинный код, который компьютер может исполнить.

Чтобы превратить исходный код в машинный, компилятор проводит несколько промежуточных преобразований. Если очень упростить, то компилятор делает две вещи:

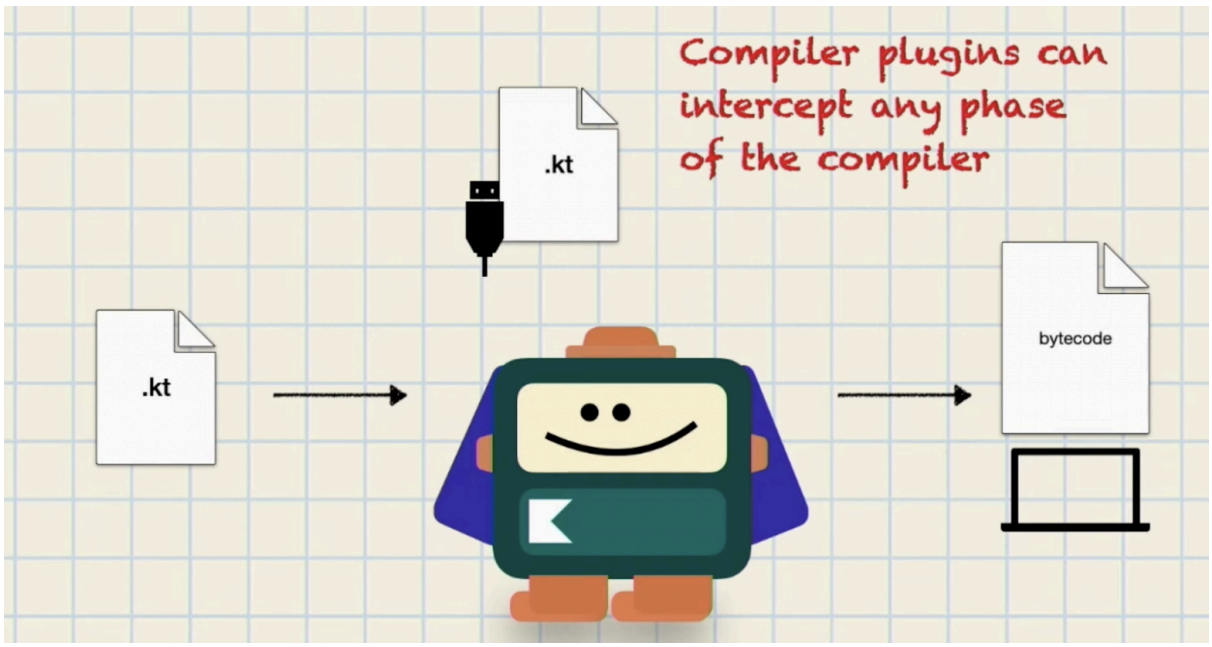
- меняет формат данных (compiling)

- упрощает его (lowering)



Иногда стандартных функций Kotlin недостаточно. В таких случаях разработчики выходят за рамки стандартных инструментов и используют **плагины компилятора**.

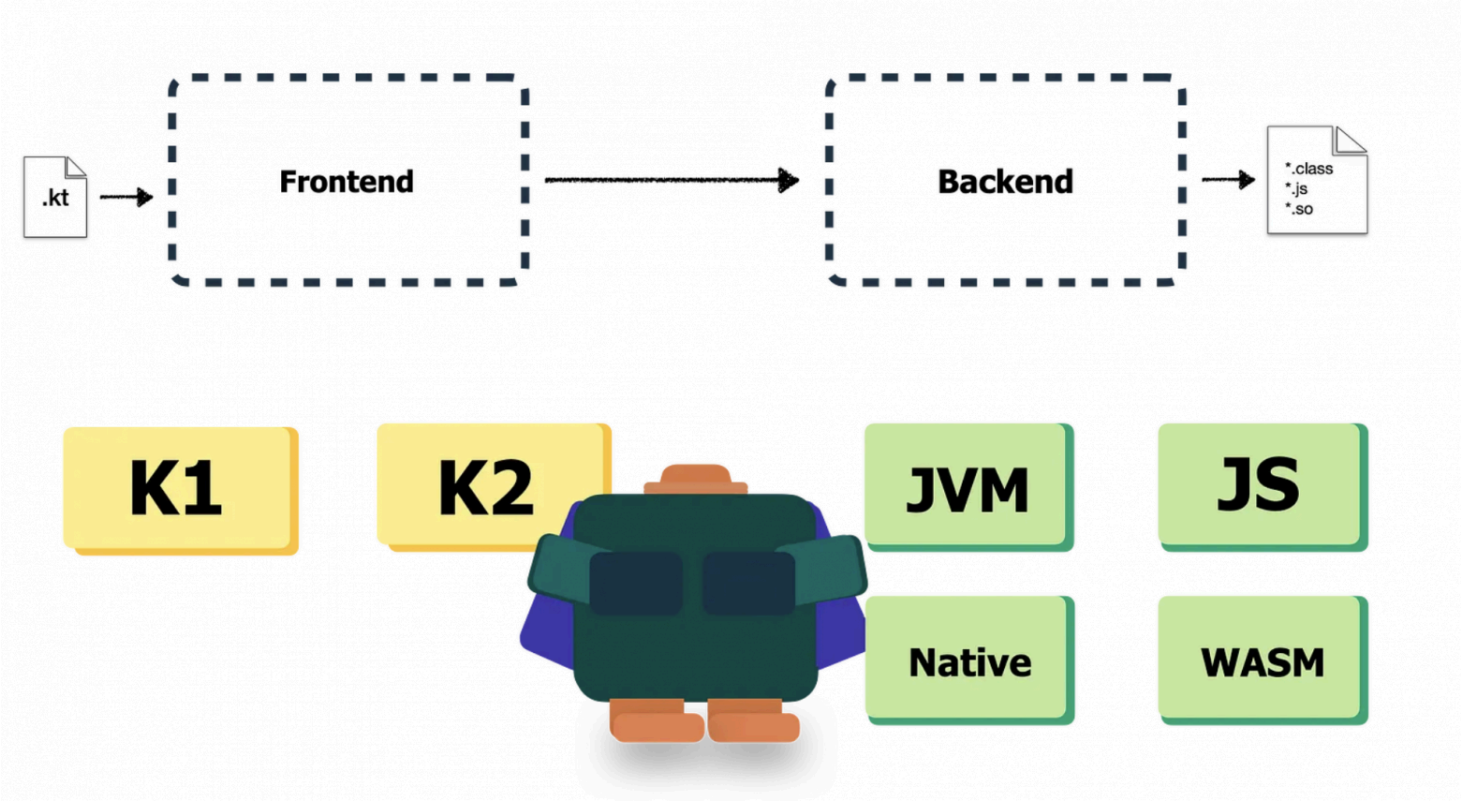
Плагины встраиваются в работу компилятора Kotlin на различных фазах компиляции и меняют стандартное преобразование кода компилятором.



> Плагины компилятора используют такие библиотек как Jetpack Compose, Kotlinx serialization.

Kotlin компилятор состоит из 2 частей:

- Frontend
  - отвечает за построение **синтаксического дерева (структуры кода)** и добавление **семантической информации (смысл кода)**
- Backend
  - отвечает за генерацию кода для целевой (target) платформы: JVM, JS, Native, WASM (экспериментальный)



Остановимся подробнее на каждой из частей.

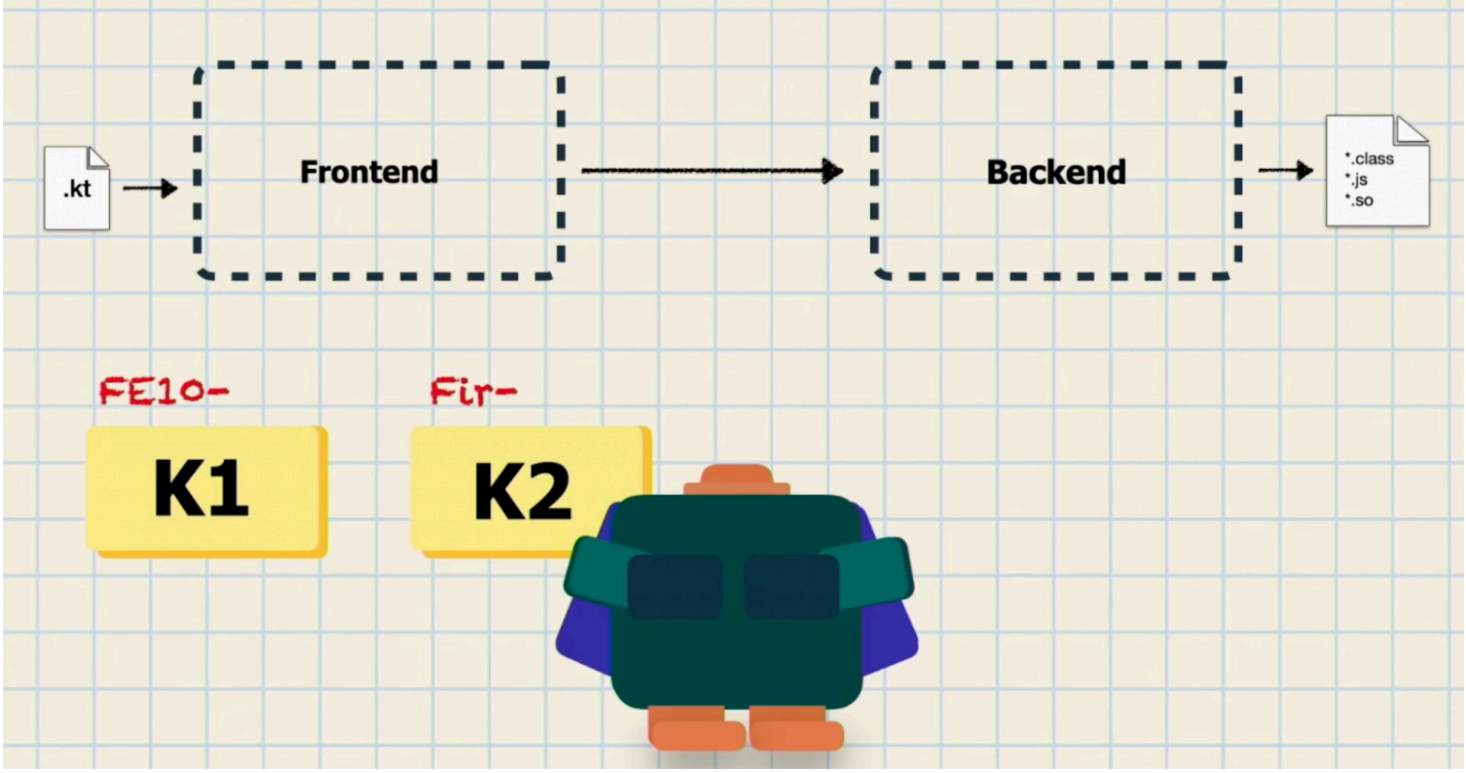
Frontend

У компилятора Kotlin есть две фронтенд-реализации:

- K1 (FE10-)
- K2 (Fir-)

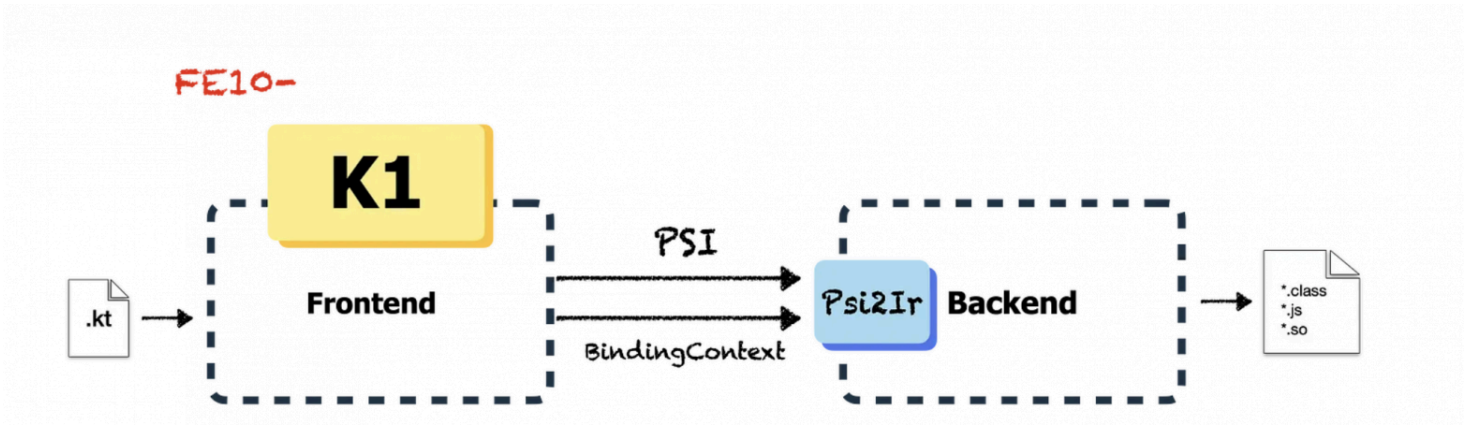
Упрощенно весь процесс работы фронтенда выглядит так:

- Компилятор принимает исходный код
- Производит синтаксический и семантический анализ кода
- Отправляет данные на бэкенд для последующей генерации IR (Intermediate representation) и целевого кода платформы (target)



Обе реализации похожи, но K2 вводит дополнительное форматирование данных перед тем, как отправить преобразованный код на бэкенд.

Реализация K1



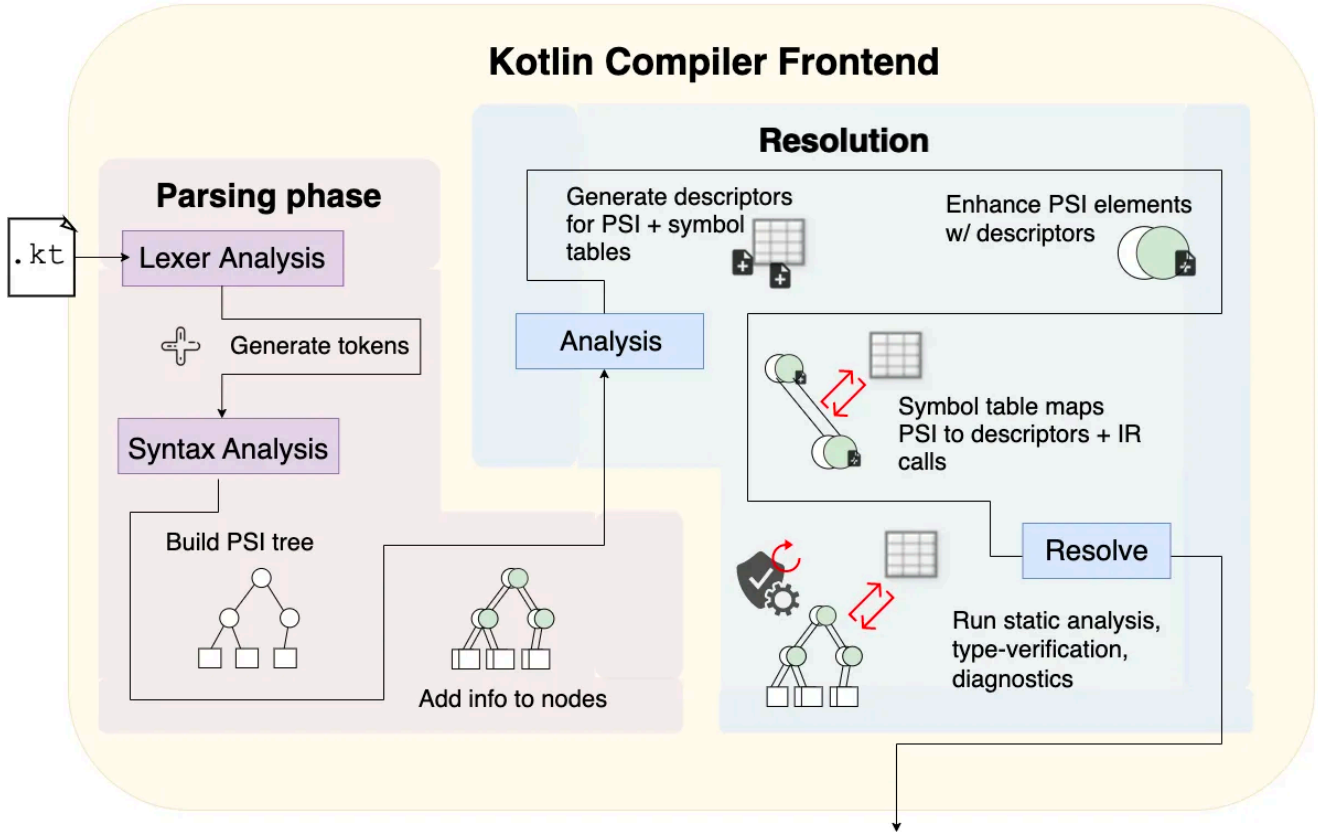
Реализация K1 работает с:

- **PSI** (Programming Structure Interface)
  - PSI — это слой платформы IntelliJ, который занимается парсингом файлов и созданием синтаксических и семантических моделей кода (что такое синтаксис и семантика рассмотрим позже)
- **Дескрипторами**
  - В зависимости от типа элемента дескрипторы могут содержать разную информацию о нем. Например дескриптор класса: контент класса, overrides, companion object и тд.
- **BindingContext** — это Map, которая содержит информацию о программе и дополняет PSI.

Во время обработки K1 отправляет **PSI** и **BindingContext** на Backend, который на входе использует преобразование PSI в IR (Psi2Ir) для дальнейшей работы.

Как работает K1

Общая схема работы выглядит следующим образом



Есть 2 фазы:

- **Parsing Phase** — разбирает что именно написал разработчик. На этой фазе еще неизвестно, будет ли компилироваться написанный код. Здесь проверяется, что код написан **синтаксически** верно.
- **Resolution Phase** — производит анализ, необходимый для понимания, будет ли код компилироваться и является ли он **семантически** правильным (более детальный разбор Resolution Phase).

Разберём дальше разницу между синтаксисом и семантикой.

Чтобы разобраться в работе компилятора K1, предположим, что у нас есть исходный код

```
fun two() = 2
```

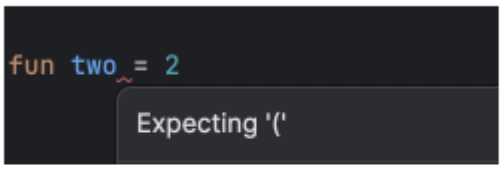
Пока что для компилятора это только строка текста.

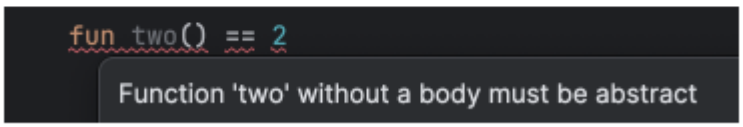
Parsing Phase

Задача синтаксического парсинга

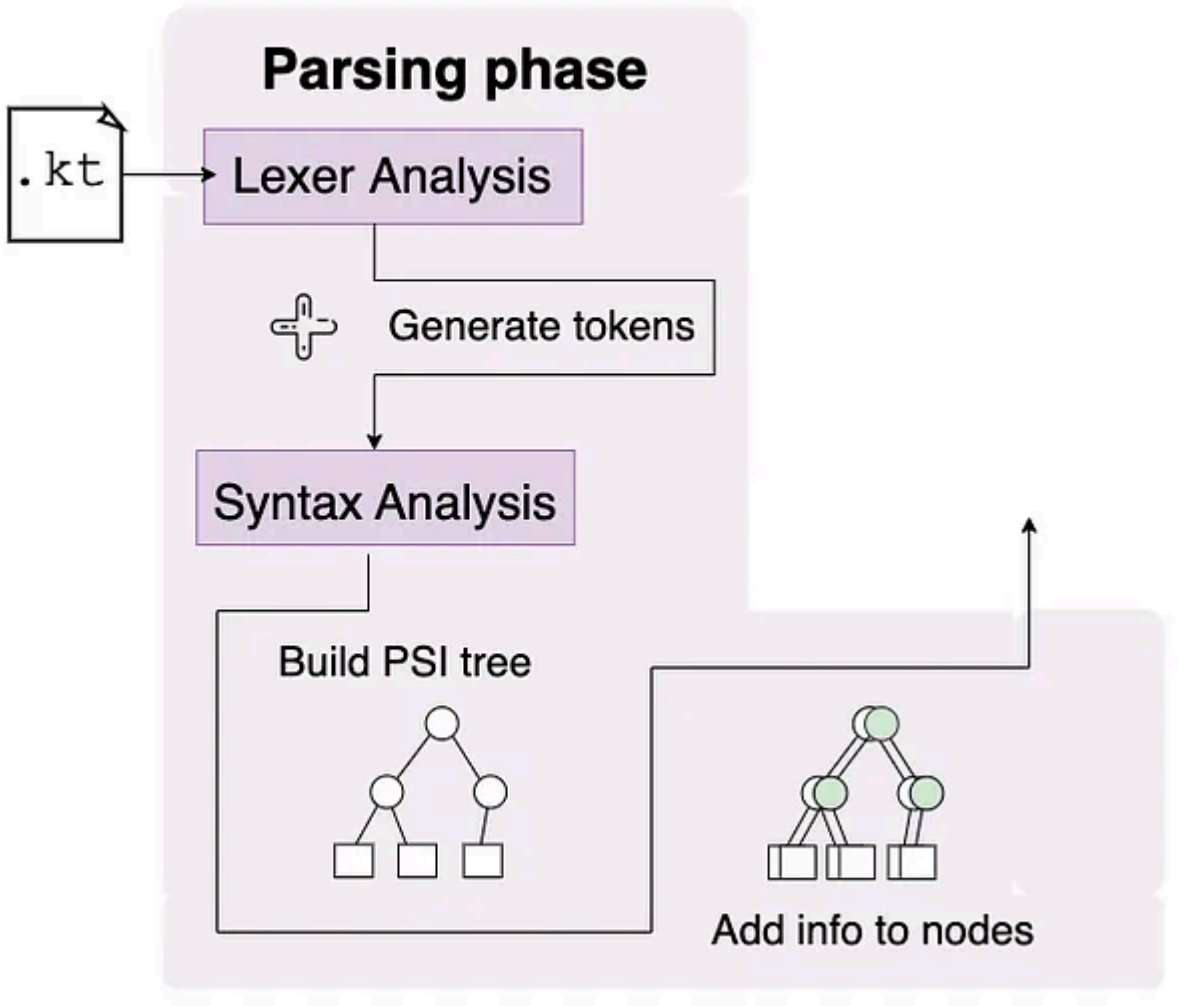
Задача парсинга — проверить **структуру** программы и убедиться, что она верная и следует **грамматическим правилам** языка.

Пример:



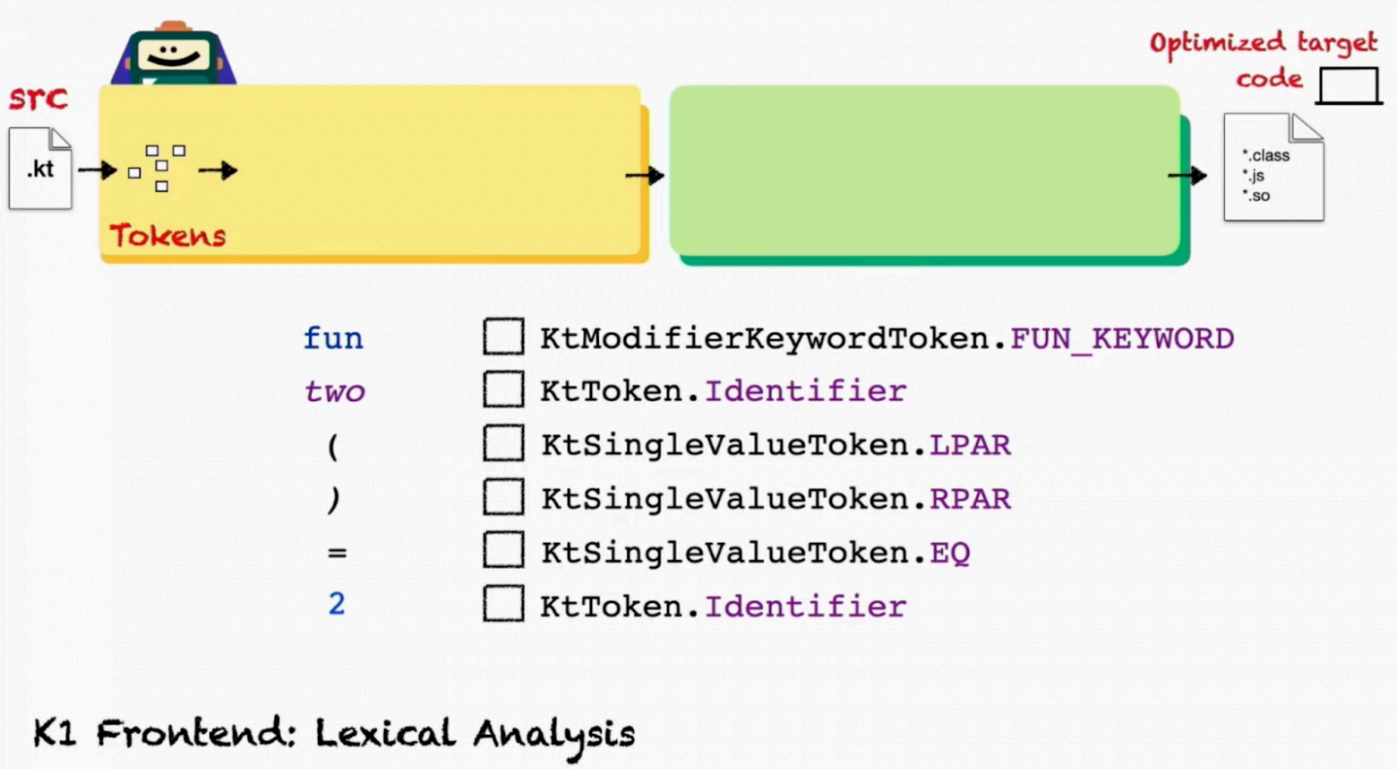


Этапы Parsing phase



1. Все начинается с **лексического анализа (Lexer Analysis)**. Сначала Kotlin Parser сканирует исходный код (строку) и разбивает его на лексические части (**KtTokens**). Например:
   - `(` —> `KtSingleValueToken.LPAR`
   - `)` —> `KtSingleValueToken.RPAR`
   - `=` —> `KtSingleValueToken.EQ`

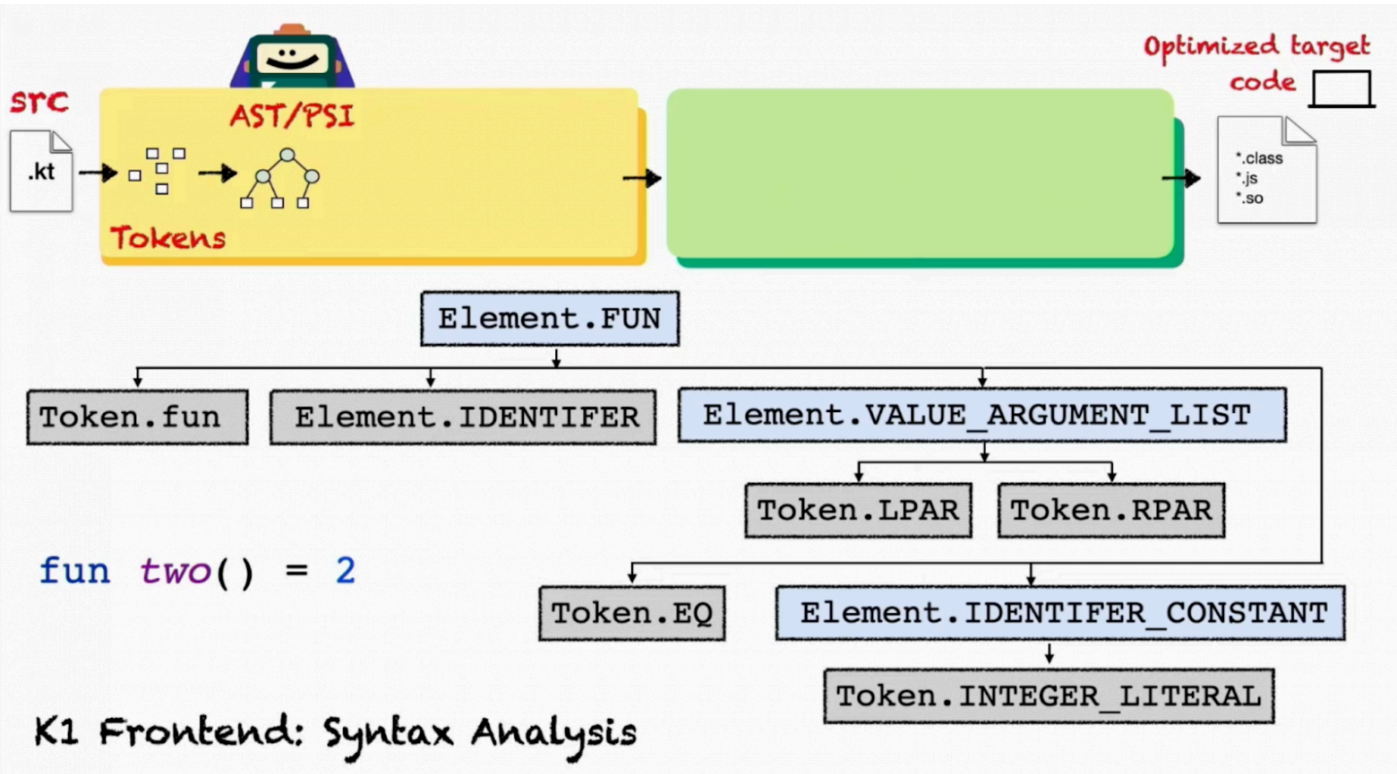
   В нашем примере разбиение на токены выглядит так:



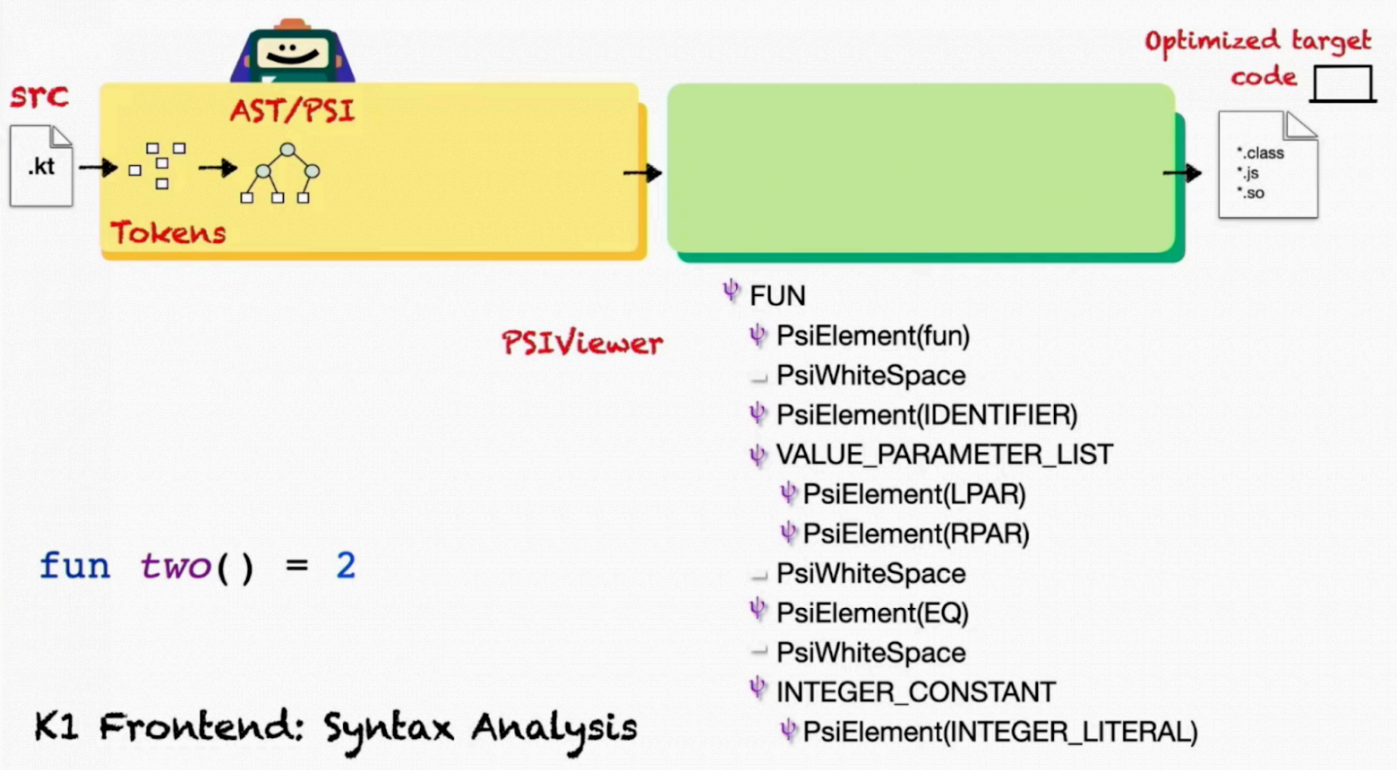
Пробелы тоже превращаются в токены. Но они не обозначены, чтобы не перегружать лишней информацией.

2. Далее начинается **синтаксический анализ (Syntax Analysis)**. Его первый этап — это построение **абстрактного синтаксического дерева (AST)**, которое состоит из лексических токенов. Важная особенность: дерево уже представляет из себя не последовательность символов, а **имеет структуру**.

   Обратите внимание, что структура этого дерева совпадает со структурой нашего исходного кода. В нашем примере это выглядит так:



3. Следующим этапом мы накладываем PSI на узлы абстрактного синтаксического дерева. Теперь у нас есть **PSI-дерево,** которое содержит дополнительную информацию по каждому элементу:



На этом этапе мы ищем только синтаксические ошибки. Другими словами, мы видим правильно написанный код, но пока не можем предсказать, скомпилируется он или нет и самое главное не понимаем смысл этого кода.

- В IntelliJ IDEA, вы можете загрузить плагин PSIViewer чтобы смотреть PSI для кода, написанного вами.

Resolution Phase

Задача семантического парсинга

Чтобы понять **смысл** написанного кода используется **Resolution Phase**.

Здесь PSI-дерево проходит набор этапов, в процессе которых генерируется семантическая информация, позволяющая понять (resolving) имена **функций, типов, переменных** и откуда они взялись.

Все мы в коде видели ошибки из этой стадии.

```
two(5) // Error: Too many arguments

three() // Unresolved reference
```

Семантическая информация содержит данные о:

- Переменных и параметрах
- Типах
- Вызовах функций

Эта стадия отвечает на похожие вопросы:

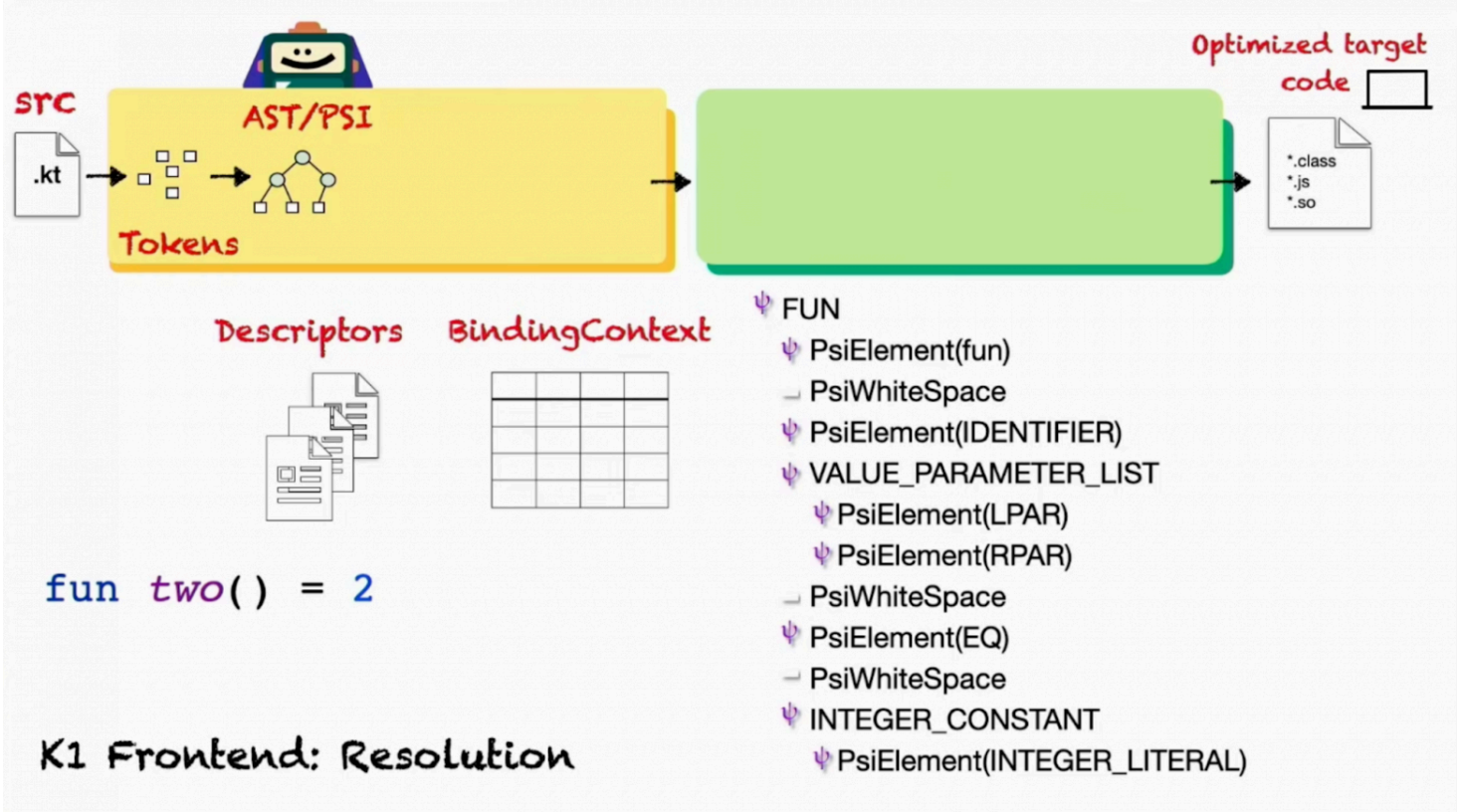
- Откуда появилась эта функция/переменная/тип для использования
- Точно ли две строки относятся к одной и той же переменной или это разные переменные в зависимости от контекста где они используются?
- Какой тип у конкретного объекта?

- Производится выведение типов

Этапы Resolution Phase

Реализация K1 выполняет вычисление на **PSI-дереве**, создает дескрипторы и **BindingContext** map, а затем передает все это на бэкенд. В итоге бэкенд преобразует полученную информацию в IR с помощью **Psi2Ir**.

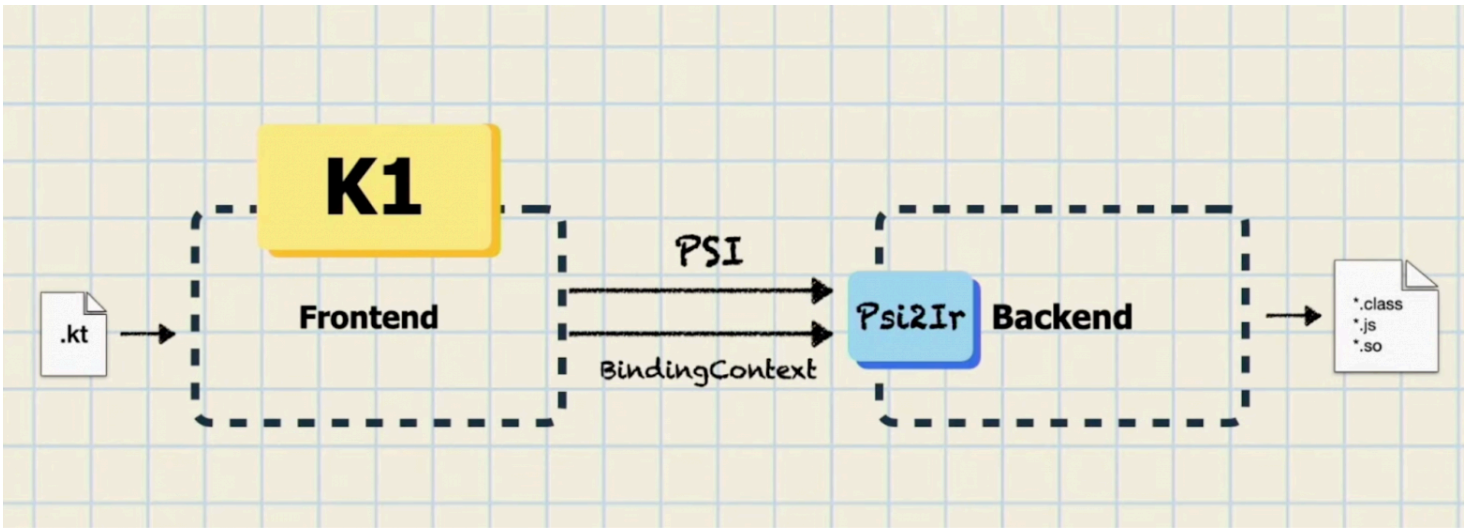
Вся эта информация будет использоваться дальше на бэкенде:



- Пример с деревом PSI и семантической информацией BindingContext

Проблемы K1

Мы разобрались с тем, как работает фронтенд-реализация K1. Обозначим этот процесс на схеме K1:



У этой реализации есть свои недостатки (объяснение Дмитрия Новожилова в kotlin slack). Преобразование через **PSI** и **BindingContext** приводит к проблемам с производительностью компилятора.

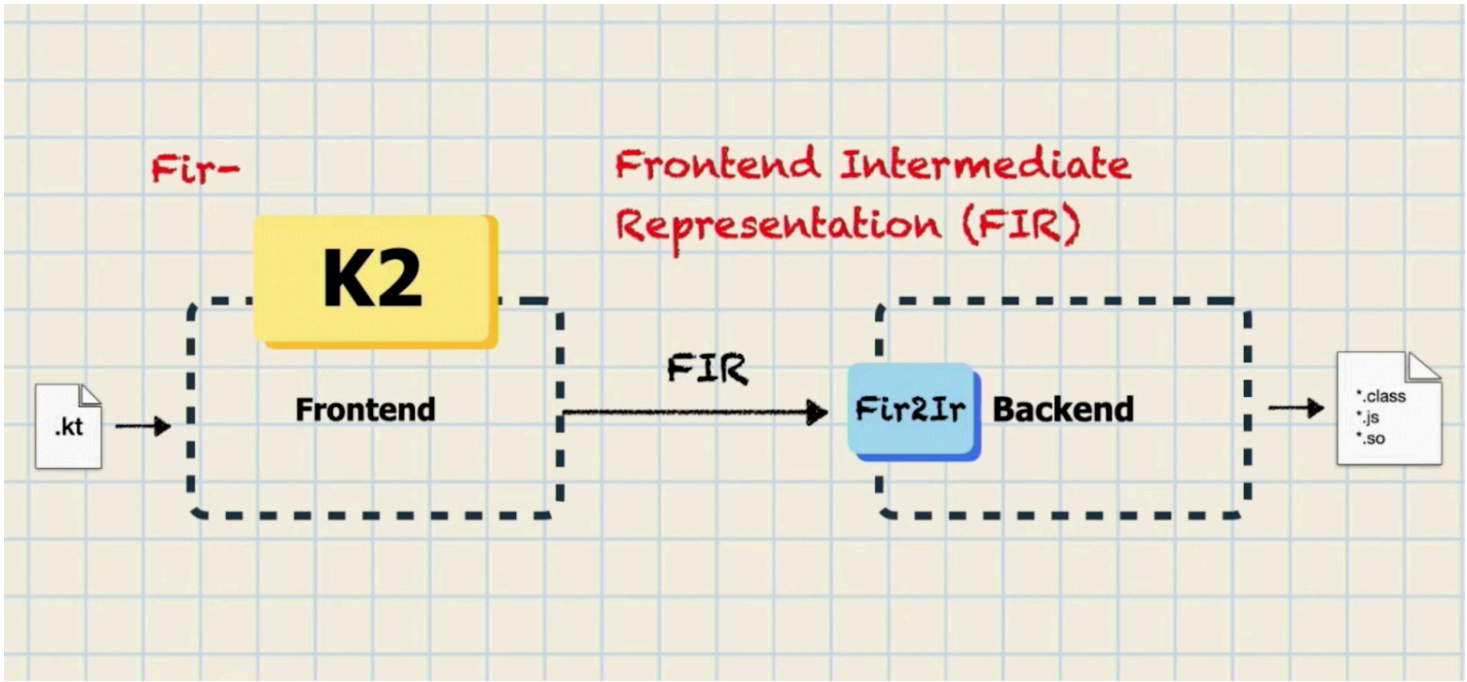
Тк компилятор работает с ленивыми дескрипторами, то выполнение постоянно прыгает между разными частями кода, это сводит на нет некоторые JIT-оптимизации. Кроме этого много информации resolution phase хранится в большом **BindingContext**, из-за этого ЦП не может хорошо кэшировать объекты.

Все это ведет к проблемам с производительностью.

Реализация K2

Чтобы повысить производительность компилятора и улучшить архитектуру решения, JB создала новую фронтенд-реализацию **K2 (Fir frontend)**.

Изменение произошло в структурах данных. Вместо старых структур (**PSI** + **BindingContext**) реализация K2 использует **Fir (Frontend Intermediate Representation)**. Это **семантическое дерево** (дерево, в узлах которого к структуре кода добавлена семантическая информация). Оно представляет исходный код.



Как работает K2

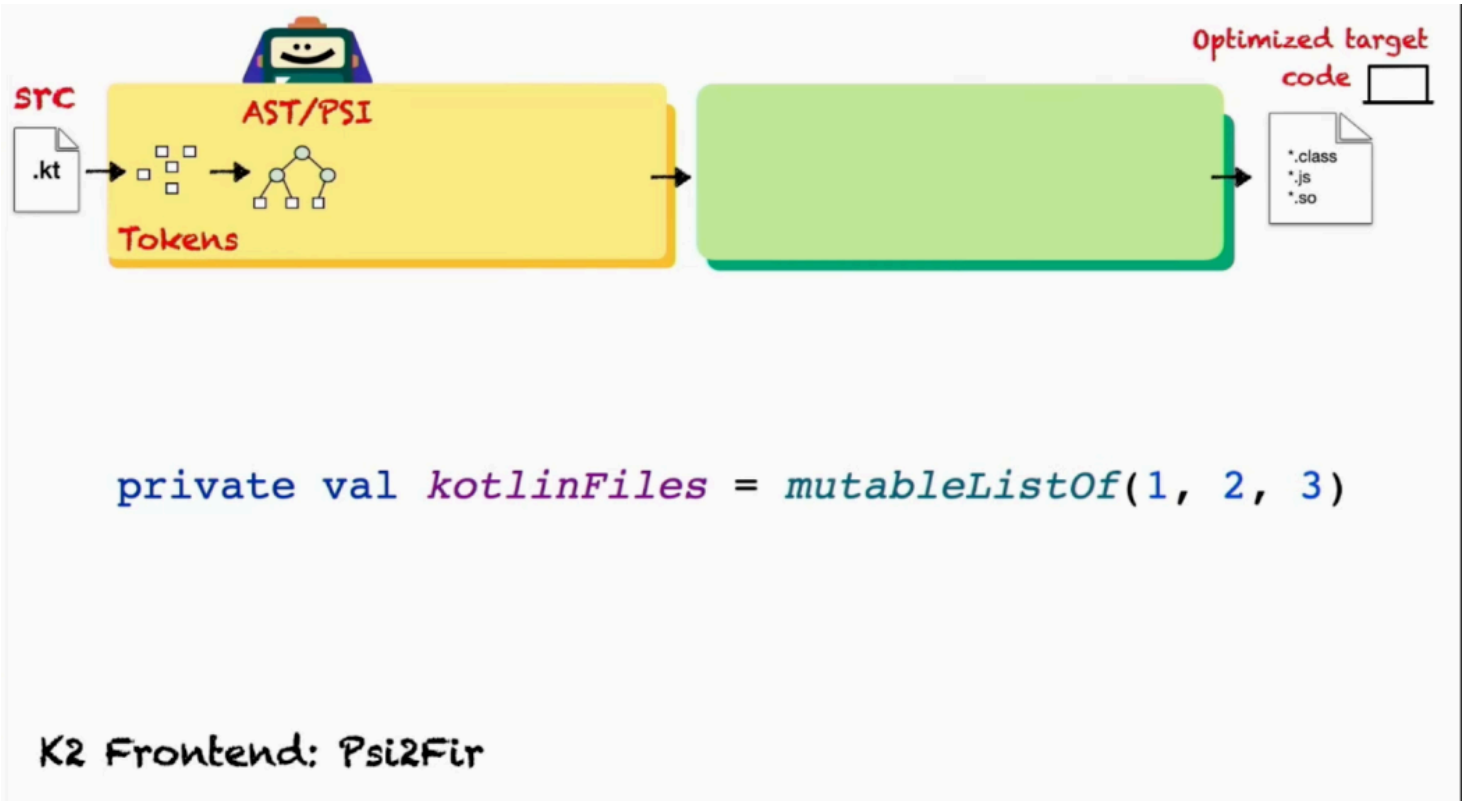
Вместо того, чтобы как раньше отправлять **PSI** и **BindingContext** в Backend, в K2 на Frontend происходят дополнительные преобразования:

1. Компилятор принимает raw PSI на входе (Parsing)
2. Производит незаполненное raw Fir-дерево (Psi2Fir)
3. Семантическая информация заполняет Fir-дерево (Resolution + Checking)
4. Преобразует Fir в Ir (Fir2Ir)
5. Передает Ir бэкенду

В случае с K2, мы рассмотрим более сложный случай и пропустим те фазы, в которых K1 и K2 работают одинаково.

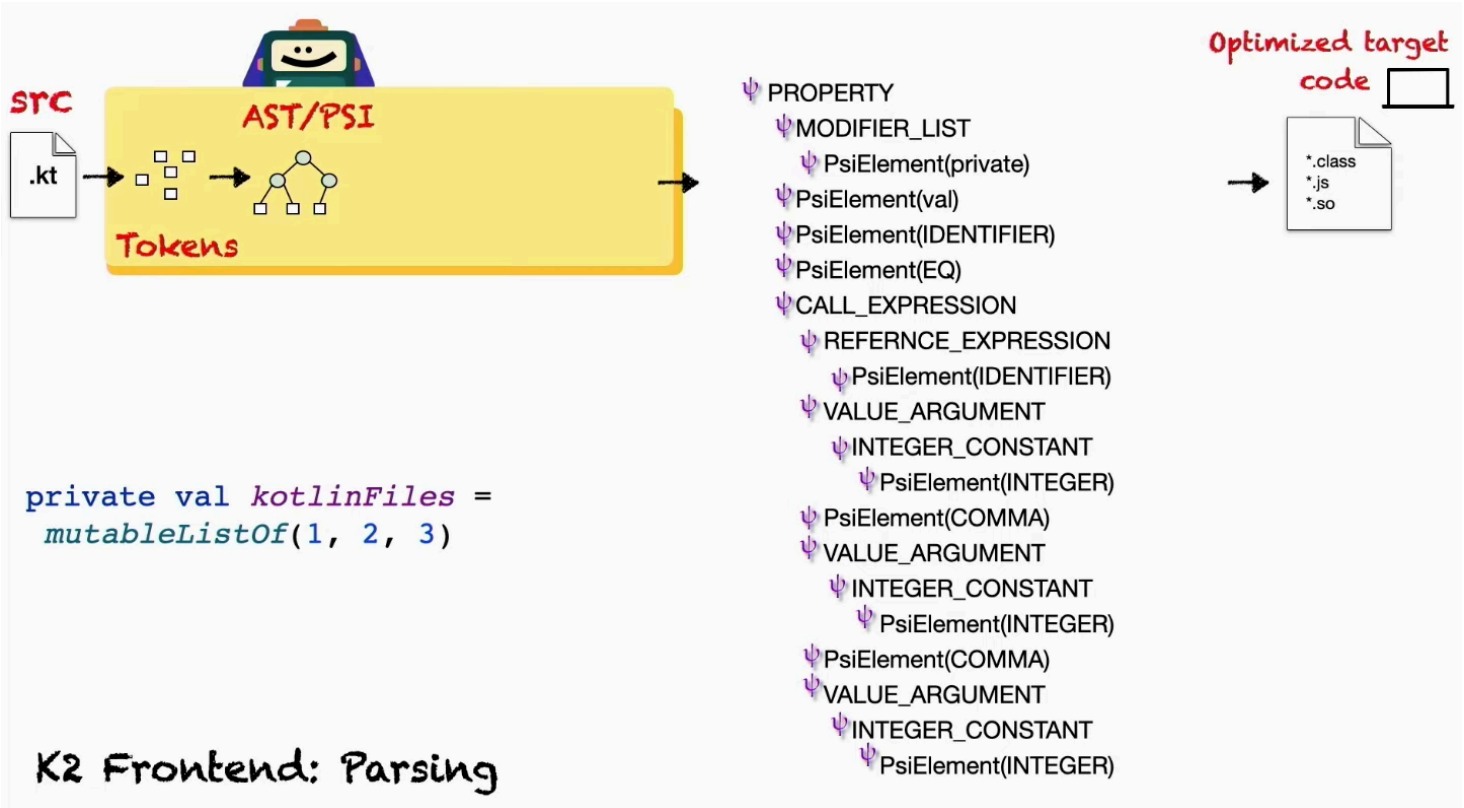
Для примера возьмем исходный код

```
private val kotlinFiles = mutableListOf(1, 2, 3)
```



Parsing

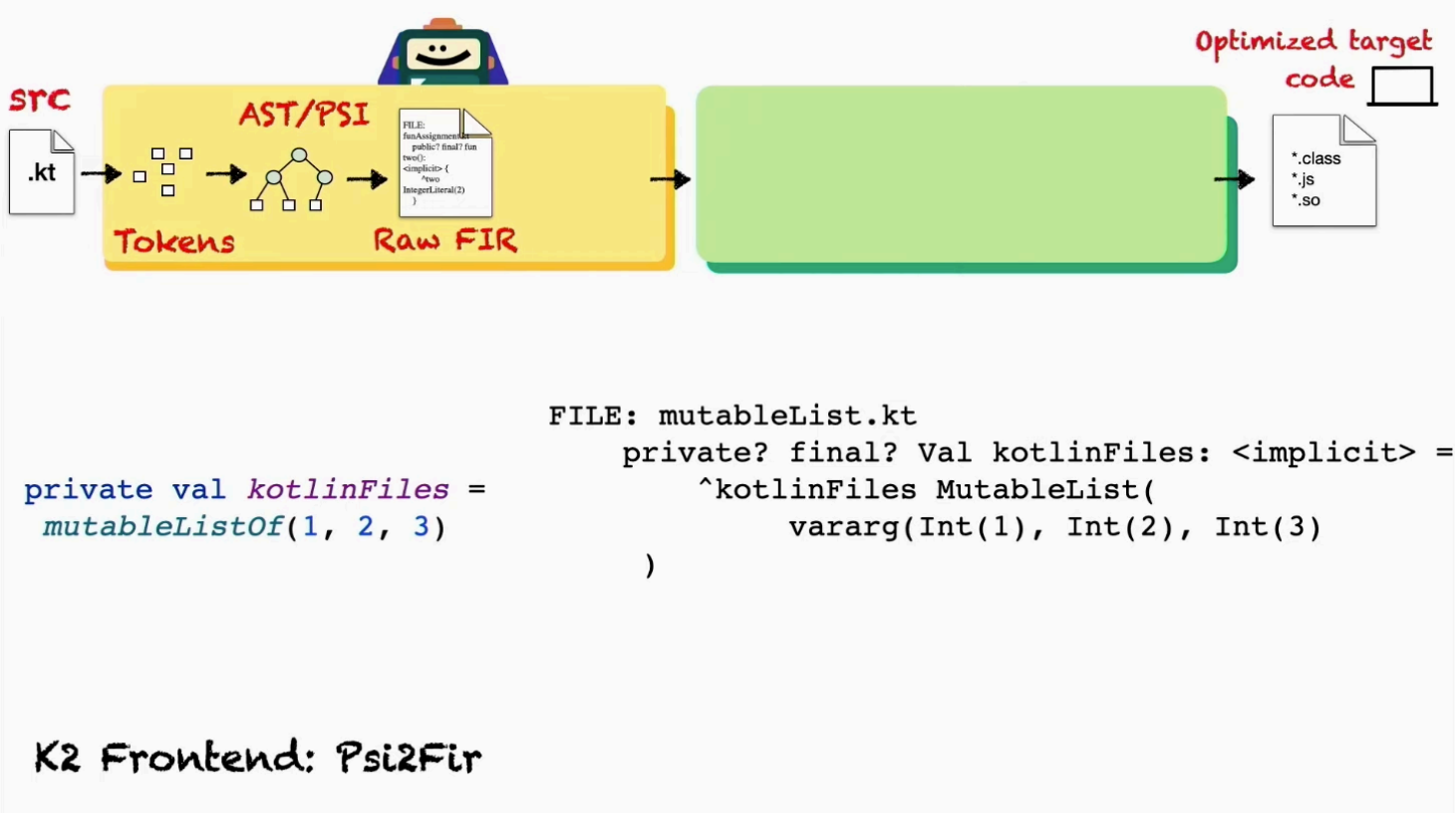
Начнем с уровня PSI (до этого K1 и K2 схожи):



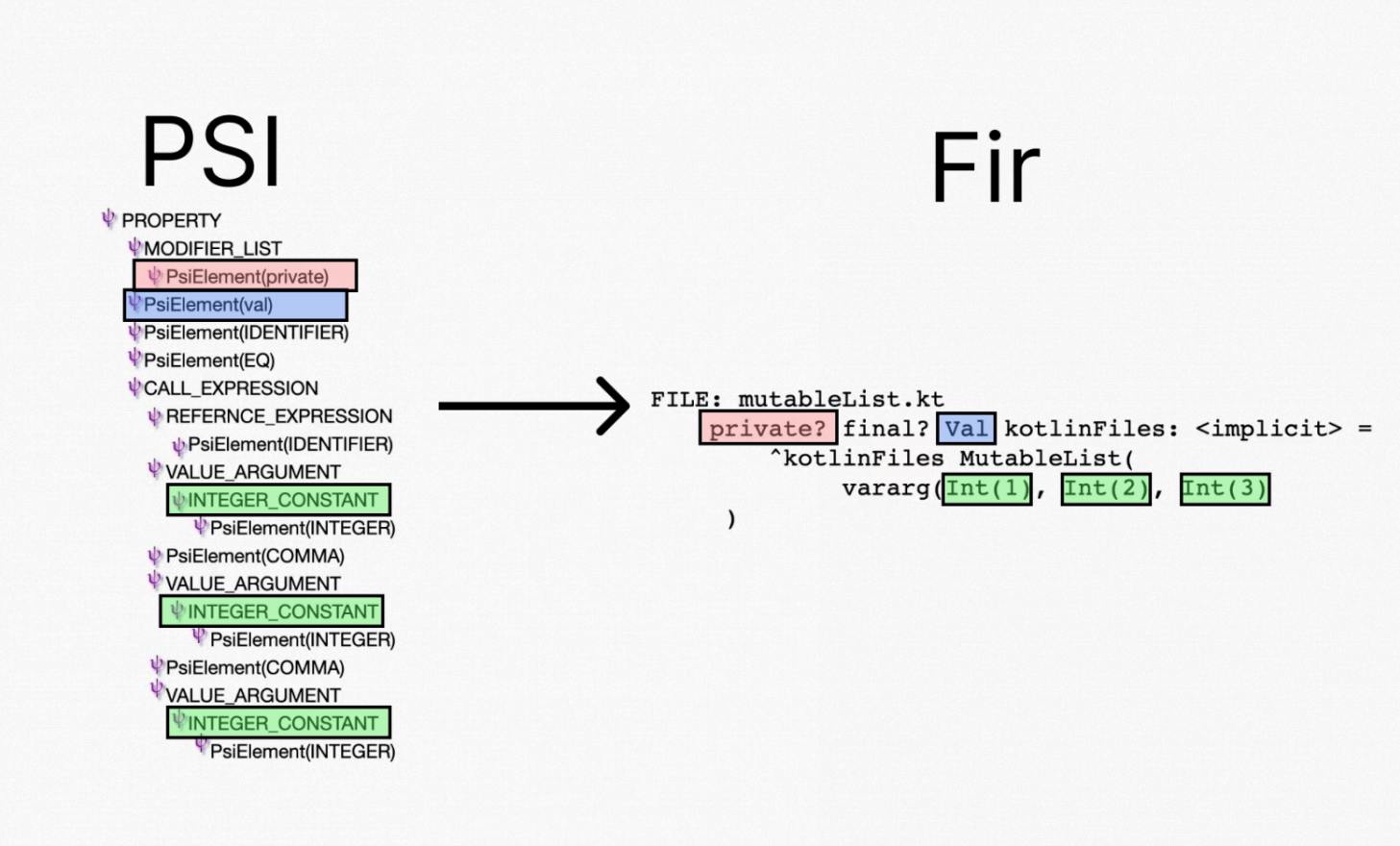
В этом случае, реализация K1 просто генерировала бы дополнительные данные для отправки PSI на бэкенд.

Psi2Fir

В отличие от нее, K2 компилирует этот промежуточный формат данных перед созданием Ir. В итоге получается **Fir** — это *mutable*-дерево, построенное из результатов парсера (PSI-дерева):

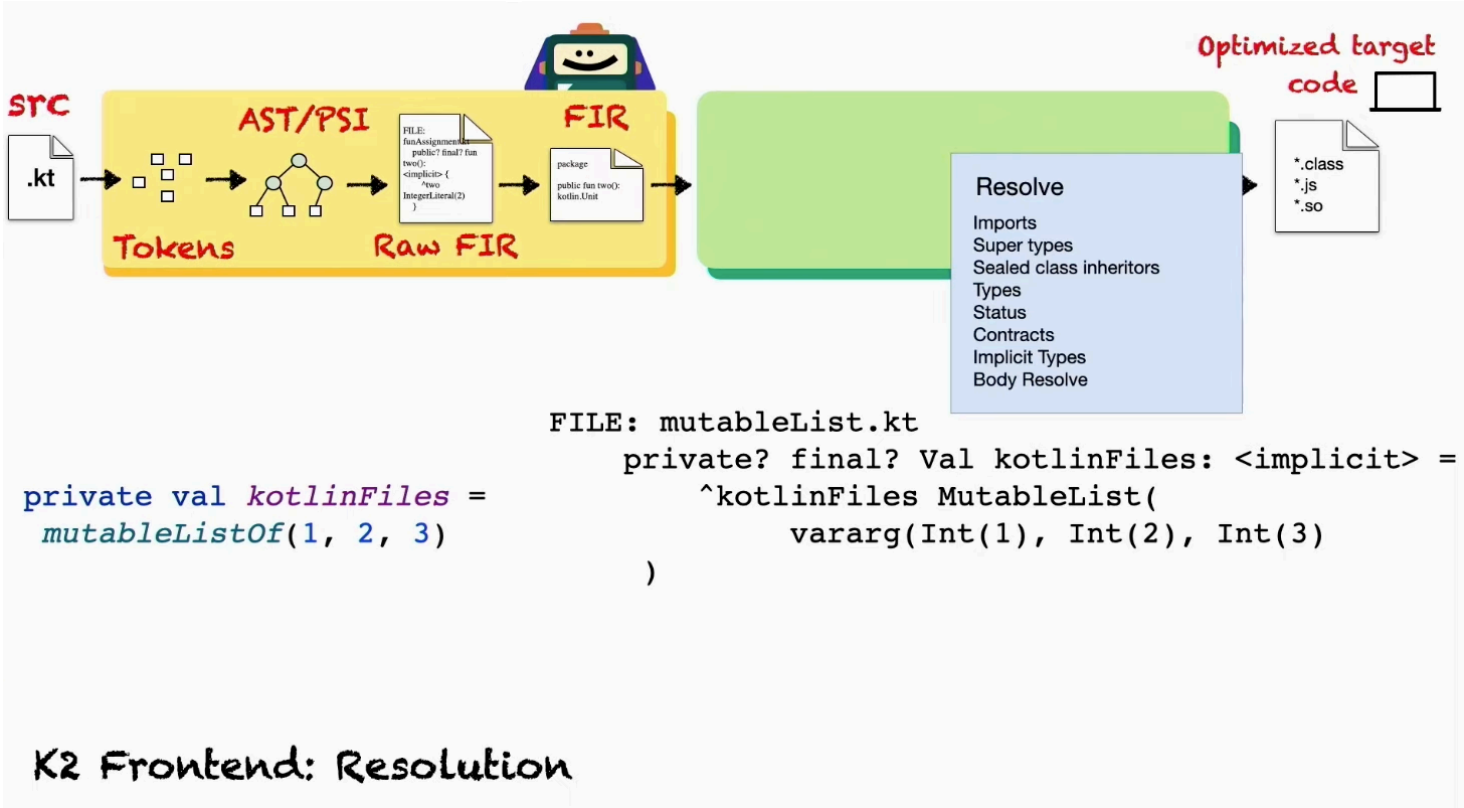


Тут видно сходство между PSI и raw Fir.



Resolution

Итак, мы создаем raw FIR и пересылаем его нескольким обработчикам. Они представляют собой различные этапы пайплайна компилятора и заполняют дерево семантической информацией:

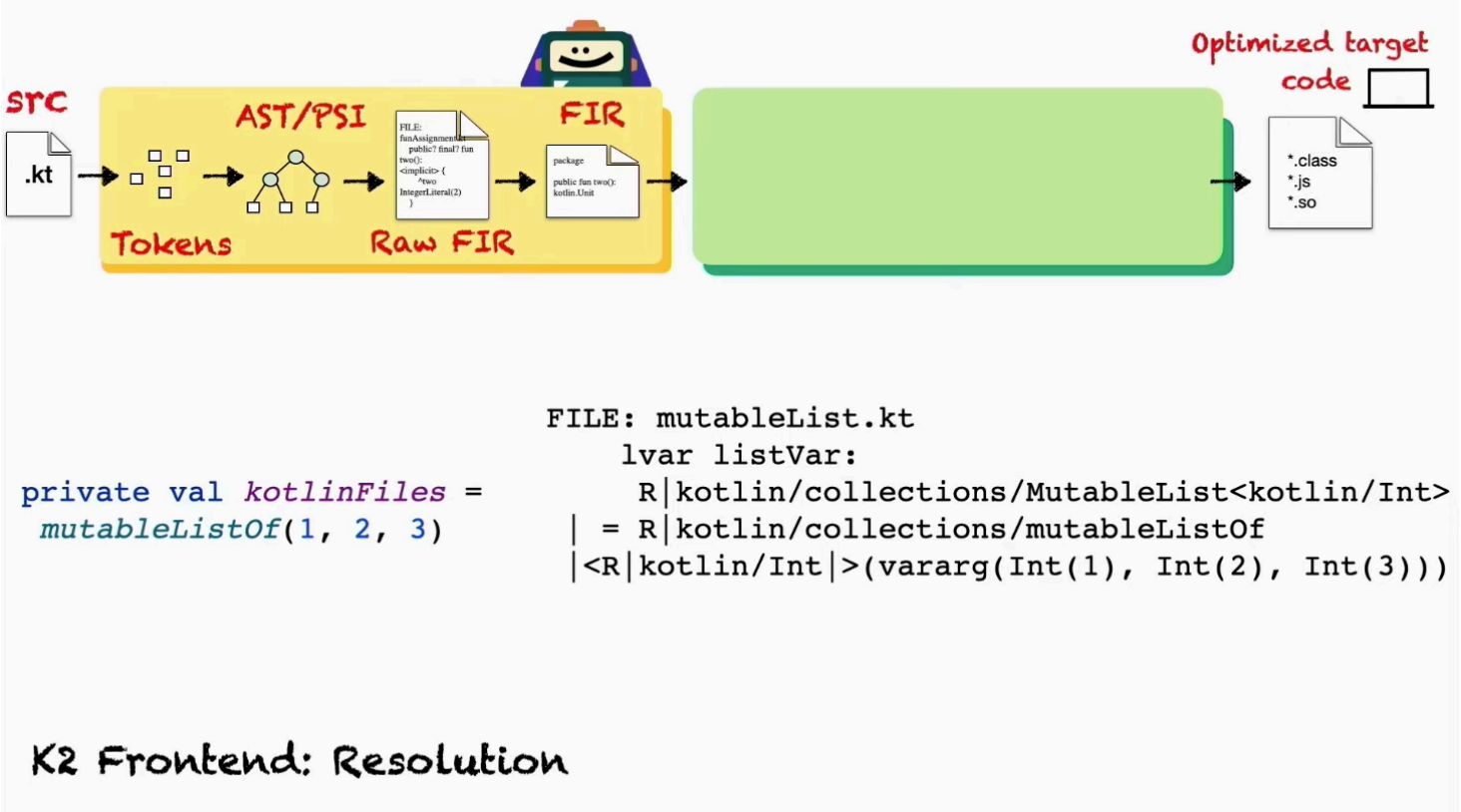


Все файлы в модуле проходят эти этапы одновременно.

В это время выполняются такие действия, как:

- Вычисление import’ов
- Поиск идентификаторов класса для супертипов и наследников sealed классов
- Вычисление модификаторов видимости
- Вычисление контрактов функций
- Выведение типов возвращаемого значения
- Анализ тел функций и свойств

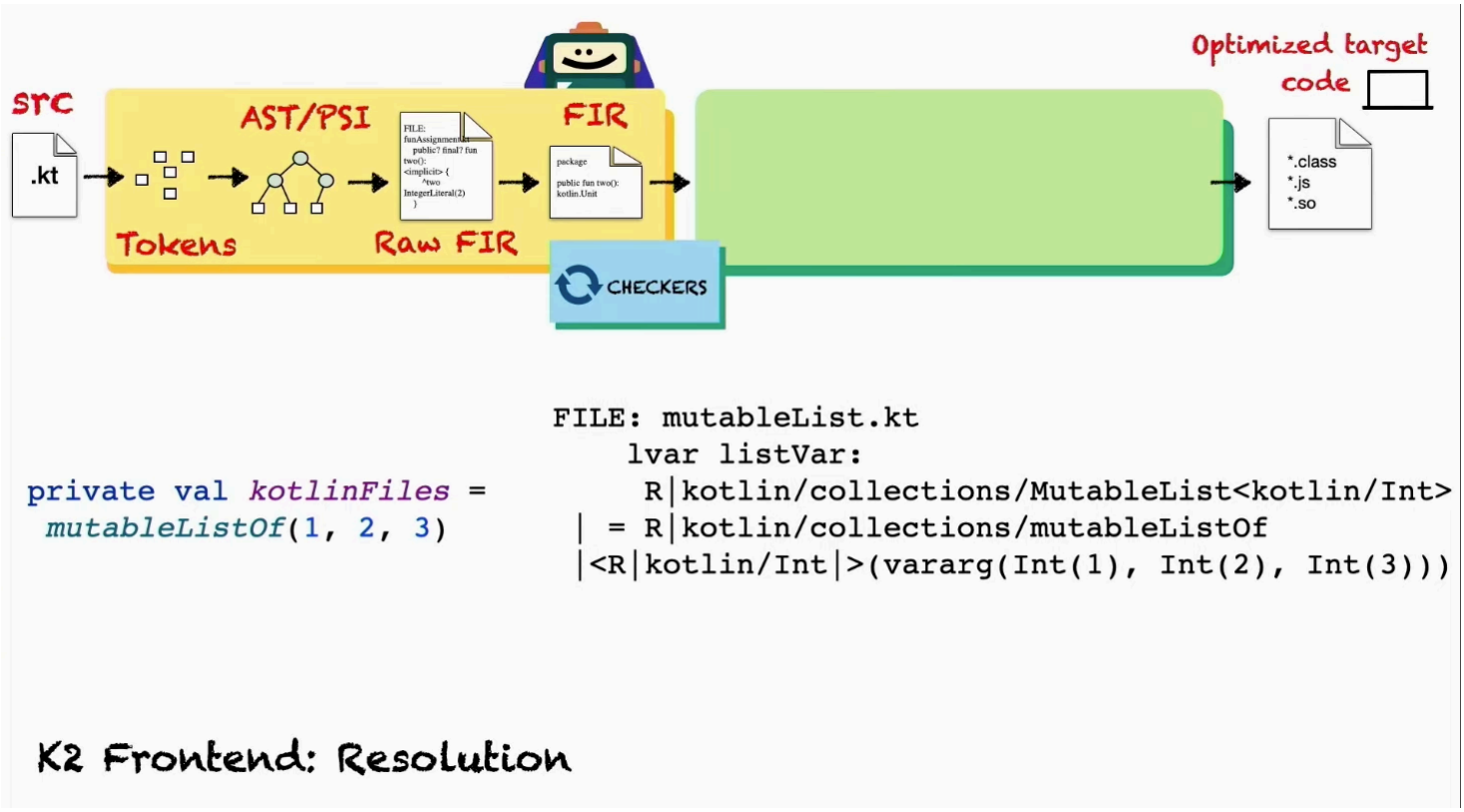
На стадии Resolution компилятор преобразует сгенерированное raw Fir, заполняя дерево семантической информацией:



На этих этапах компилятор выполняет desugaring. Например, `if` можно заменить на `when` или `for` — на `while`.

Раньше мы делали все это на Backend, но благодаря K2 то же самое можно делать на Frontend.

Checking

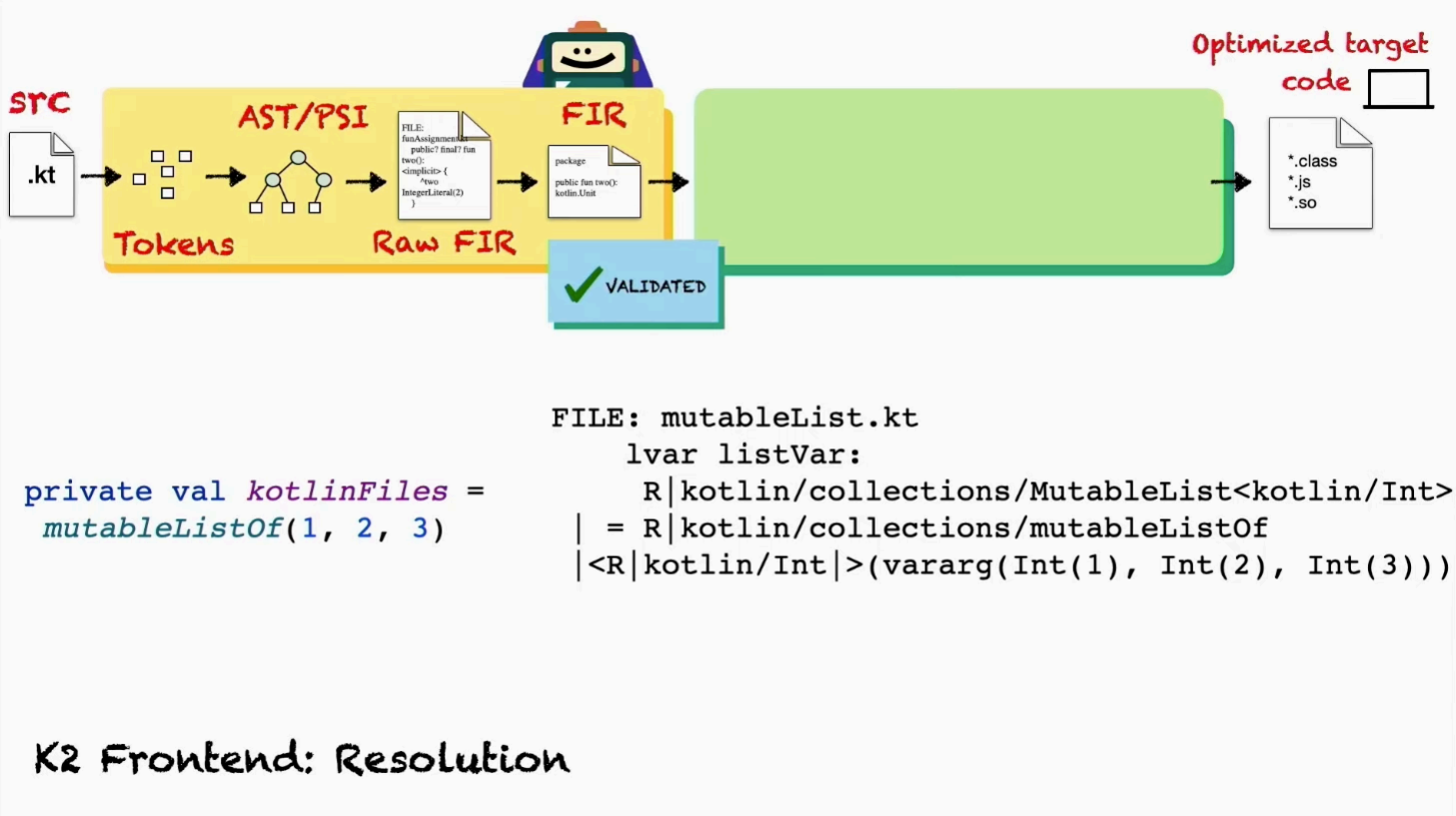


Мы пришли к последней стадии — стадии проверки. Здесь компилятор берет Fir и проводит диагностику на предмет предупреждений и ошибок.

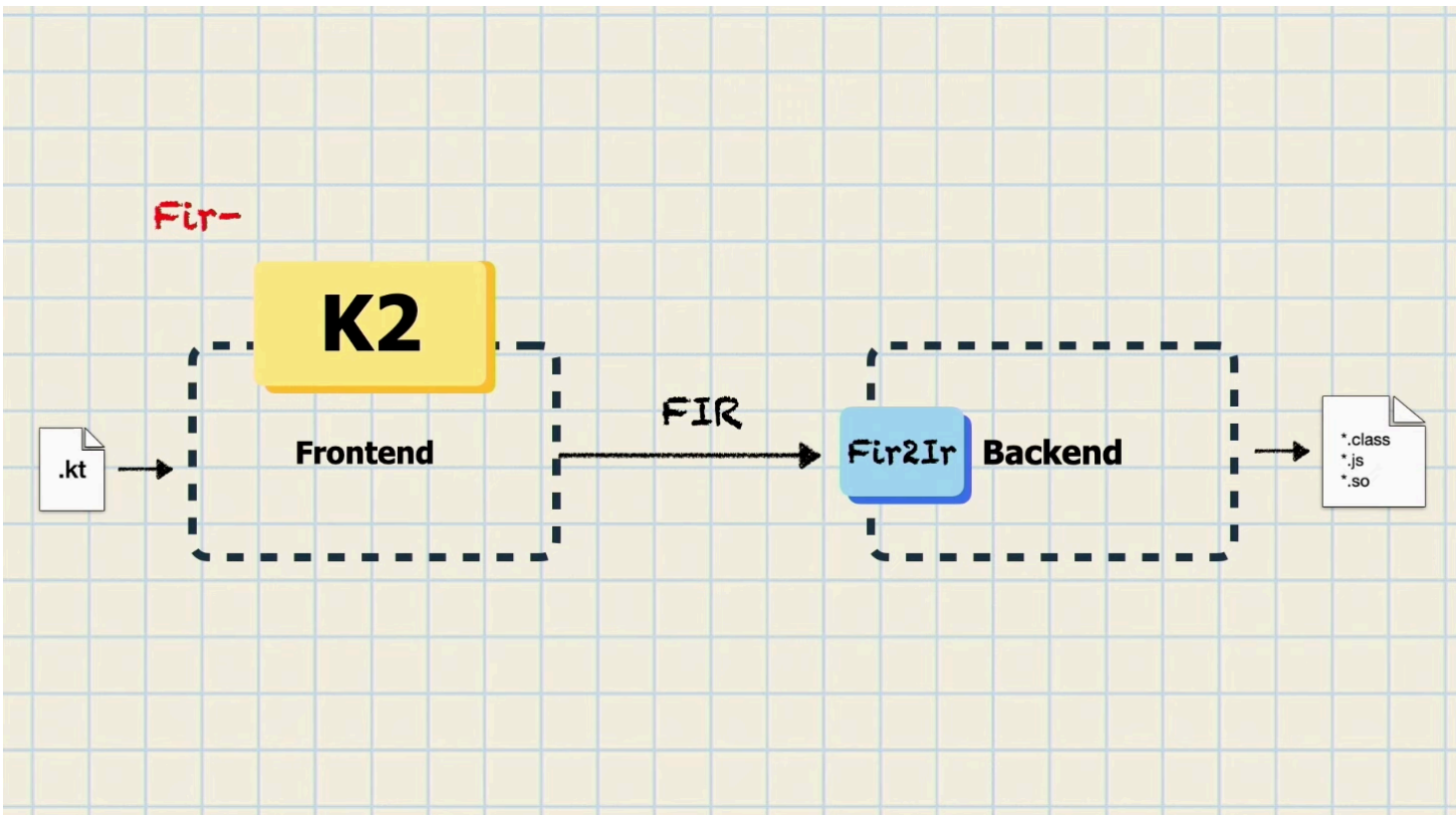
Если ошибки появились, данные отправляются плагину IDEA — именно он подчеркивает ошибку красной линией. Компиляция останавливается, чтобы не отправлять неверный код на бэкенд.

Fir2Ir

Если ошибок нет

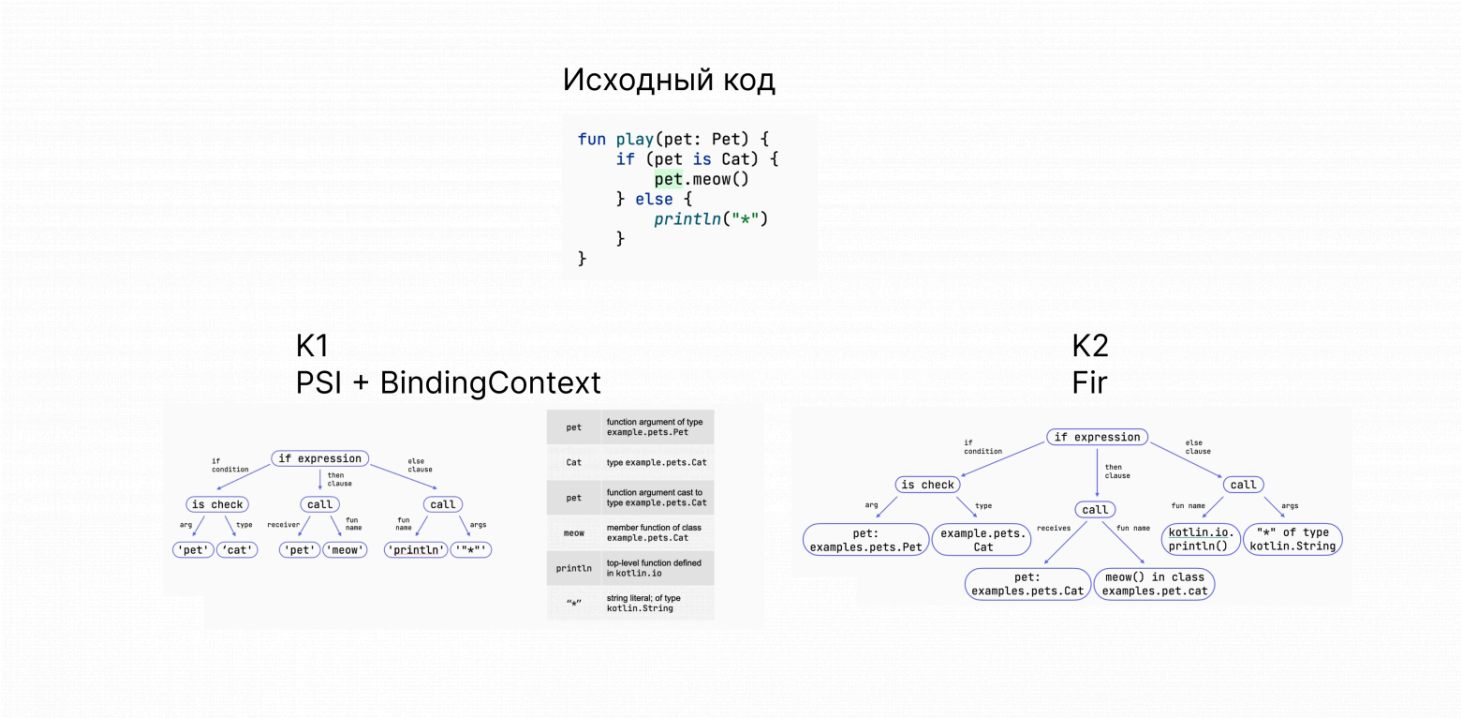


FIR трансформируется в IR



В итоге мы получаем входные данные для бэкенда

Визуальное сравнение результатов K1 и K2



Итого в K2 мы получили Fir (одну структуру данных вместо двух в K1 версии)

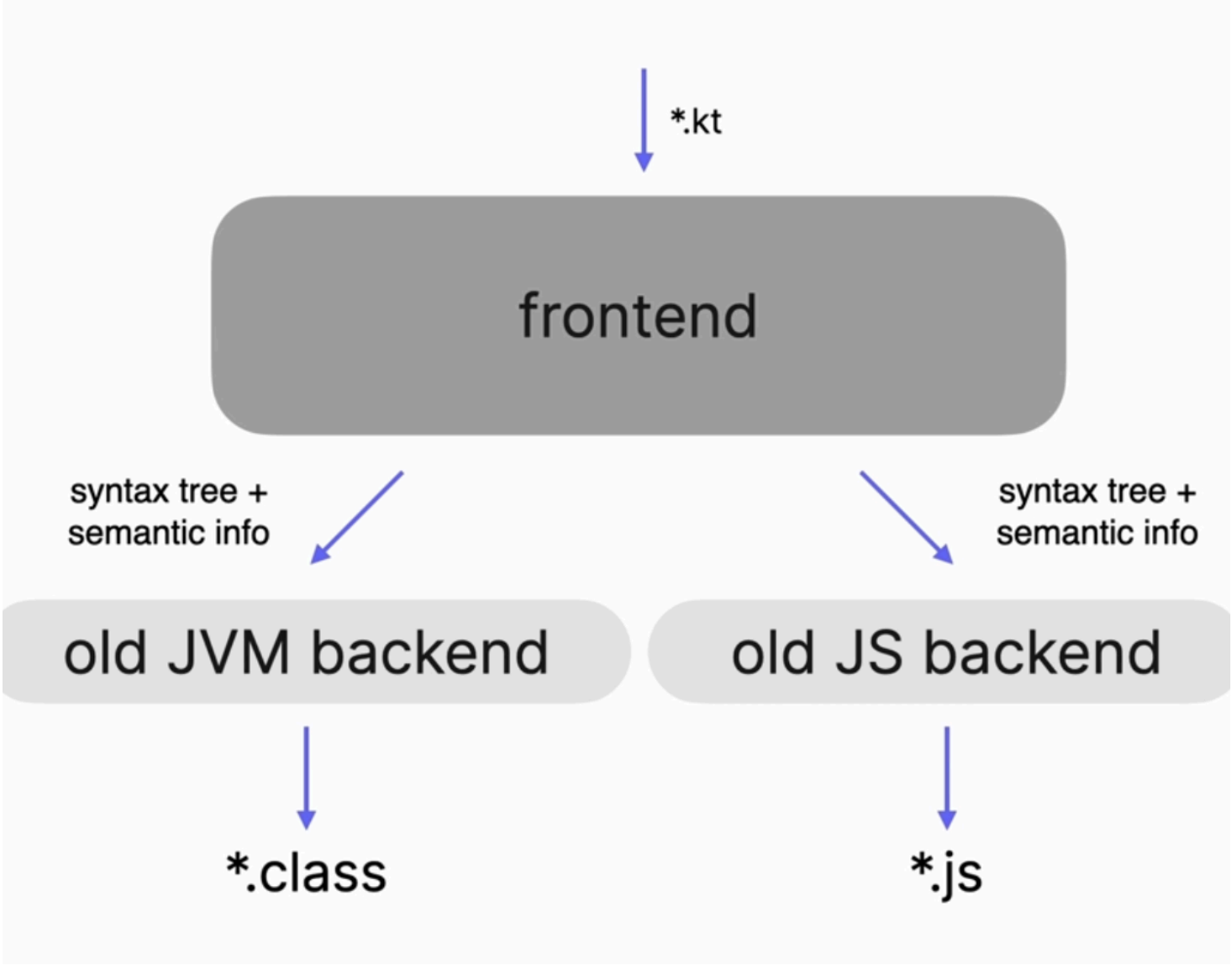
Backend

Цели улучшения Backend

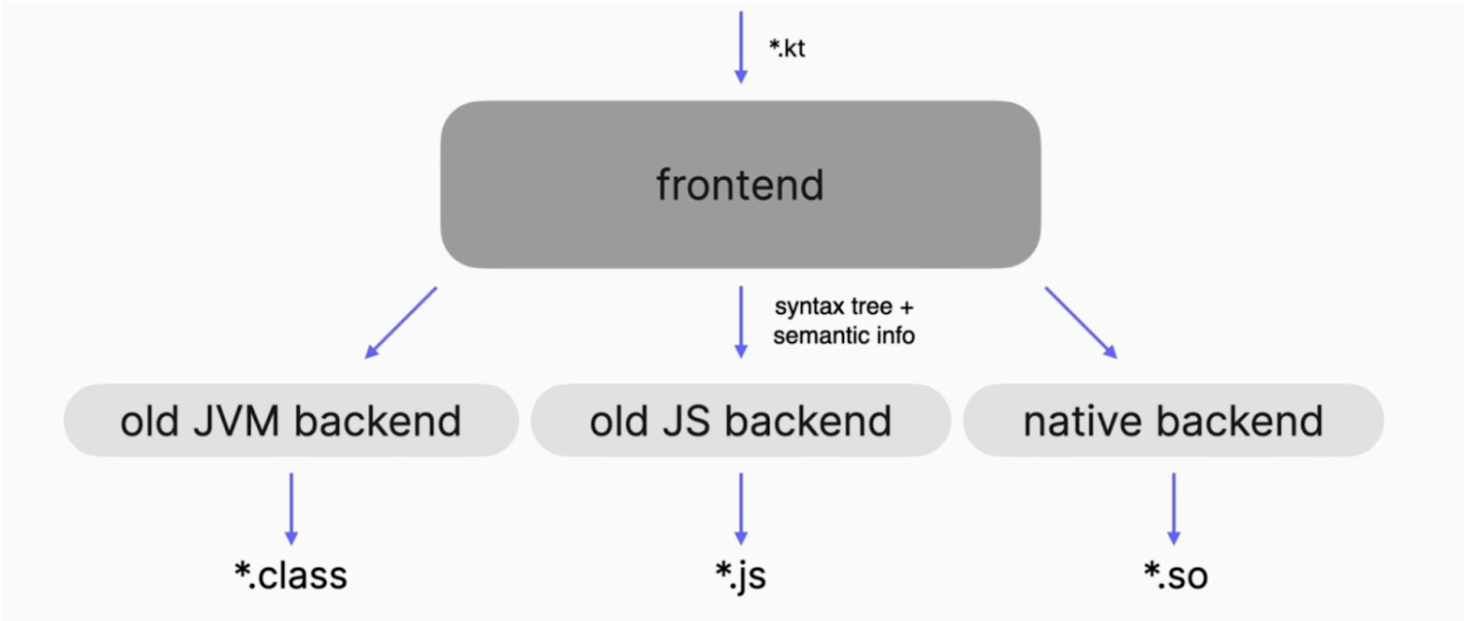
Изначально при разработке Kotlin компилятор не использовал промежуточное представление (IR). Тк это необязательная стадия при разработке компиляторов.

- Каноничная архитектура компилятора

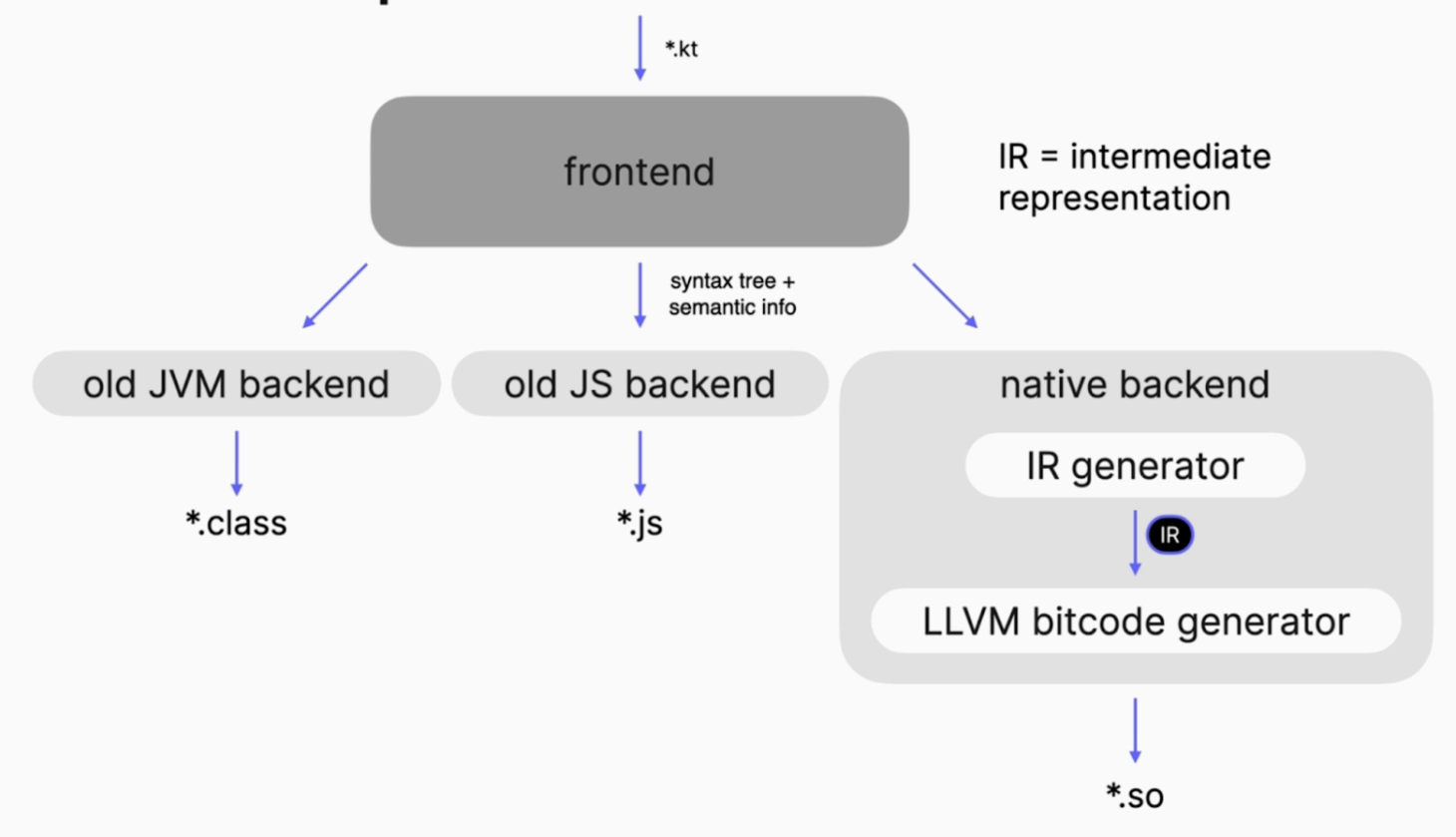
Отсутствие IR позволило Kotlin быстрее эволюционировать на ранних стадиях. Kotlin компилировался не только в JVM, но так же и в JS. JS высокоуровневый язык и он похож больше на Kotlin, чем на JVM байткод. Поэтому IR бы только тормозило разработку преобразования.



Но вскоре появился еще 1 native backend.



Стало понятно что все 3 бэкенда могут использовать общую логику по трансформации и упрощению кода, что позволит поддерживать фичи для разных бекендов проще. Поэтому начиная с native-backend команда Kotlin решила использовать IR, сначала только для него самого.



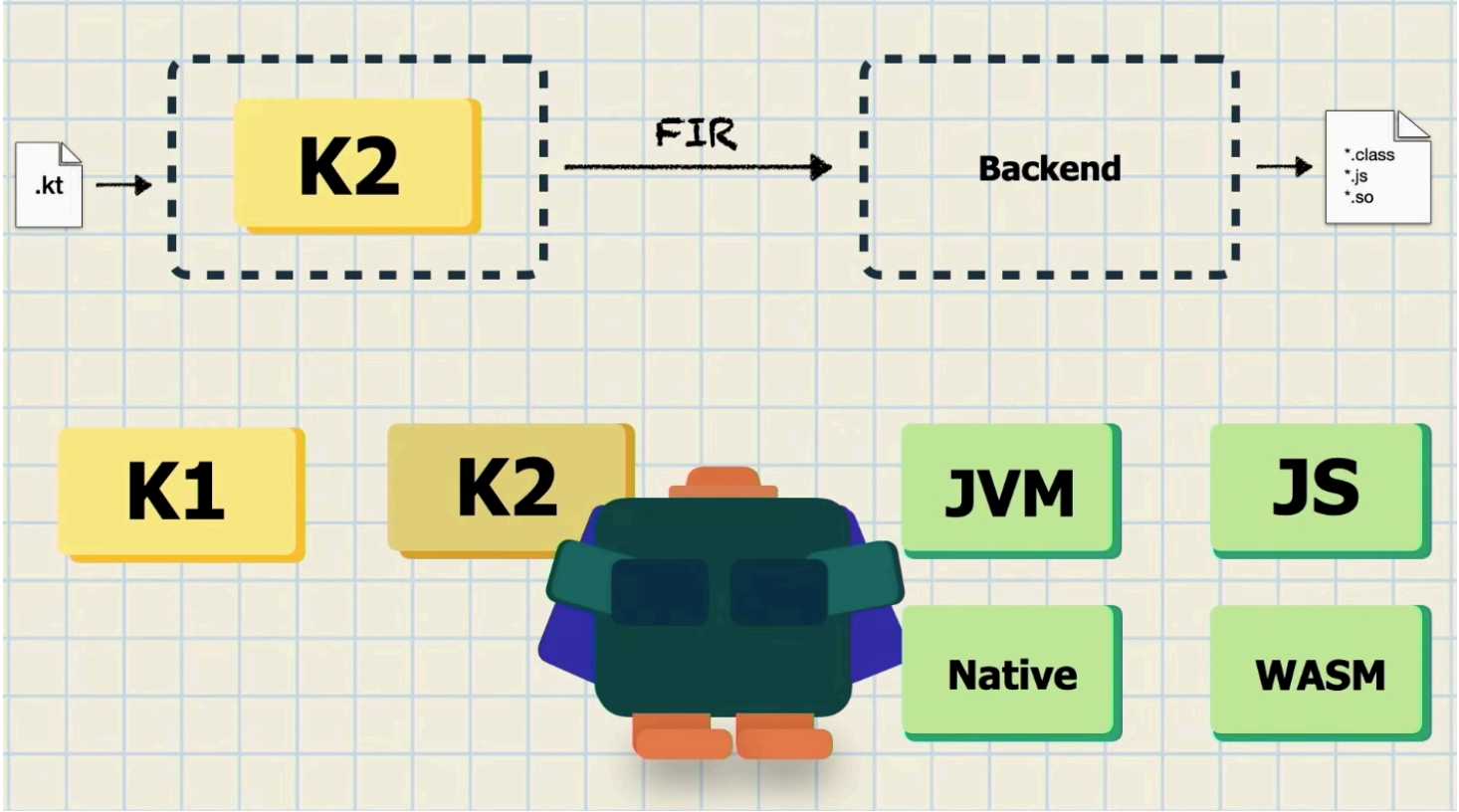
Итак, в качестве целей разработки новой версии можно выделить:

- переиспользование логики между разными бекендами
- упрощение поддержки новых языковых возможностей

Необходимо отметить, что ускорение Backend не являлось целью. В отличие от ускорения Frontend-части.

Новая реализация Backend позволила расширять компилятор с помощью компиляторных плагинов. В частности реализация Jetpack Compose полагается на новую версию бекенда с использованием IR.

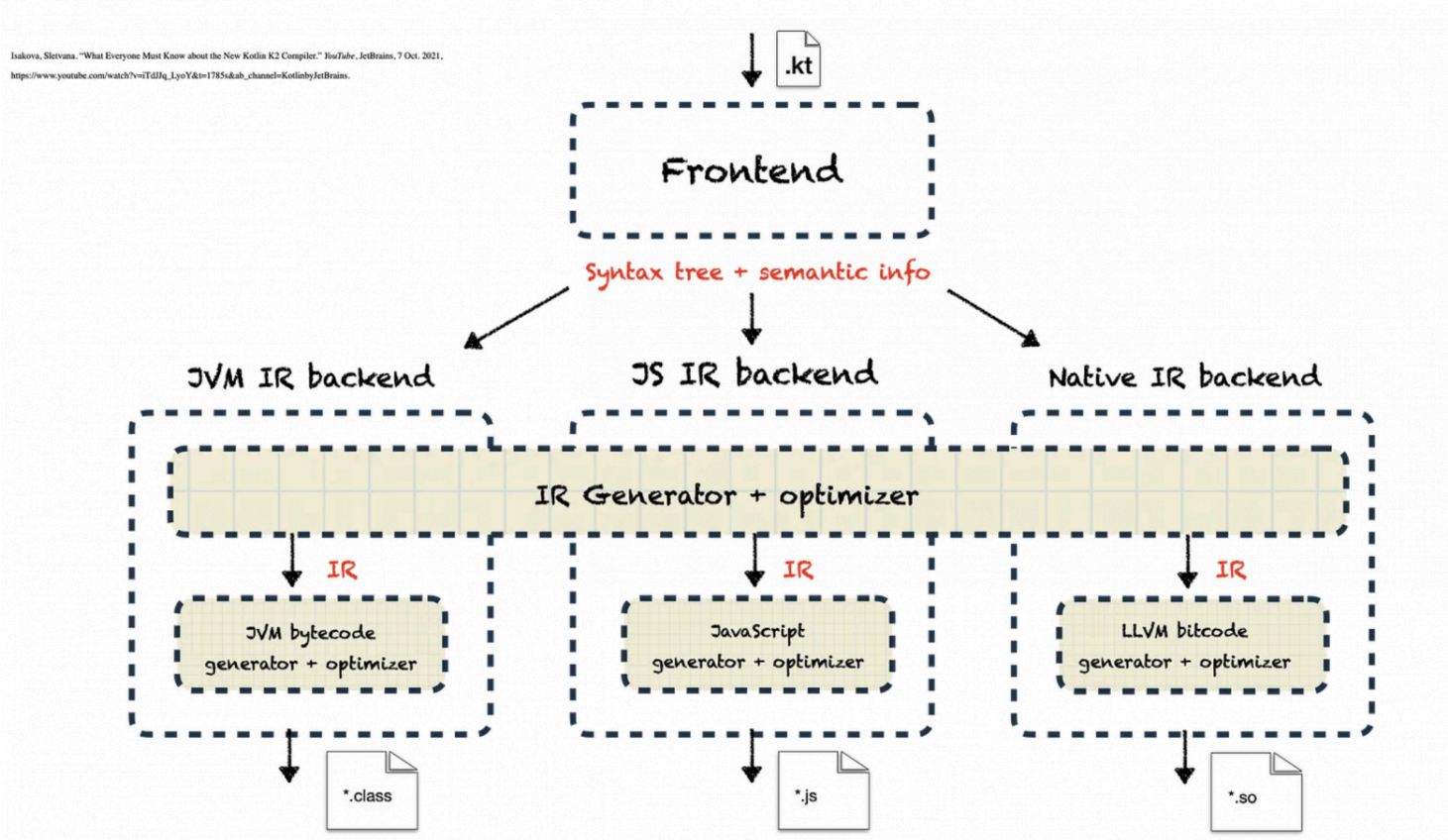
Как работает Backend



Сейчас в Kotlin есть четыре реализации backend:

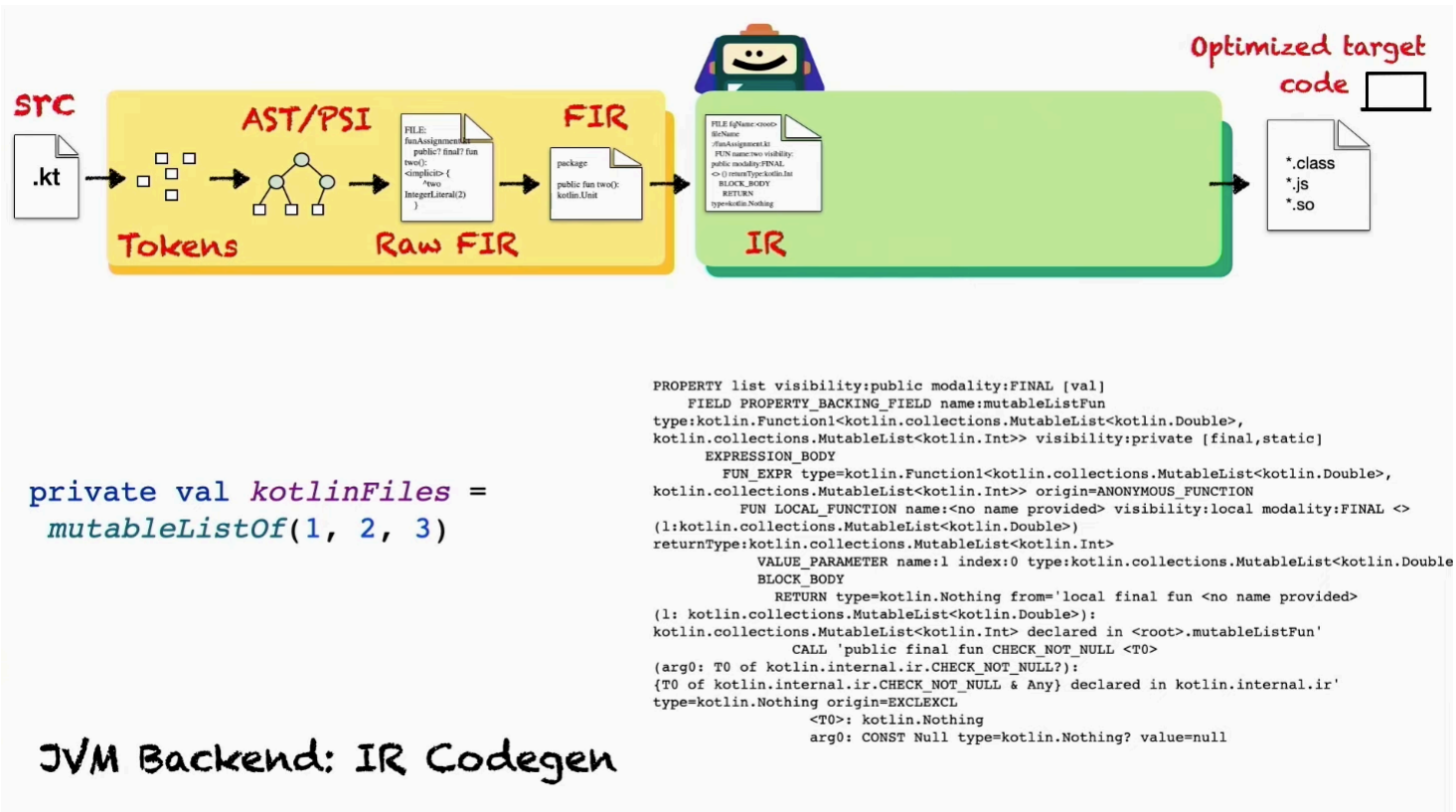
- JVM
- JS
- Native
- WASM

Какой бы бэкенд вы ни использовали, обработка в нем всегда начинается с генератора IR и оптимизатора. Затем выбранная конфигурация запускает сгенерированный код:



Для примера выберем бэкенд JVM и продолжим работу с mutableList

IR

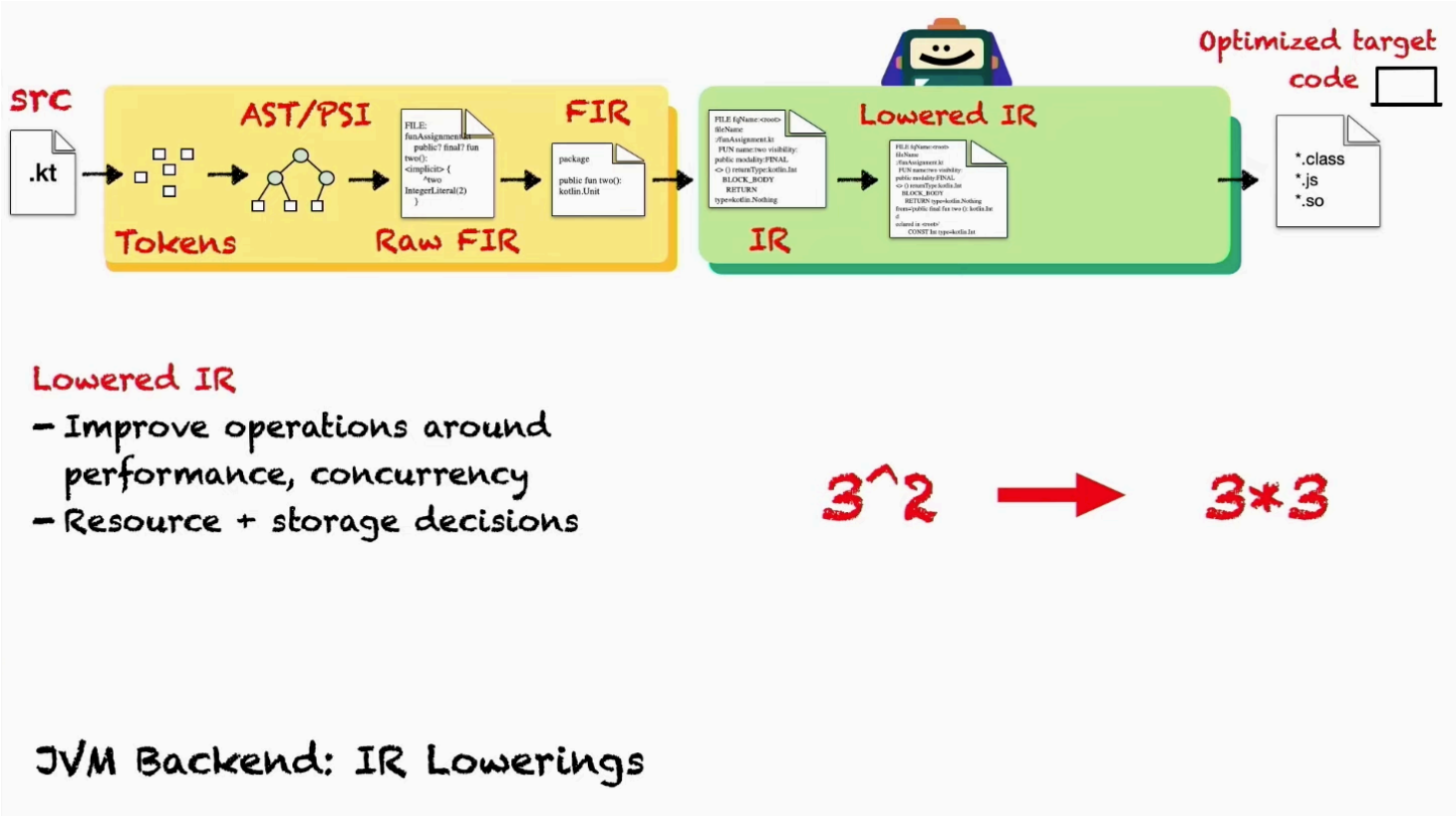


IR это также дерево. В IR добавлена вся семантическая информация. Для человека IR на выглядит не очень читабельно, но зато такой вид гораздо понятнее для компилятора. В момент анализа IR создаются поток управления и стеки вызовов.

Можно рассматривать IR как представление кода в виде дерева, которое **разработано специально для генерации кода target-платформы**. В отличие от Fir на фронтенде, который проверяет смысл написанного кода.

IR lowering

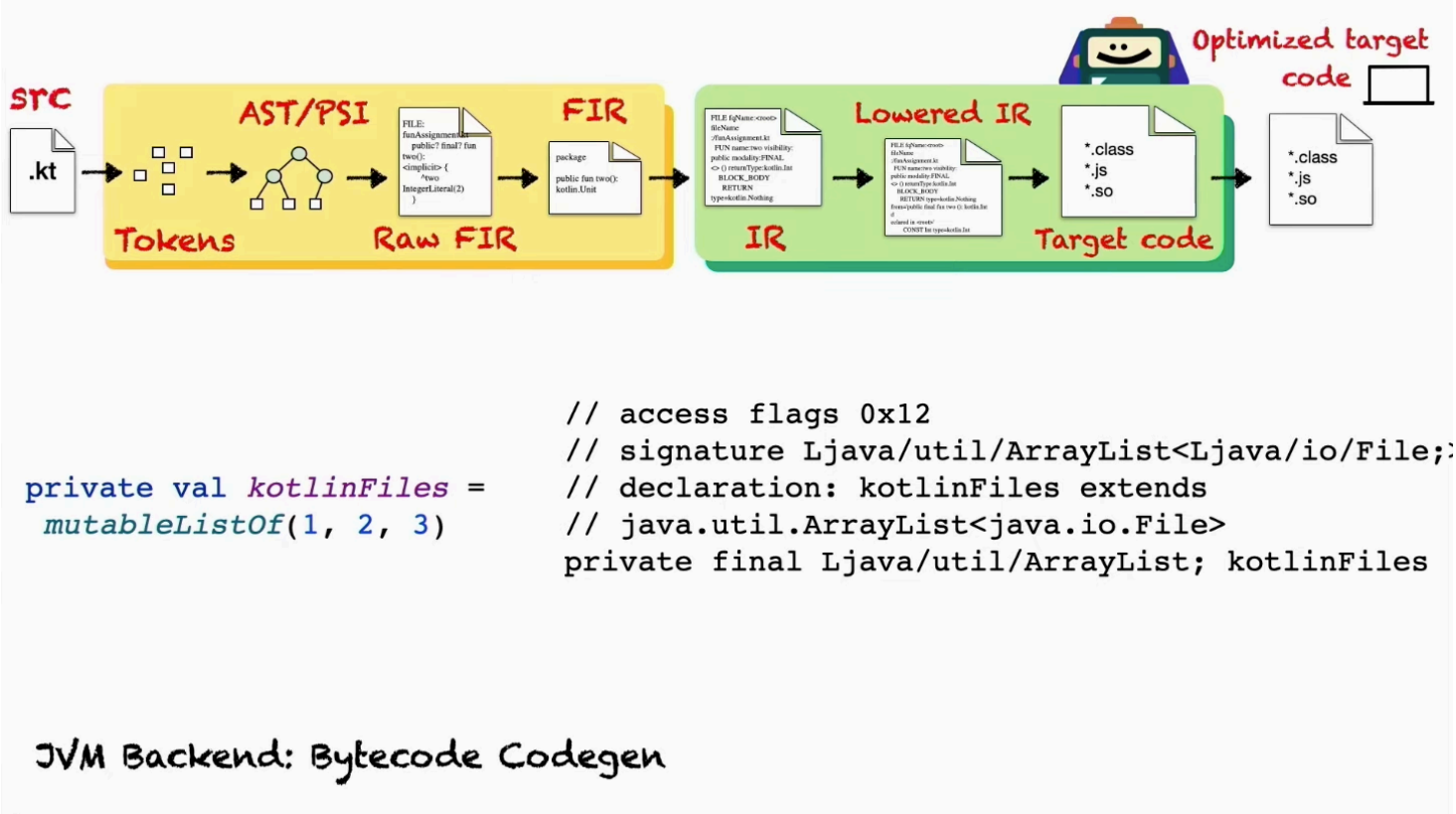
Затем на IR выполняются оптимизации и упрощения это называется **lowering**. Таким образом упрощается сложность Kotlin. Например в IR отсутствуют local и internal классы, они заменены отдельными классами со специальными именами. С помощью упрощений разным backend'ам не нужно бороться со сложностью языка Kotlin и реализация становится проще.



Также на этом этапе упрощаются операции, улучшается производительность и качество машинного кода, особенно с точки зрения многопоточности.

Target code

Теперь у нас есть сгенерированный код целевой платформы. В нашем примере это JVM байт-код. Далее он отправляется на виртуальную машину, где он и будет исполняться:



На этом задачи компилятора считаются выполненными.

Выводы

Что дает новый компилятор:

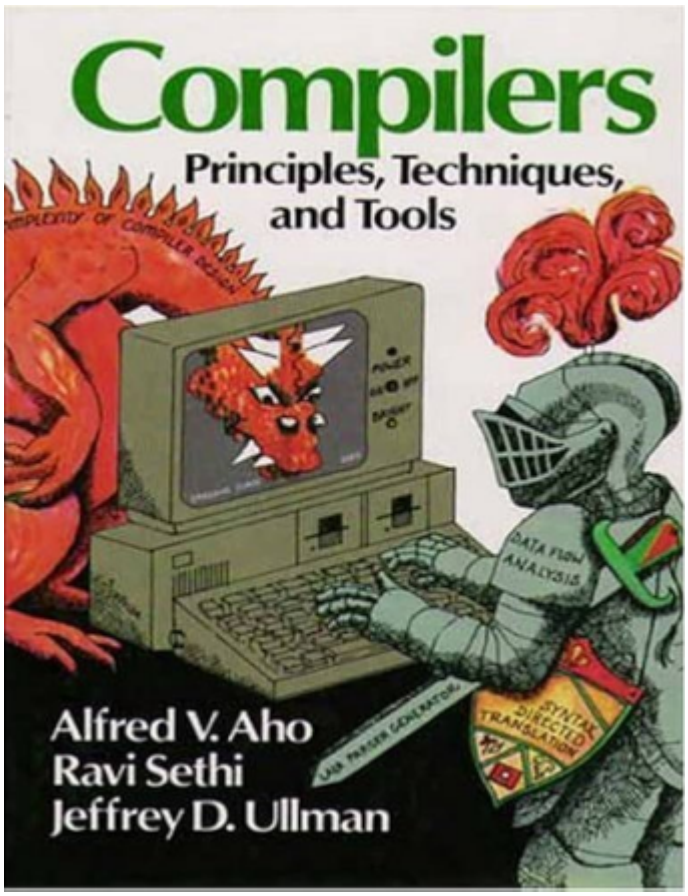
- Единая структура данных Fir для представления кода и семантики
- Возможность добавления плагинов компилятора
- Упрощение поддержки новых языковых возможностей для разных target-платформ.
- Улучшение производительности компилятора и IDE (плагин Kotlin в IDE использует Frontend компилятора). По результатам замеров JB:
  - ускорение компиляции на 94%
  - ускорение фазы инициализации на 488%
  - ускорение фазы анализа на 376%

В этой статье мы рассмотрели как работают разные версии компилятора Kotlin. Непосредственно в эту тему редко углубляются разработчики. Я надеюсь, что материал позволил заглянуть вам под капот и узнать, что же там происходит, когда вы каждый день собираете приложения на работе.

Дополнительные материалы

- Видео Crash Course on the Kotlin Compiler by Amanda Hinchman-Dominguez

- Видео KotlinConf 2018 - Writing Your First Kotlin Compiler Plugin by Kevin Most
- Видео What Everyone Must Know About The NEW Kotlin K2 Compiler
- Видео A Hitchhiker's Guide to Compose Compiler: Composers, Compiler Plugins, and Snapshots
- Репозиторий Kotlin-Compiler-Crash-Course с заметками об отладке, тестировании и настройке компилятора Kotlin
- Цикл статей Crash course on the Kotlin compiler
  - K1 + K2 Frontends, Backends
  - 1. Frontend: Parsing phase
  - 2. Frontend: Resolution phase
- Книга “Compilers: Principles, Techniques, and Tools” by Alfred Aho, Jeffrey Ullman, Ravi Sethi, Monica Lam



Другие наши статьи по Android разработке:

- Developer Keynote Google I/O 2024: официальная поддержка KMP, развитие Gemini и AI в Андроиде
- 4 сценария, когда нужно сделать ставку на Kotlin Multiplatform, а не Flutter
- Kotlin Multiplatform перешёл в stable. Что это значит?
- Кот в мешке: мастерство обработки ошибок внешних ключей SQLite

Теги: компиляторы, kotlin, k1, k2, релиз kotlin, kotlin 2.0, android, kmp

Хабы: Блог компании KTS, Разработка мобильных приложений, Разработка под Android, Компиляторы, Kotlin

+75 79 16 +16

KTS

Создаем цифровые продукты для бизнеса

Подписаться

Сайт

47 Карма 0.1 Рейтинг

Мялкин Максим @MAX1993M

Мобильный разработчик

Подписаться

Комментарии 16

Panzerschrek 9 мая 2024 в 08:40

А почему LLVM не используется? Не очень то подходит для Kotlin? Или есть другие причины?

+1 Ответить

Devchik 9 мая 2024 в 12:12

Kotlin/Native включает в себя серверную часть на основе LLVM для компилятора Kotlin.

+3 Ответить

equeim 9 мая 2024 в 17:17

Вероятно потому что LLVM не очень подходит для компиляции в Java байткод. Бэкэнд для компиляции в нативный код использует LLVM.

+4 Ответить

miksmiks 9 мая 2024 в 12:35

Ничего особенного с точки зрения компиляторов. Одна из возможных архитектур в развитии. Показательна ссылка на Dragon book как одну из базовых и уже достаточно устаревших.

0 Ответить

LionZXY 9 мая 2024 в 14:18

А расскажете какие сейчас актуальные тренды в компиляторостроении? Где об этом можно почитать?

+10 Ответить

miksmiks 9 мая 2024 в 14:42

Нет, я не эксперт здесь. Я понимаю, что это ирония. Я читаю классический курс по компиляторам - ничего особенного.

Но действительно, архитектура K1 - совершенно классическая, архитектура K2, где синтаксическое дерево затем украшается семантикой, - тоже классическая. Выгода здесь понятно в рефакторинге. И два бонуса. Основной - это масштабирование на несколько целевых архитектур. Я думаю, ради этого всё затевалось. Второй - ускорение компиляции - это бонусом. Повторюсь, не случайно в списке литературы абсолютно стандартная и древняя Dragon book.

Из альтернатив, тоже довольно стандартных, является построение по синтаксису отдельного семантического дерева. Здесь есть свои преимущества и недостатки, но их никто толком не анализировал.

Про сахарные преобразования в статье сказано скупо, что этот этап есть. А это самое интернюесное, потому что именно здесь могут быть конфликты - некоторые сахарные преобразования можно выполнять только до других, а иные рекурсивно зависят от других, и нет регулярной литературы, описывающей решения. Наконец, понятно, что в сахарных преобразованиях участвует семантика, но проблема в том, что семантика может быть на момент сахарных преобразований еще не до конца выведена. Вот об этом было бы интересно послушать. Об этом не пишут в учебниках и не сравнивают с другими реализациями.

+6 Ответить

MAX1993M 10 мая 2024 в 02:24

> Выгода здесь понятно в рефакторинге. И два бонуса. Основной - это масштабирование на несколько целевых архитектур. Я думаю, ради этого всё затевалось. Второй - ускорение компиляции - это бонусом.

Языку уже 13 лет, цели изменились, много нового добавилось.

Из-за появления новых целевых архитектур замедлялась разработка фичей языка. Поэтому JB в последних релизах и в том числе в 2.0 не выпускают ничего нового. Но в будущем новый компилятор открывает дорогу.

Я бы не сказал что была основная цель, а что-то далось бонусом. Тк ускорение фронтенда и масштабирование на разные архитектуры происходит в разных частях компилятора. Поэтому логично предположить что работа над этими улучшениями была разделена.

> Ничего особенного с точки зрения компиляторов. Одна из возможных архитектур в развитии.

Чем плоха классическая архитектура, если она решает нужные задачи и уже опробована в действии много раз. Kotlin не экспериментальный язык, чтобы испытывать что-то особенное в компиляторе)

> Из альтернатив, тоже довольно стандартных, является построение по синтаксису отдельного семантического дерева. Здесь есть свои преимущества и недостатки, но их никто толком не анализировал.

Тк компилятор переписывался то я думаю что в JB анализировали разные подходы.

Наличие нескольких структур данных в предыдущей версии компилятора ухудшала JIT-оптимизацию и кеширование процессора. Видится, что в варианте с наличием нескольких деревьев может быть аналогичная проблема.

Но, конечно, интересно, какие альтернативы JB рассматривали. Я эту информацию в открытых источниках не нашел.

> Про сахарные преобразования в статье сказано скупо, что этот этап есть. А это самое интернюесное, потому что именно здесь могут быть конфликты - некоторые сахарные преобразования можно выполнять только до других, а иные рекурсивно зависят от других, и нет регулярной литературы, описывающей решения. Наконец, понятно, что в сахарных преобразованиях участвует семантика, но проблема в том, что семантика может быть на момент сахарных преобразований еще не до конца выведена.

Так это часть алгоритма компилятора. Благо компилятор открытый. Статья и так получилась нагруженной и не было задачи разобрать все нюансы работы компилятора.

+3 Ответить

equeim 9 мая 2024 в 16:55

94% - маркетинговая цифра из одного бенчмарка. В реальности сокращение времени компиляции не более 10-15%.

+2 Ответить

MAX1993M 10 мая 2024 в 00:51

94% это ускорение по чистому билду. Помимо этого в статье с замерами есть и другие сценарии.

Замеры проведены на реальных развивающихся опенсорсных проектах: Anki-Android, Exposed. Понятно, что эти проекты не эталонные, и будет интересно увидеть цифры от крупных компаний в будущем.

+2 Ответить

rombell 9 мая 2024 в 20:13

Ускорение на 94% - это ускорение в 16 раз! Кто-то сильно *приукрашивает*

-1 Ответить

MAX1993M 10 мая 2024 в 00:41

Вы неправильно трактуете это число. Как трактовать эти числа можете почитать в статье.

Также в этой статье привожу пример на официальные замеры JB. Там явно видно числа: 94% это не в 16 раз ускорение, у чуть меньше чем в 2.

+5 Ответить

rombell 10 мая 2024 в 10:05

Извините, но как раз я *правильно* толкую. Счёт ведётся от старого значения. было 100%, осталось (100-94)=6%, ускорение 100/6=16 раз. Правильно было сказать ускорение на 45%, но на 94% звучит гораздо маркетологичнее

+2 Ответить

MAX1993M 12 мая 2024 в 01:20

К сожалению, вы невнимательно читаете. Речь идет не ***об уменьшении времени компиляции на 94%,*** а ***об увеличении производительности (скорости) компилятора на 94%.***

Таким образом:

- Раньше я мог выполнять работу X за время Y - **начальная производительность**.
- Теперь я могу выполнять работу X за время Y / 2 - **конечная производительность**.

Итого: конечная производительность в 2 раза больше начальной.

+2 Ответить

dkirienko 10 мая 2024 в 11:21

Пусть прибыль в прошлом году была миллион, а в этом году - 2 миллиона. На сколько увеличилась прибыль? Все согласны, что на 100%. К миллиону нужно прибавить 100% от миллиона, получится два миллиона. Никто не скажет, что прибыль увеличилась на 50%.
Если время сборки проекта составляло 10 секунд, а теперь составляет 5 секунд, на сколько оно уменьшилось? На 50%, нужно из 10 секунд вычесть 50% от 10 секунд, получится 5 секунд. А вовсе не на 100%, хотя по-видимому маркетологам JetBrains хотелось бы сделать из этих 50% целых 100%.

+2 Ответить

4p4 11 мая 2024 в 18:37

Все верно, только в вашем примере увеличение в два раза, а в статье уменьшение в два раза. Если бы прибыль упала до 550 миллионов, вы бы написали что она упала на 45% а не на 95%.

Ответить

MAX1993M 12 мая 2024 в 01:22

В статье говорится об ускорении компилятора на 94%, а не уменьшении времени компиляции на 94%. См объяснение выше.

Ответить

Вы можете оставлять комментарии только к свежим публикациям

Публикации

ЛУЧШИЕ ЗА СУТКИ ПОХОЖИЕ

Erwinmal 12 часов назад

Чапаев и Матрица: почему культура 90-х бунтовала против пластмассового мира? Часть 1

Простой 9 мин 3.6K

Ретроспектива

+37 23 15 +15

Lunathecat 8 часов назад

Электрогитара по доступной цене, не нуждающаяся в доработках

Простой 7 мин 4.1K

Обзор

+23 7 4 +4

DavidAsatryan 10 часов назад

Agile умер: из-за своего сострадания к product- и project-менеджерам (с) Фридрих Ницше

Простой 8 мин 7.1K

Мнение

+22 48 8 +8

erbanovanastasia 13 часов назад

Индия продолжает экспансию на рынок полупроводников: чего ожидать в ближайшем будущем

4 мин 1.6K

+22 3 12 +12

tw0face 14 часов назад

Покажи свой стартап/пет-проект (Январь)

1 мин 1.4K

+22 16 30 +30

Giox_Nostr 13 часов назад

Классика научной фантастики: хронология

Простой 22 мин 4.4K

Ретроспектива

+17 64 30 +30

Parfenovlgor 12 часов назад

Опыт написания компилятора вручную

Средний 9 мин 3.2K

+16 39 8 +8

Lexx_Nimofff 12 часов назад

Игра, вдохновлённая UFO и Jagged Alliance: интервью с главным разработчиков «Спарты 2035»

Простой 9 мин 1K

Интервью

+12 3 2 +2

guselnikov 3 часа назад

То о чем многие молчат, или может не знают...

3 мин 1.7K

+11 8 9 +9

bodyawm 7 часов назад

Я купил легендарный игровой смартфон из утиля и отремонтировал его — смотрим на Nokia N-Gage Classic

10 мин 2.9K

Ретроспектива

+10 3 9 +9

Показать еще

ИНФОРМАЦИЯ

Сайт	kts.tech
Дата регистрации	10 марта 2021
Дата основания	9 ноября 2015
Численность	101–200 человек
Местоположение	Россия

ССЫЛКИ

Сайт
kts.tech

Школа Metaclass
metaclass.kts.studio

Услуги DevOps
devops.kts.tech

Вакансии
hh.ru

Tg канал: Как программисты делают бизнес
t.me

ВКОНТАКТЕ

ВИДЖЕТ

ВИДЖЕТ

ВИДЖЕТ

ВИДЖЕТ

ВИДЖЕТ

ВИДЖЕТ

ВИДЖЕТ

БЛОГ НА ХАБРЕ

17 янв в 16:28

Дополненная реальность в Web: какие библиотеки актуальны в 2025?

1.8K 2 +2

30 дек 2024 в 10:26

Ory Kratos — конструктор для сборки цифрового продукта любой сложности

6K 7 +7

25 дек 2024 в 15:20

Firezone, или как спрятать свою инфраструктуру от посторонних глаз

5.3K 3 +3

19 дек 2024 в 19:36

New Year DevOps Challenge: подводим итоги и делимся решением

1.9K 0

12 дек 2024 в 12:05

DevOps Challenge: помогите Деду Морозу с оповещениями и получите новогодний мерч

2.7K 0