VÍCTOR COELHO

Software Engineer @ Codex Utilities

VÍCTOR COELHO

Software Engineer @ Codex Utilities

- linkedin: https://linkedin.com/in/rabbitvictor

VÍCTOR COELHO

Software Engineer @ Codex Utilities

- linkedin: https://linkedin.com/in/rabbitvictor
- tech blog: https://rabbitvictor.com

KOTLIN COMPILER

1. O que é um Compilador?
    1. Frontend
    2. Backend
2. Kotlin Compiler
    1. Overview
    2. Frontend
    3. Backend
    4. Compiler Plugins e Annotation Processors

O QUE É UM COMPILADOR?

De forma simples, um compilador é um programa que pode ler um programa em uma linguagem — *a linguagem de origem* — e traduzi-lo para um programa equivalente em outra linguagem — *a linguagem de destino*.

Compilers: Principles, Techniques, and Tools

FRONTEND

Identifica a **estrutura** e **significado** da linguagem de programação de **origem**

- Existem muitos paralelos entre o trabalho do frontend de um compilador e a **Linguística**

- Existem muitos paralelos entre o trabalho do frontend de um compilador e a **Linguística**
- Fazemos análises léxicas, sintáticas e semânticas nas linguagens naturais também!

ANÁLISE LÉXICA:
LEXER

- O *lexer* identifica quais **palavras** foram escritas no código-fonte

- O *lexer* identifica quais **palavras** foram escritas no código-fonte
- Categoriza estas palavras em tipos de palavras válidas para a linguagem

- O *lexer* identifica quais **palavras** foram escritas no código-fonte
- Categoriza estas palavras em tipos de palavras válidas para a linguagem
- Cada linguagem vai definir estes **tipos válidos** ou *Token Types*

- O *lexer* identifica quais **palavras** foram escritas no código-fonte
- Categoriza estas palavras em tipos de palavras válidas para a linguagem
- Cada linguagem vai definir estes **tipos válidos** ou *Token Types*
- Nesta etapa, tipos de tokens (ou palavras) desconhecidos são detectados como erros de compilação

TOKEN TYPES: KOTLIN

Tipo de Token	Exemplos
LBRACKET	[
RBRACKET	]
PLUSPLUS	++
ELVIS	?:
SUSPEND_KEYWORD	suspend
WHEN_KEYWORD	when

```
suspend fun foo() {
  val hello = "Olá Kotlin Devs Brasil!"
}
```

```
SUSPEND_KEYWORD FUN_KEYWORD
WHITE_SPACEIDENTIFIER(foo)
LPAR RPAR
WHITE_SPACE LBRACE
WHITE_SPACE IDENTIFIER(println)
LPAR REGULAR_STRING_PART("Olá Kotlin Devs Brasil!")
RPAR WHITE_SPACE RBRACE
```

ANÁLISE SINTÁTICA:
PARSER

GRAMÁTICA

GRAMÁTICA

- Identifica se as **frases** formadas pelos tokens formam sentenças válidas definidas pela **gramática** da linguagem

GRAMÁTICA

- Identifica se as **frases** formadas pelos tokens formam sentenças válidas definidas pela **gramática** da linguagem
- Comumente se usam Gramáticas Livres de Contexto (CFG), que também são usadas no campo da Linguística

EXTENDED BACKUS–NAUR FORM OU EBNF

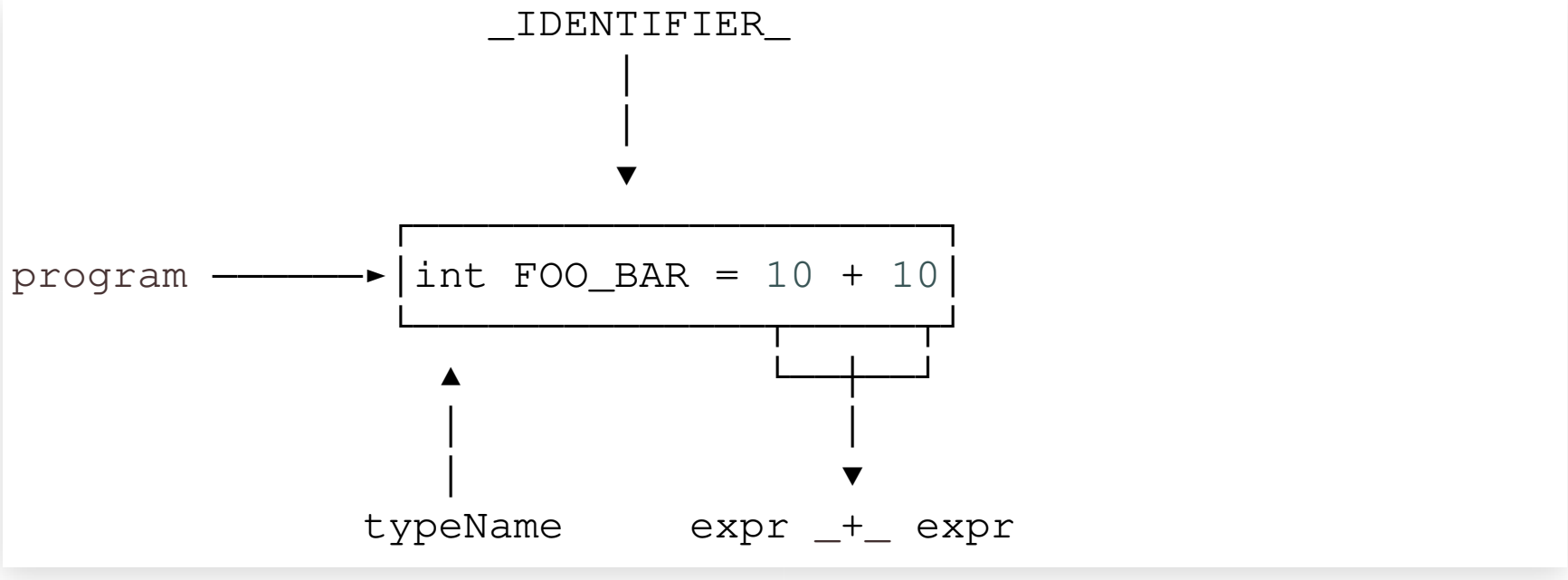
```
program ::= decl* stmt*

decl ::= typeName _IDENTIFIER_ [= expr] _;_

stmt ::= expr _;_
       | _if (_ expr _)_ stmt _else_ stmt
       | _while (_ expr _)_ stmt
       | _{_ stmt*\; _}_
```

```
expr ::= _INT_LITERAL_
       | _FLOAT_LITERAL_
       | _IDENTIFIER_
       | IDENTIFIER _=_ expr
       | expr _+_ expr
       | expr _==_ expr

typeName ::= _int_ | _float_
```


IDENTIFIER
program
int FOO_BAR = 10 + 10
typeName
expr _+_ expr

PARSER

- Gerado a partir da definição gramatical da linguagem que pode estar, por exemplo, em EBNF

PARSER

- Gerado a partir da definição gramatical da linguagem que pode estar, por exemplo, em EBNF
- Utiliza o resultado do *Lexer* para montar uma *Abstract Syntax Tree* (AST) do código-fonte

PARSER

- Gerado a partir da definição gramatical da linguagem que pode estar, por exemplo, em EBNF
- Utiliza o resultado do *Lexer* para montar uma *Abstract Syntax Tree* (AST) do código-fonte
- É mais uma **transformação** aplicada no código alterando sua **estrutura** e **representação**

PARSER

- Gerado a partir da definição gramatical da linguagem que pode estar, por exemplo, em EBNF
- Utiliza o resultado do *Lexer* para montar uma *Abstract Syntax Tree* (AST) do código-fonte
- É mais uma **transformação** aplicada no código alterando sua **estrutura** e **representação**
- Neste momento, erros gramaticais são detectados

PARSER

- Gerado a partir da definição gramatical da linguagem que pode estar, por exemplo, em EBNF
- Utiliza o resultado do *Lexer* para montar uma *Abstract Syntax Tree* (AST) do código-fonte
- É mais uma **transformação** aplicada no código alterando sua **estrutura** e **representação**
- Neste momento, erros gramaticais são detectados
- A **AST** é utilizada nas próximas etapas para identificar outros tipos de erros de compilação

ANÁLISE SEMÂNTICA

- Avalia o **sentido** das expressões do código-fonte

- Avalia o **sentido** das expressões do código-fonte
- Utiliza a **AST** gerada na etapa anterior para realizar esta análise

- Avalia o **sentido** das expressões do código-fonte
- Utiliza a **AST** gerada na etapa anterior para realizar esta análise
    - Utiliza uma **Symbol Table** para armazenar as informações e contexto

- Avalia o **sentido** das expressões do código-fonte
- Utiliza a **AST** gerada na etapa anterior para realizar esta análise
    - Utiliza uma **Symbol Table** para armazenar as informações e contexto
        - Escopo de variáveis

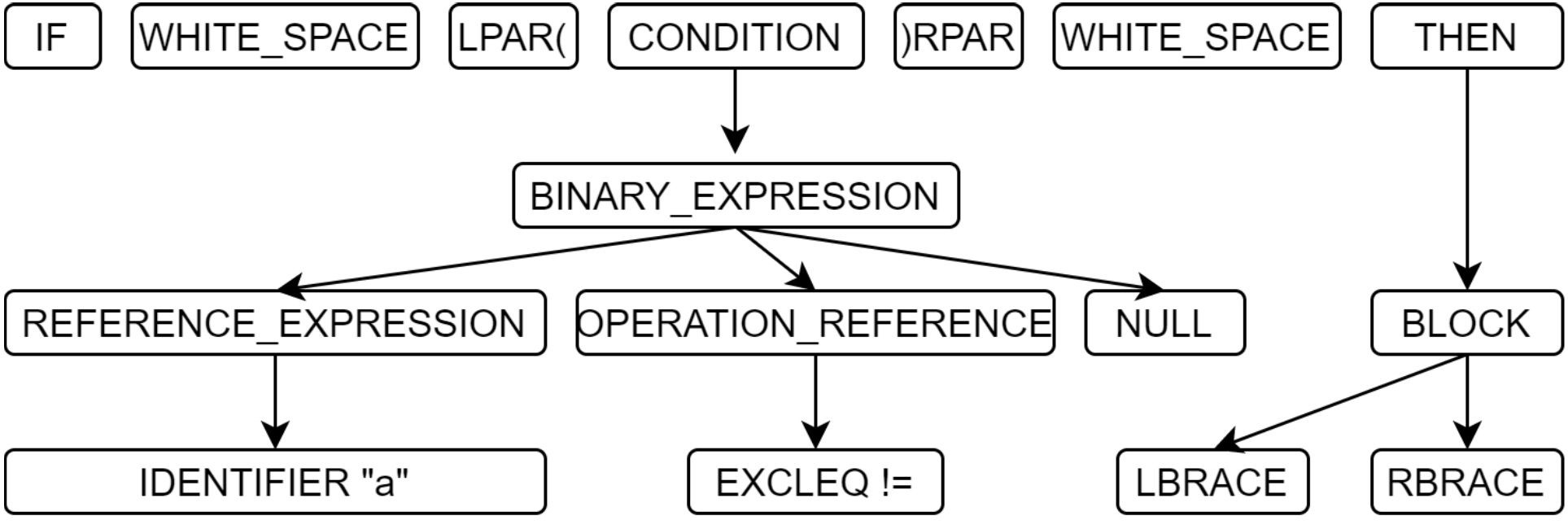
- Avalia o **sentido** das expressões do código-fonte
- Utiliza a **AST** gerada na etapa anterior para realizar esta análise
    - Utiliza uma **Symbol Table** para armazenar as informações e contexto
        - Escopo de variáveis
        - Escopo de funções

- Avalia o **sentido** das expressões do código-fonte
- Utiliza a **AST** gerada na etapa anterior para realizar esta análise
    - Utiliza uma **Symbol Table** para armazenar as informações e contexto
        - Escopo de variáveis
        - Escopo de funções
        - Operações com tipos inválidos (somar um inteiro com uma string)

- Avalia o **sentido** das expressões do código-fonte
- Utiliza a **AST** gerada na etapa anterior para realizar esta análise
    - Utiliza uma **Symbol Table** para armazenar as informações e contexto
        - Escopo de variáveis
        - Escopo de funções
        - Operações com tipos inválidos (somar um inteiro com uma string)
        - Assinaturas e retornos de funções

- Avalia o **sentido** das expressões do código-fonte
- Utiliza a **AST** gerada na etapa anterior para realizar esta análise
    - Utiliza uma **Symbol Table** para armazenar as informações e contexto
        - Escopo de variáveis
        - Escopo de funções
        - Operações com tipos inválidos (somar um inteiro com uma string)
        - Assinaturas e retornos de funções

```
if   (a!= null) {}
```


IF
WHITE_SPACE
LPAR(
CONDITION
)RPAR
WHITE_SPACE
THEN
BINARY_EXPRESSION
REFERENCE_EXPRESSION
OPERATION_REFERENCE
NULL
BLOCK
IDENTIFIER "a"
EXCLEQ !=
LBRACE
RBRACE

REPRESENTAÇÃO INTERMEDIÁRIA (IR)

- Formas de representar um código-fonte que facilitem a análise e otimização deste código

- Formas de representar um código-fonte que facilitem a análise e otimização deste código
- Existem múltiplas IRs ao longo de um processo de compilação

- Formas de representar um código-fonte que facilitem a análise e otimização deste código
- Existem múltiplas IRs ao longo de um processo de compilação
- Possuem objetivos diferentes dependendo da fase de compilação

- Formas de representar um código-fonte que facilitem a análise e otimização deste código
- Existem múltiplas IRs ao longo de um processo de compilação
- Possuem objetivos diferentes dependendo da fase de compilação
    - Uma IR do frontend precisa de informações diferentes de uma IR do backend

- Formas de representar um código-fonte que facilitem a análise e otimização deste código
- Existem múltiplas IRs ao longo de um processo de compilação
- Possuem objetivos diferentes dependendo da fase de compilação
    - Uma IR do frontend precisa de informações diferentes de uma IR do backend
- De modo geral, conforme as fases avançam, a IR se torna mais **baixo nível** e mais próxima da linguagem de destino.

- Formas de representar um código-fonte que facilitem a análise e otimização deste código
- Existem múltiplas IRs ao longo de um processo de compilação
- Possuem objetivos diferentes dependendo da fase de compilação
    - Uma IR do frontend precisa de informações diferentes de uma IR do backend
- De modo geral, conforme as fases avançam, a IR se torna mais **baixo nível** e mais próxima da linguagem de destino.
- O frontend do compilador gera uma IR que será consumida pelo backend

BACKEND

Transforma uma IR na linguagem de programação alvo

- Recebe a IR do Frontend e aplica diversas transformações nesta representação;

- Recebe a IR do Frontend e aplica diversas transformações nesta representação;
- Cada transformação tem um objetivo: otimização, *lowering* da IR

Data-Flow Analysis

- Uma série de etapas muito importantes é conhecida como *data-flow analysis* ou Análise de Fluxo de Código.

Data-Flow Analysis

- Uma série de etapas muito importantes é conhecida como *data-flow analysis* ou Análise de Fluxo de Código.
- São etapas que recebem a IR do frontend e realizam etapas de otimização gerais ou, ainda, otimizações específicas para uma linguagem alvo.

- Alguns exemplos destas fases de transformações:

- Alguns exemplos destas fases de transformações:
  - *Dead code elimination* (Eliminação de código morto): remoção de variáveis não utilizadas;

- Alguns exemplos destas fases de transformações:
  - *Dead code elimination* (Eliminação de código morto): remoção de variáveis não utilizadas;
  - Eliminação de subexpressões comuns: remoção de cálculos de uma expressão repetida;

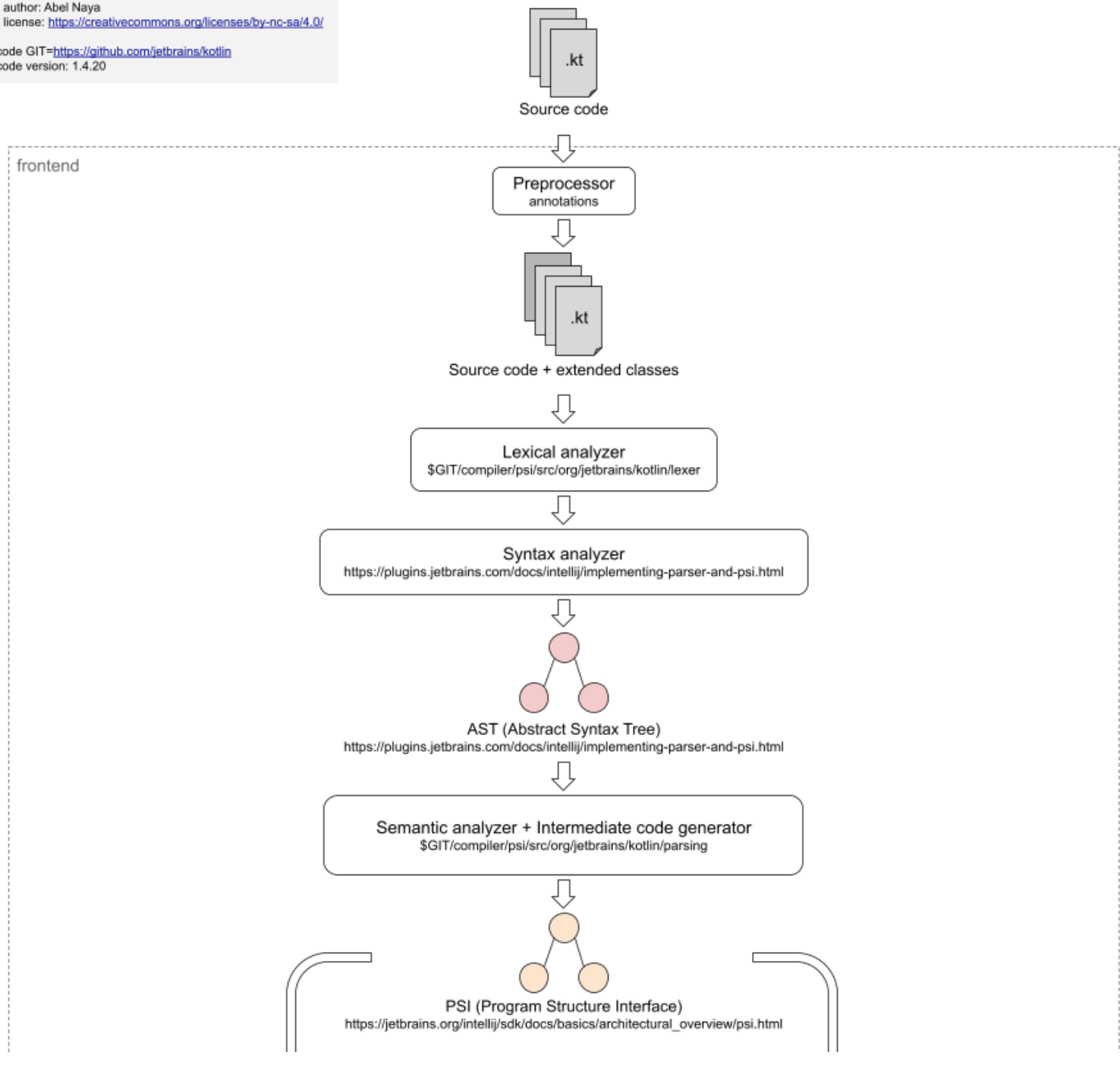
- Alguns exemplos destas fases de transformações:
  - *Dead code elimination* (Eliminação de código morto): remoção de variáveis não utilizadas;
  - Eliminação de subexpressões comuns: remoção de cálculos de uma expressão repetida;
  - *Constant Folding*: cálculo de expressões constantes em tempo de compilação;

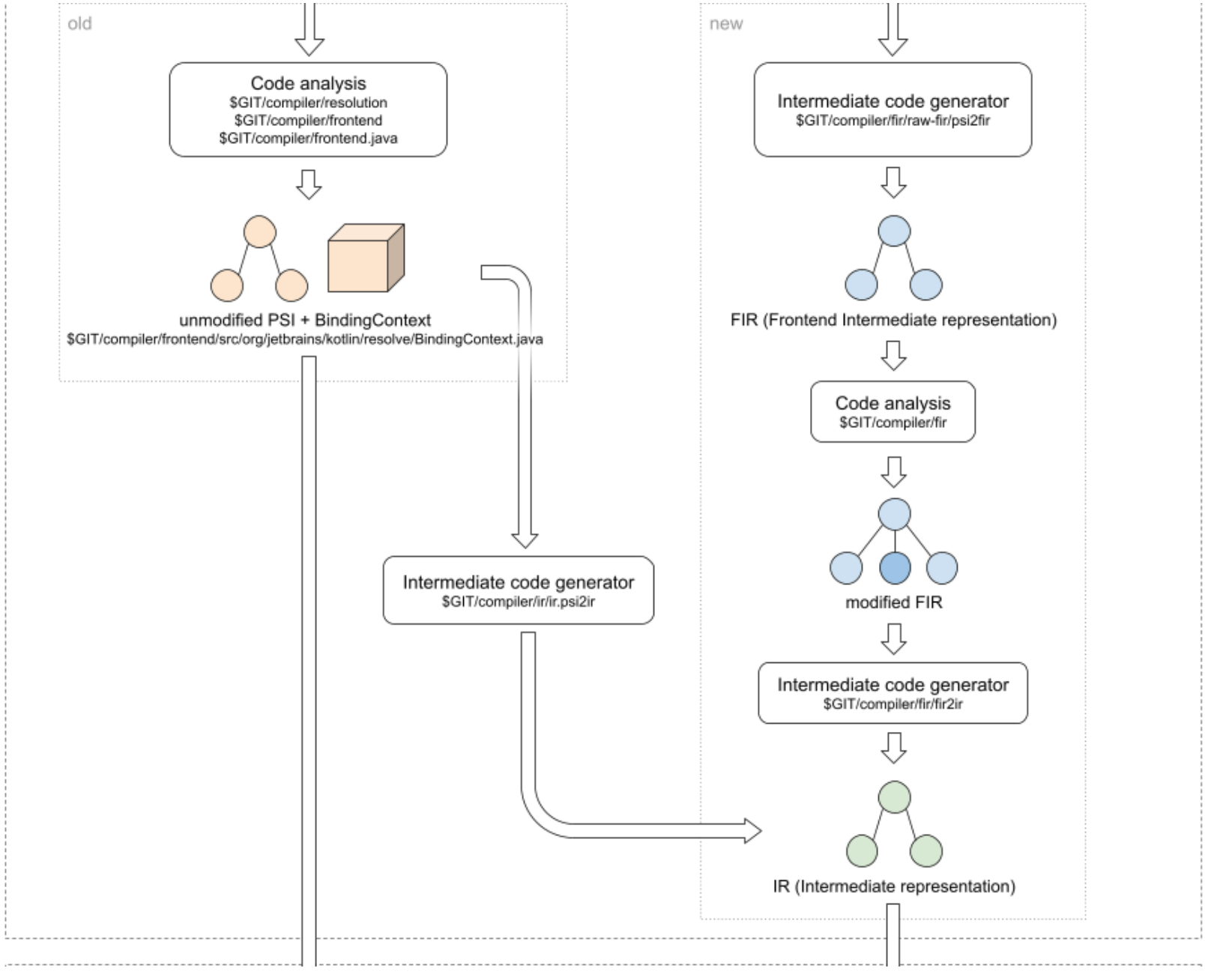
- Alguns exemplos destas fases de transformações:
  - *Dead code elimination* (Eliminação de código morto): remoção de variáveis não utilizadas;
  - Eliminação de subexpressões comuns: remoção de cálculos de uma expressão repetida;
  - *Constant Folding*: cálculo de expressões constantes em tempo de compilação;
  - Alocação de registradores: reutilização de registradores durante a execução do código;

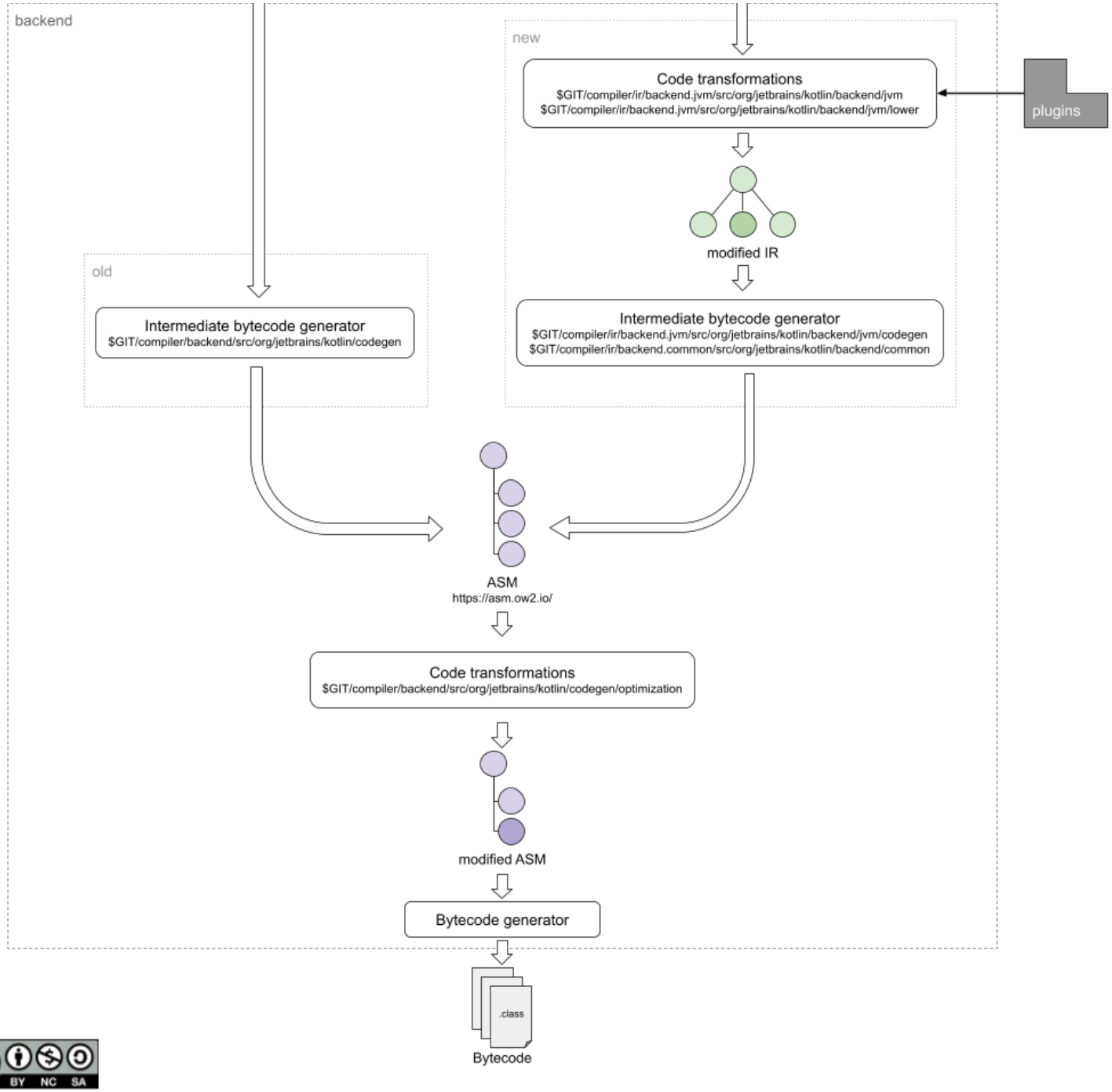
- Alguns exemplos destas fases de transformações:
  - *Dead code elimination* (Eliminação de código morto): remoção de variáveis não utilizadas;
  - Eliminação de subexpressões comuns: remoção de cálculos de uma expressão repetida;
  - *Constant Folding*: cálculo de expressões constantes em tempo de compilação;
  - Alocação de registradores: reutilização de registradores durante a execução do código;
- A etapa final é, de fato, retornar um código, muitas vezes código de máquina ou bytecode, na linguagem alvo.

KOTLIN COMPILER OVERVIEW

Source code GIT=https://github.com/jetbrains/kotlin
Source code version: 1.4.20




old
Code analysis
$GIT/compiler/resolution
$GIT/compiler/frontend
$GIT/compiler/frontend.java
unmodified PSI + BindingContext
$GIT/compiler/frontend/src/org/jetbrains/kotlin/resolve/BindingContext.java
Intermediate code generator
$GIT/compiler/ir/ir.psi2ir
new
Intermediate code generator
$GIT/compiler/fir/raw-fir/psi2fir
FIR (Frontend Intermediate representation)
Code analysis
$GIT/compiler/fir
modified FIR
Intermediate code generator
$GIT/compiler/fir/fir2ir
IR (Intermediate representation)


backend
new
Code transformations
$GIT/compiler/ir/backend.jvm/src/org/jetbrains/kotlin/backend/jvm
$GIT/compiler/ir/backend.jvm/src/org/jetbrains/kotlin/backend/jvm/lower
plugins
modified IR
old
Intermediate bytecode generator
$GIT/compiler/backend/src/org/jetbrains/kotlin/codegen
Intermediate bytecode generator
$GIT/compiler/ir/backend.jvm/src/org/jetbrains/kotlin/backend/jvm/codegen
$GIT/compiler/ir/backend.common/src/org/jetbrains/kotlin/backend/common
ASM
https://asm.ow2.io/
Code transformations
$GIT/compiler/backend/src/org/jetbrains/kotlin/codegen/optimization
modified ASM
Bytecode generator
.class
Bytecode

Algumas representações intermediárias:

- PSI (*Program Structure Interface*)
- FIR (*Frontend intermediate Representation*)
- IR (*Intermediate Representation*)
- MIR (*Modified Intermediate Representation*)

FRONTEND

PSI (*PROGRAM STRUCTURE INTERFACE*)

PSI (*PROGRAM STRUCTURE INTERFACE*)

- Representação do programa utilizada pelas IDEs da JetBrains para realizar análise de código e outras funcionalidades:

PSI (*PROGRAM STRUCTURE INTERFACE*)

- Representação do programa utilizada pelas IDEs da JetBrains para realizar análise de código e outras funcionalidades:
  - Code Refactoring

PSI (*PROGRAM STRUCTURE INTERFACE*)

- Representação do programa utilizada pelas IDEs da JetBrains para realizar análise de código e outras funcionalidades:
  - Code Refactoring
  - Code Completion

PSI (*PROGRAM STRUCTURE INTERFACE*)

- Representação do programa utilizada pelas IDEs da JetBrains para realizar análise de código e outras funcionalidades:
  - Code Refactoring
  - Code Completion
  - Navigation

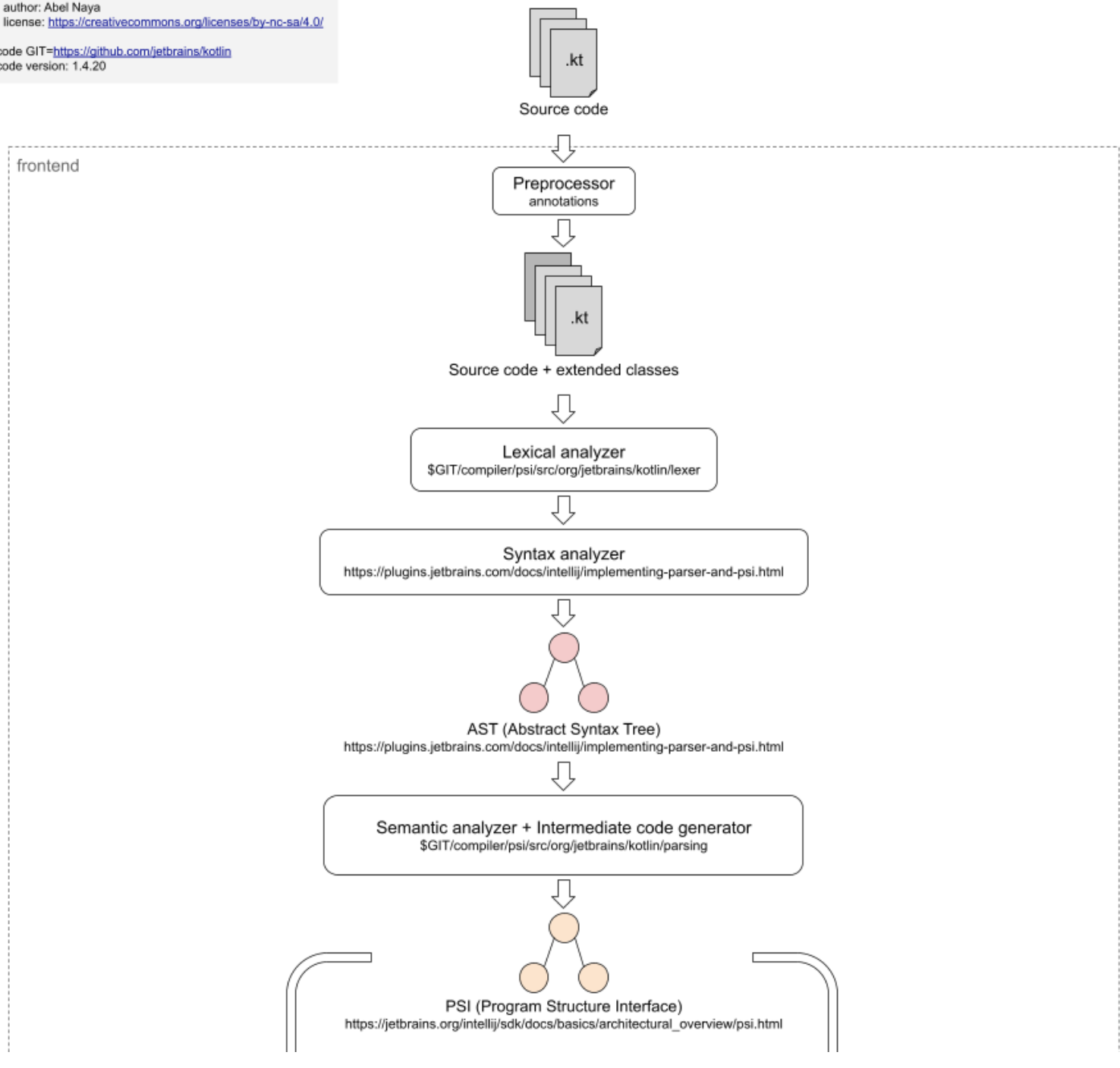
PSI (*PROGRAM STRUCTURE INTERFACE*)

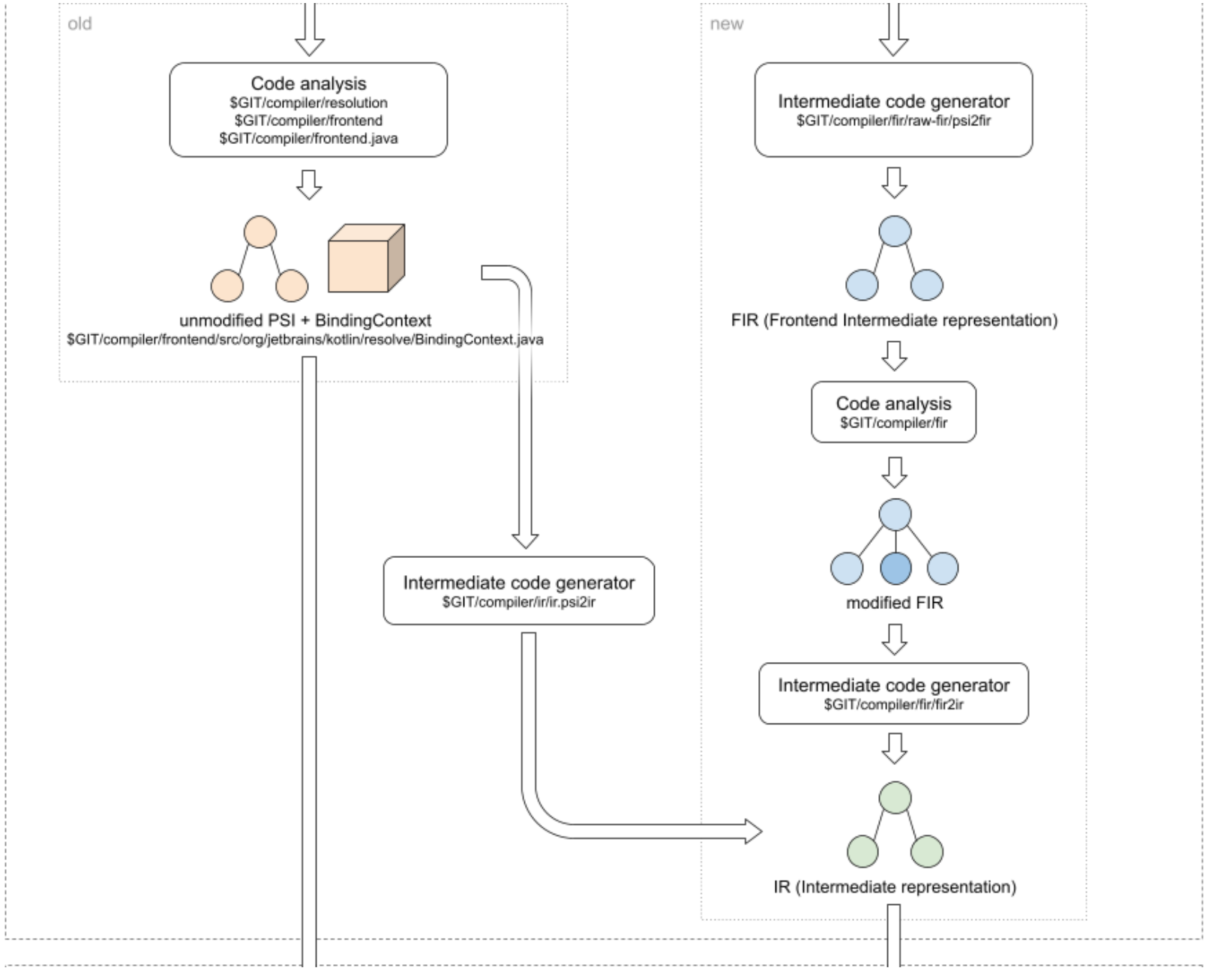
- Representação do programa utilizada pelas IDEs da JetBrains para realizar análise de código e outras funcionalidades:
  - Code Refactoring
  - Code Completion
  - Navigation
  - Code Inspection

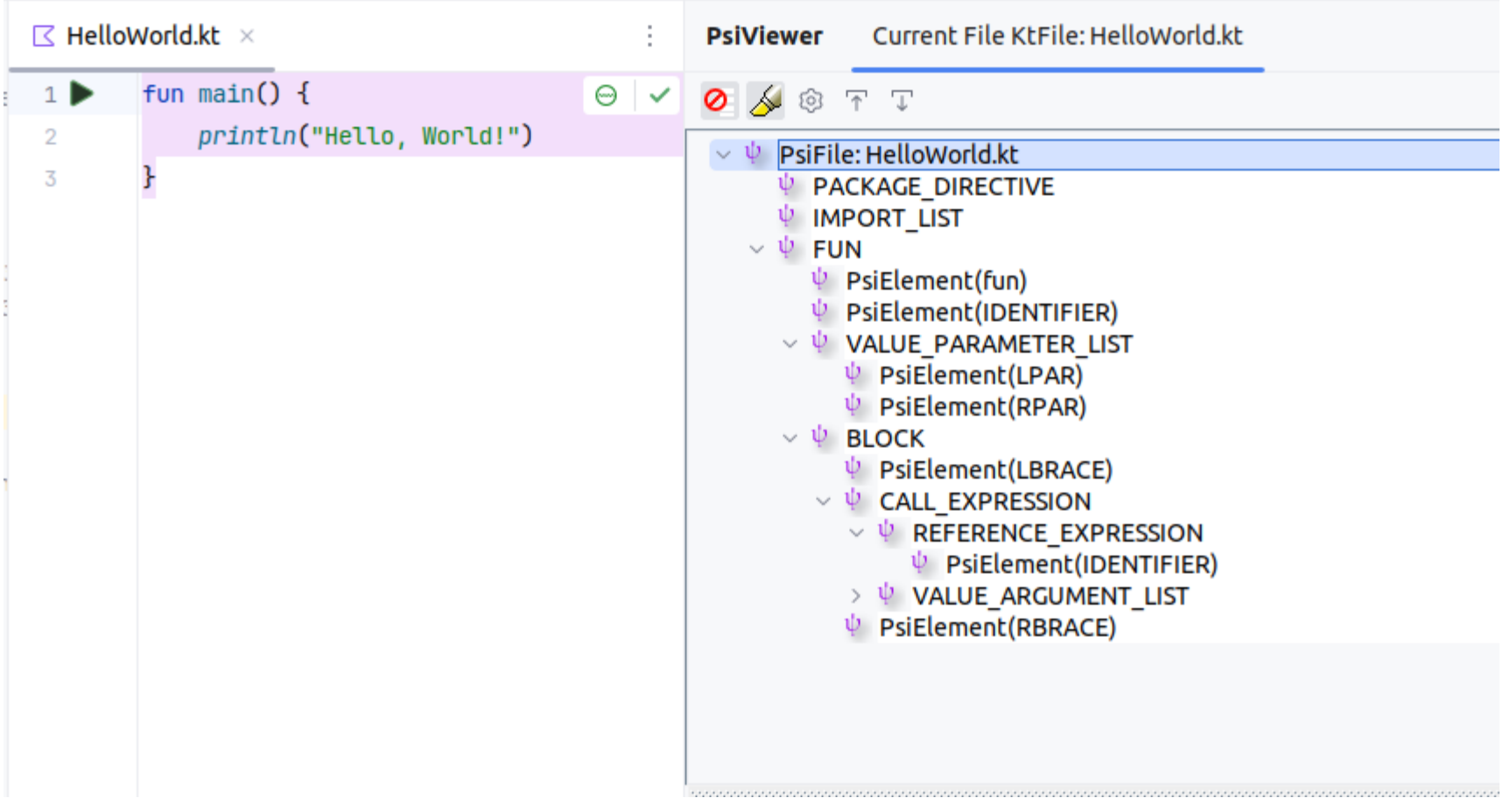
PSI (*PROGRAM STRUCTURE INTERFACE*)

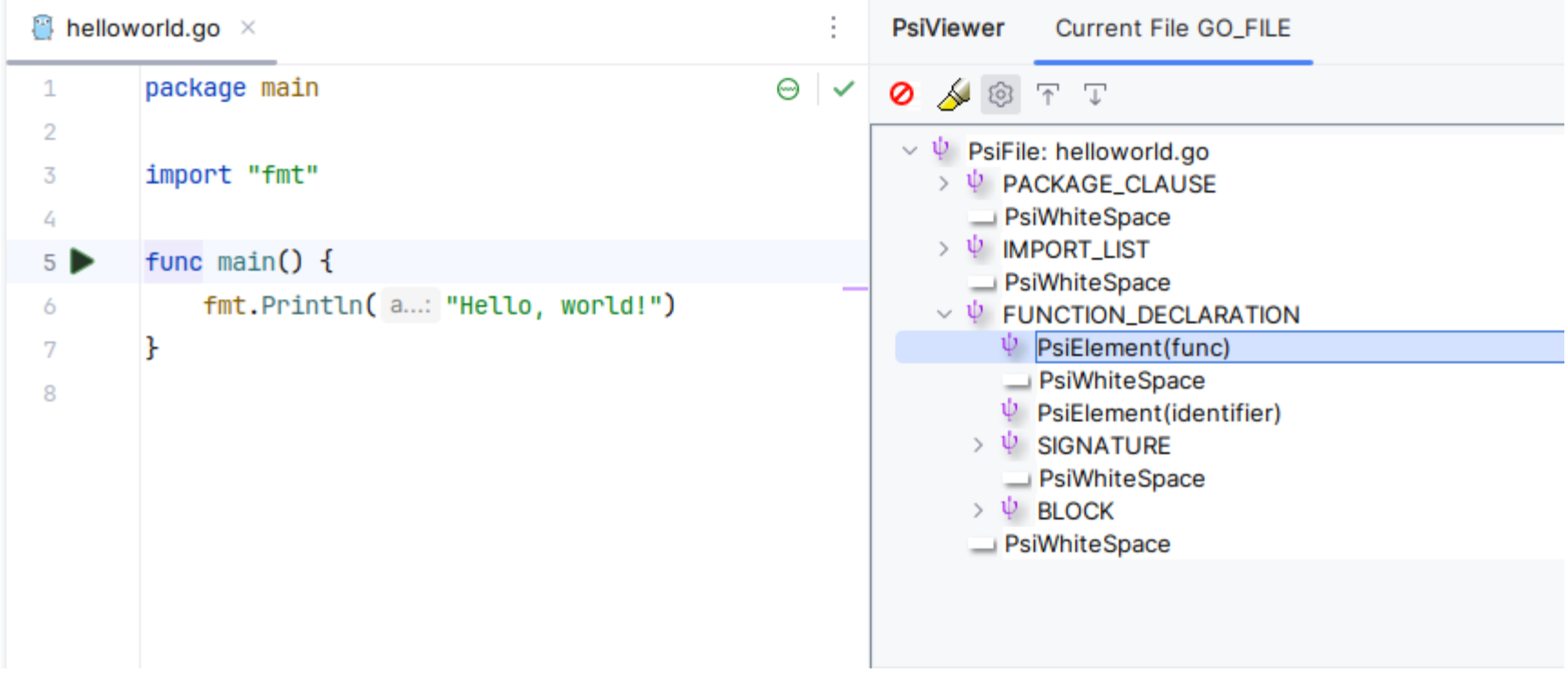
- Representação do programa utilizada pelas IDEs da JetBrains para realizar análise de código e outras funcionalidades:
  - Code Refactoring
  - Code Completion
  - Navigation
  - Code Inspection
  - Intention Actions

Source code GIT=https://github.com/jetbrains/kotlin
Source code version: 1.4.20

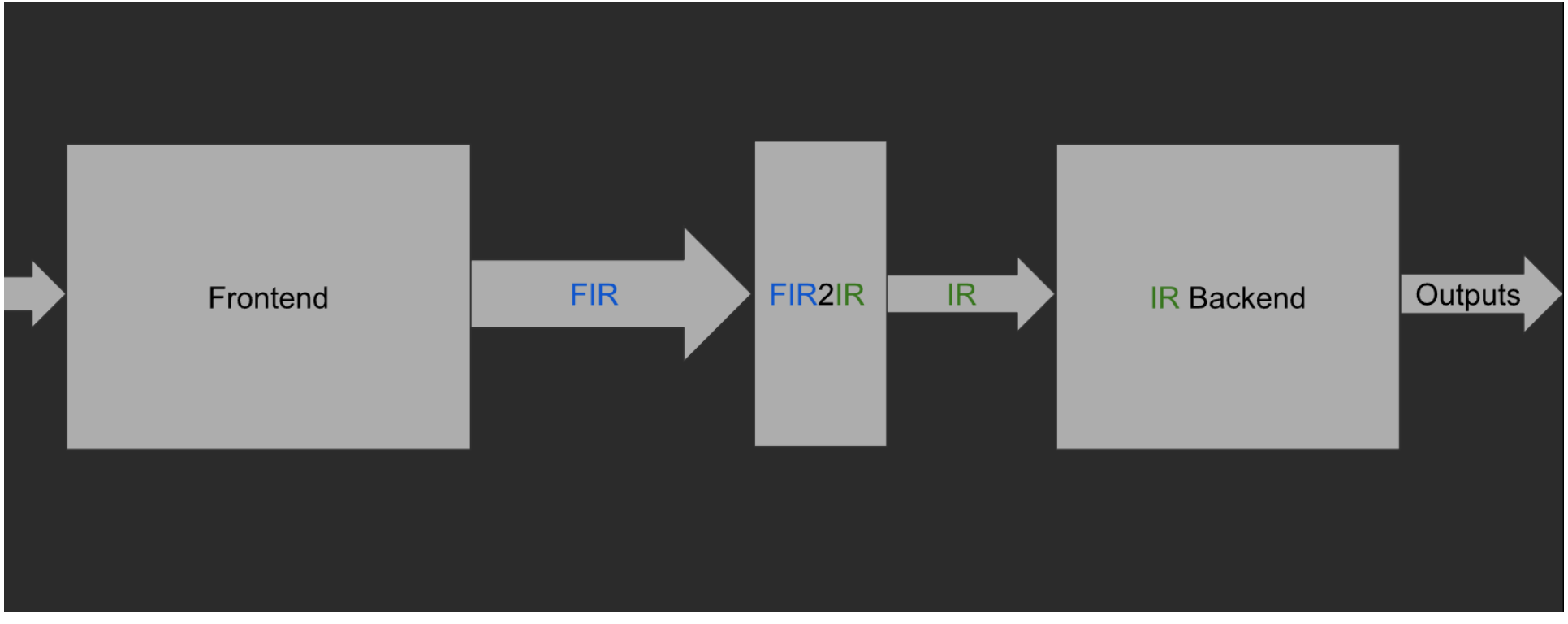


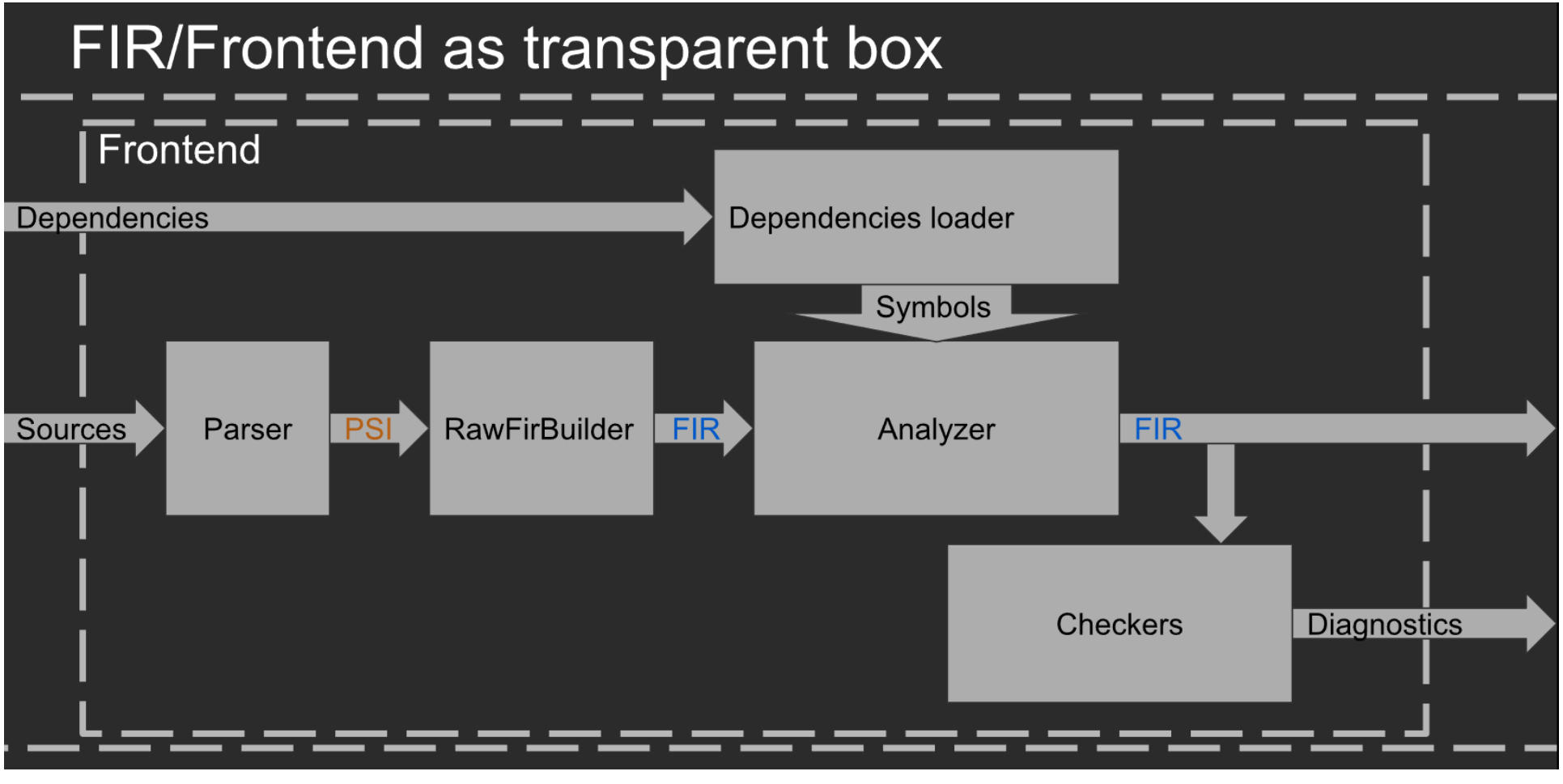

old
Code analysis
$GIT/compiler/resolution
$GIT/compiler/frontend
$GIT/compiler/frontend.java
unmodified PSI + BindingContext
$GIT/compiler/frontend/src/org/jetbrains/kotlin/resolve/BindingContext.java
Intermediate code generator
$GIT/compiler/ir/ir.psi2ir
new
Intermediate code generator
$GIT/compiler/fir/raw-fir/psi2fir
FIR (Frontend Intermediate representation)
Code analysis
$GIT/compiler/fir
modified FIR
Intermediate code generator
$GIT/compiler/fir/fir2ir
IR (Intermediate representation)


HelloWorld.kt
1 fun main() {
2 println("Hello, World!")
3 }
PsiViewer
Current File KtFile: HelloWorld.kt
PsiFile: HelloWorld.kt
PACKAGE_DIRECTIVE
IMPORT_LIST
FUN
PsiElement(fun)
PsiElement(IDENTIFIER)
VALUE_PARAMETER_LIST
PsiElement(LPAR)
PsiElement(RPAR)
BLOCK
PsiElement(LBRACE)
CALL_EXPRESSION
REFERENCE_EXPRESSION
PsiElement(IDENTIFIER)
VALUE_ARGUMENT_LIST
PsiElement(RBRACE)

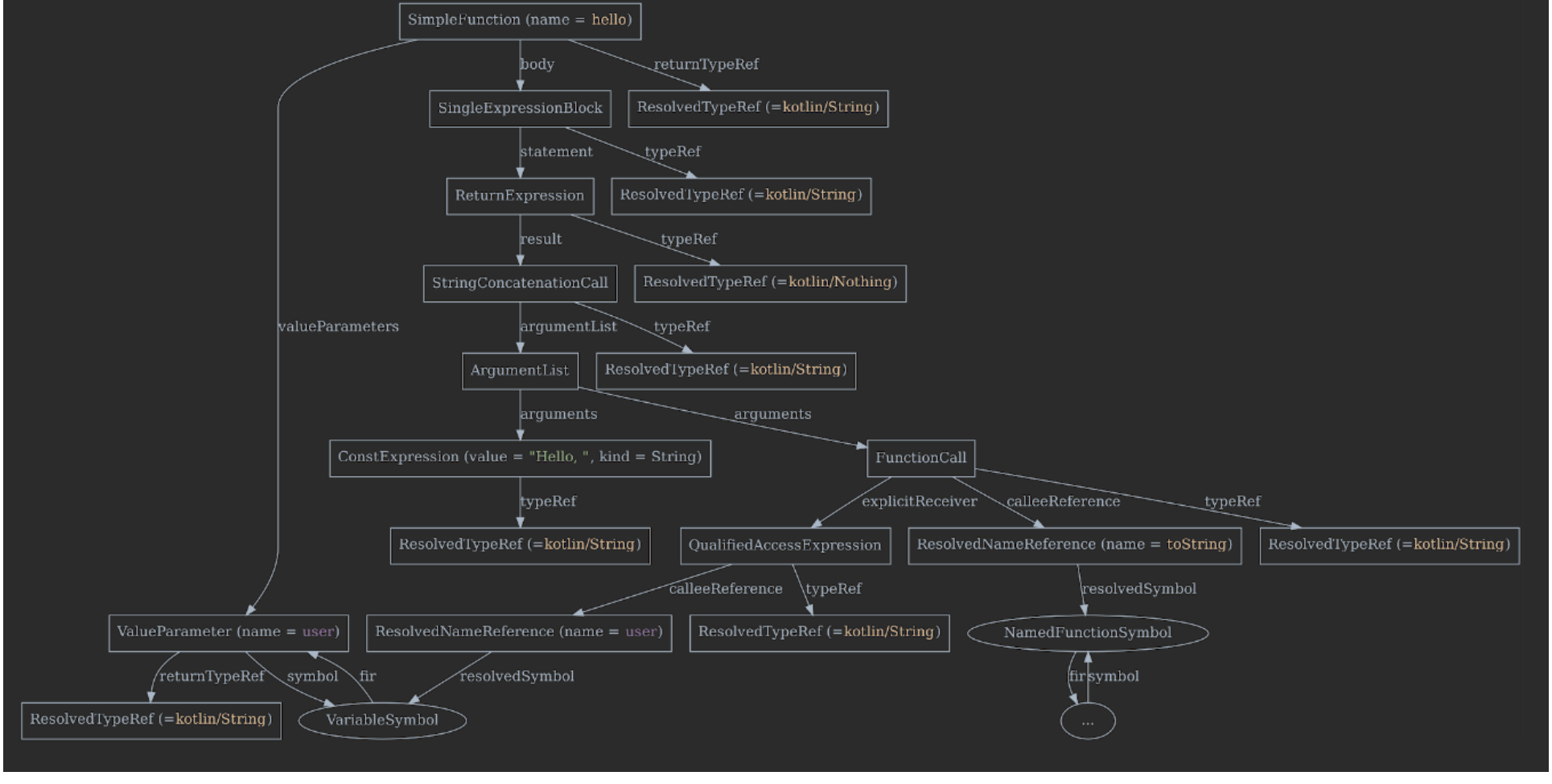

helloworld.go
1 package main
2
3 import "fmt"
4
5 func main() {
6 fmt.Println(a...: "Hello, world!")
7 }
8
PsiViewer
Current File GO_FILE
PsiFile: helloworld.go
PACKAGE_CLAUSE
PsiWhiteSpace
IMPORT_LIST
PsiWhiteSpace
FUNCTION_DECLARATION
PsiElement(func)
PsiWhiteSpace
PsiElement(identifier)
SIGNATURE
PsiWhiteSpace
BLOCK
PsiWhiteSpace

FIR (*FRONTEND INTERMEDIATE REPRESENTATION*)


Frontend
FIR
FIR2IR
IR
IR Backend
Outputs

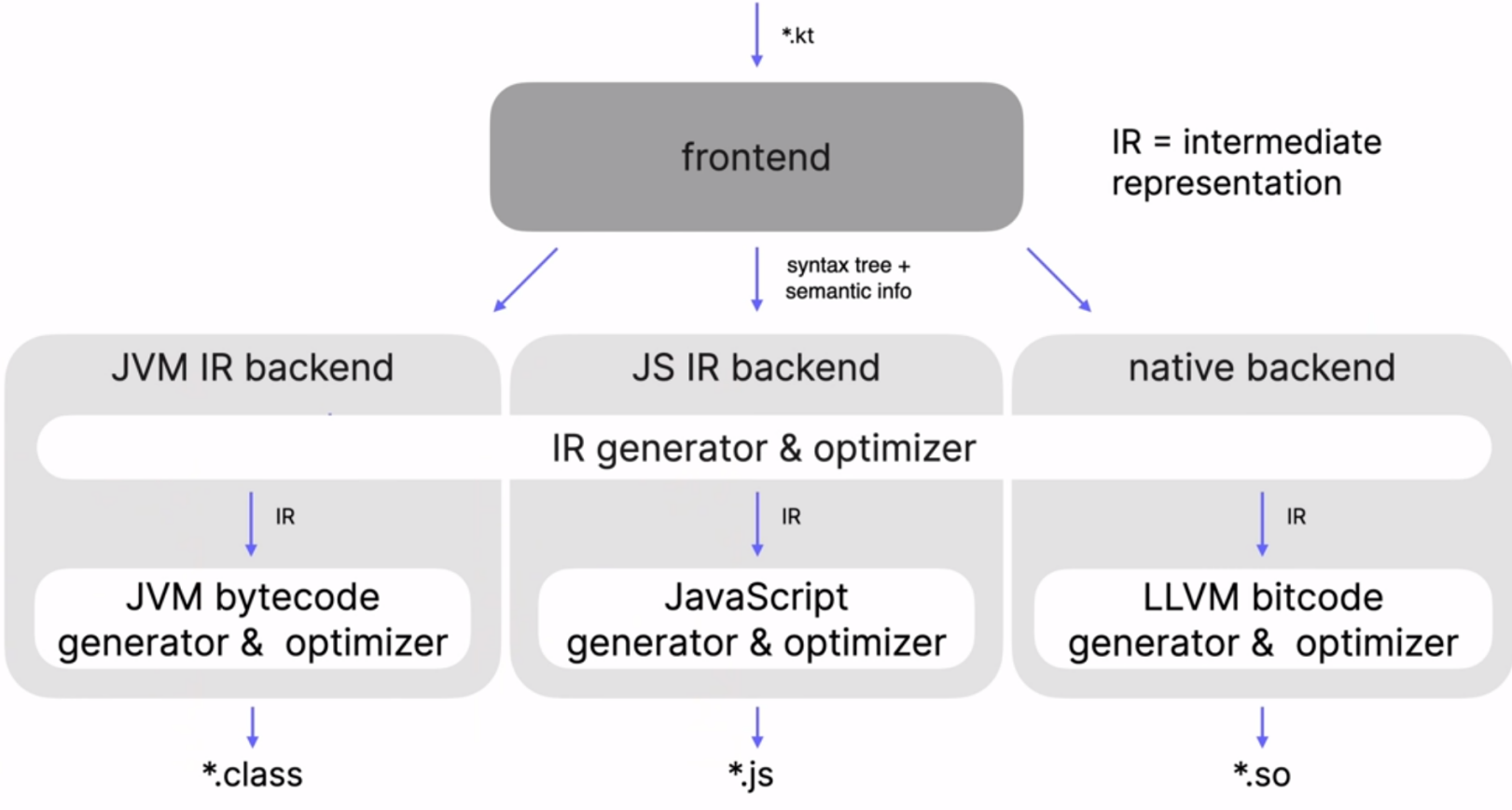

FIR/Frontend as transparent box
Frontend
Dependencies
Dependencies loader
Symbols
Sources
Parser
PSI
RawFirBuilder
FIR
Analyzer
FIR
Checkers
Diagnostics

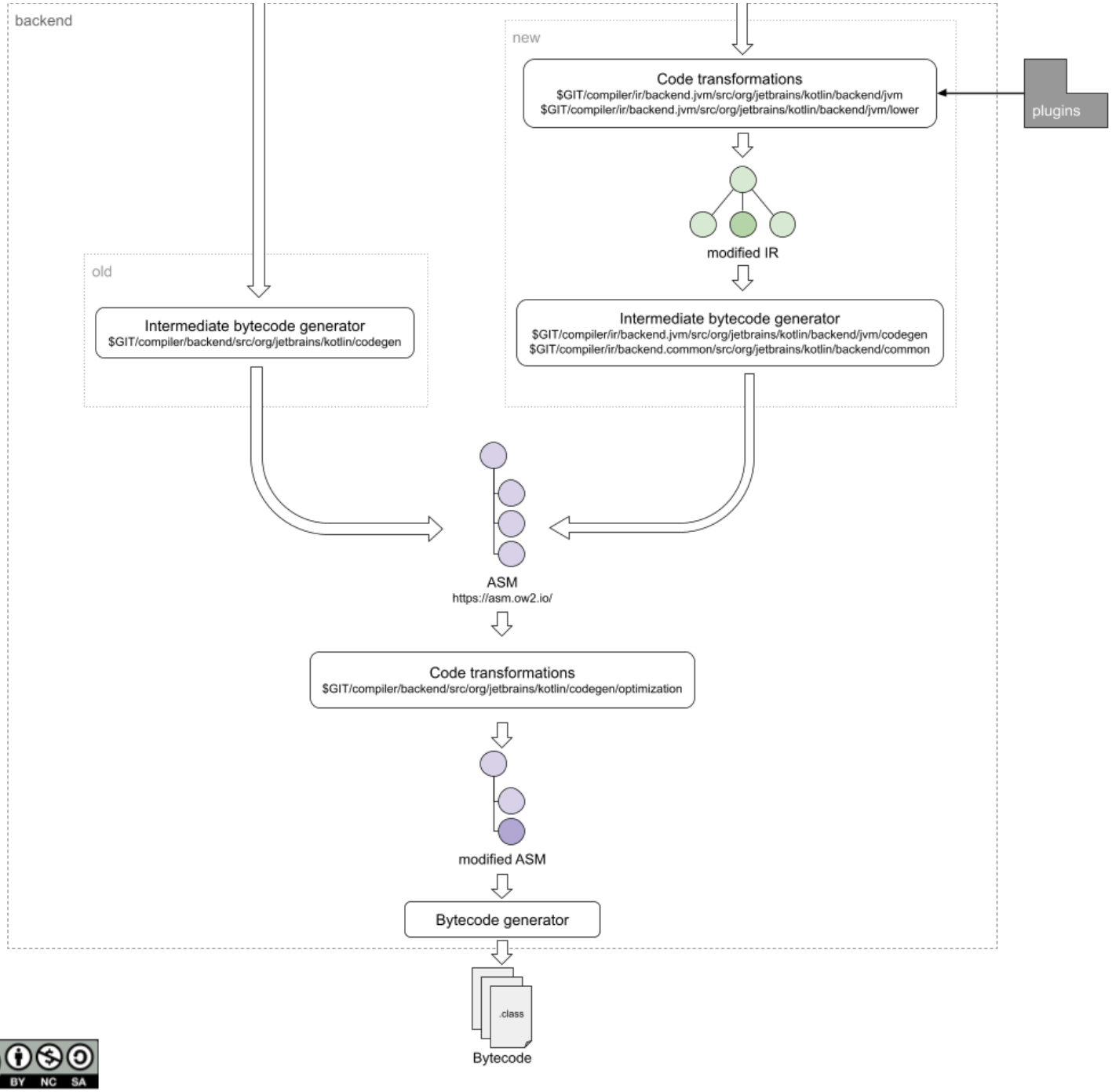
```
fun hello(user: String) = "Hello, $user"
```



BACKEND

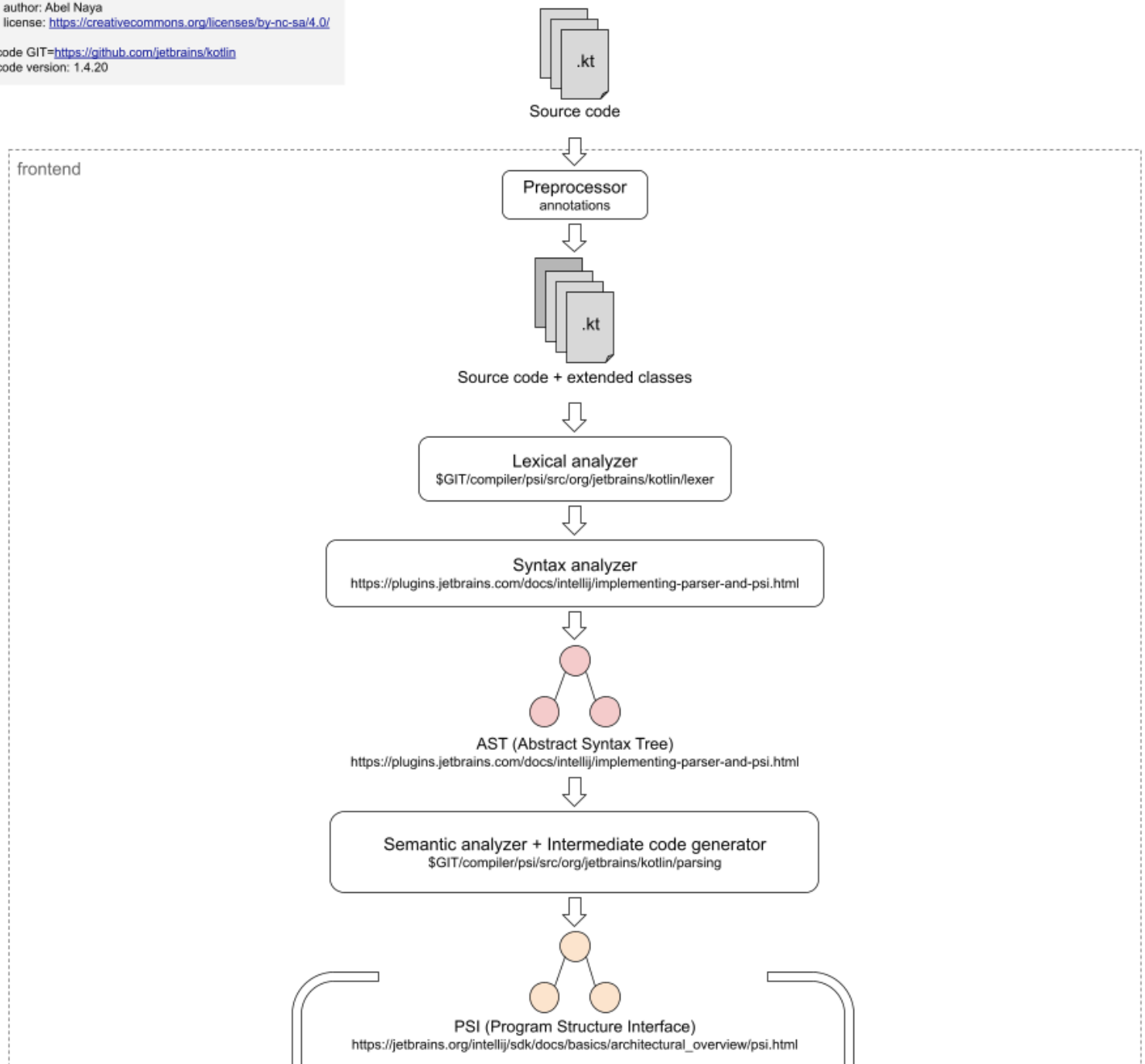
Kotlin compiler




backend
new
Code transformations
$GIT/compiler/ir/backend.jvm/src/org/jetbrains/kotlin/backend/jvm
$GIT/compiler/ir/backend.jvm/src/org/jetbrains/kotlin/backend/jvm/lower
plugins
modified IR
old
Intermediate bytecode generator
$GIT/compiler/backend/src/org/jetbrains/kotlin/codegen
Intermediate bytecode generator
$GIT/compiler/ir/backend.jvm/src/org/jetbrains/kotlin/backend/jvm/codegen
$GIT/compiler/ir/backend.common/src/org/jetbrains/kotlin/backend/common
ASM
https://asm.ow2.io/
Code transformations
$GIT/compiler/backend/src/org/jetbrains/kotlin/codegen/optimization
modified ASM
Bytecode generator
.class
Bytecode

PREPROCESSORS

Source code GIT=https://github.com/jetbrains/kotlin
Source code version: 1.4.20



COMPILER PLUGINS

- Permitem modificar a IR em tempo de compilação

- Permitem modificar a IR em tempo de compilação
  - Frontend Plugins alteram a FIR

- Permitem modificar a IR em tempo de compilação
  - Frontend Plugins alteram a FIR
    - Estender o Type System

- Permitem modificar a IR em tempo de compilação
  - Frontend Plugins alteram a FIR
    - Estender o Type System
    - Realizar uma análise de código específica para a sua situação

- Permitem modificar a IR em tempo de compilação
  - Frontend Plugins alteram a FIR
    - Estender o Type System
    - Realizar uma análise de código específica para a sua situação
    - Emitir erros em tempo de compilação que façam sentido para o seu projeto

- Permitem modificar a IR em tempo de compilação
  - Frontend Plugins alteram a FIR
    - Estender o Type System
    - Realizar uma análise de código específica para a sua situação
    - Emitir erros em tempo de compilação que façam sentido para o seu projeto
  - Backend Plugins alteram a IR (ou backend IR)

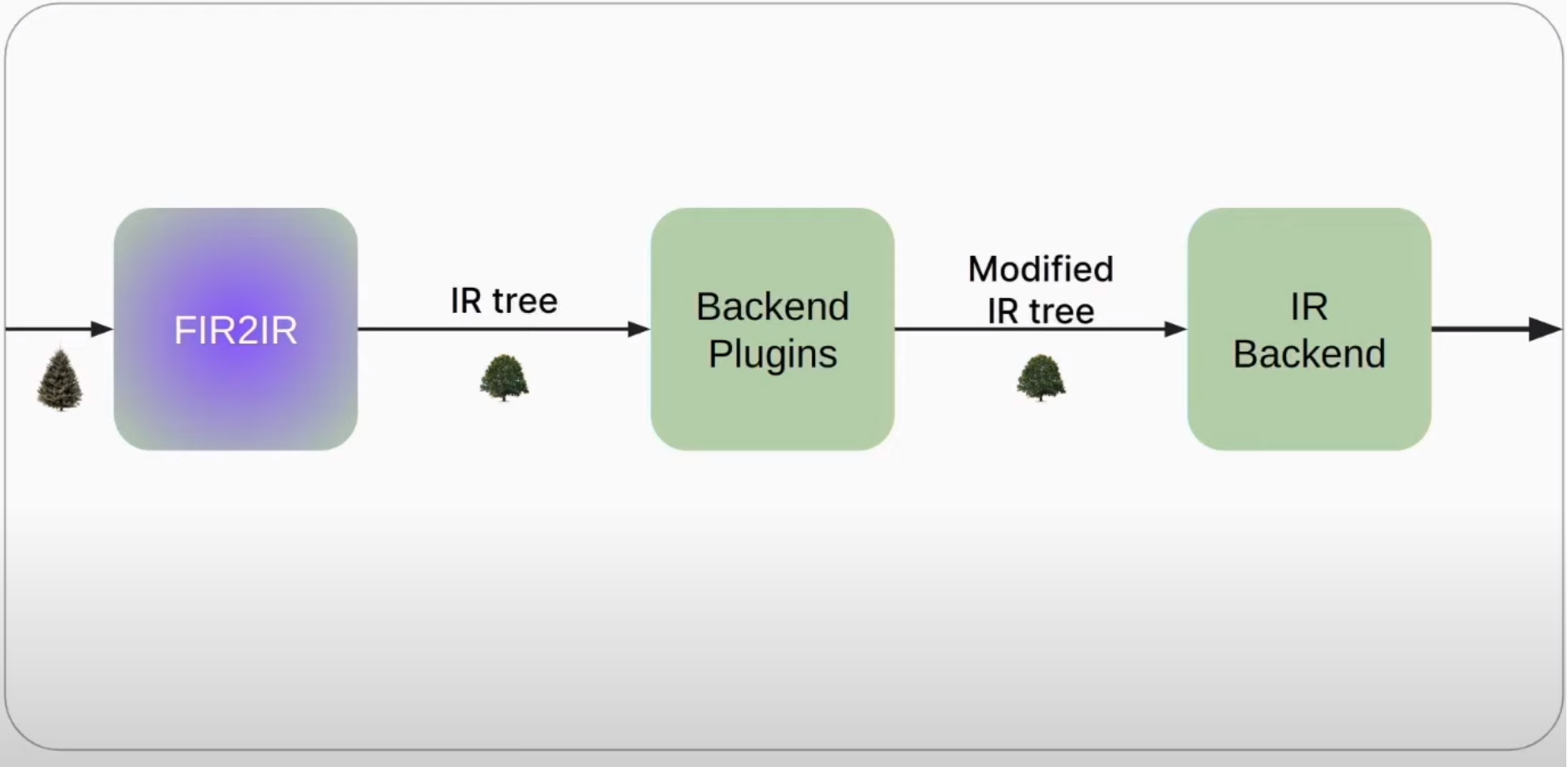
- Permitem modificar a IR em tempo de compilação
  - Frontend Plugins alteram a FIR
    - Estender o Type System
    - Realizar uma análise de código específica para a sua situação
    - Emitir erros em tempo de compilação que façam sentido para o seu projeto
  - Backend Plugins alteram a IR (ou backend IR)
    - Modificar todas as ocorrências de uma classe

- Reduzir boilerplate

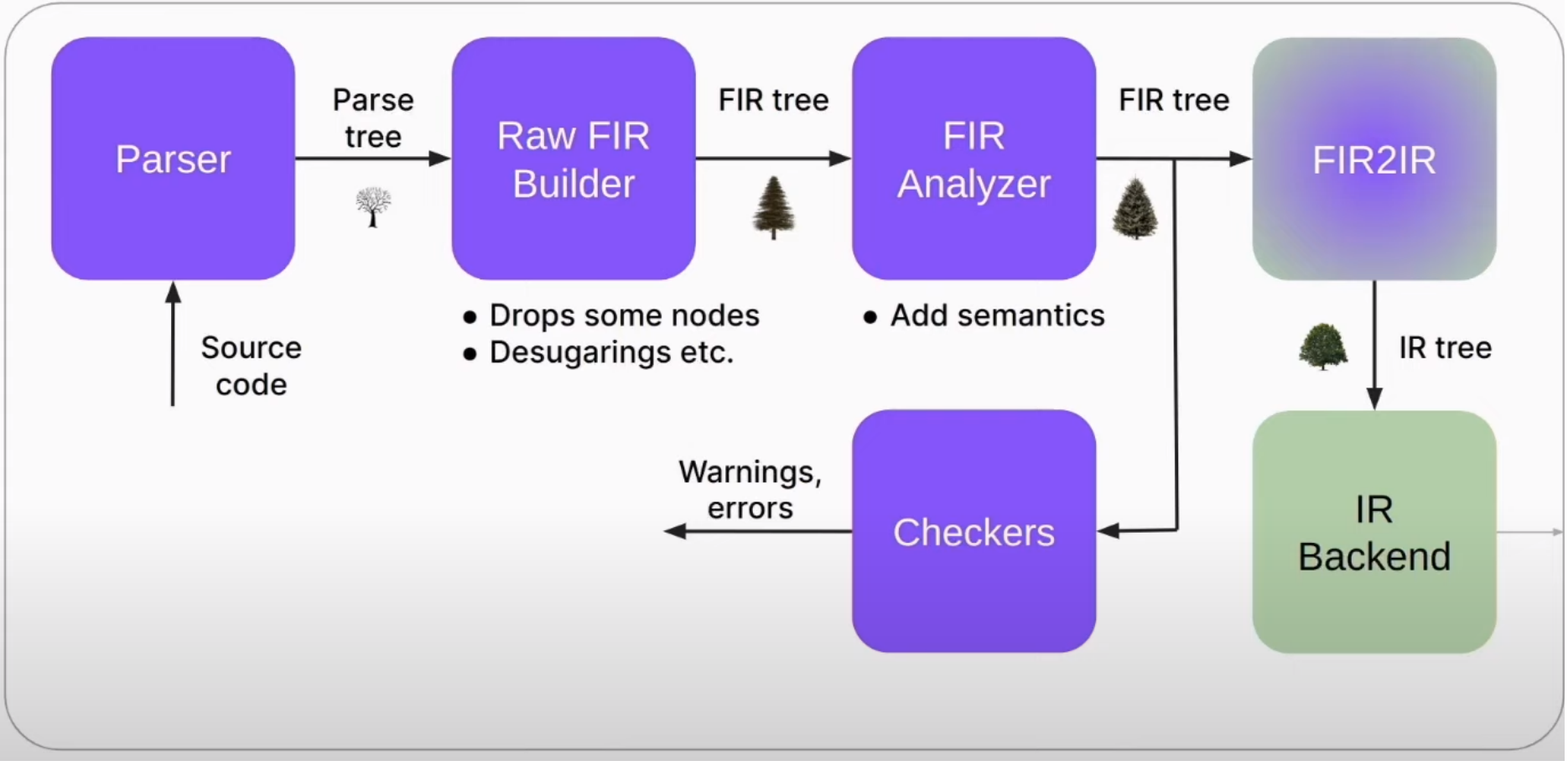
- Reduzir boilerplate
- Melhorar o uso de DSLs

- Reduzir boilerplate
- Melhorar o uso de DSLs
- Realizar tarefas complexas que o compilador não realiza por padrão

Backend plugins



K2 Frontend structure



PERGUNTAS?

Muito Obrigado!

REFERÊNCIAS E MATERIAIS ADICIONAIS

1. discussions and diagrams about the Kotlin compiler part 1
2. discussions and diagrams about the Kotlin compiler part 2
3. Kotlin Compiler Internals
4. What Everyone Must Know About The NEW Kotlin K2 Compiler
5. Kotlin Compiler Crash Course repository
6. Exploring Kotlin IR blog post
7. Kotlin Power Assert - a Kotlin compiler plugin
8. K2 Compiler plugins by Mikhail Glukhikh

9. all-open compiler plugin