От Kotlin до пикселей

Compiler, Compose, Concurrency

ДВФУ · ИМКТ · Департамент МиКМ · Android-разработка · 2025

Agenda

Compiler

Composable Runtime

Concurrency

Что мы уже знаем?

- composable функции
- state (remember, ViewModel)
- navigation и back stack

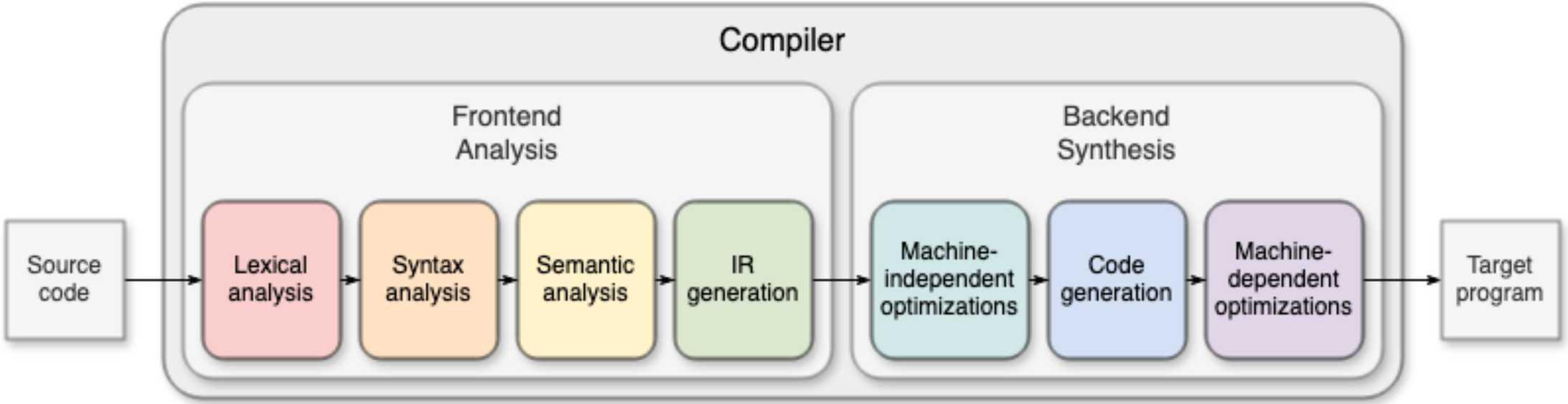
Compiler

Из чего состоит компилятор?

Из чего состоит компилятор?

- Source code → … → Target code
- Frontend: “понимает” код
- Backend: “выпускает” код

Compiler Design



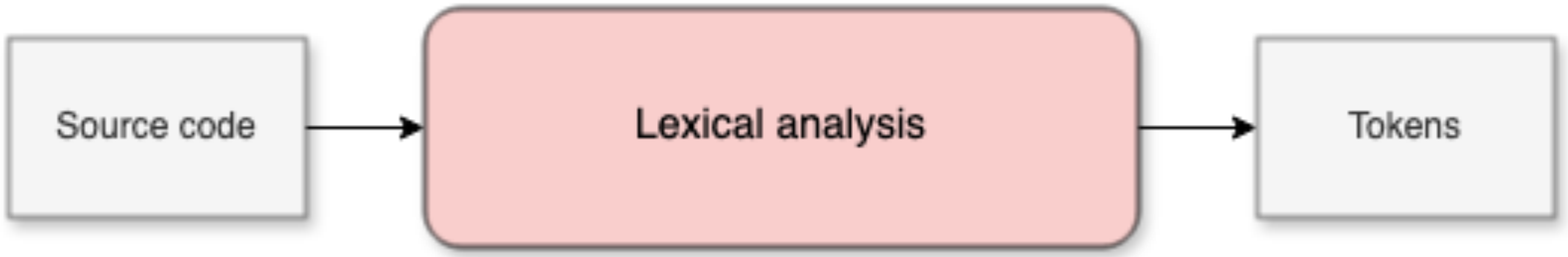
https://antimonit.github.io/2024/08/15/compiler-design.html

Фазы фронтенда

- Lexing (сканер) -> токены
- Parsing -> дерево синтаксиса (AST)
- Semantic analysis -> “смысл” (типы/разрешение имён)

Lexer (лексер): из текста в токены

- Вход: строка символов
- Выход: поток токенов (IDENT, NUMBER, PLUS, IF…)
- Лексер игнорирует “смысл”, только форма



https://antimonit.github.io/2024/08/15/compiler-design.html

Lexer (лексер): из текста в токены

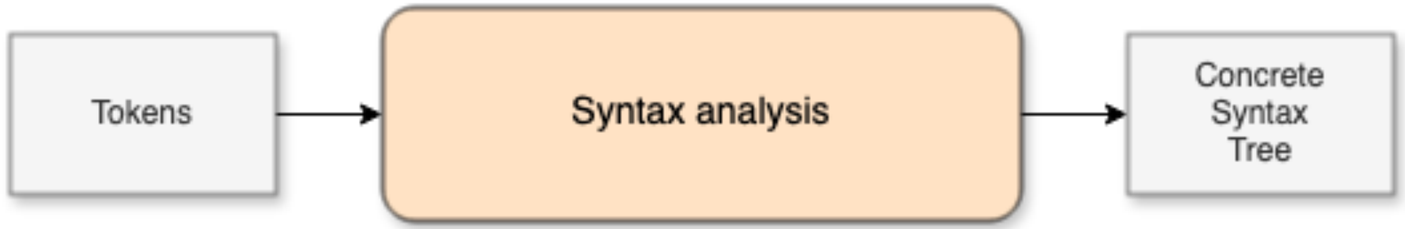
```
package sample.hello

fun hello(user: String) = "Hello, $user"
```

```
PACKAGE("package")
WS(" ")
Identifier("sample")
DOT(".")
Identifier("hello")
NL("\n")
NL("\n")
FUN("fun")
WS(" ")
Identifier("hello")
LPAREN("(")
Identifier("user")
COLON(":")
Inside_WS(" ")
Identifier("String")
RPAREN(")")
WS(" ")
ASSIGNMENT("=")
WS(" ")
QUOTE_OPEN(""")
LineStrText("Hello, ")
LineStrRef("$user")
QUOTE_CLOSE(""")
```

https://antimonit.github.io/2024/08/15/compiler-design.html

Parser (парсер): токены -> дерево

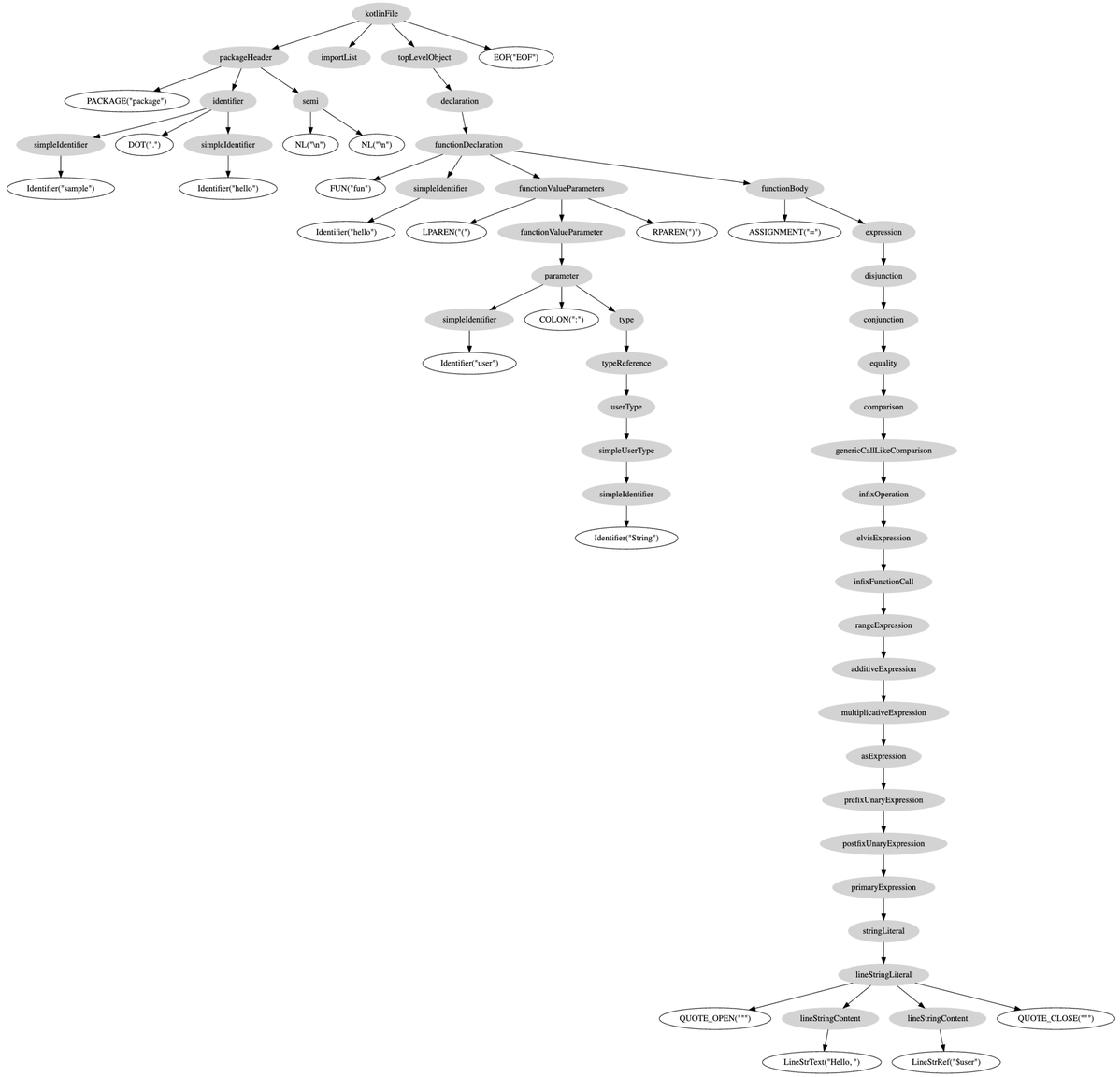
- Вход: токены
- Выход: дерево разбора / AST
- Опирается на грамматику языка (правила)



https://antimonit.github.io/2024/08/15/compiler-design.html

Parser (парсер): токены -> дерево

```
package sample.hello

fun hello(user: String) = "Hello, $user"
```



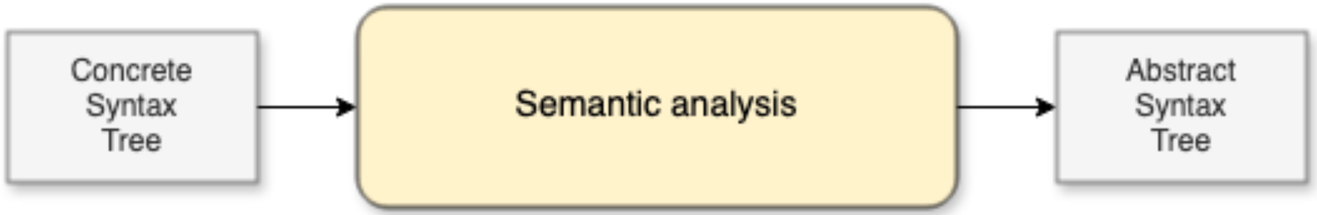
https://antimonit.github.io/2024/08/15/compiler-design.html

AST vs “смысл”

- AST: синтаксис (“как написано”)
- Semantic model: смысл (“что это за сущности”)
  - переменные, функции, типы, области видимости

Semantic analysis: таблица символов и типы

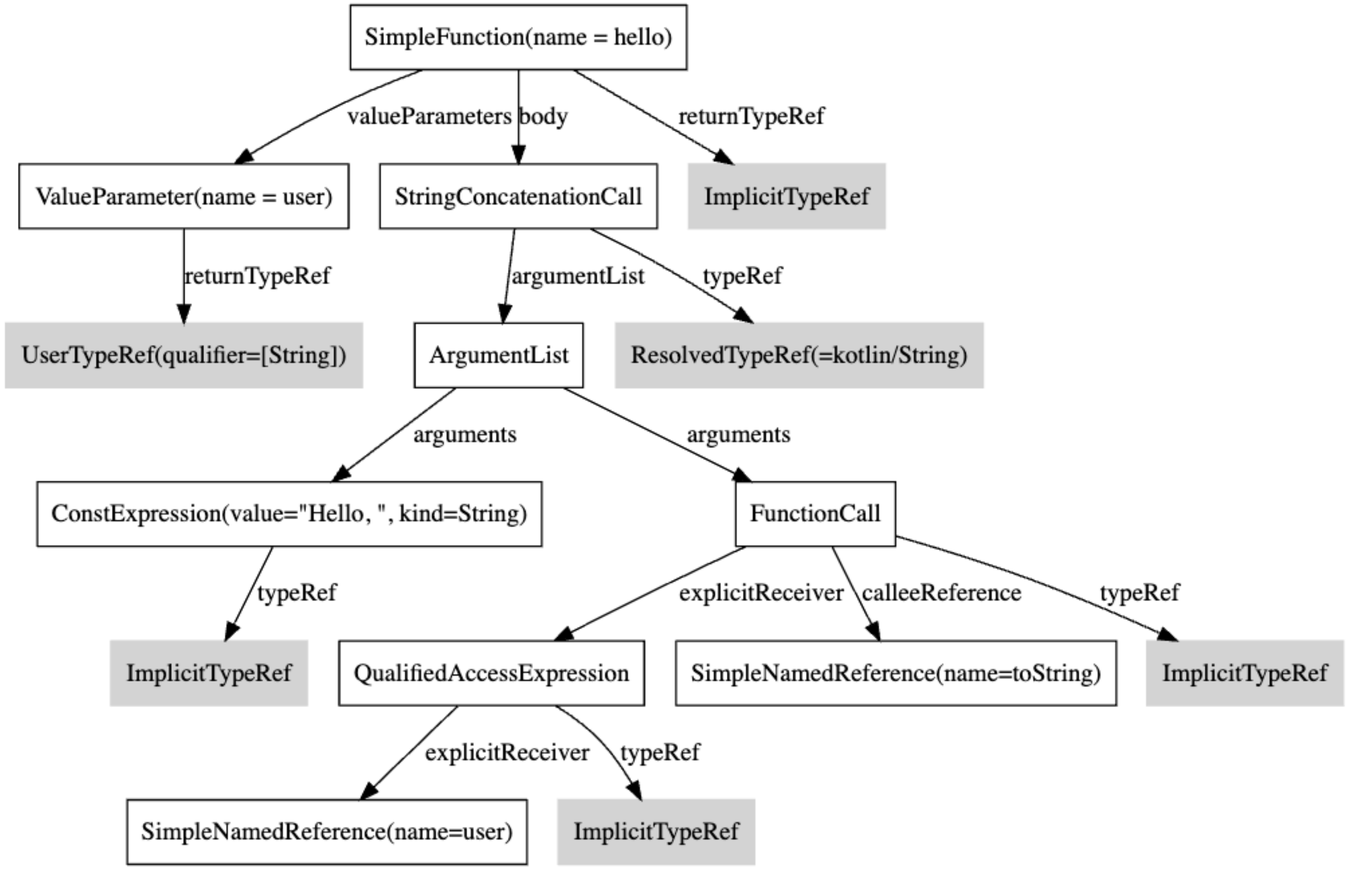
- Symbol table: “кто где объявлен”
- Name resolution: “к какой foo() мы обращаемся”
- Type inference/check: “какой тип у выражения”
- Ошибки компиляции чаще всего здесь



https://antimonit.github.io/2024/08/15/compiler-design.html

Semantic analysis: таблица символов и типы

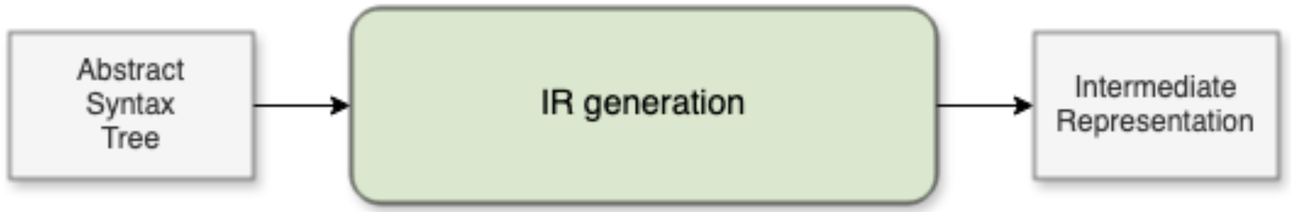
```
fun hello(user: String) = "Hello, $user"
```



https://antimonit.github.io/2024/08/15/compiler-design.html

IR: промежуточное представление

- IR = “универсальная форма программы”
- Удобно:
  - оптимизировать
  - **трансформировать**
  - генерировать код под разные платформы



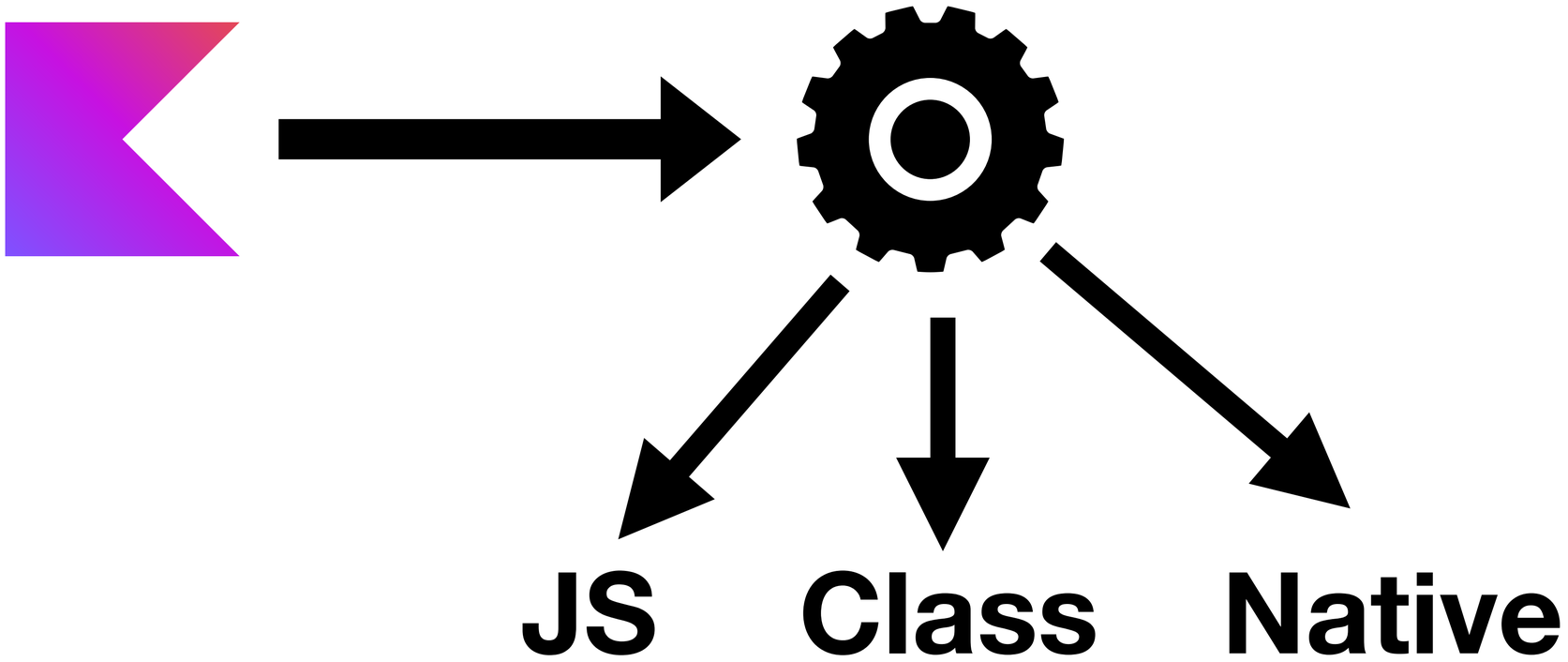
Lowering/Optimizations/Deshugaring

- Lowering: превращаем “сложное” в “простое”
- Оптимизации:
  - убрать мёртвый код
  - инлайн
  - упрощение выражений
-

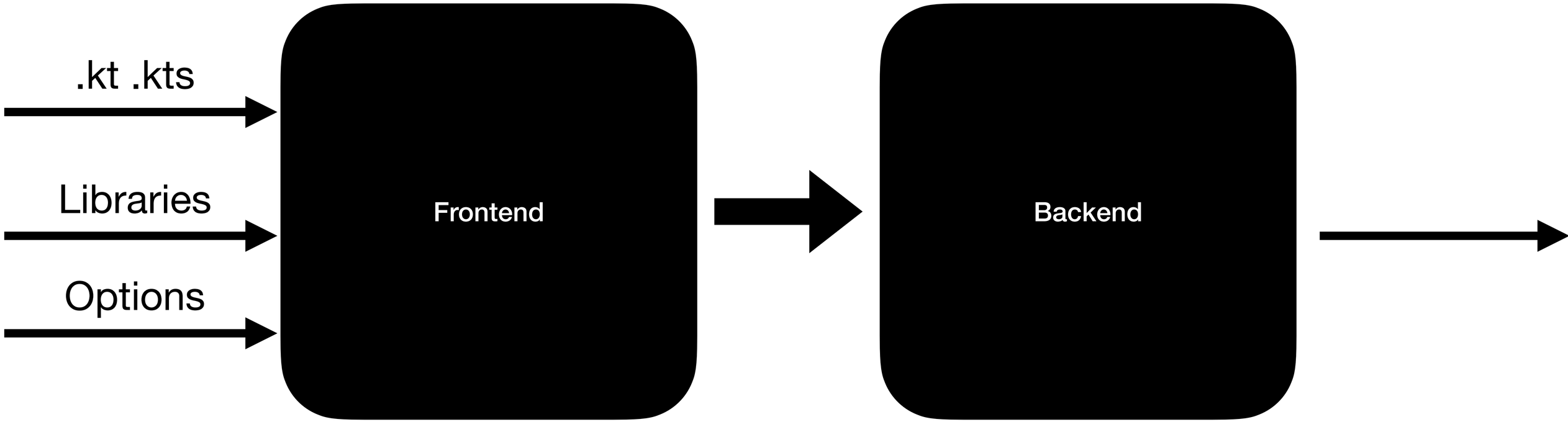
Backend: генерация кода

- IR -> target:
  - JVM bytecode
  - JS
  - Native
  - WASM

Не только Android

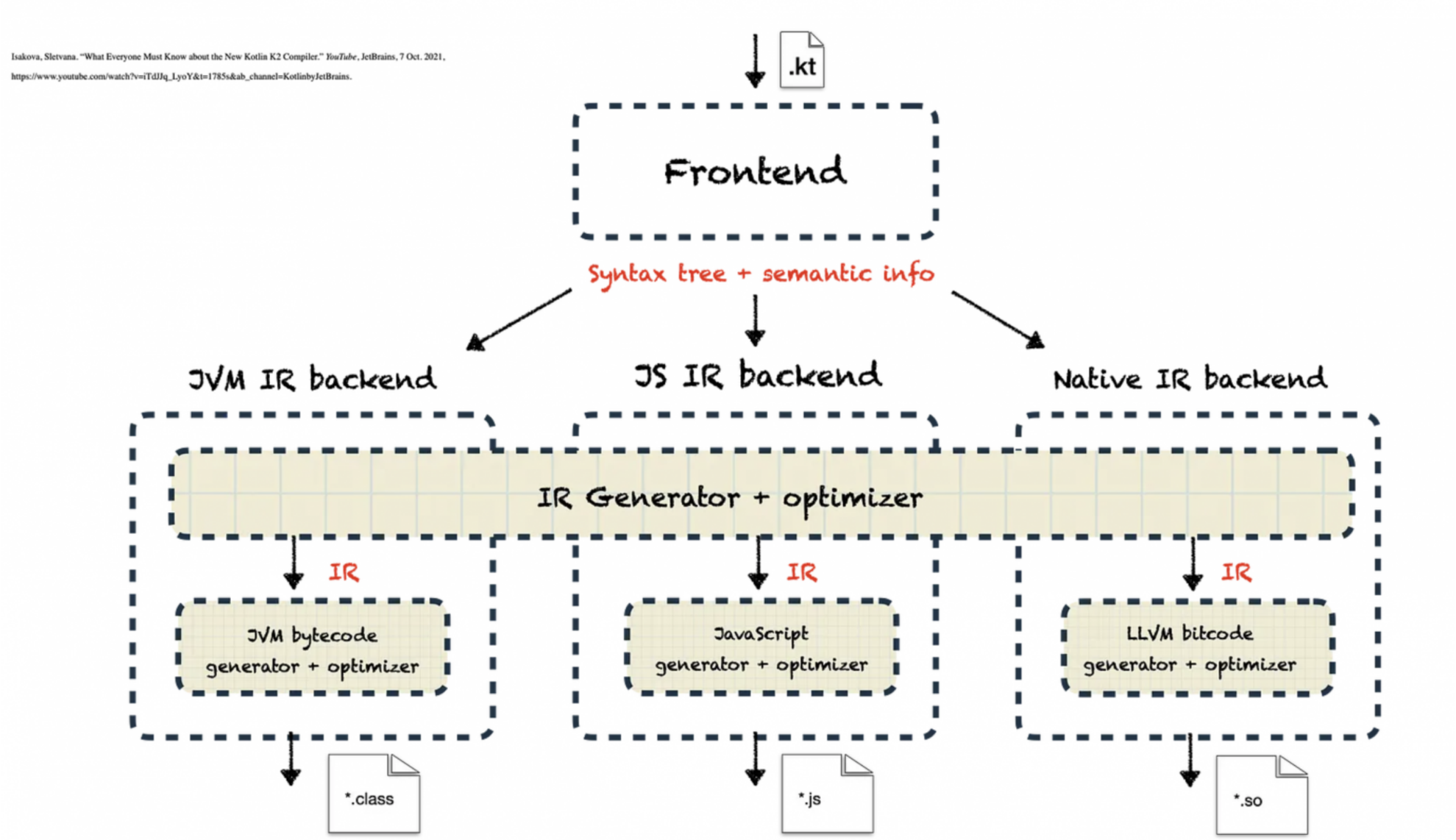


Kotlin Compiler



Compile

Isakova, Sletvana. "What Everyone Must Know about the New Kotlin K2 Compiler." *YouTube*, JetBrains, 7 Oct. 2021, https://www.youtube.com/watch?v=iTdJJq_LyoY&t=1785s&ab_channel=KotlinbyJetBrains.



Compiler Plugins

- Добавлять проверки (линт на этапе компиляции)
- Генерировать код/метаданные
- Трансформировать программу (метапрограммирование)

https://github.com/JetBrains/kotlin/blob/master/docs/fir/fir-plugins.md

https://resources.jetbrains.com/storage/products/kotlinconf2018/slides/5_Writing%20Your%20First%20Kotlin%20Compiler%20Plugin.pdf

Compiler Plugins - точка встраивания

- Frontend extensions (на уровне анализа/символов)
- IR transformations (на уровне промежуточного представления)
- Codegen hooks (в генерации)

https://github.com/JetBrains/kotlin/blob/master/docs/fir/fir-plugins.md
https://github.com/JetBrains/kotlin/blob/master/docs/fir/fir-basics.md

Compose compiler plugin

- трансформирует @Composable
- Компилятор “переписывает” composable-функции
- Встраивает служебные вызовы для runtime
- Runtime потом управляет composition/recomposition

https://android.googlesource.com/platform/frameworks/support/%2B/HEAD/compose/runtime/design/how-compose-works.md

Compose

Compose - из чего состоит?

Composable Runtime

2D library (Skia)

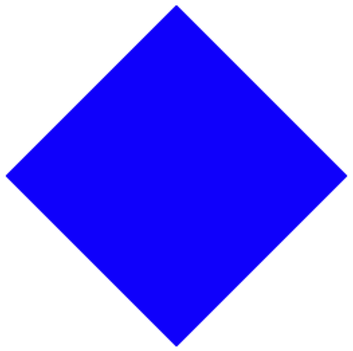
Compiler Plugin

UI Elements Library (Material)

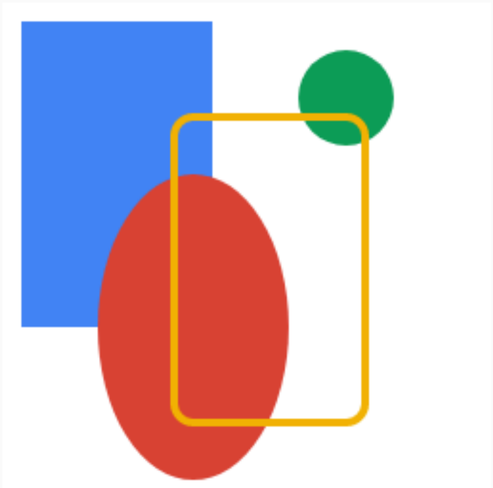
Skia

```
1  void draw(SkCanvas* canvas) {
2      canvas->save();
3      canvas->translate(SkIntToScalar(128), SkIntToScalar(128));
4      canvas->rotate(SkIntToScalar(45));
5      SkRect rect = SkRect::MakeXYWH(-90.5f, -90.5f, 181.0f, 181.0f);
6      SkPaint paint;
7      paint.setColor(SK_ColorBLUE);
8      canvas->drawRect(rect, paint);
9      canvas->restore();
10 }
```

RUN Pop-out



Shapes



Bézier Curves

Translations and Rotations

Text Rendering

Compose - это не только UI-компоненты. Это связка: компилятор переписывает функции, рантайм управляет recomposition, рисование идёт через графический пайплайн, а Material - всего лишь библиотека поверх всего этого.

Pipeline

@Composable code

| (compiler plugin)

transformed code + runtime calls

| (runtime: composition/recomposition)

UI tree (layout nodes)

| (layout + drawing)

Canvas draw ops

| (Android graphics / Skia)

pixels on screen

Compiler Plugin Role

- @Composable - это не “обычная аннотация”
- Нужна автоматическая генерация “служебной логики”
- Это делает компилятор, а не библиотека

Compiler Plugin Role

“transform @Composable”

- Компилятор:
  - добавляет служебные параметры/вызовы
  - ставит “метки” для отслеживания изменений
  - подготавливает код к selective recomposition

```
fun A($composer: Composer) {
  $composer.startGroup(1234)
  val data = remember($composer) { Data() }
  …
  $composer.endGroup()
}

@Composable
fun B($composer: Composer) {
  $composer.startGroup(4567)
  A($composer)
  A($composer)
  $composer.endGroup()
}
```

Compose Runtime Role

- Выполняет composable-функции (composition)
- Следит за state и запускает recomposition
- Хранит “память UI” (remember)
- Выпускает изменения в UI tree

Compose Runtime Phases

- Composition - что показывать
- Layout - где размещать
- Drawing - как рисовать (в Canvas)

https://developer.android.com/develop/ui/compose/phases?hl=ru

runtime знает “что перерисовать”

- Composable читает state
- Runtime запоминает “кто читал”
- При изменении state -> помечает зависимые части “dirty”

https://developer.android.com/develop/ui/compose/phases?hl=ru

Threads

Process vs Thread

Процесс vs Поток

- Process: изолированное “пространство” приложения (память, ресурсы)
- Thread: “дорожка выполнения” внутри процесса
- У одного процесса может быть много потоков

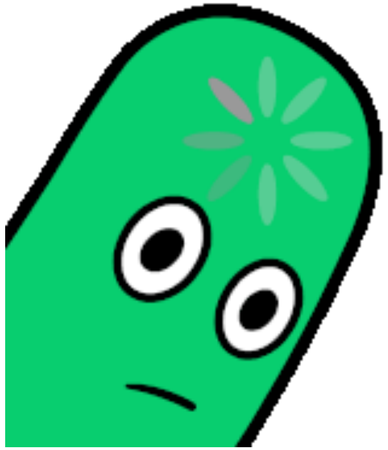
Поток (thread): простое определение

- отдельная последовательность выполнения инструкций
- со своим стеком вызовов
- но общей памятью процесса (heap)

Главный/UI-поток (Main thread / UI thread)

- По умолчанию компоненты приложения стартуют в одном процессе и в основном потоке Android Developers
- UI-поток отвечает за:
  - обработку ввода
  - отрисовку UI
  - Если он занят…

Главный/UI-поток (Main thread / UI thread)



“Правильная” картинка приложения

- Main/UI thread:
  - ввод
  - отрисовка
  - короткие операции
- Background:
  - тяжёлая работа в thread pool

UI thread (EDT) — это выделенный поток, который последовательно (one-by-one) снимает UI-задачи/события из очереди и диспетчеризует их в обработчики; чтобы сохранить потокобезопасность через thread confinement, все UI-объекты должны читаться/меняться только из этого потока.

Java Concurrency in Practice (Goetz et al.)

UI-поток (main thread) - это один выделенный поток приложения, который по очереди обрабатывает события интерфейса (тапы, жизненный цикл, перерисовку) из очереди сообщений (Looper/MessageQueue).
Поэтому любой доступ к UI (создать/изменить View или Compose-state, который влияет на UI) должен происходить только в этом потоке; иначе получишь гонки/краши/“странные” баги.

Main thread = очередь событий -> обработчики -> рисование.

Ui-поток есть во всех GUI системах т.к. Отрисовка интерфейса должна происходить согласованно в любой момент времени и распараллелить это невозможно

А зачем он нужен?

Async and await..?

Синхронно vs асинхронно

- Синхронно: вызываю функцию -> жду результат -> продолжаю
- Асинхронно: вызываю -> не жду -> результат придёт позже

Асинхронность - это про время: результат появляется не сейчас.
Многопоточность - это про параллельность: выполнение может идти на другом потоке. Асинхронность может быть и без потоков (через event loop), но в Android часто сочетается с потоками.

Зачем асинхронность в Android

- UI-поток должен быть свободен
- Долгое -> асинхронно
- UI обновляем после завершения

Callback: базовая форма асинхронности

- Callback = функция, которую передаём как параметр
- Её вызывают когда операция завершилась
- Часто два колбэка:
  - onSuccess(result)
  - onError(error)

Типичный API с колбэками

- “callback hell” при цепочках
- обработка ошибок размазывается
- отмена/таймаут - сложно

```
fun loadSomething(
  onSuccess: (Data) -> Unit,
  onError: (Throwable) -> Unit
)
```

Callback Hell

```
function hell(win) {
 // for listener purpose
 return function() {
   loadLink(win, REMOTE_SRC+'/assets/css/style.css', function() {
     loadLink(win, REMOTE_SRC+'/lib/async.js', function() {
       loadLink(win, REMOTE_SRC+'/lib/easyXDM.js', function() {
         loadLink(win, REMOTE_SRC+'/lib/json2.js', function() {
           loadLink(win, REMOTE_SRC+'/lib/underscode.min.js', function() {
             loadLink(win, REMOTE_SRC+'/lib/backbone.min.js', function() {
               loadLink(win, REMOTE_SRC+'/dev/base_dev.js', function() {
                 loadLink(win, REMOTE_SRC+'/assets/js/deps.js', function() {
                   loadLink(win, REMOTE_SRC+'/src/' + win.loader_path + '/loader.js', function() {
                     async.eachSeries(SCRIPTS, function(src, callback) {
                       loadScript(win, BASE_URL+src, callback);
                     });
                   });
                 });
               });
             });
           });
         });
       });
     });
   });
 };
}
```