Compiler plugins in Kotlin

Identify yourself as a **System Programmer**

Andrey Kuleshov

LOMONOSOV MOSCOW STATE UNIVERSITY

intel

Deutsche Bank

HUAWEI

https://github.com/**akuleshov7**

saveourtool/diktat Public

Strict coding standard for Kotlin and a custom set of rules for detecting code smells, code style issues and bugs

Kotlin 490 37

ktoml Public

Kotlin Multiplatform parser and compile-time serializer/deserializer for TOML format (Native, JS, JVM) based on KxS

Kotlin 418 22

Motivation of this talk

01

Overview

Brief overview of high-level Compiler design and IR understanding

02

Other languages and getting familiar with frameworks

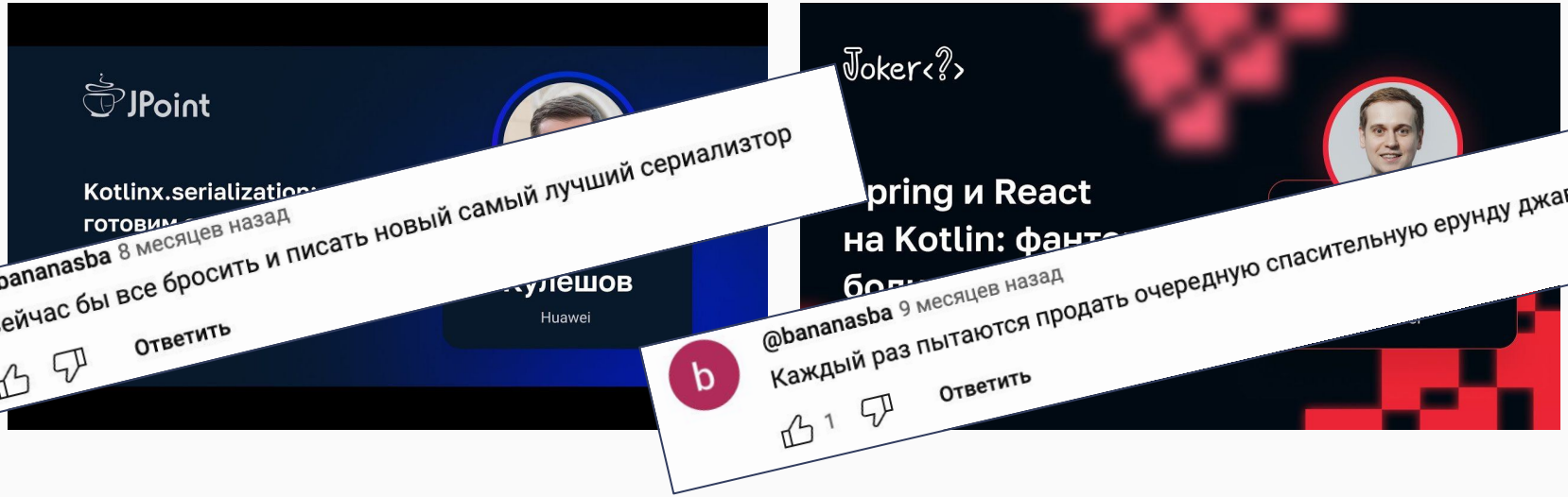
Historical Overview of Compiler Plugin Design Concepts in Multiple Programming Languages

03

Kotlin

Evolution of Plugins concept in Kotlin

Motivation of this talk



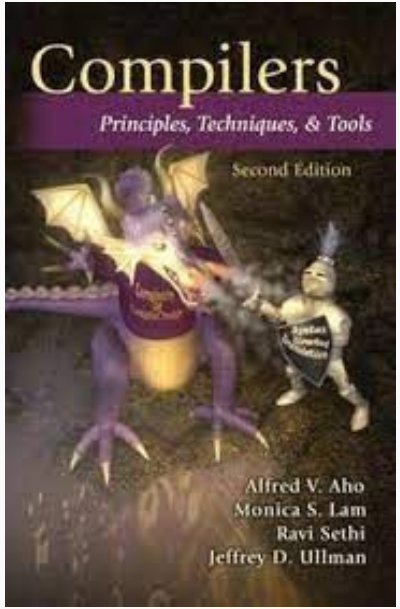
01.

Introduction

All you need to know today about compilers

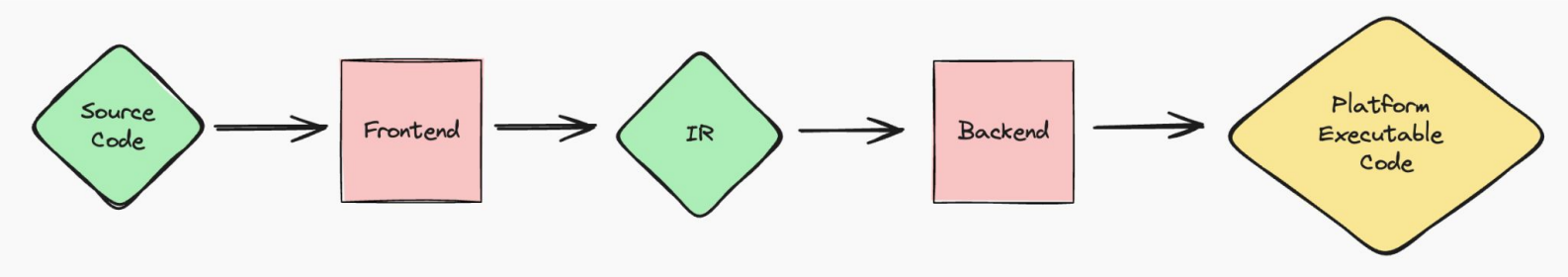
Start from the “Dragon Book”

Aho, Sethi, Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley, 1986. ISBN 0-201-10088-6

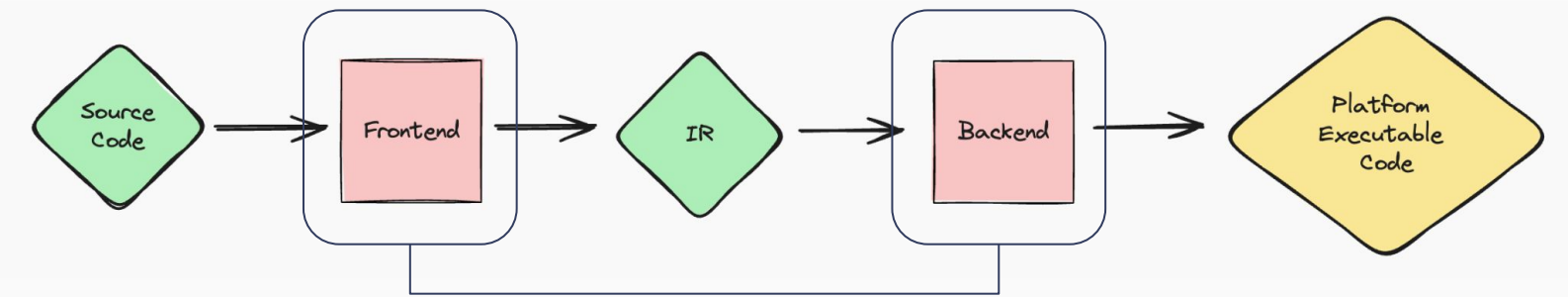



son you know...
once you start there's no going back

As simple as possible



As simple as possible



passes

Quiz for fellow kids!

Match the error type on the left to its problem on the right

Lexical error

Syntactic error

Semantic error

```
fun func() {
    return listOf("ERROR")
}
```

```
fun func() {
    glistOf("ERROR")
}
```

```
fun func() {
    listOf("ERROR").forEach {
        println(it)
}
```

Quiz for fellow kids!

Match the error type on the left to its problem on the right

Lexical error

Syntactic error

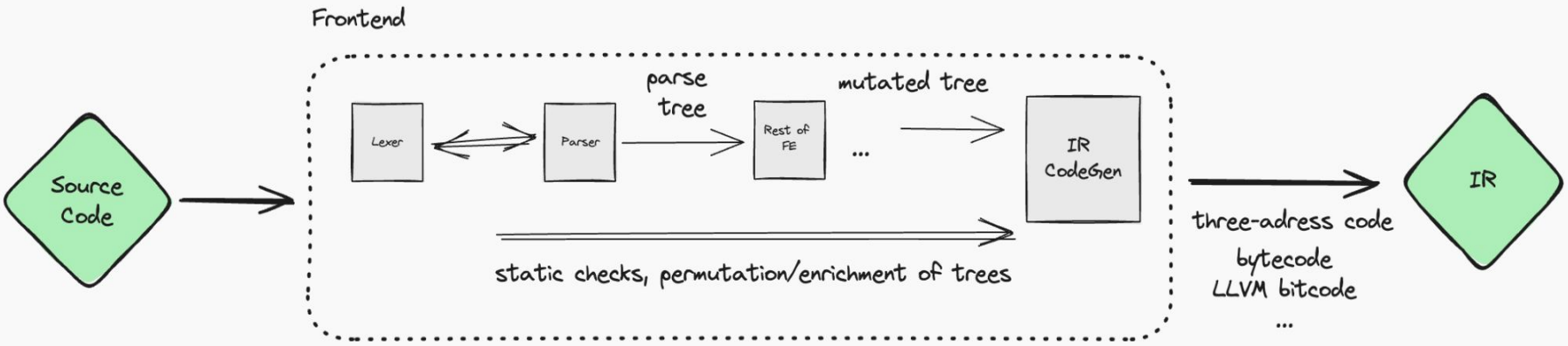
Semantic error

```
fun func() {
    return listOf("ERROR")
}
```

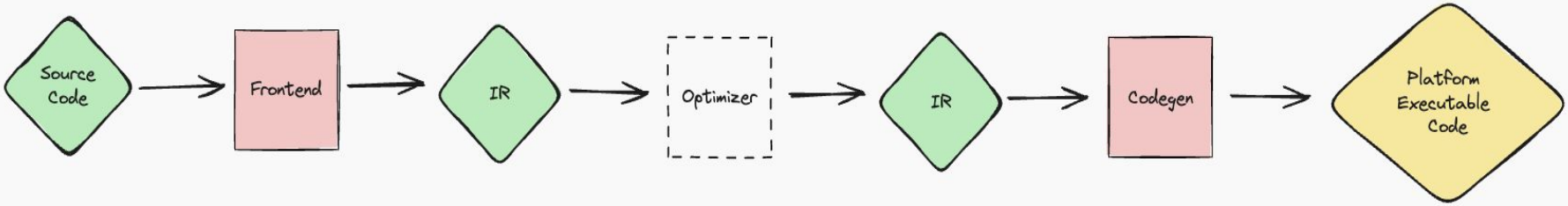
```
fun func() {
    glistOf("ERROR")
}
```

```
fun func() {
    listOf("ERROR").forEach {
        println(it)
}
```

Frontend


Frontend
Source Code
Lexer
Parser
parse tree
Rest of FE
...
mutated tree
IR CodeGen
static checks, permutation/enrichment of trees
three-adress code
bytecode
LLVM bitcode
...
IR

Backend

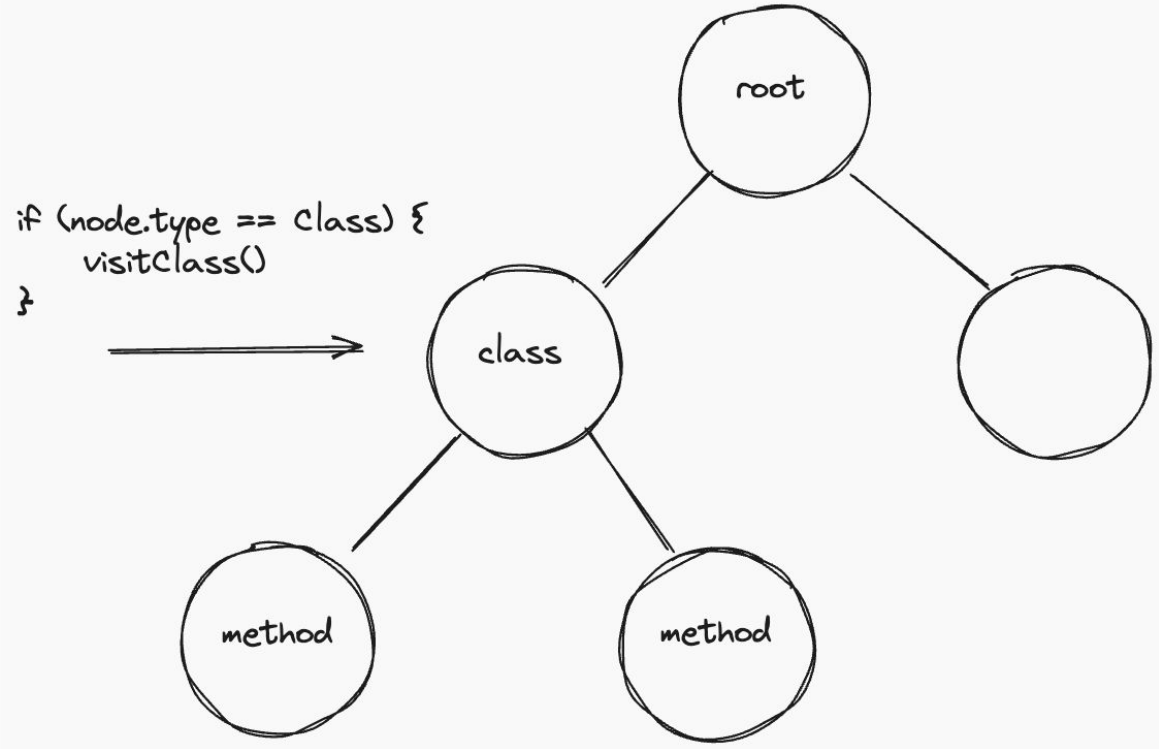

Source Code
Frontend
IR
Optimizer
IR
Codegen
Platform Executable Code

02.

Old-fashioned

Existing approaches in main programming languages

How usually plugin frameworks work



C/C++

C/C++

Clang

A plugin is loaded from a dynamic library at runtime by the compiler.

```
class PrintFunctionsConsumer : public ASTConsumer {
	bool HandleTopLevelDecl(DeclGroupRef DG) override {
		// Consumer with logic for node checks or modification
	}
}
```

https://github.com/llvm/llvm-project/blob/main/clang/examples/PrintFunctionNames/PrintFunctionNames.cpp

C/C++

Clang

A plugin is loaded from a dynamic library at runtime by the compiler.

```
class PrintFunctionsConsumer : public ASTConsumer {
            ...
}

class PrintFunctionNamesAction : public PluginASTAction {
        <...>CreateASTConsumer(CompilerInstance &CI, llvm::StringRef) override {
          //
          return std::make_unique<PrintFunctionsConsumer>(CI, ParsedTemplates);
        }



}
```

https://github.com/llvm/llvm-project/blob/main/clang/examples/PrintFunctionNames/PrintFunctionNames.cpp

C/C++

Clang

A plugin is loaded from a dynamic library at runtime by the compiler.

```
class PrintFunctionsConsumer : public ASTConsumer {
            ...
}

class PrintFunctionNamesAction : public PluginASTAction {
        <...>CreateASTConsumer(CompilerInstance &CI, llvm::StringRef) override {
          return std::make_unique<PrintFunctionsConsumer>(CI, ParsedTemplates);
        }

      bool ParseArgs(const CompilerInstance &CI, const std::vector<std::string> &args) override {
            // possibility to add logic for cli arguments
      }
}
```

https://github.com/llvm/llvm-project/blob/main/clang/examples/PrintFunctionNames/PrintFunctionNames.cpp

C/C++

Clang

A plugin is loaded from a dynamic library at runtime by the compiler.

```
class PrintFunctionsConsumer : public ASTConsumer {
            ...
}

class PrintFunctionNamesAction : public PluginASTAction {
        <...>CreateASTConsumer(CompilerInstance &CI, llvm::StringRef) override {
          return std::make_unique<PrintFunctionsConsumer>(CI, ParsedTemplates);
        }

       ...
}

//registring action in FrontendPluginRegistry
static FrontendPluginRegistry::Add<PrintFunctionNamesAction>X("print-fns", "print function names");
```

https://github.com/llvm/llvm-project/blob/main/clang/examples/PrintFunctionNames/PrintFunctionNames.cpp

C/C++

Clang

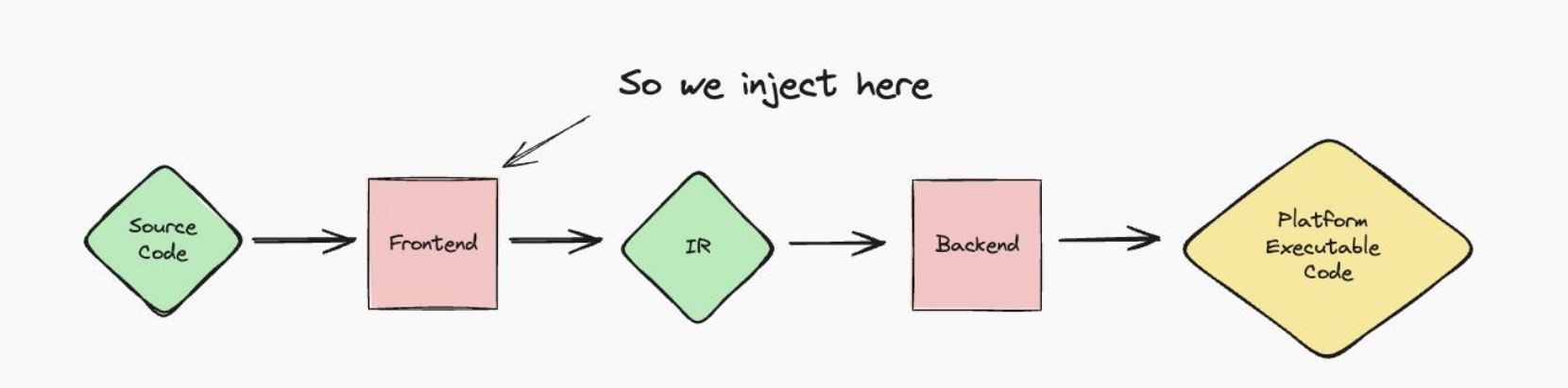
A plugin is loaded from a dynamic library at runtime by the compiler.

```
class PrintFunctionsConsumer : public ASTConsumer {
            ...
}

class PrintFunctionNamesAction : public PluginASTAction {
        <...>CreateASTConsumer(CompilerInstance &CI, llvm::StringRef) override {
          return std::make_unique<PrintFunctionsConsumer>(CI, ParsedTemplates);
        }

      ...
}

//registring action in FrontendPluginRegistry
static FrontendPluginRegistry::Add<PrintFunctionNamesAction>X("print-fns", "print function names");
```

```
cclang -cc1 -load <>/libPrintFunctionNames.so -plugin print-fns  --plugin-arg-print-fns --example-argument
```

https://github.com/llvm/llvm-project/blob/main/clang/examples/PrintFunctionNames/PrintFunctionNames.cpp

C/C++


So we inject here
Source Code
Frontend
IR
Backend
Platform Executable Code

Don't write a clang plugin

Contents

Make sure you really want to write a clang plugin.

- The clang plugin api is not stable. If you write a plugin, *you* are responsible for making sure it's updated when we update clang.
- If you're adding a generally useful warning, it should be added to upstream clang, not to a plugin.
- You should not use a clang plugin to do things that can be done in a PRESUBMIT check (e.g. checking that the headers in a file are sorted).

Valid reasons for writing a plugin are for example:

- You want to add a chromium-specific error message.
- You want to write an automatic code rewriter.

In both cases, please inform clang@chromium.org of your plans before you pursue them.

Java

Java

Javac plugins

(starting from Java 8 → but reworked in Java 9, 16. For example: tools.jar will not be so easy to find 😅)

```java
package com.akuleshov7;

// com.sun.source.util.*
import com.sun.source.util.JavacTask;
import com.sun.source.util.Plugin;

public class SampleJavacPlugin implements Plugin {
   @Override
   public String getName() {
       return "MyPlugin";
   }

   @Override
   public void init(JavacTask task, String... args) {
       System.out.println("Hello world");
   }
}
```

Java 8 Baeldung example:

https://github.com/eugenp/tutorials/blob/master/core-java-modules/core-java-sun/src/main/java/com/baeldung/javac/SampleJavacPlugin.java

Java

Javac plugins: ServiceLoader

```
resources
    META-INF.services
        com.sun.source.util.Plugin
```

```
com.akuleshov7.SampleJavacPlugin
```

```
$ javac
      -cp ./target/classes/com/akuleshov7
      -Xplugin:SampleJavacPlugin
      ./src/main/java/com/akuleshov7/Main.java
```

Java

Javac plugins: phases and life cycle.

TaskEvent.Kind per source file:

- **PARSE** -> AST
  - **ENTER** -> imports
    - **ANALYZE** -> analysis
      - **GENERATE** -> codegen

Java

Javac plugins: task listeners

```
@Override
public void init(JavacTask task, String... args) {
    task.addTaskListener(new TaskListener() {
        public void started(TaskEvent taskEvent) {
                ...
        }

        public void finished(TaskEvent taskEvent) {
                // taskEvent.getKind()
                ...
        }
    });
}
```

Java

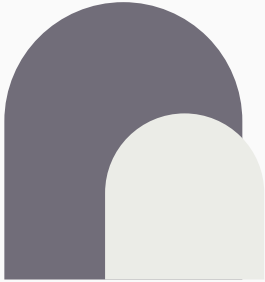
Javac plugins: phases and lifecycle

```
taskEvent.getCompilationUnit().accept(new TreeScanner<Void, Void>() {
    @Override
    public Void visitClass(ClassTree node, Void aVoid) {
        ...
    }

    @Override
    public Void visitMethod(MethodTree node, Void aVoid) {
        ...
    }
}, null);
```

+ You can modify AST

```
com.sun.tools.javac.tree.JCTree;
```

```
JCTree.JCBlock body = (JCTree.JCBlock) method.getBody();
body.stats = body.stats.prepend(...);
```



+ You can modify AST

```
com.sun.tools.javac.tree.JCTree;
```

```
JCTree.JCBlock body = (JCTree.JCBlock) method.getBody();
body.stats = body.stats.prepend(...);
```

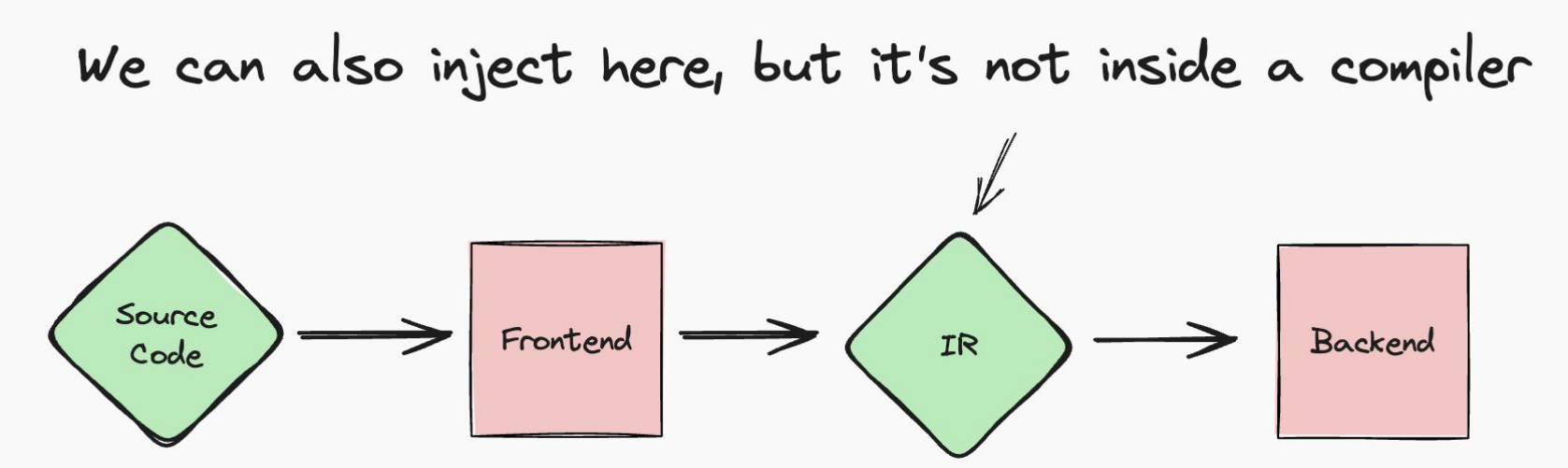
Which project first comes to mind when you think about such modification of AST?

But it’s not quite true

JSR 269: Pluggable Annotation Processing API

Compile-time code generation with annotations for creating of new data

Java


We can also inject here, but it's not inside a compiler
Source Code
Frontend
IR
Backend

Java

ASM

BSD **License**

Reading docs:

- the **OpenJDK**, to generate the lambda call sites
- the **Groovy** compiler and the **Kotlin** compiler
- **Cobertura** and **Jacoco**, to instrument classes
- **Gradle**, to generate some classes at runtime

Java

BSD **License**

Reading docs:

- the **OpenJDK**, to generate the lambda call sites
- the **Groovy** compiler and the **Kotlin** compiler
- **Cobertura** and **Jacoco**, to instrument classes
- **Gradle**, to generate some classes at runtime

```
public FieldVisitor visitField(String name, ...) {
    if (removeField(name)) {
        // Do nothing, in order to remove field
        return null;
    } else {
        // Keep it
        return super.visitField(name, ...);
    }
}
```

Java

Soot **SootUp**

LGPL

- Interprocedual **CallGraph** generation
- Inter-procedural **Data-flow** Analysis
- Any manipulations with **Jimple**

Jimple (3-address like)

```
public class HelloWorld extends java.lang.Object
{
  public void <init>()
  {
    HelloWorld r0;
    r0 := @this: HelloWorld;
    specialinvoke r0.<java.lang.Object: void <init>()>();
    return;
  }

  public static void main(java.lang.String[])
  {
    java.lang.String[] r0;
    java.io.PrintStream r1;

    r0 := @parameter0: java.lang.String[];
    r1 = <java.lang.System: java.io.PrintStream out>;
    virtualinvoke r1.<java.io.PrintStream:
    void println(java.lang.String)>("Hello world!");
    return;
  }
}
```

03.

Kotlin

Compiler intro

Kotlin is evolving to K2

Joker<?> Schedule Speakers Media Partners About Archive Experts Hosts More Become a speaker

If you have a ticket, log in to watch the video Login

TALK Kotlin 13.10 / 18:45 – 19:30 (UTC+8)

In the Meantime Kotlin: What's Been Happening During the past Year

RU Presentation pdf

The active development of Kotlin is pleasing us with its frequent releases, and the leitmotiv of the year becomes the work on the new compiler.

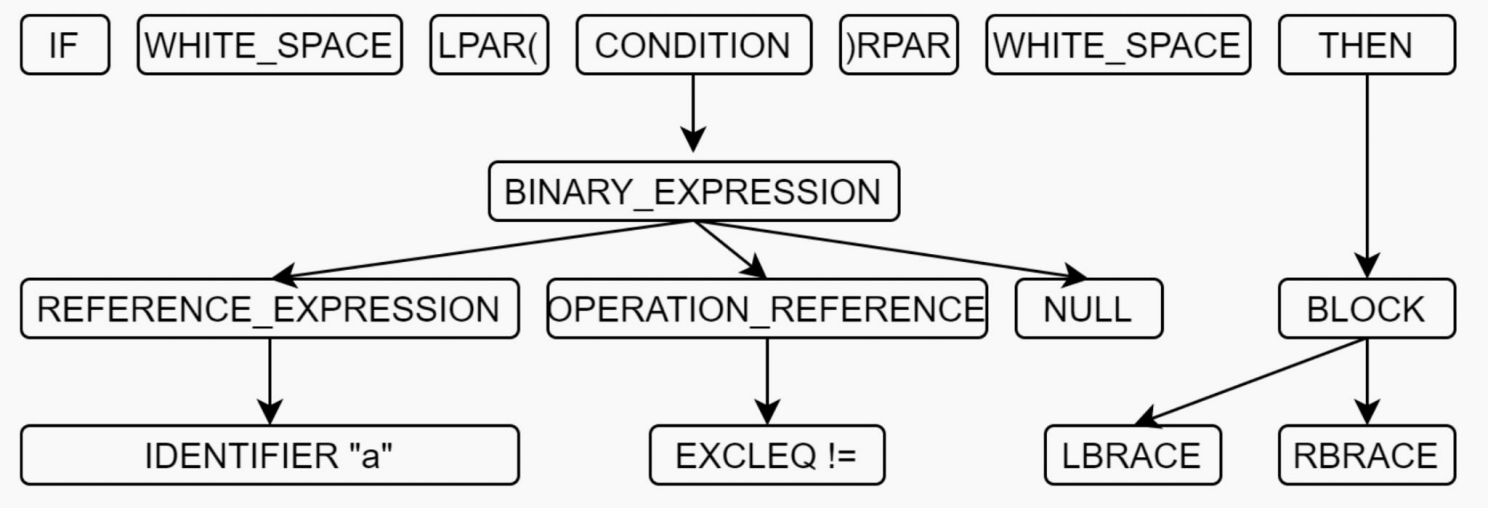
In this talk, we will uncover the purpose of K2 and why the compiler is being actively rewritten till nowadays as well as discuss language updates starting with version 1.7. Furthermore, we'll talk about the developers' future plans and other updates that you might have missed.

Speakers

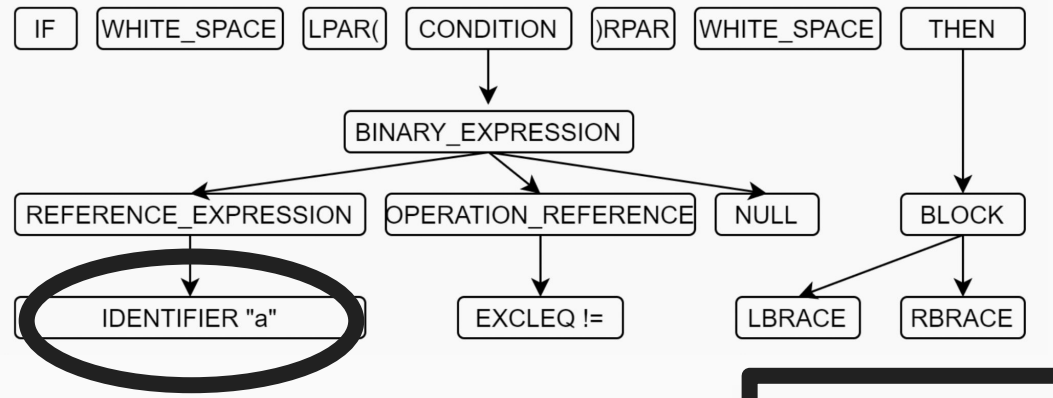
Andrey Kuleshov
Huawei

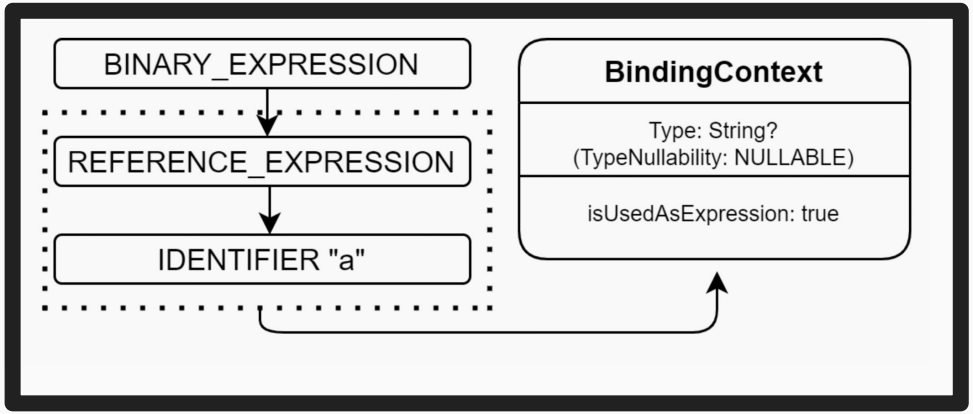
Anzhelika Pokhodun

AST/PSI: syntax info

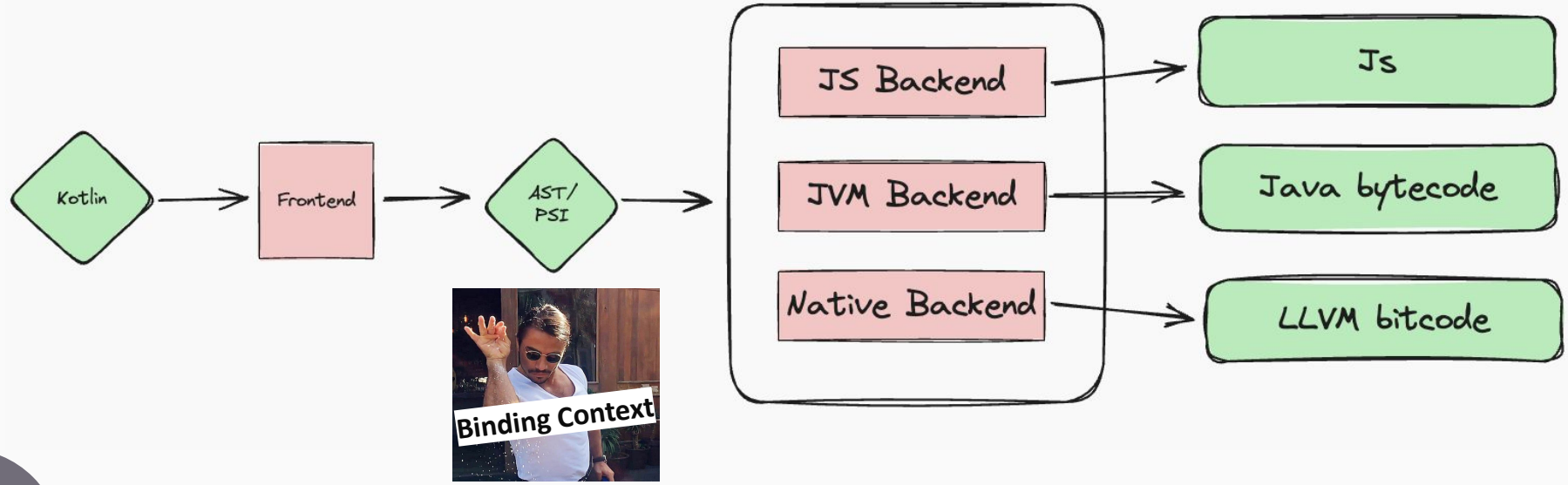


Binding Context: semantic info

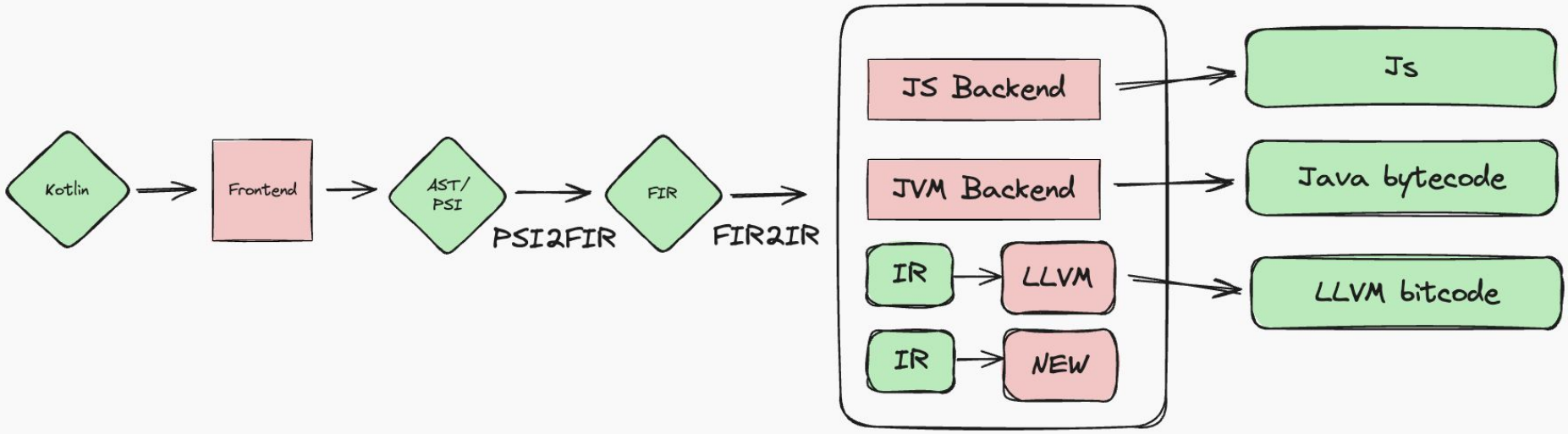




Old Kotlin compiler

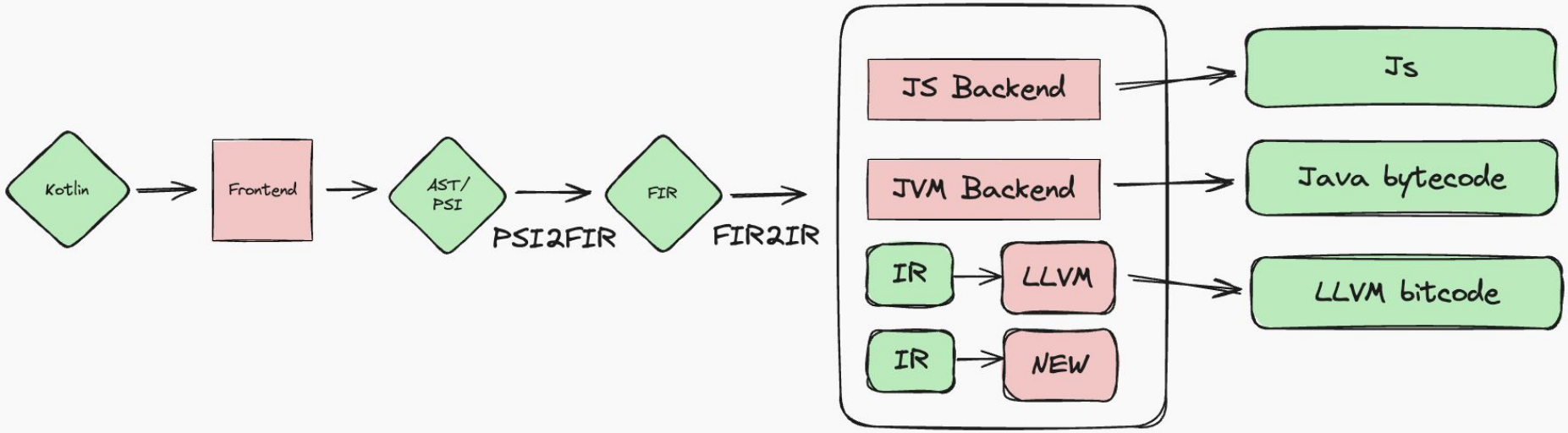


New Kotlin compiler



+optimizer

New Kotlin compiler



+optimizer

compiler/fir/raw-fir/**psi2fir**/src/org/jetbrains/kotlin/fir/builder/**PsiRawFirBuilder**.kt

compiler/fir/**fir2ir**/src/org/jetbrains/kotlin/fir/backend/**Fir2IrConverter**.kt

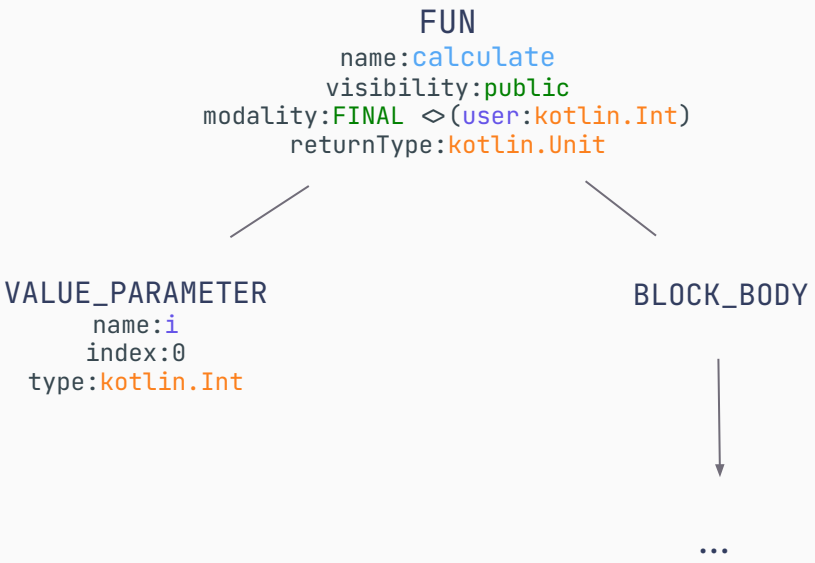
Backend IR: Ooops, it's a tree

```
fun calculate(i: Int) {
    ...
}
```

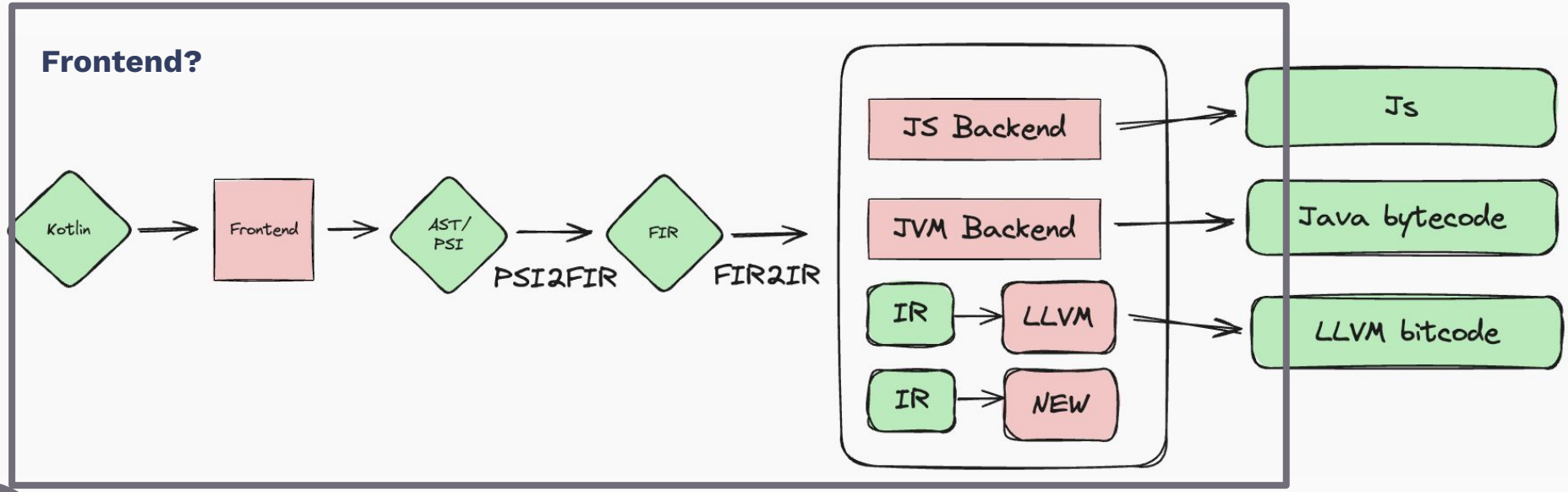
```
FUN
name:calculate
visibility:public
modality:FINAL <>(user:kotlin.Int)
returnType:kotlin.Unit
```

Backend IR: Ooops, it's a tree

```
fun calculate(i: Int) {
    ...
}
```



New Kotlin compiler

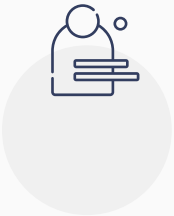


compiler/fir/raw-fir/**psi2fir**/src/org/jetbrains/kotlin/fir/builder/**PsiRawFirBuilder**.kt

compiler/fir/**fir2ir**/src/org/jetbrains/kotlin/fir/backend/**Fir2IrConverter**.kt

But! Let’s treat it separately

With Kotlin compilers we can invoke into any compiler pass



Frontend

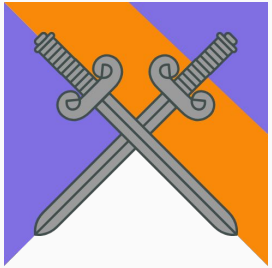
Backend

But! Let’s treat it separately

With Kotlin compilers we can invoke into any compiler pass

Frontend

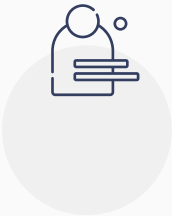
Good for the **simple** code analysis and manipulations with AST-like structures



Backend

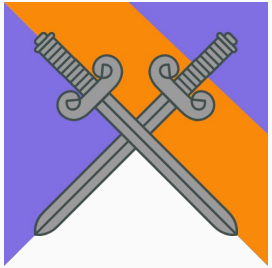
But! Let’s treat it separately

With Kotlin compilers we can invoke into any compiler pass



Frontend

Good for the **simple** code analysis and manipulations with AST-like structures



Backend

Cannot affect the code analysis. But extremely useful for modifications of IR

But! Let’s treat it separately

Examples of Kotlin compiler plugins

Frontend

Allopen

Backend

Reflekt

But! Let’s treat it separately

Examples of Kotlin compiler plugins

Frontend

Allopen

Backend

Reflekt

kotlinx.serialization

Until K2 API is unstable, especially for FIR

at least try to use methods/structures starting with **FIR**

04.

Writing a plugin

Let’s write all-open FIR plugin

Entry point: register plugin

```
@OptIn(ExperimentalCompilerApi::class)
class AllOpenComponentRegistrar : CompilerPluginRegistrar() {





    override val supportsK2: Boolean
        get() = true
}
```

https://github.com/JetBrains/kotlin/blob/f32367d2c2f84ce42ebeaec134228579d83c5a77/plugins/allopen/allopen.cli/src/org/jetbrains/kotlin/allopen/AllOpenPlugin.kt

Entry point: register plugin

```
@OptIn(ExperimentalCompilerApi::class)
class AllOpenComponentRegistrar : CompilerPluginRegistrar() {

  override fun ExtensionStorage.registerExtensions(configuration: CompilerConfiguration) {

      FirExtensionRegistrarAdapter.registerExtension(FirAllOpenExtensionRegistrar(...))

  }

   override val supportsK2: Boolean
       get() = true
}
```

https://github.com/JetBrains/kotlin/blob/f32367d2c2f84ce42ebeaec134228579d83c5a77/plugins/allopen/allopen.cli/src/org/jetbrains/kotlin/allopen/AllOpenPlugin.kt

Entry point: extension

```
class FirAllOpenExtensionRegistrar(val allOpenAnnotationFqNames: List<String>) : FirExtensionRegistrar() {
    override fun ExtensionRegistrarContext.configurePlugin() {

        +::FirAllOpenStatusTransformer
        . . .
    }
}
```

https://github.com/JetBrains/kotlin/blob/f32367d2c2f84ce42ebeaec134228579d83c5a77/plugins/allopen/allopen.k2/src/org/jetbrains/kotlin/allopen/fir/FirAllOpenExtensionRegistrar.kt

Logic itself

```
class FirAllOpenStatusTransformer(session: FirSession) : FirStatusTransformerExtension(session) {
    override fun needTransformStatus(declaration: FirDeclaration): Boolean {
        return when (declaration) {
            is FirRegularClass -> declaration.classKind == ClassKind.CLASS &&
                                  session.allOpenPredicateMatcher.isAnnotated(declaration.symbol)
                . . .
        }
    }

    override fun transformStatus(status: FirDeclarationStatus, declaration: FirDeclaration): FirDeclarationStatus {
        return if (status.modality == null) {
            status.copy(modality = Modality.OPEN)
        } else {
            status
        }
    }
}
```

https://github.com/JetBrains/kotlin/blob/f32367d2c2f84ce42ebeaec134228579d83c5a77/plugins/allopen/allopen.k2/src/org/jetbrains/kotlin/allopen/fir/FirAllOpenStatusTransformer.kt

Logic itself

```kotlin
private inline fun FirMemberDeclaration.applyExtensionTransformers(
   operation: FirStatusTransformerExtension.(FirDeclarationStatus) -> FirDeclarationStatus
) {
   . . .

   val newStatus = statusExtensions.fold(status) { acc, it ->
       if (it.needTransformStatus(declaration)) {
           it.operation(acc)
       } else {
           acc
       }
   } as FirDeclarationStatusImpl

   . . .

   replaceStatus(resolvedStatus)
}
```

https://github.com/JetBrains/kotlin/blob/808d4353e5bef8831e5d08f34fa634e390112eba/compiler/fir/java/src/org/jetbrains/kotlin/fir/java/FirJavaFacade.kt

Usage

K1:

Gradle plugins 🤬

K2:

***PLANNED* DSL!**

```
dependencies {
    compilerPlugin("com.akuleshov7.plugin ...")
}
```

Usage

Kotlin v1.9.22 Solutions Docs

- Home
- Get started
- Take Kotlin tour
- Kotlin overview
- What's new in Kotlin
- Releases and roadmap
- Basics
- Concepts
- Multiplatform development
- Platforms
- Standard library
- Official libraries
- API reference
- Language reference
- Tools
- Compiler and plugins
  - Compiler
  - Compiler plugins
    - All-open
    - No-arg
    - SAM with receiver
    - kapt
    - Lombok

Compiler and plugins / Compiler plugins / Lombok

Lombok compiler plugin

Edit page Last modified: 11 July 2023

The Lombok compiler plugin is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack ↗.

The Kotlin Lombok compiler plugin allows the generation and use of Java's Lombok declarations by Kotlin code in the same mixed Java/Kotlin module. If you call such declarations from another module, then you don't need to use this plugin for the compilation of that module.

The Lombok compiler plugin cannot replace Lombok ↗, but it helps Lombok work in mixed Java/Kotlin modules. Thus, you still need to configure Lombok as usual when using this plugin. Learn more about how to configure the Lombok compiler plugin.

Supported annotations

The plugin supports the following annotations:

- `@Getter`, `@Setter`
- `@Builder`

https://kotlinlang.org/docs/lombok.html

Future: K2?

K2 COMPILER PLUGINS API

- Planned new **DSL** for Compiler Plugins !
- **Generation** of new declarations, including top-level functions and properties
- Transformation of **visibility** (hello to AllOpen)
- **Checkers**: for calls and expressions

Conclusion

- All concepts were **already** invented a long before
- A compiler plugin is a good approach to interfere with compilers and change code or perform static checks
- Compiler plugins are a good way to get a first touch of system programming

Thanks!

Github: akuleshov7