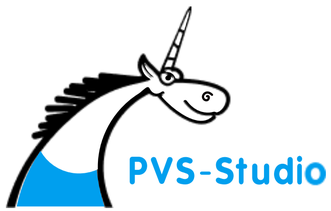

PVS-Studio

Что внутри фронтенда компилятора

И как это использовать



Константин Волоховский

Java Analyzer Lead



Константин Волоховский

Java Analyzer Lead



- Руковожу разработкой Java анализатора в PVS-Studio
- Внедрял taint в анализатор
- Веду блог и рассказываю про программирование, ООП и статические анализаторы

@kvolokhovskii

Из чего состоит компилятор

- Frontend
  - Работает с .java файлами
  - Синтаксический анализ
  - Семантический анализ
- Middle-end
  - Оптимизации исходного кода
  - В graal оптимизации делаются над IR
- Backend
  - Генерация байткода

Из чего состоит компилятор

- Frontend
  - Работает с .java файлами
  - Синтаксический анализ
  - Семантический анализ
- Middle-end
  - Оптимизации исходного кода
  - В graal оптимизации делаются над IR
- Backend
  - Генерация байткода

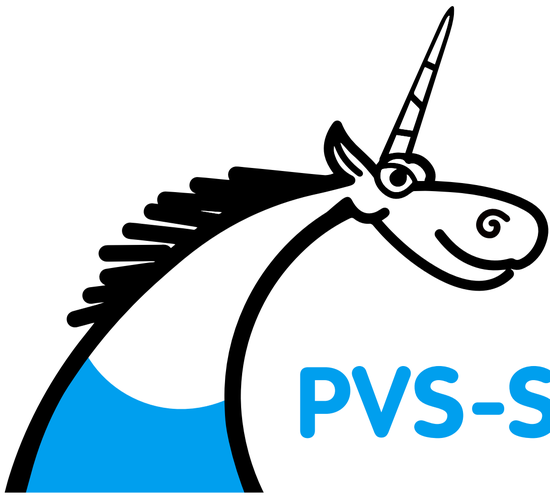
Из чего состоит компилятор

- Frontend
  - Работает с .java файлами
  - Синтаксический анализ
  - Семантический анализ

- Благодаря нему можно сделать:
  - Поиск code smell, нарушений кодстайла
  - Подсветка синтаксиса, навигация по коду
  - Поиск ошибок (статический анализ) и уязвимостей (SAST)

Что такое статический анализатор

- Анализирует код на возможные ошибки:
  - Опечатки
  - Логические
  - Стиль/качество
  - Безопасность (SAST)
- Работает с исходным кодом, не требуя его выполнения



Что такое статический анализатор

- Анализирует код на возможные ошибки:
  - Опечатки
  - Логические
  - Стиль/качество
  - Безопасность (SAST)
- Работает с исходным кодом, не требуя его выполнения




Какую вакцину тебе сделали?
Мне сделали SQL-инъекцию
Что там написано?
or |=|; drop table unicorns

Как выглядит инъекция


```
public static void read(ServletRequest request) {
    var bar = request.getParameter("bar");
    var query = "SELECT * FROM foo WHERE bar = '" + bar + "'";
    var st = conn.createStatement();
    var rs = st.executeQuery(query);
    while (rs.next()) {
        System.out.println(rs.getString("baz"));
    }
}
```

Как выглядит инъекция


```
public static void read(ServletRequest request) {
    var bar = request.getParameter("bar");
    var query = "SELECT * FROM foo WHERE bar = '" + bar + "'";
    var st = conn.createStatement();
    var rs = st.executeQuery(query);
    while (rs.next()) {
        System.out.println(rs.getString("baz"));
    }
}
```

Как выглядит инъекция


```
public static void read(ServletRequest request) {
    var bar = request.getParameter("bar");
    var query = "SELECT * FROM foo WHERE bar = '" + bar + "'";
    var st = conn.createStatement();
    var rs = st.executeQuery(query);
    while (rs.next()) {
        System.out.println(rs.getString("baz"));
    }
}
```

Как выглядит инъекция


```
public static void read(ServletRequest request) {
    var bar = request.getParameter("bar");
    var query = "SELECT * FROM foo WHERE bar = '" + bar + "'";
    var st = conn.createStatement();
    var rs = st.executeQuery(query);
    while (rs.next()) {
        System.out.println(rs.getString("baz"));
    }
}
```

Как выглядит инъекция


```
public static void read(ServletRequest request) {
    var bar = request.getParameter("bar");
    var query = "SELECT * FROM foo WHERE bar = '" + bar + "'";
    var st = conn.createStatement();
    var rs = st.executeQuery(query);
    while (rs.next()) {
        System.out.println(rs.getString("baz"));
    }
}
```

Немного терминологии

- Источник
  - `var bar = request.getParameter("bar");`
- Сток
  - `var rs = st.executeQuery(query);`
- Санитизация
  - `if (bar.equals("safe")) return;`

Как защищаются от инъекции

- Параметризированные запросы

```
var ps = conn.prepareStatement("SELECT * FROM foo WHERE bar = ?");
ps.setString(1, bar);
var rs = ps.executeQuery();
```

- Белый список

```
if (bar.equals("safe_1") || bar.equals("safe_2")) {
    // ....
}
```

- Чёрный список

```
if (bar.contains(";")) {
    // ....
}
```

Почему не чёрный список?

- Можно сделать список специальных символов для всех диалектов SQL
- Закинуть в одну библиотеку
- И экранировать им любой ввод



PVS-Studio

Как выглядит инъекция


```
public static void read(ServletRequest request) {
    var bar = request.getParameter("bar");
    var query = "SELECT * FROM foo WHERE bar = '" + bar + "'";
    var st = conn.createStatement();
    var rs = st.executeQuery(query);
    while (rs.next()) {
        System.out.println(rs.getString("baz"));
    }
}
```

Как выглядит инъекция


```
public static void read(ServletRequest request) {
    var bar = request.getParameter("bar");
    var query = "SELECT * FROM foo WHERE bar = '" + bar + "'";
    var st = conn.createStatement();
    var rs = st.executeQuery(query);
    while (rs.next()) {
        System.out.println(rs.getString("baz"));
    }
}
```



Как работать с кодом?

- Разбить на лексемы
  - Преобразование кода в последовательность осмысленных единиц - токенов
- Построить абстрактное синтаксическое дерево
  - Формирование дерева, отражающего операции и выражения
- Собрать семантическую информацию
  - Связывание идентификаторов с их объявлениями

Разбор кода

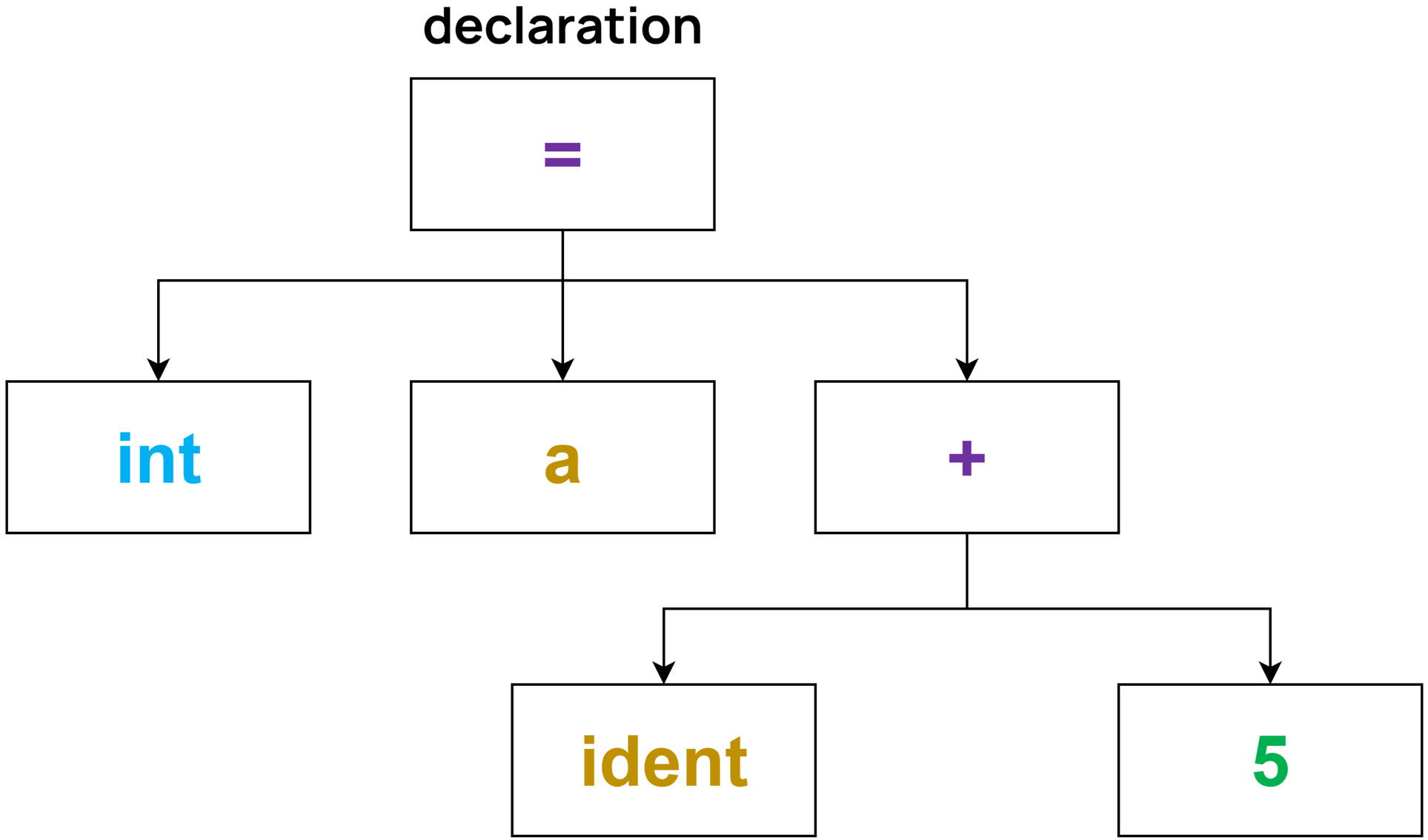

```
int a = ident + 5;
```

Разбор кода

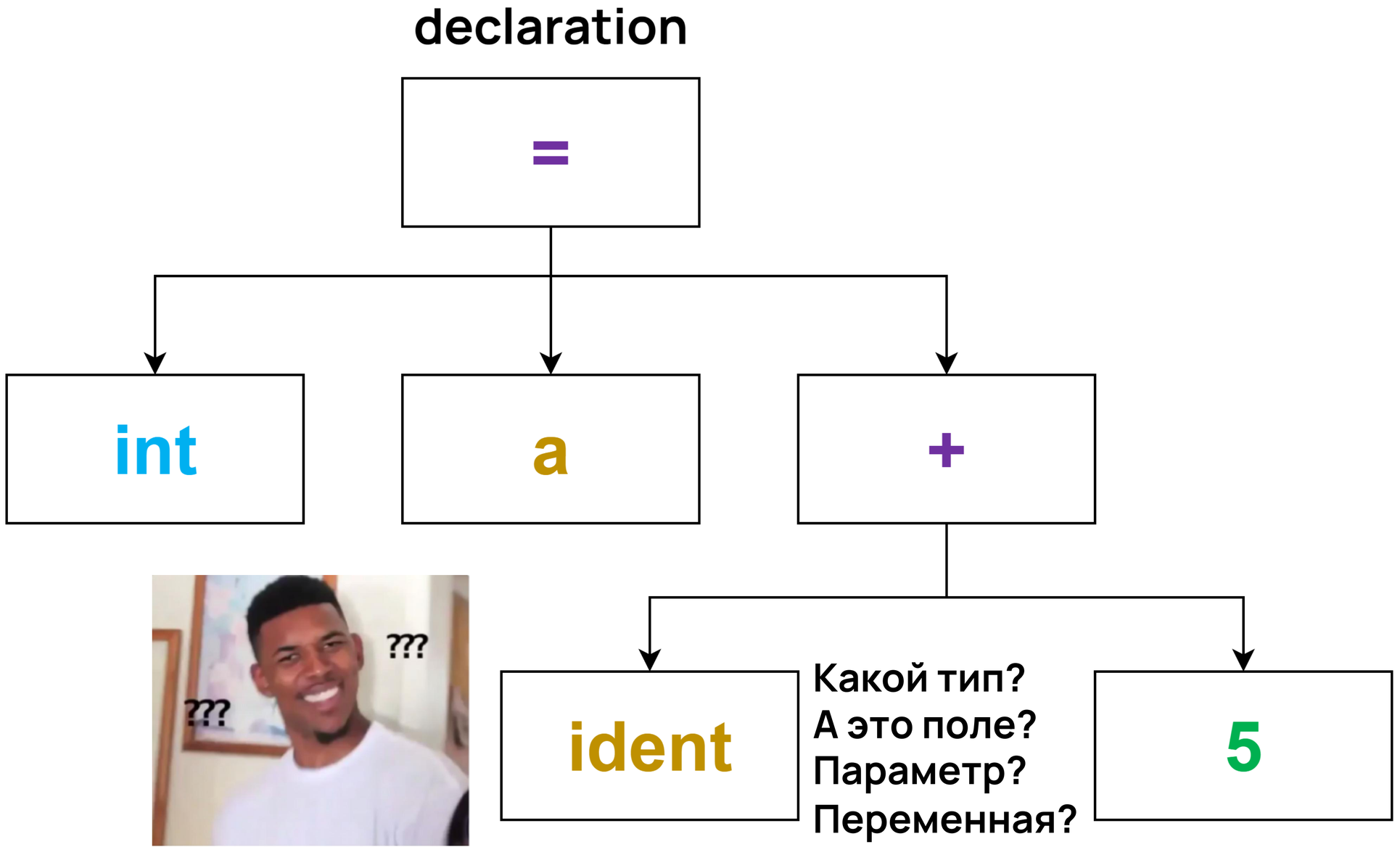

```
int a = ident + 5;
```

- Фиолетовый: оператор
- Коричневый: идентификатор
- Голубой: ключевое слово
- Зелёный: значение

Разбор кода



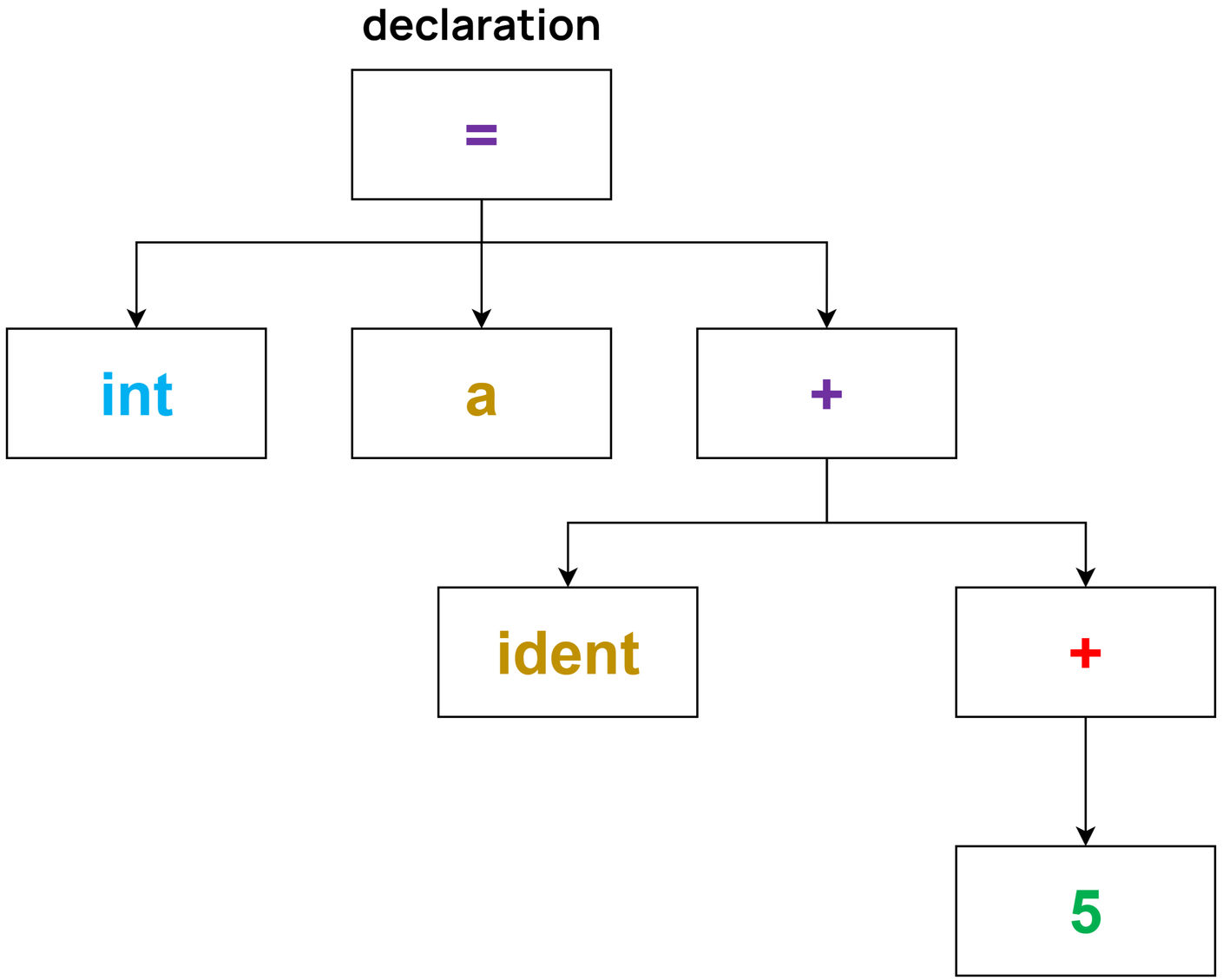
Разбор кода


declaration
=
int
a
+
ident
5
Какой тип?
А это поле?
Параметр?
Переменная?
???
???

Разбор кода


```
int a = ident + 5 + ;
```

Разбор кода



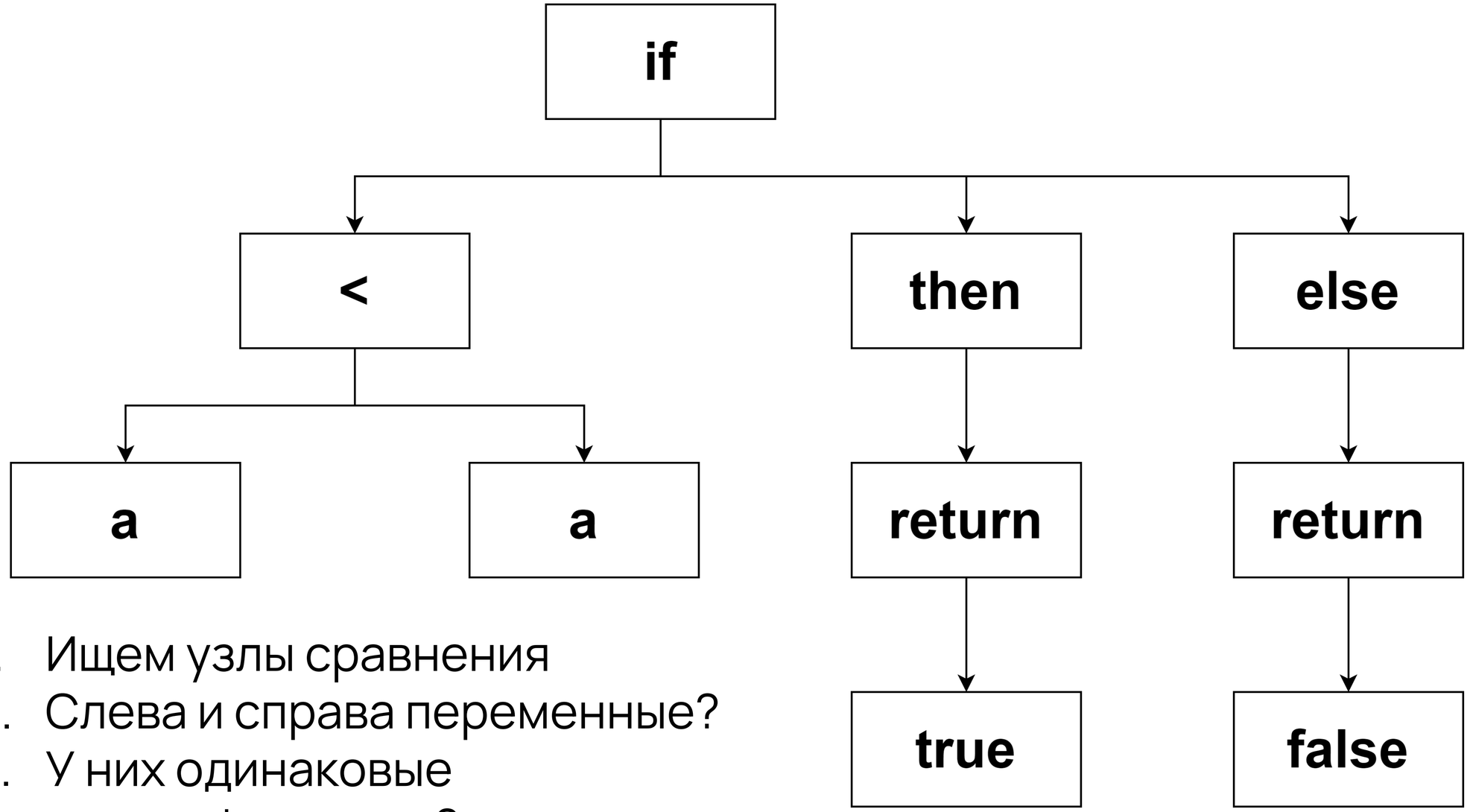
Ищем ошибку в коде


```
private void isLessThan(int a, int b) {
    if (a < a) {
        return true;
    } else {
        return false;
    }
}
```

Ищем ошибку в коде


```
private void isLessThan(int a, int b) {
    if (a < a) {
        return true;
    } else {
        return false;
    }
}
```

Ищем ошибку в коде



1. Ищем узлы сравнения
2. Слева и справа переменные?
3. У них одинаковые идентификаторы?

Как построить AST

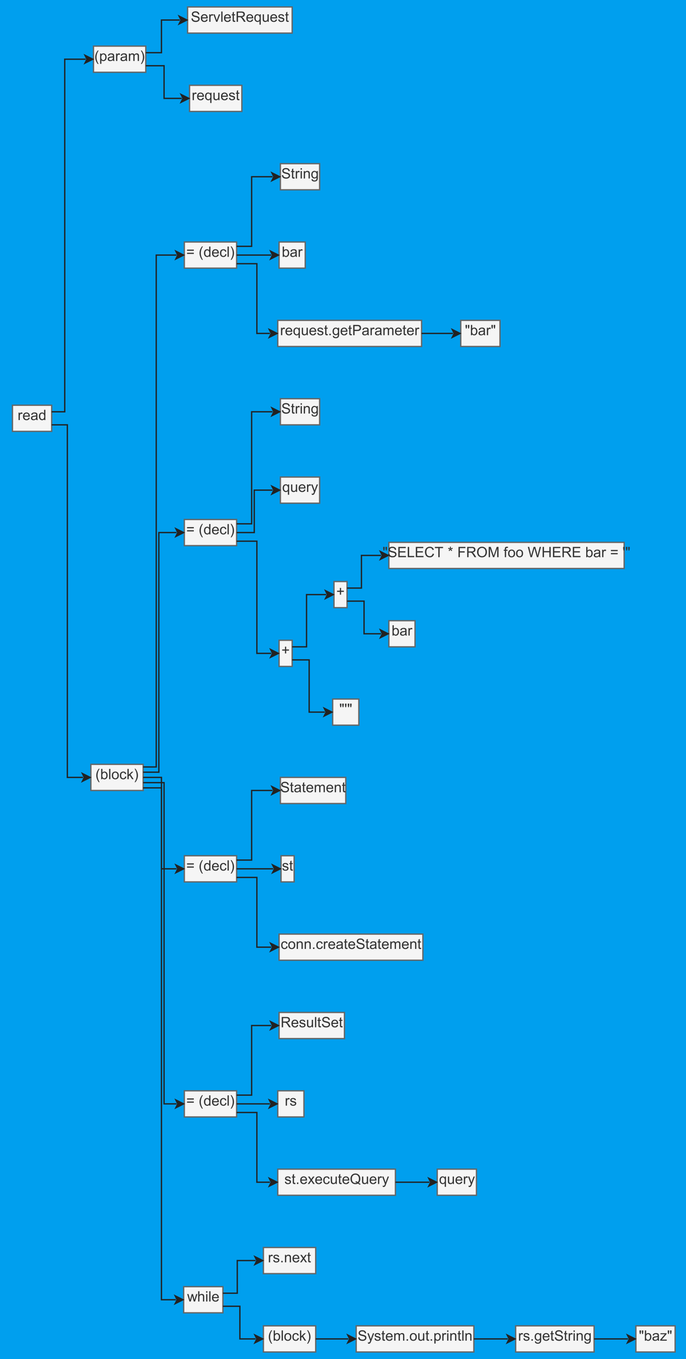
- Генераторы парсеров: ANTLR или Tree-Sitter
  - Нужно задать грамматику, парсер выдаст AST
  - Уже готовые грамматики можно найти
  - Придётся делать свой семантический анализ

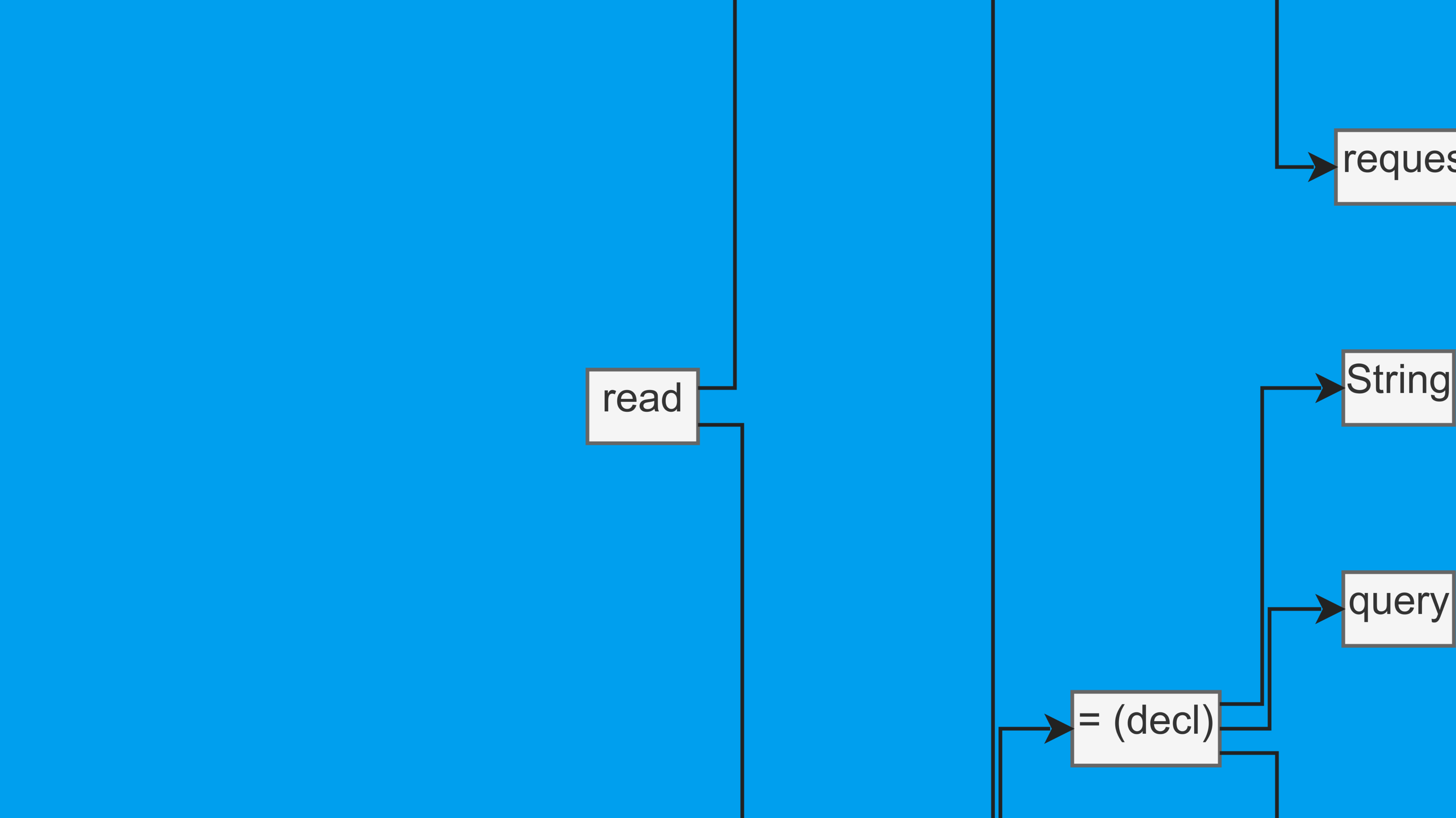
- Использовать уже готовые решения
  - Spoon
  - JavaParser
  - JDT
  - Инфраструктура JDK

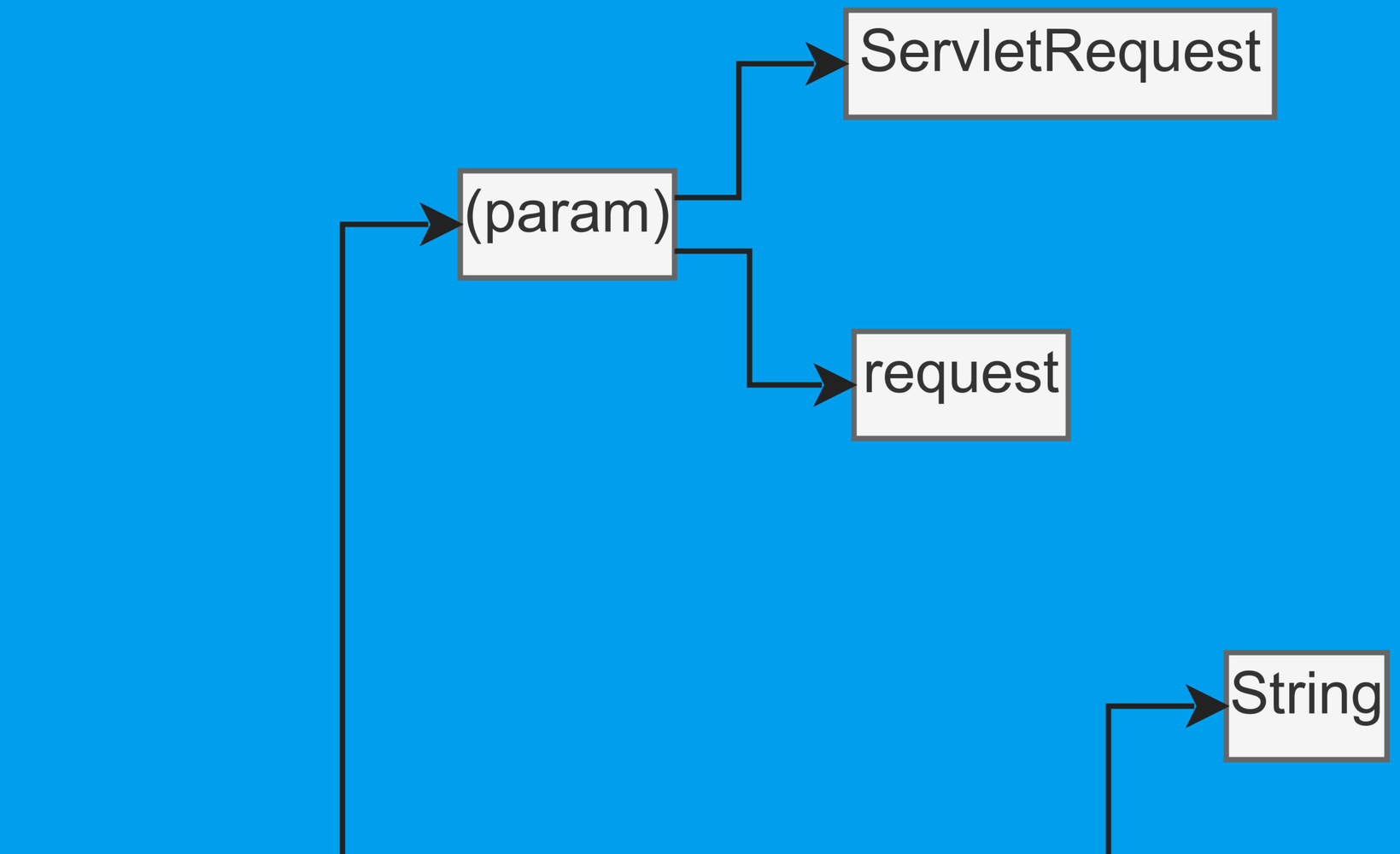
Обратно к коду

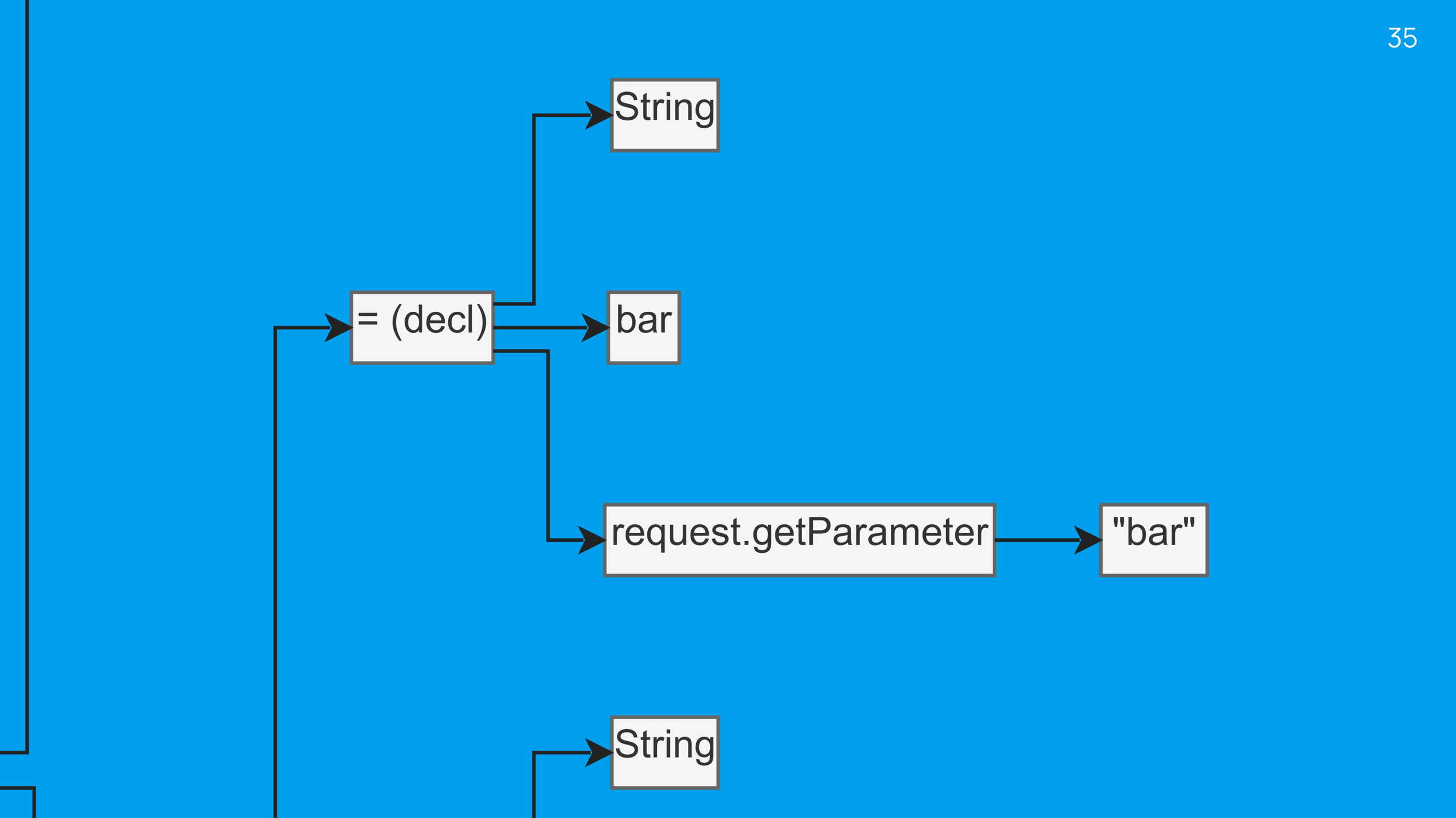

```
public static void read(ServletRequest request) throws SQLException {
    var bar = request.getParameter("bar");
    var query = "SELECT * FROM foo WHERE bar = '" + bar + "'";
    var st = conn.createStatement();
    var rs = st.executeQuery(query);
    while (rs.next()) {
        System.out.println(rs.getString("baz"));
    }
}
```

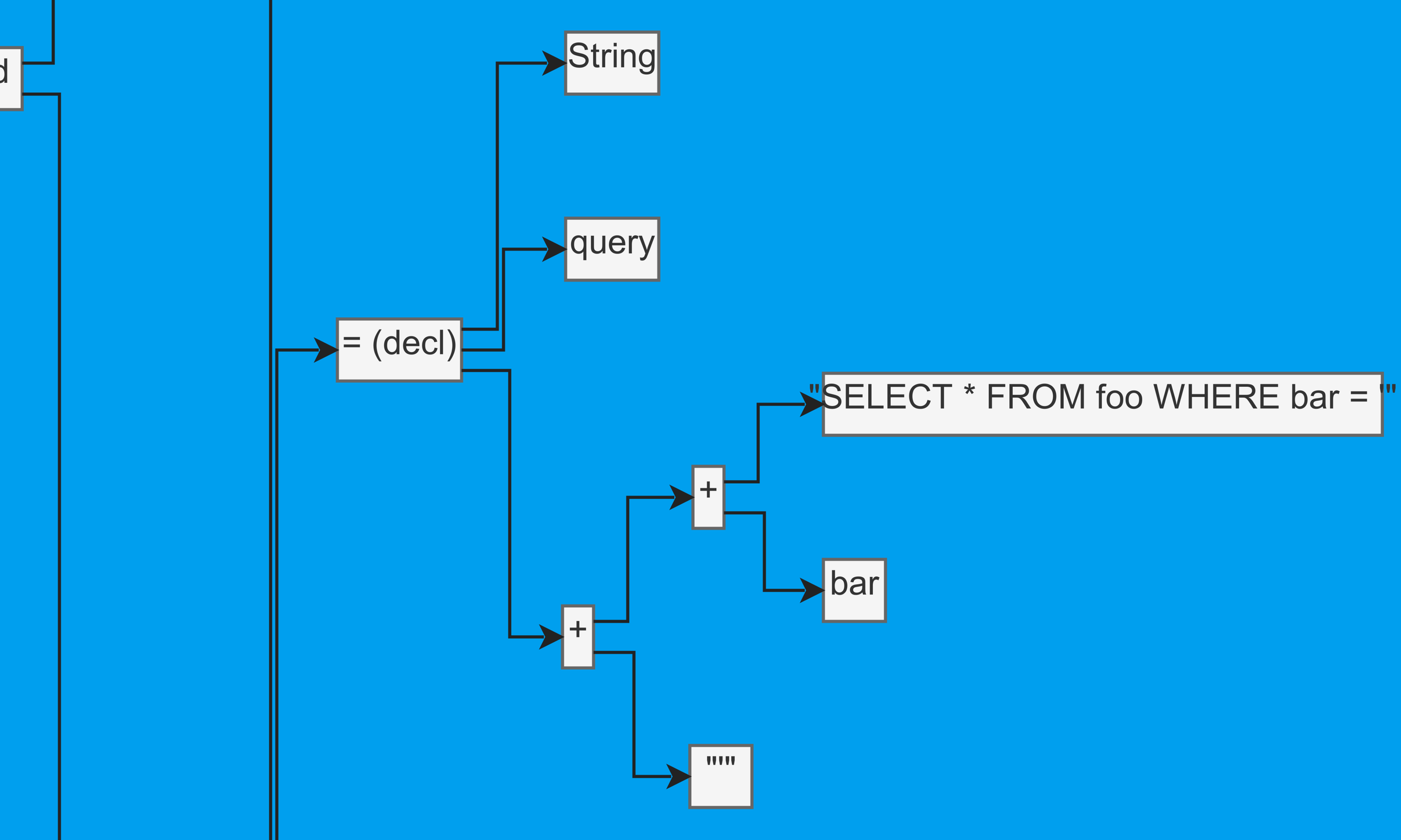
Абстрактное синтаксическое дерево

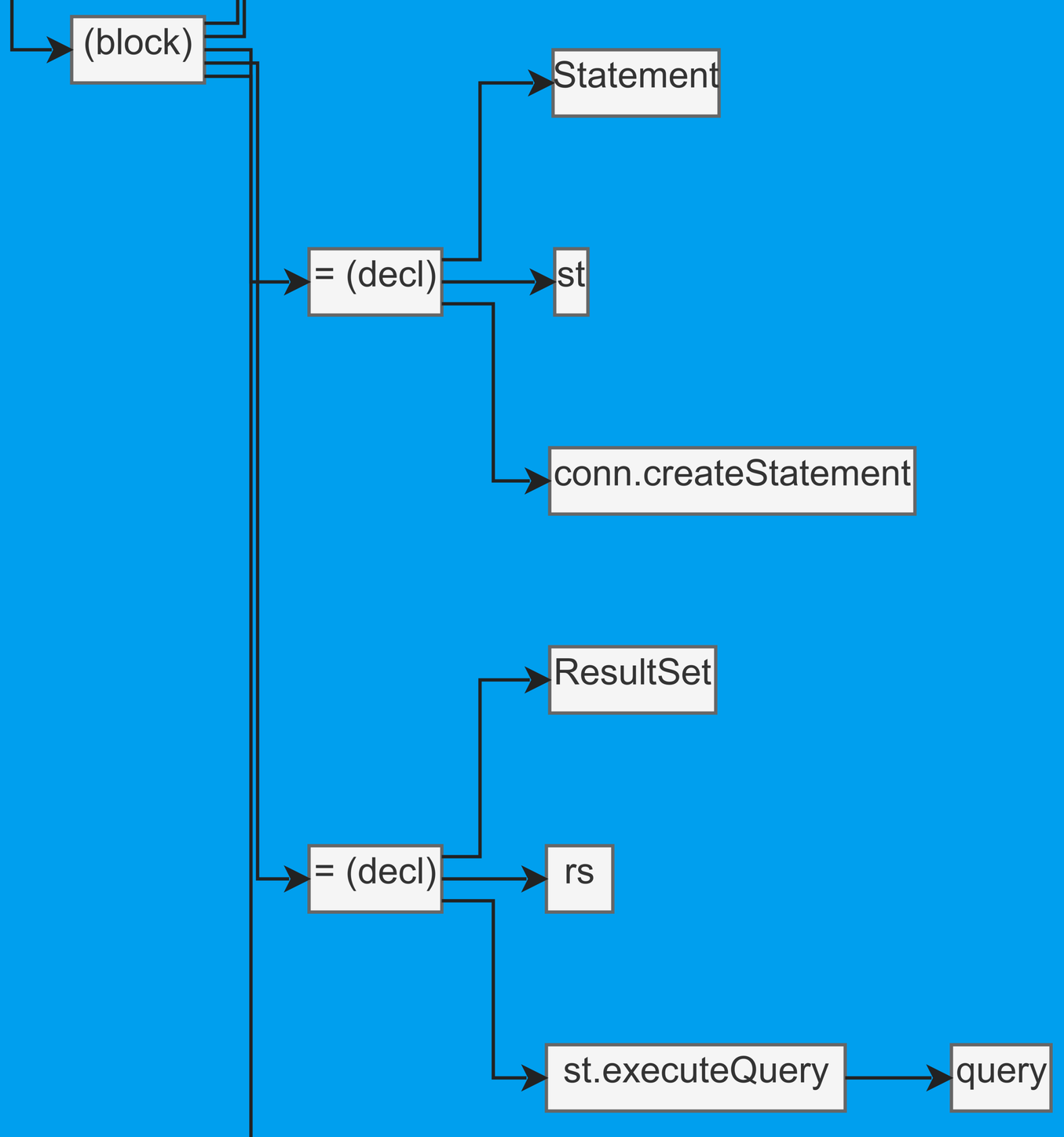


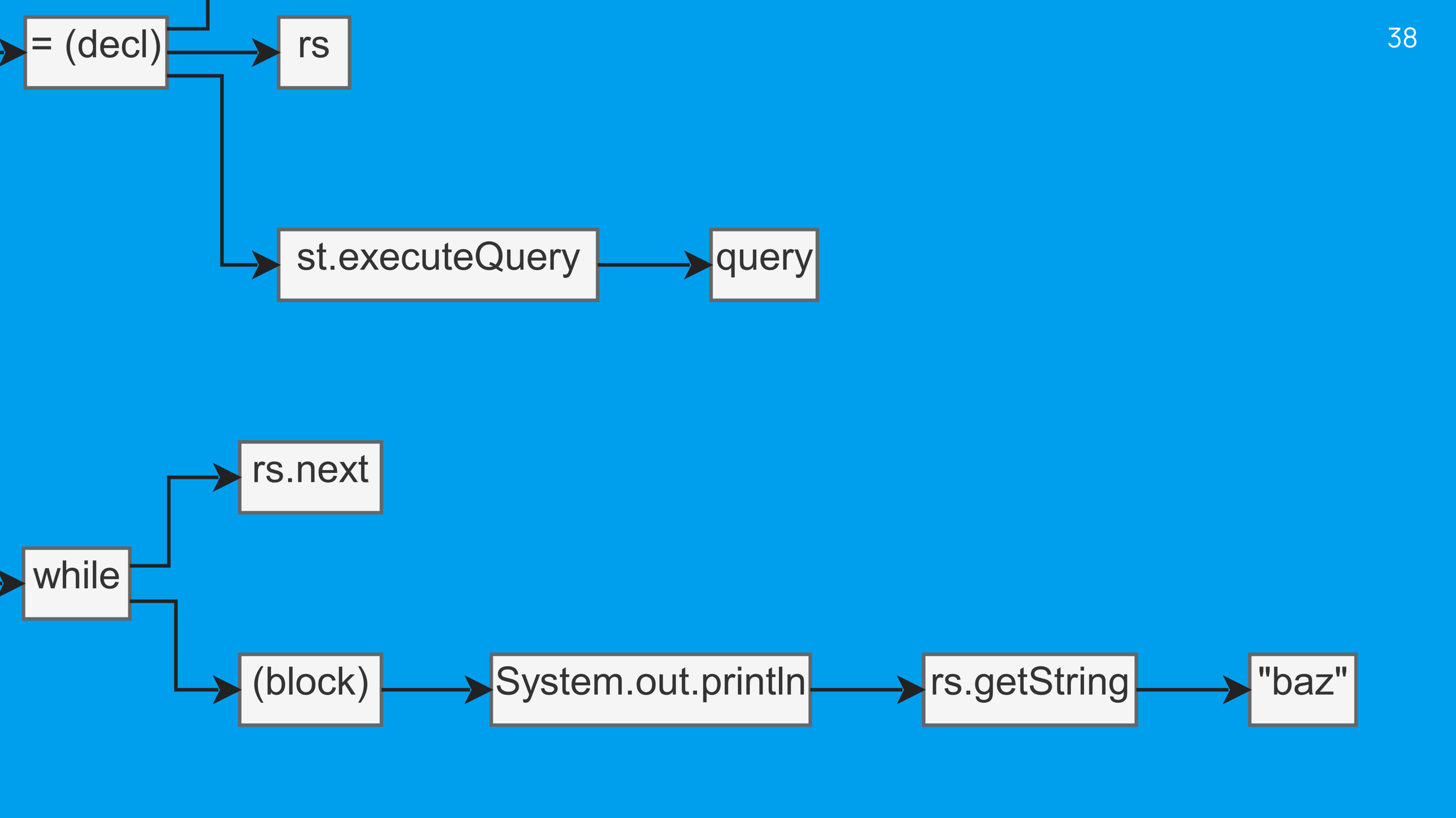

reques
read
String
query
= (decl)


ServletRequest
(param)
request
String


String
= (decl)
bar
request.getParameter
"bar"
String


d
String
query
= (decl)
"SELECT * FROM foo WHERE bar = '"
+
bar
+
""


(block)
Statement
= (decl)
st
conn.createStatement
ResultSet
= (decl)
rs
st.executeQuery
query


= (decl)
rs
st.executeQuery
query
while
rs.next
(block)
System.out.println
rs.getString
"baz"

Абстрактное синтаксическое дерево

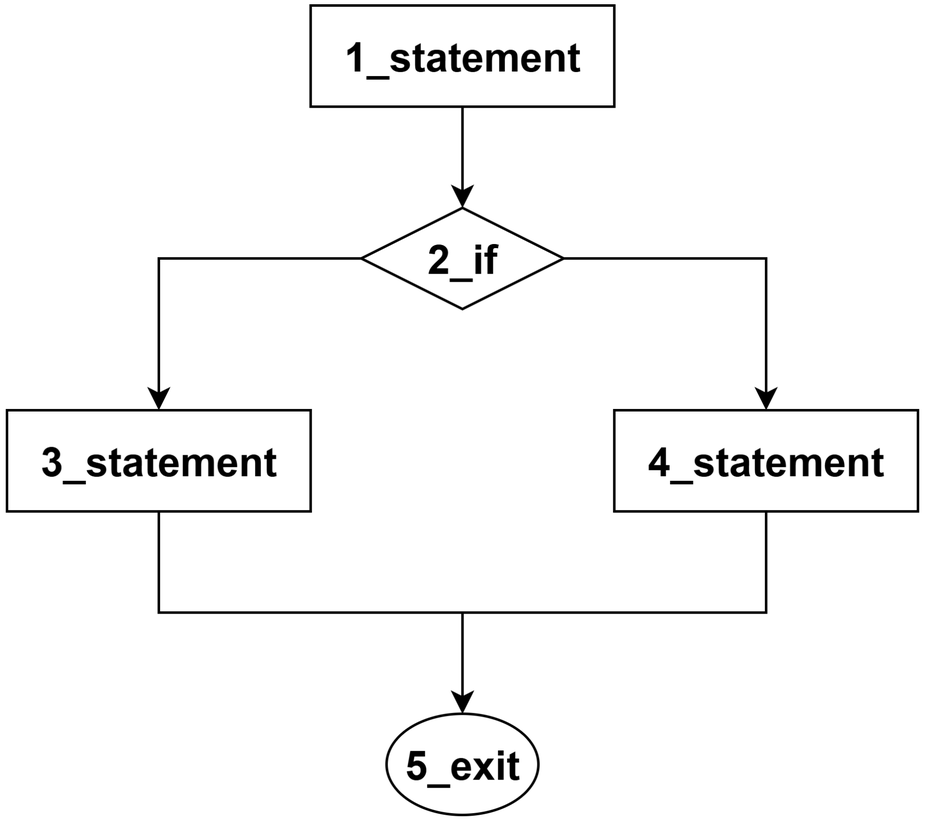
- У нас есть удобная для работы структура и достаточно информации из семантики
- На ней можно находить ошибочные паттерны в коде
- У нас достаточно инструментов, чтобы находить уязвимости

- AST не приспособлен для самой задачи отслеживания потока данных
- Нам нужно что-то, на чём можно будет отслеживать путь от источника к стоку

Отслеживаем поток управления

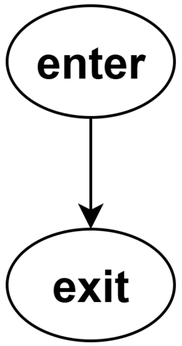
Граф потока управления

- Граф потока управления
  - Показывает, по каким инструкциям идёт выполнение
  - В нём вершины состоят из инструкций, а рёбра – переход управления

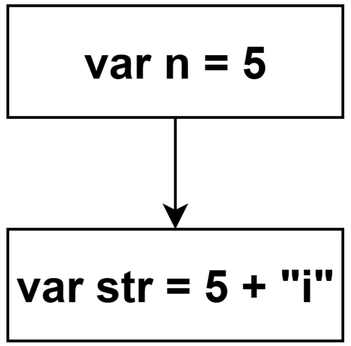


Граф потока управления

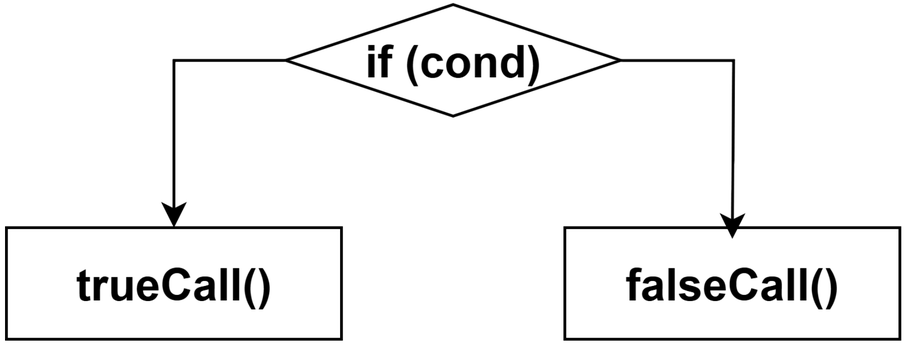
Точки входа и выхода



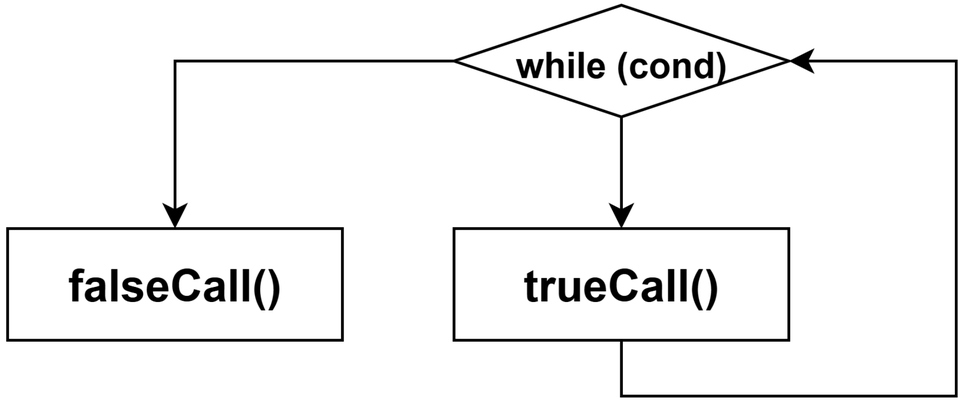
Линейные инструкции



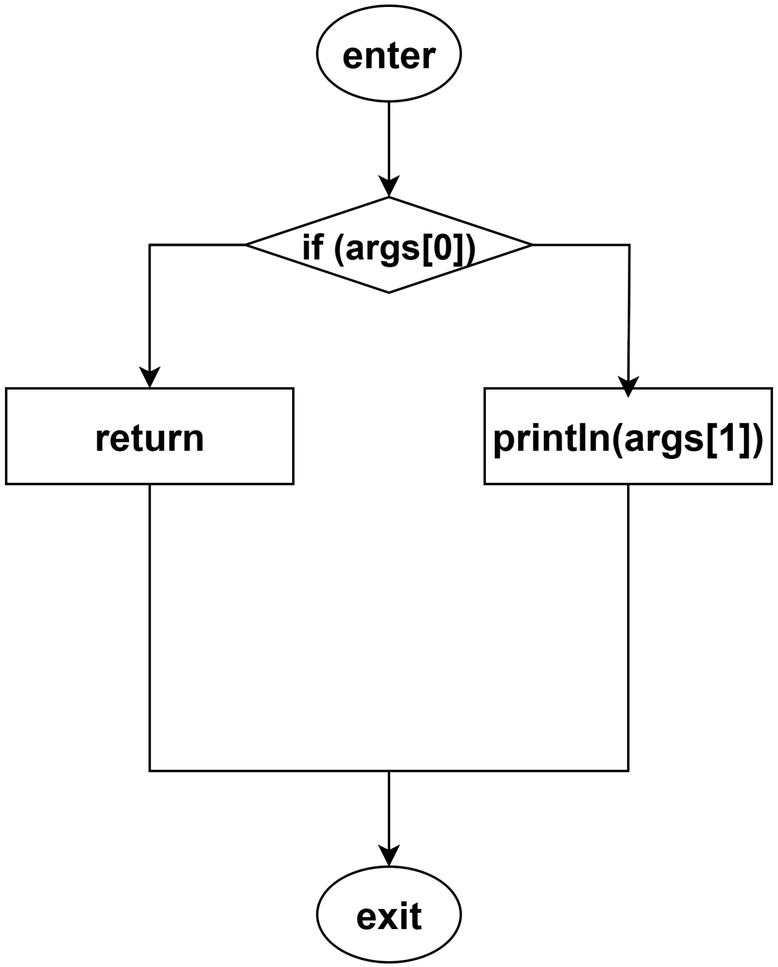
Условия



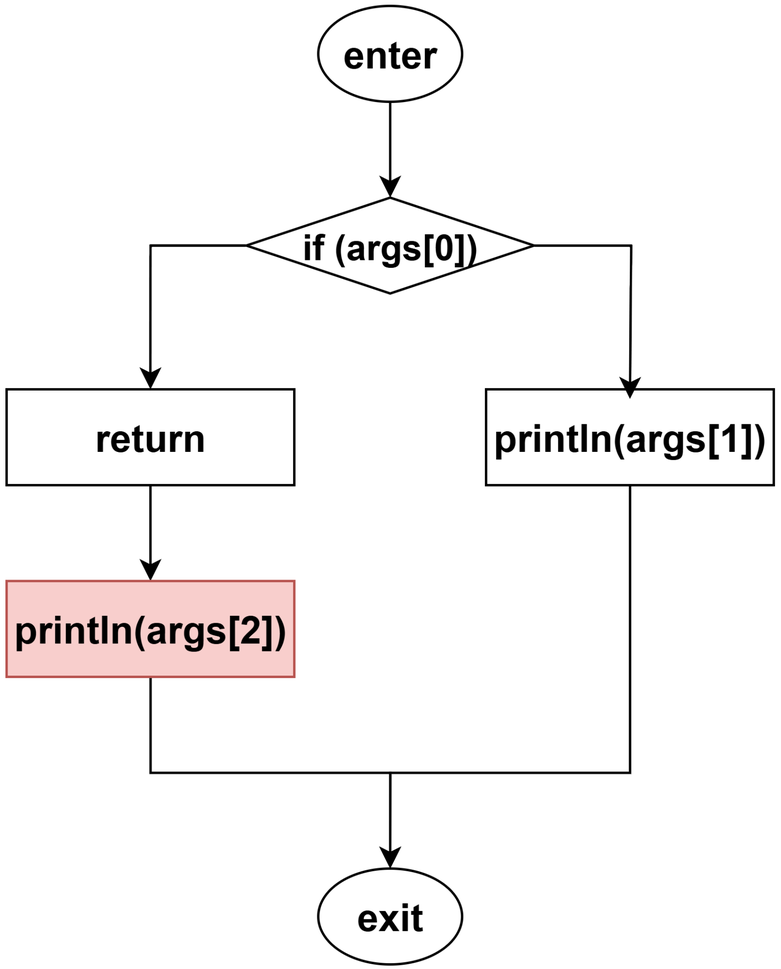
Циклы



Граф потока управления



Граф потока управления



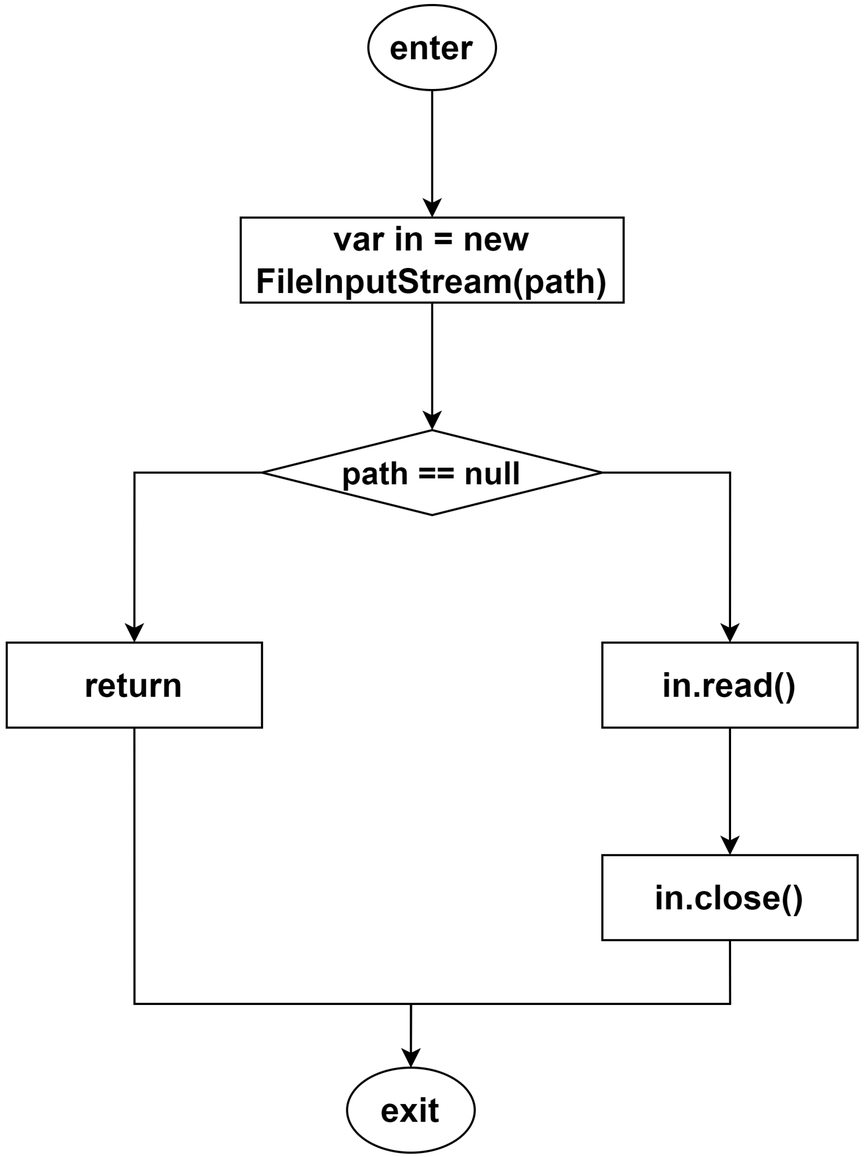
Ищем ошибку в коде


```
void readFile(String path) {
    var in = new FileInputStream(path);
    if (path == null)
        return;
    in.read();
    in.close();
}
```

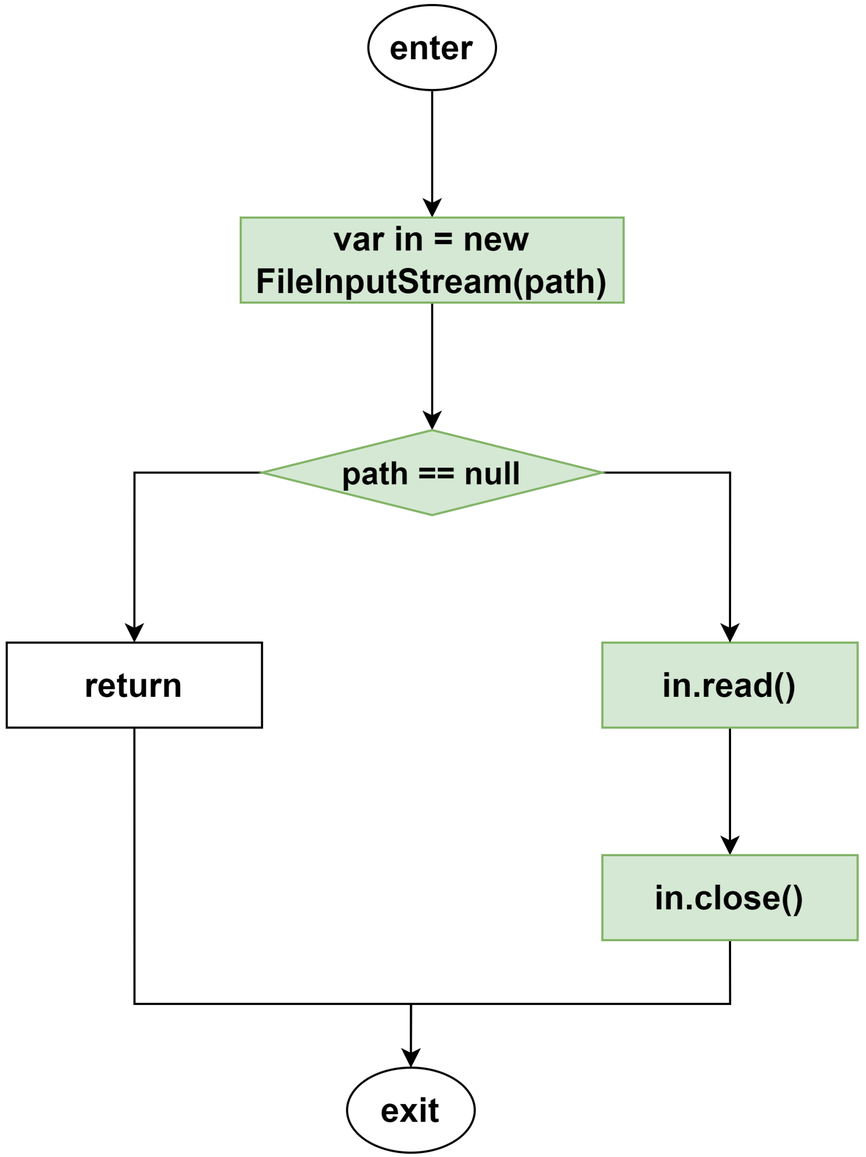
Ищем ошибку в коде


```
void readFile(String path) {
    var in = new FileInputStream(path);
    if (path == null)
        return;
    in.read();
    in.close();
}
```

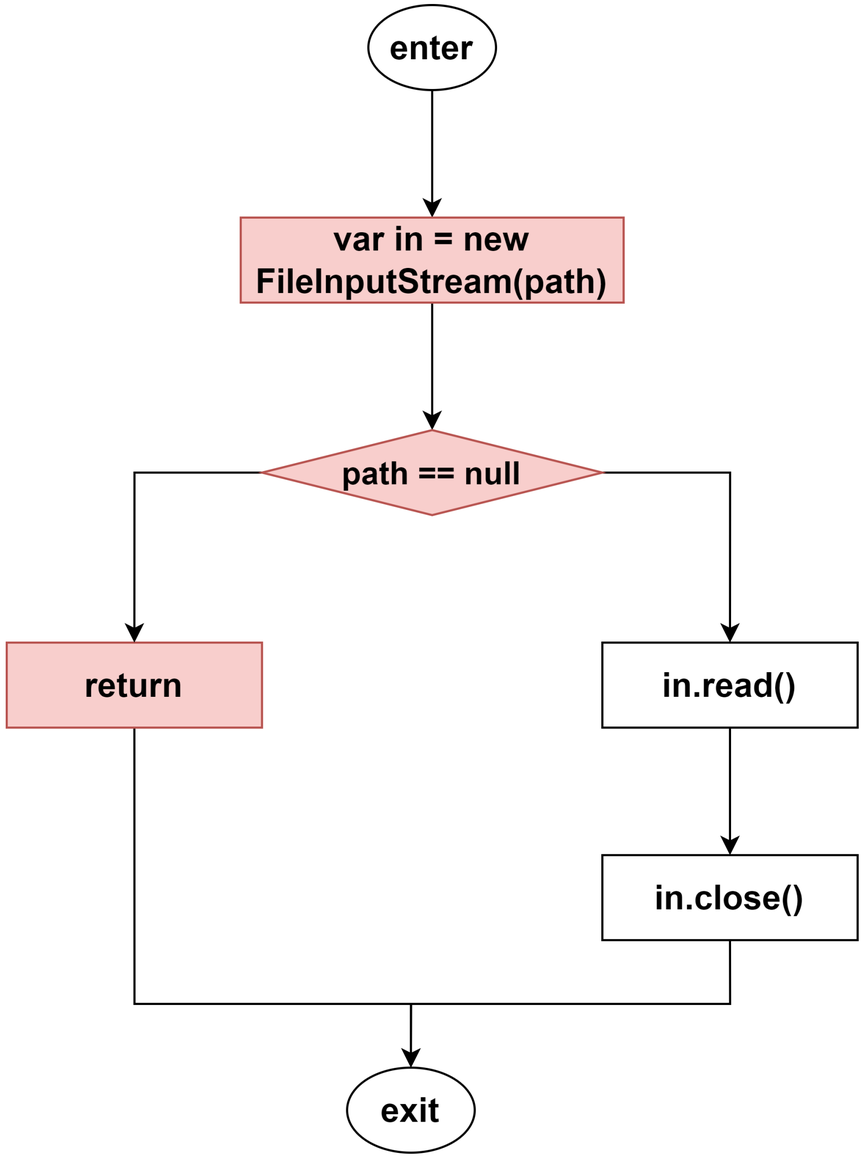
Ищем ошибку в коде



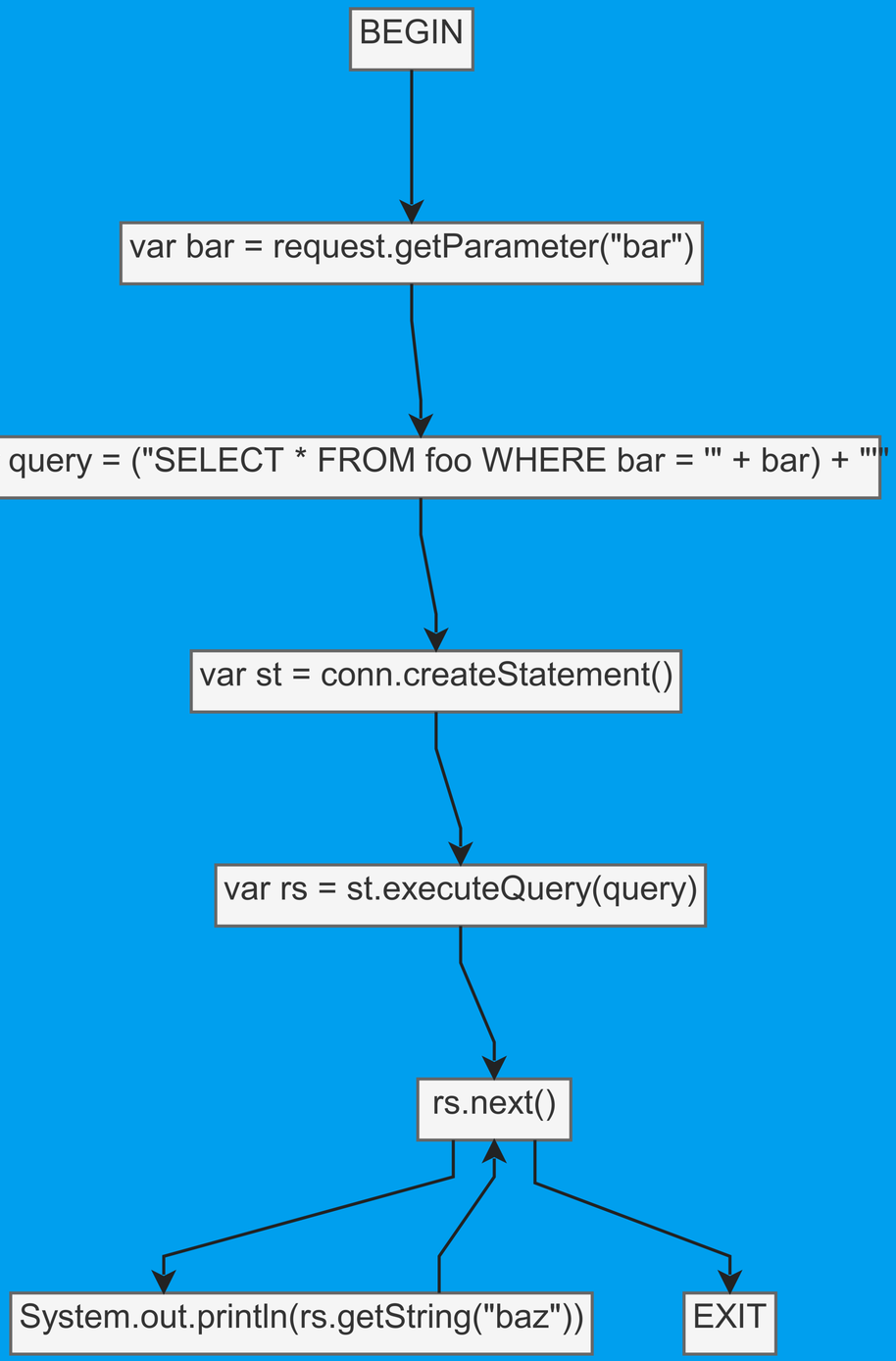
Ищем ошибку в коде

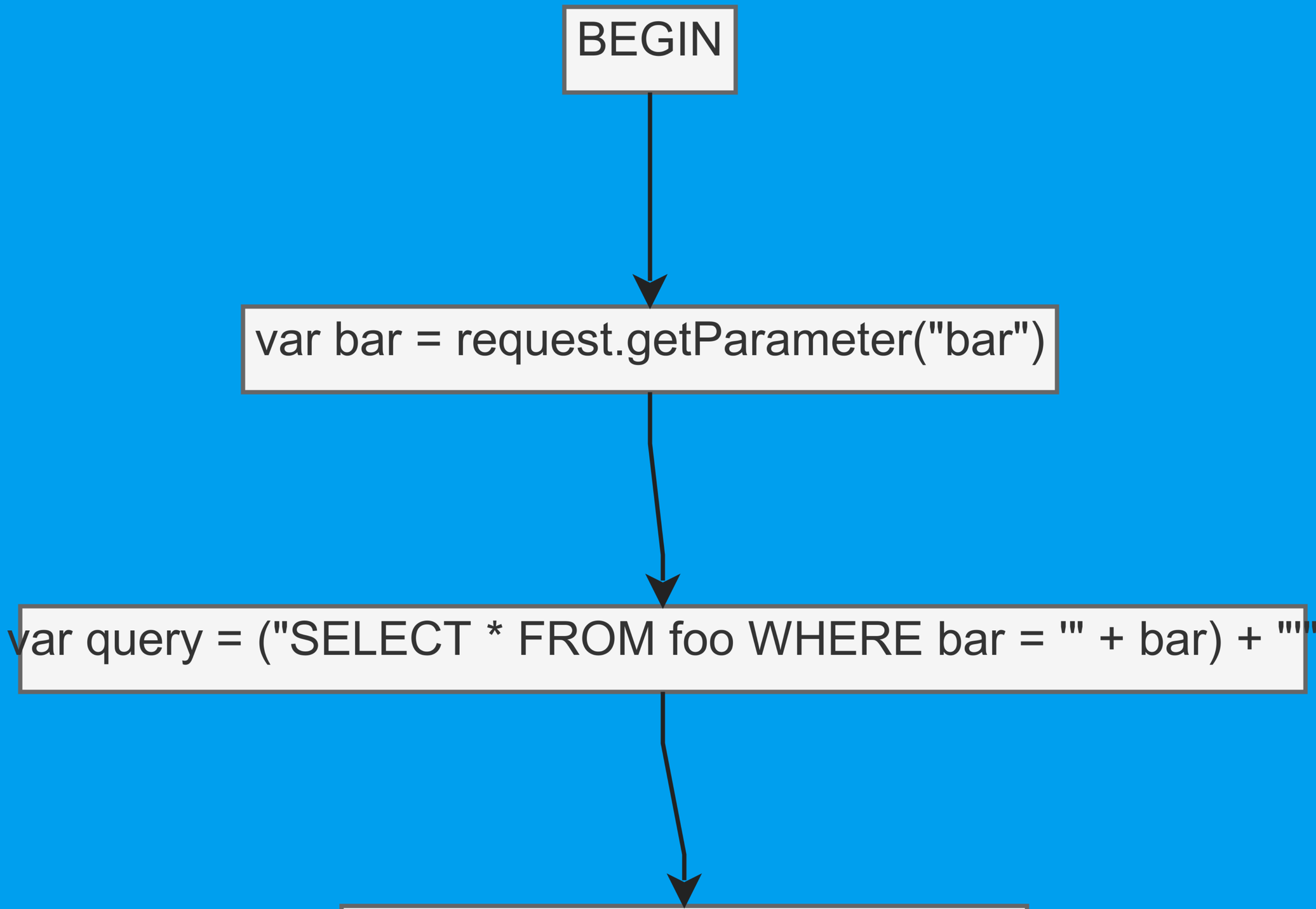


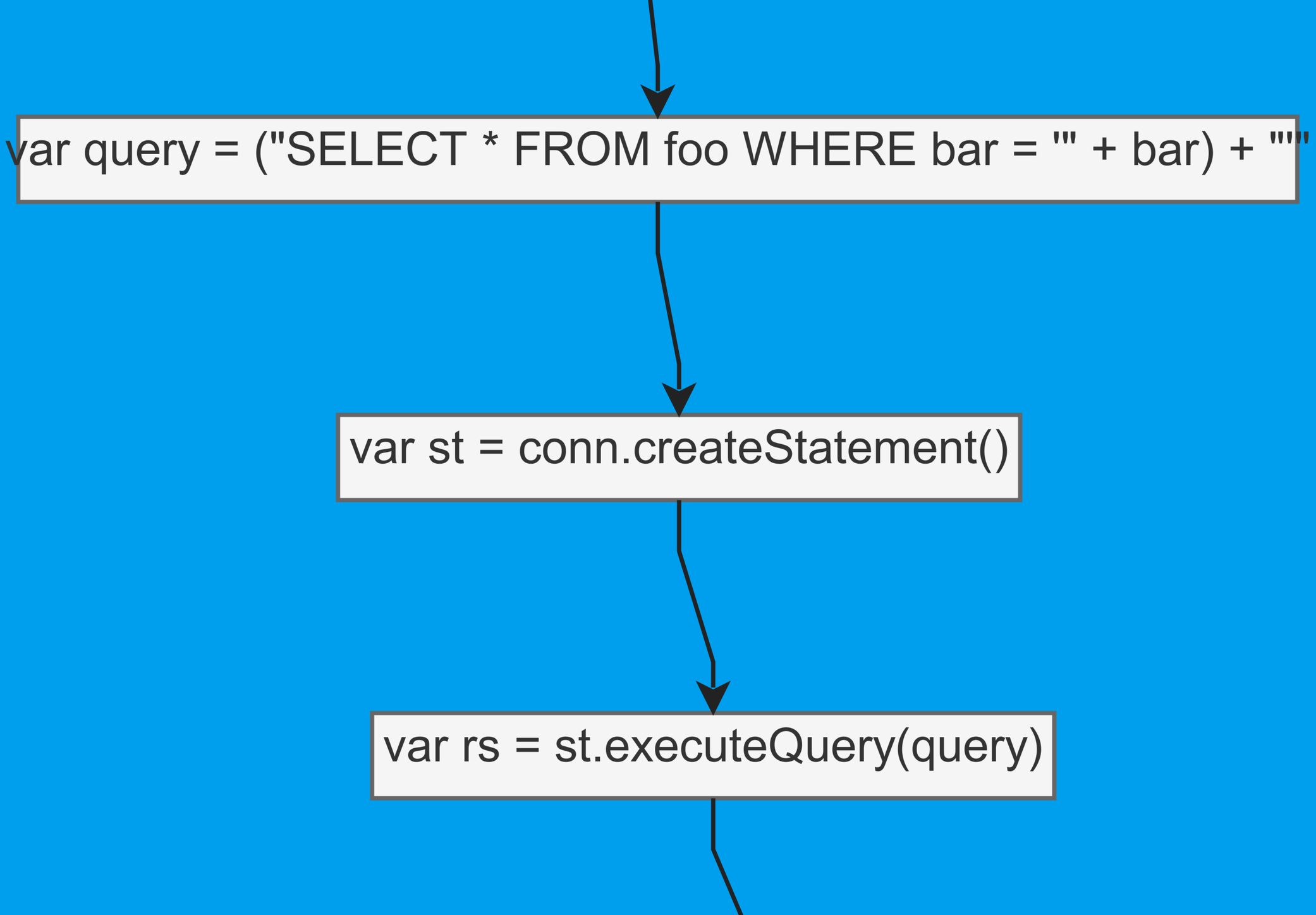
Ищем ошибку в коде

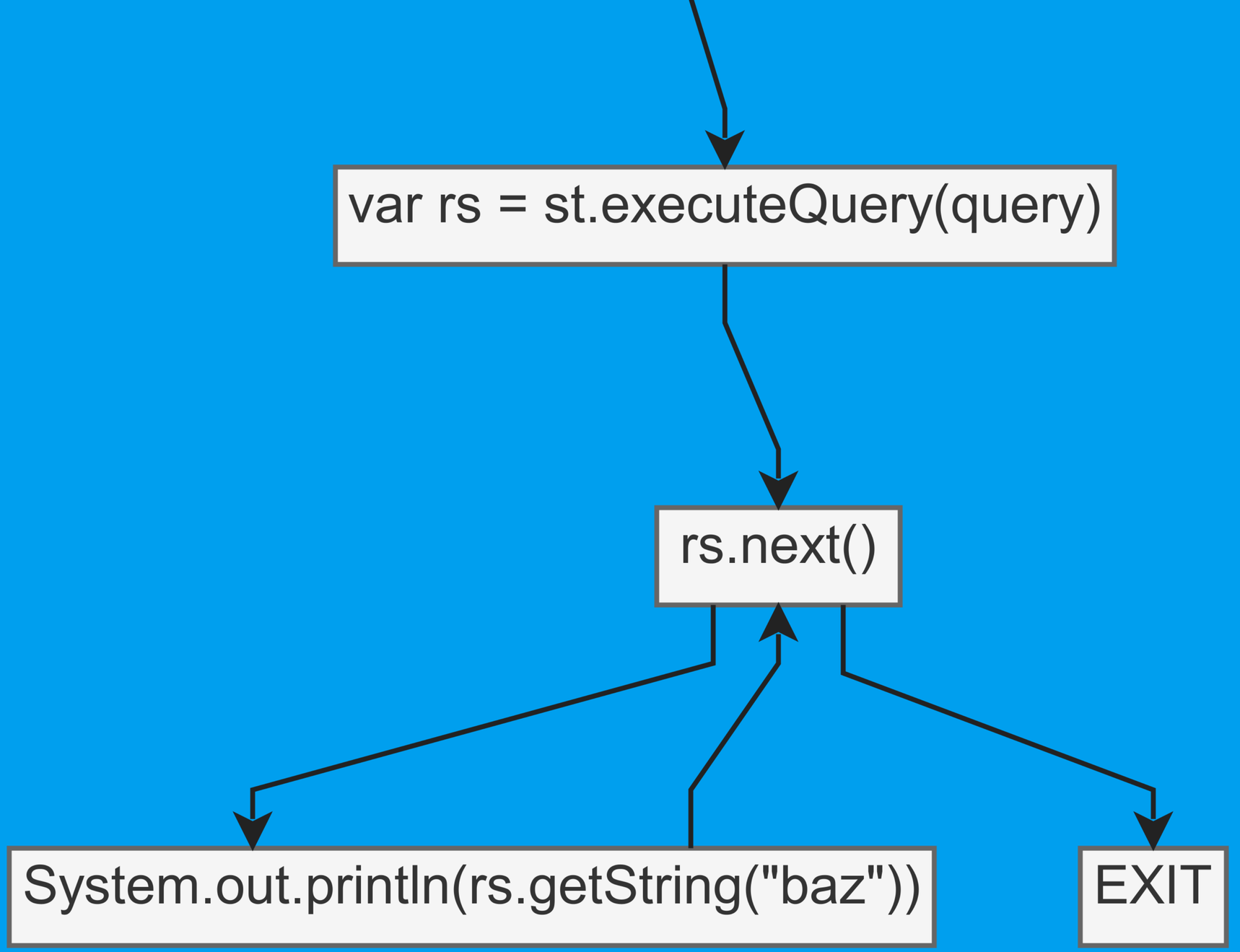


Граф потока управления




BEGIN
var bar = request.getParameter("bar")
var query = ("SELECT * FROM foo WHERE bar = '" + bar) + "'"


var query = ("SELECT * FROM foo WHERE bar = '" + bar) + "'"
var st = conn.createStatement()
var rs = st.executeQuery(query)


var rs = st.executeQuery(query)
rs.next()
System.out.println(rs.getString("baz"))
EXIT

Граф потока управления

- Что даёт:
  - Видим все пути исполнения, включая ветвления и циклы
  - Можем найти недостижимый код и точки выхода
  - Проверяем, **может ли** путь от источника дойти до стока

- Почему недостаточно:
  - CFG показывает **где** может пойти управление, но не **какие данные** текут по этим путям
  - Не видно, какая переменная где была определена и где использована
  - Для анализа уязвимостей нужен **поток данных**, а не только поток управления

Отслеживаем данные

DU цепи

- def-use chains:
  - граф (цепи) между местами, где переменная определена (definition), и местами, где она используется (use)

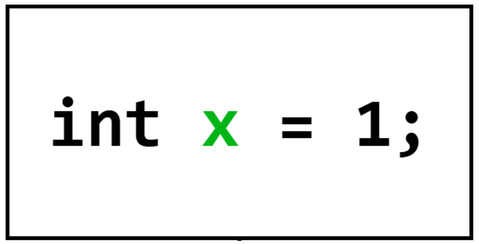
- Как строятся:
  - Идём по графу потока управления
  - Для каждого определения переменной создаём новую цепь
  - Если находим присваивание такой переменной, то начинаем новую

DU цепи


```
int x = 1;
x = x + 2;
println(x);
```

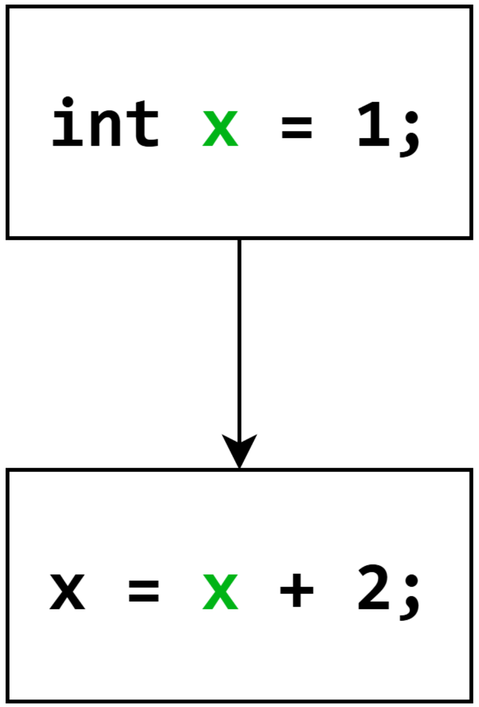
DU цепи


```
int x = 1;
x = x + 2;
println(x);
```



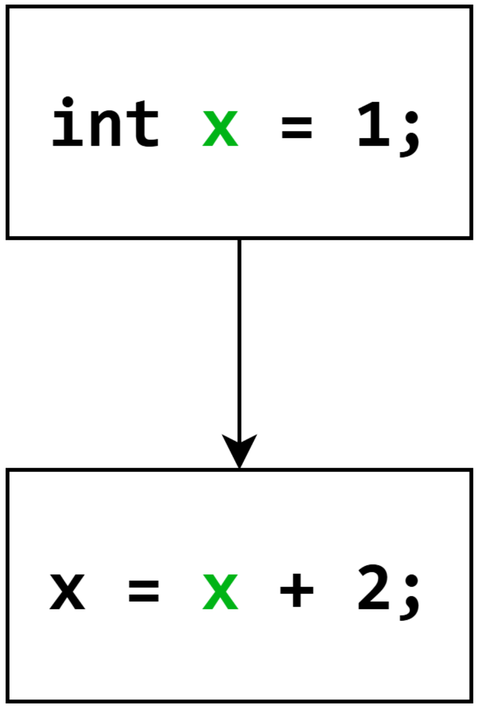
DU цепи

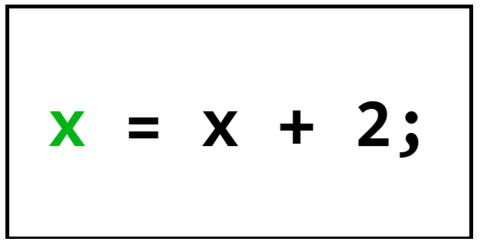

```
int x = 1;
x = x + 2;
println(x);
```



DU цепи

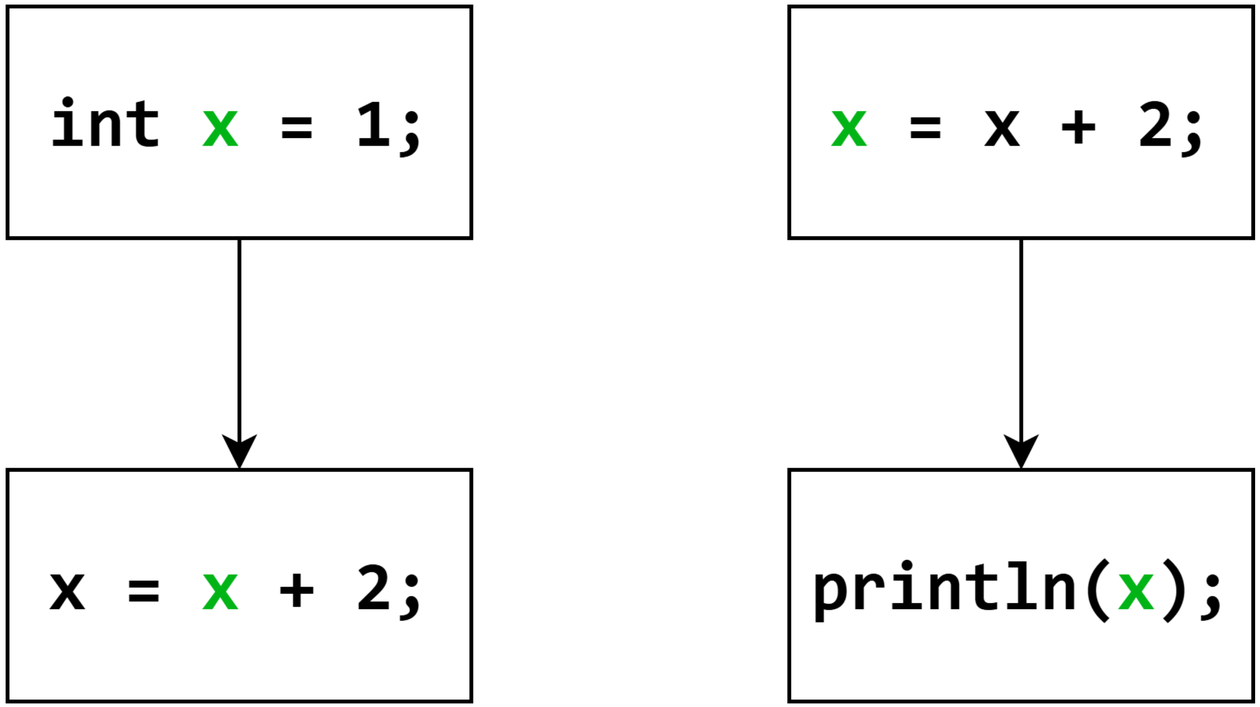

```
int x = 1;
x = x + 2;
println(x);
```





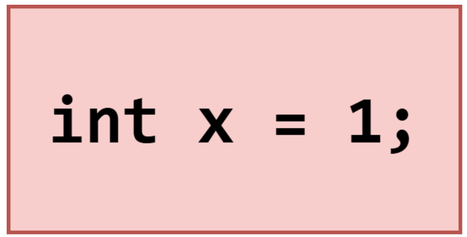
DU цепи

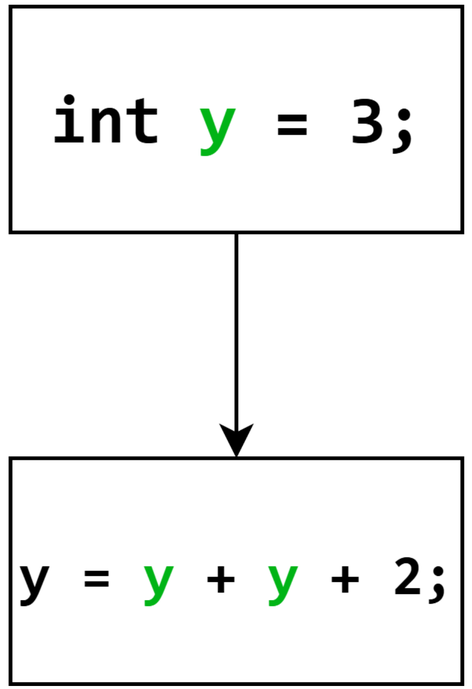

```
int x = 1;
x = x + 2;
println(x);
```

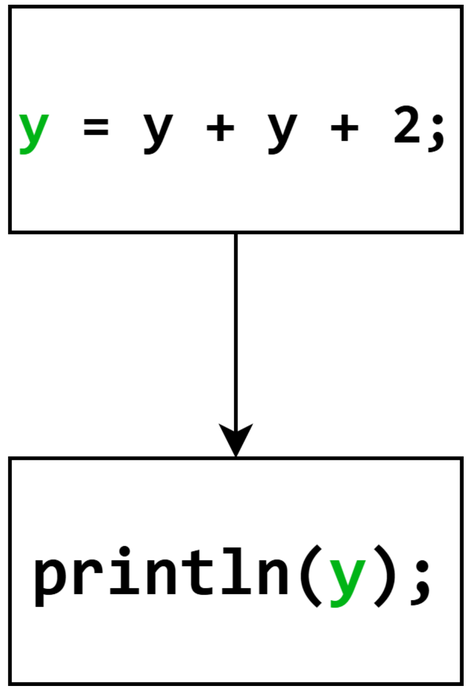


DU цепи


```
int x = 1;
int y = 3;
y = y + y + 2;
println(y);
```

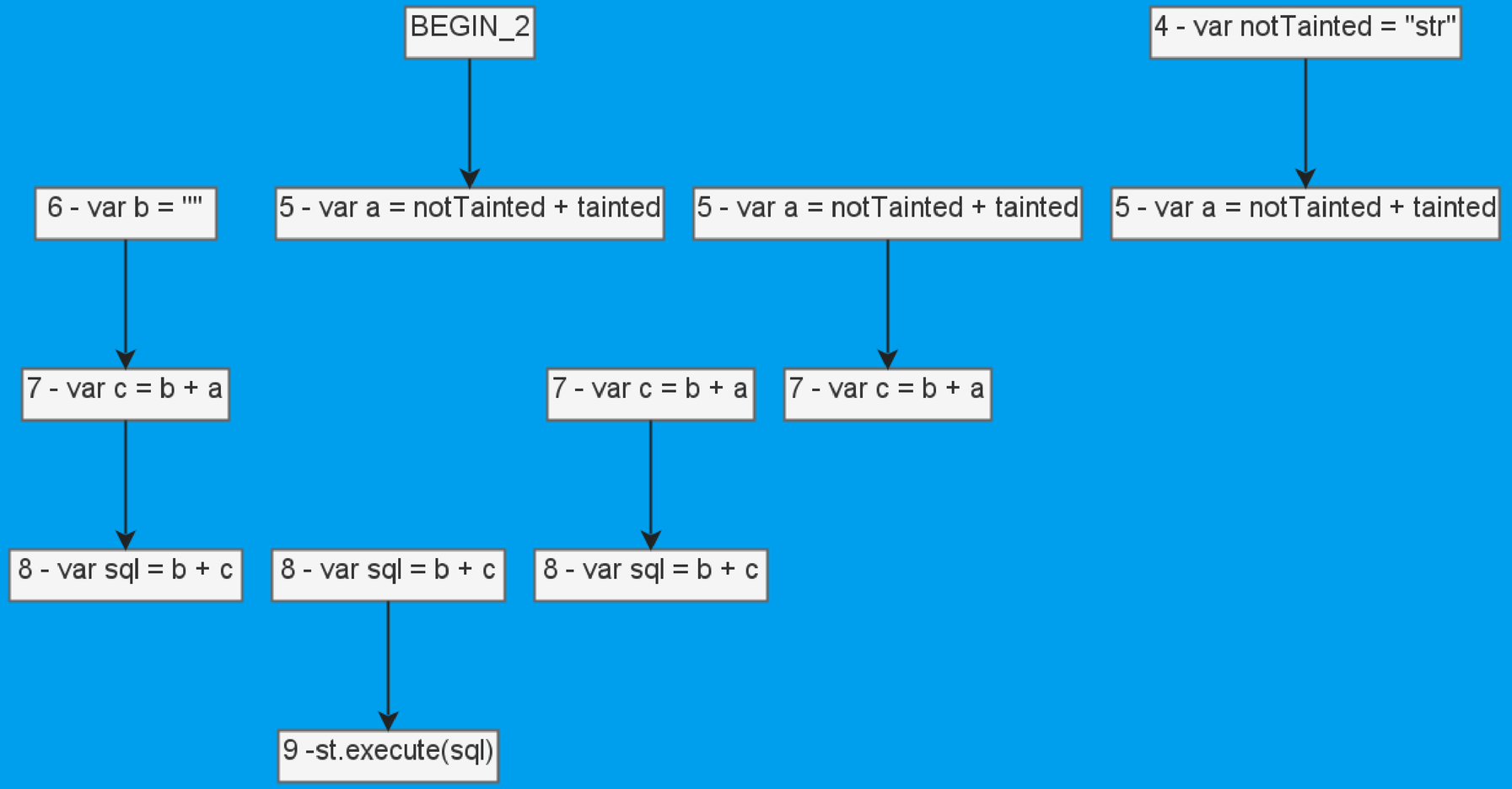




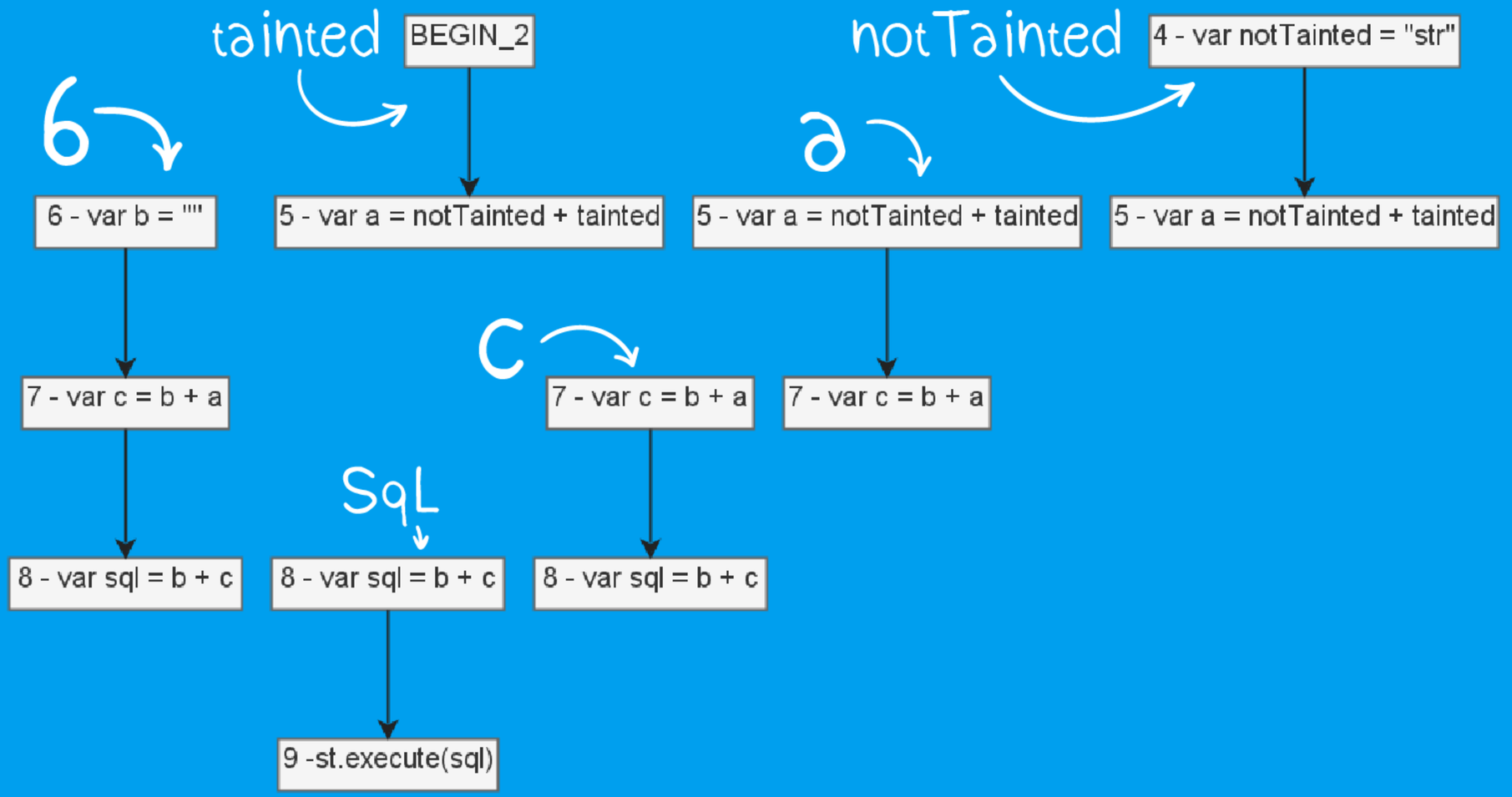


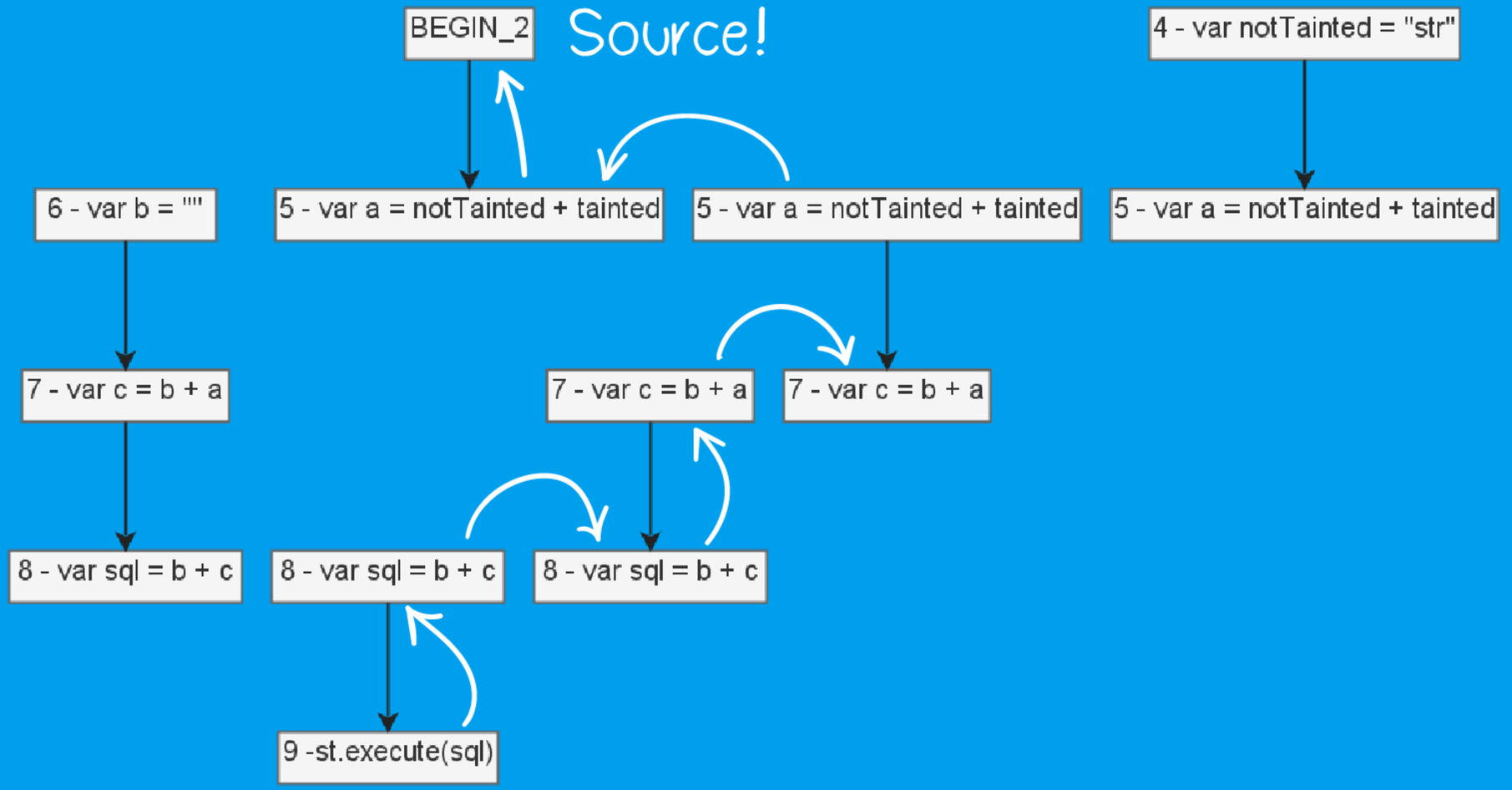
К простым примерам


```
public static void taint(String tainted) {
    var notTainted = "str";
    var a = notTainted + tainted;
    var b = "";
    var c = b + a;
    var sql = b + c;
    st.execute(sql);
}
```


BEGIN_2
4 - var notTainted = "str"
6 - var b = ""
5 - var a = notTainted + tainted
5 - var a = notTainted + tainted
5 - var a = notTainted + tainted
7 - var c = b + a
7 - var c = b + a
7 - var c = b + a
8 - var sql = b + c
8 - var sql = b + c
8 - var sql = b + c
9 -st.execute(sql)

Def-use цепи


tainted
BEGIN_2
notTainted
4 - var notTainted = "str"
6
a
c
SqL
6 - var b = ""
5 - var a = notTainted + tainted
5 - var a = notTainted + tainted
5 - var a = notTainted + tainted
7 - var c = b + a
7 - var c = b + a
7 - var c = b + a
8 - var sql = b + c
8 - var sql = b + c
8 - var sql = b + c
9 -st.execute(sql)



Граф потока данных

Что имеем

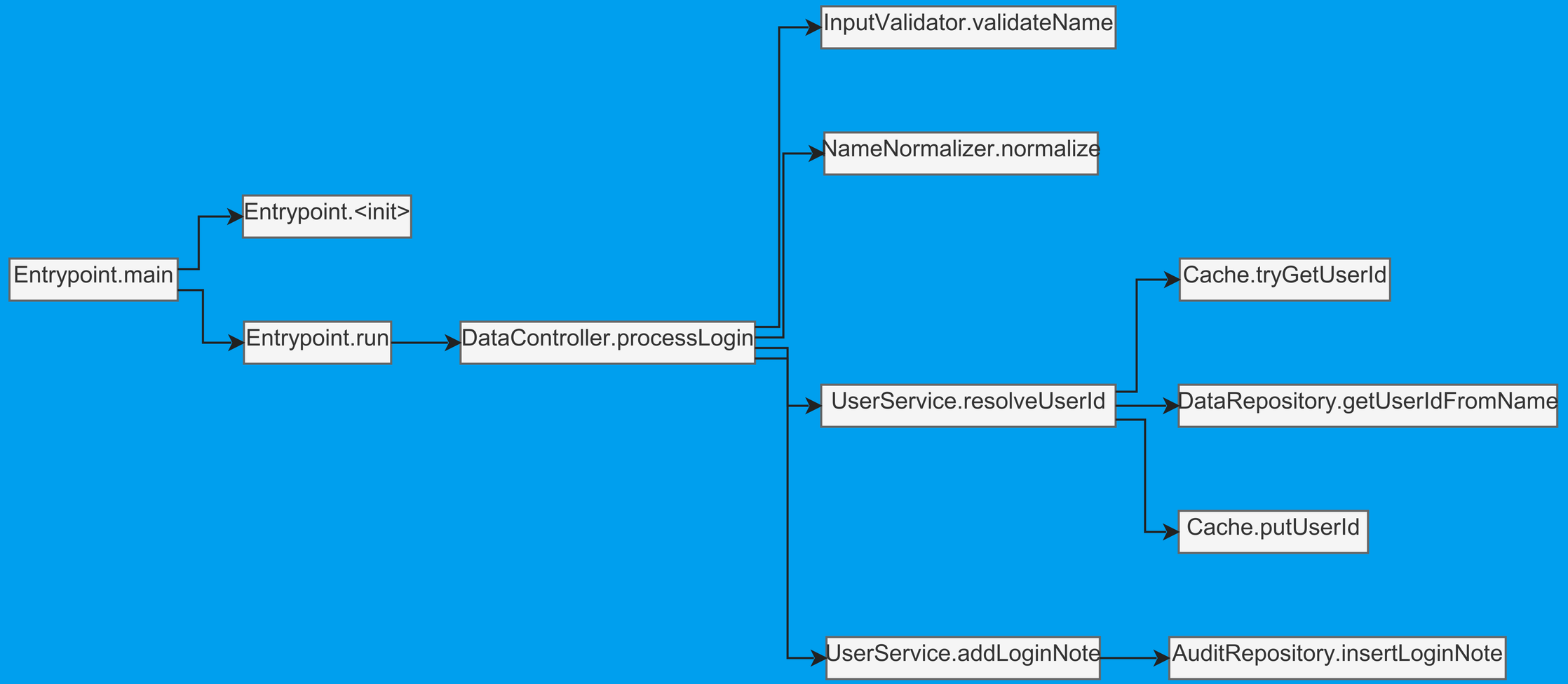
- AST: можем работать с кодом
- CFG: видим пути исполнения кода
- DU-цепи: знаем какие данные текут по этим путям

- Но уязвимости часто проходят через несколько методов

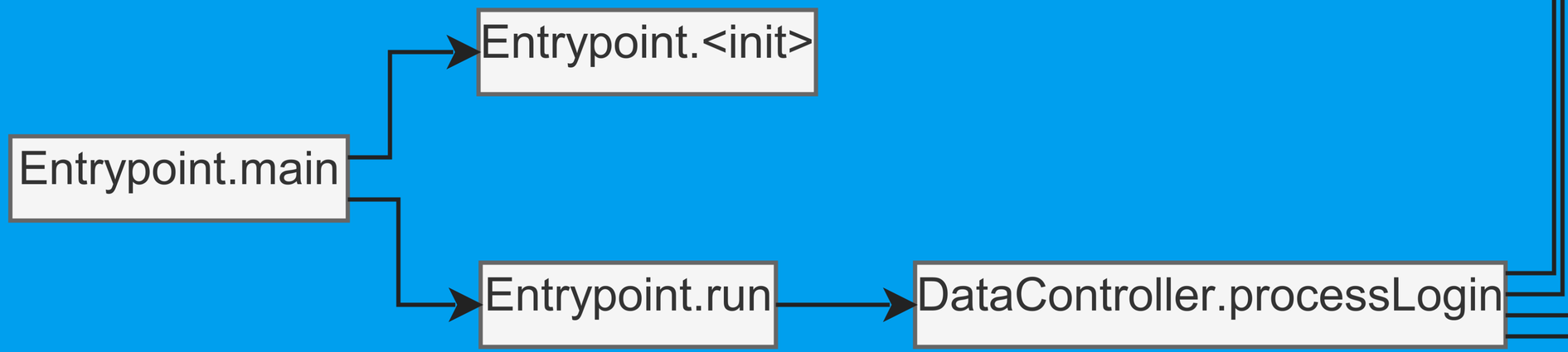
Анализ вызовов

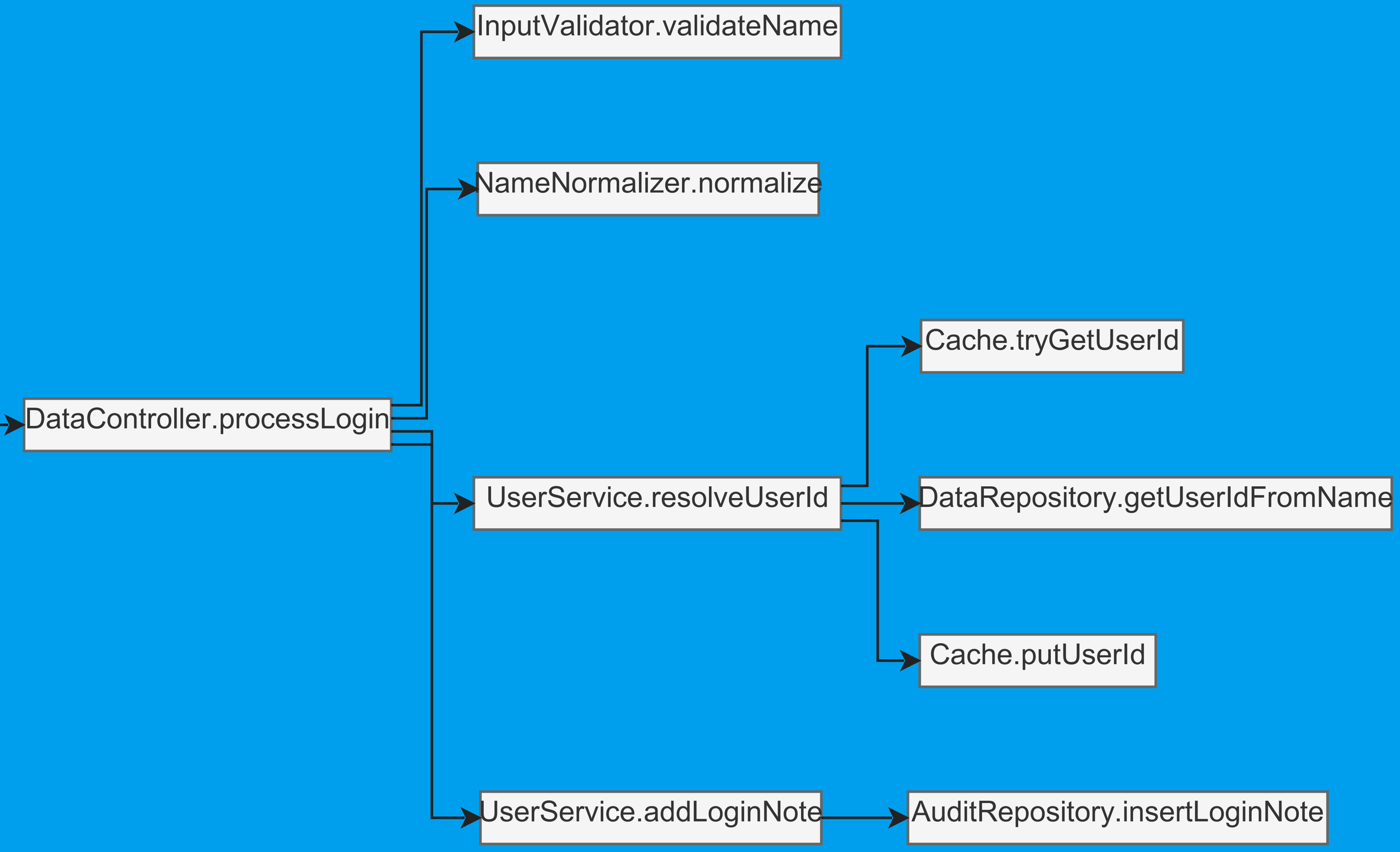
Анализ вызовов

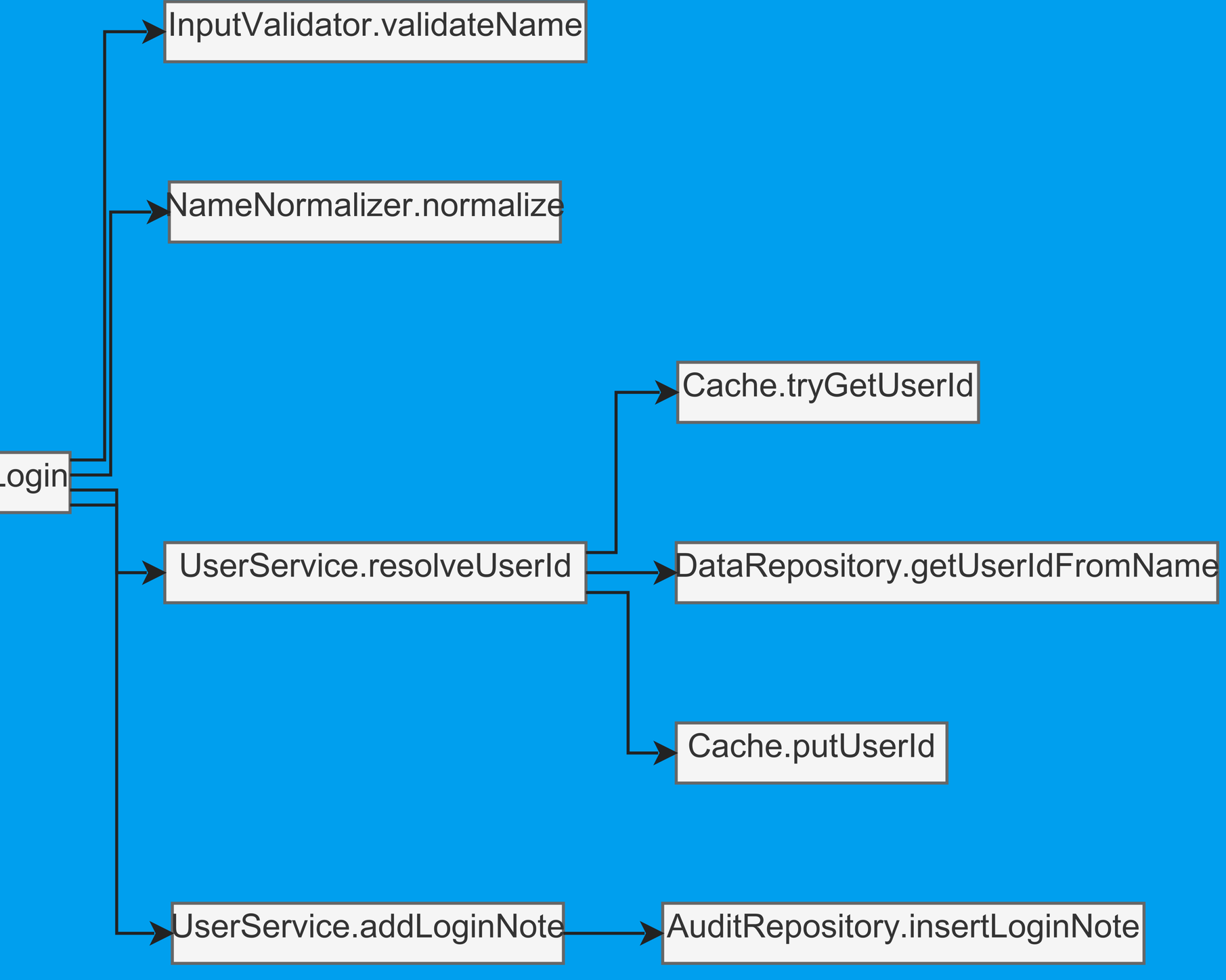
- Мы не знаем, какой метод откуда вызывают
- Источник может оказаться в вызывающем методе, а не там, где сток
- Для этого заранее посмотрим всё приложение и составим карту вызовов

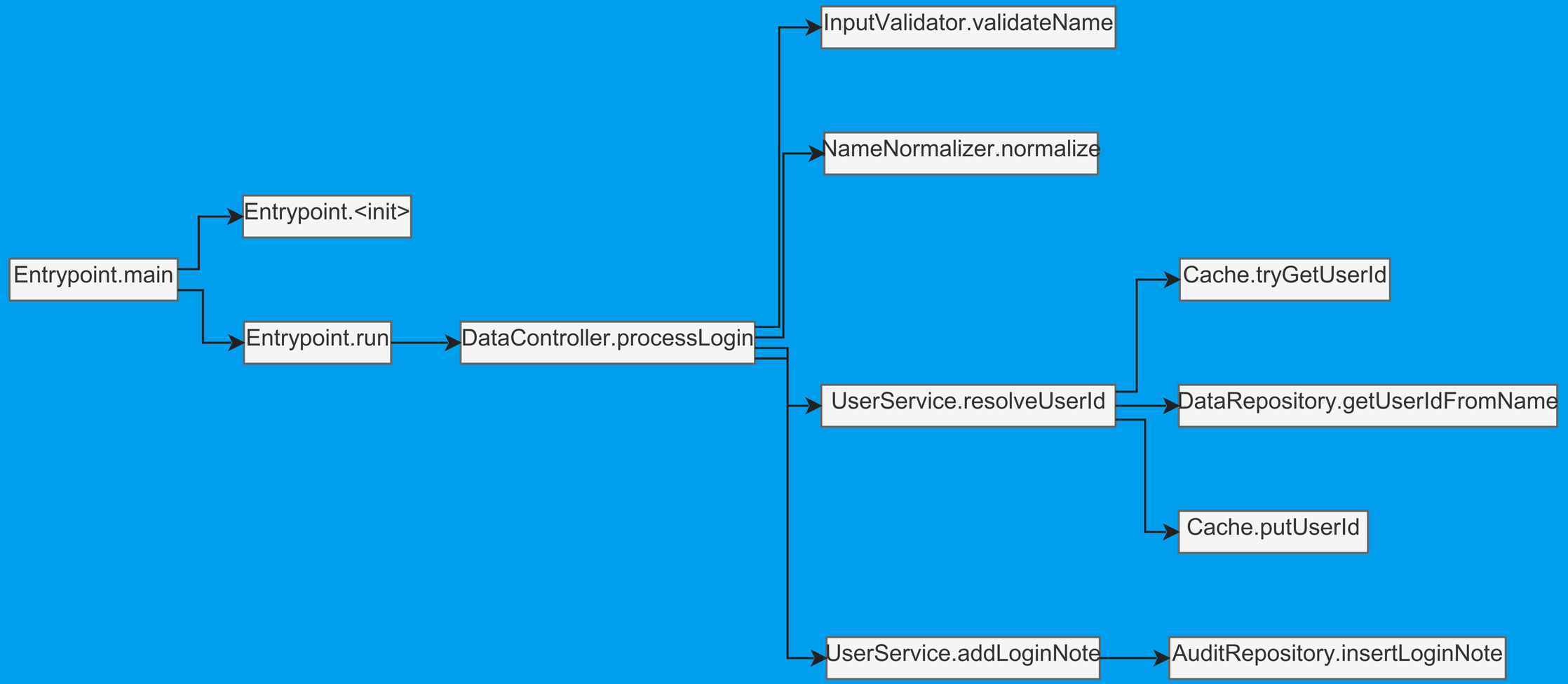


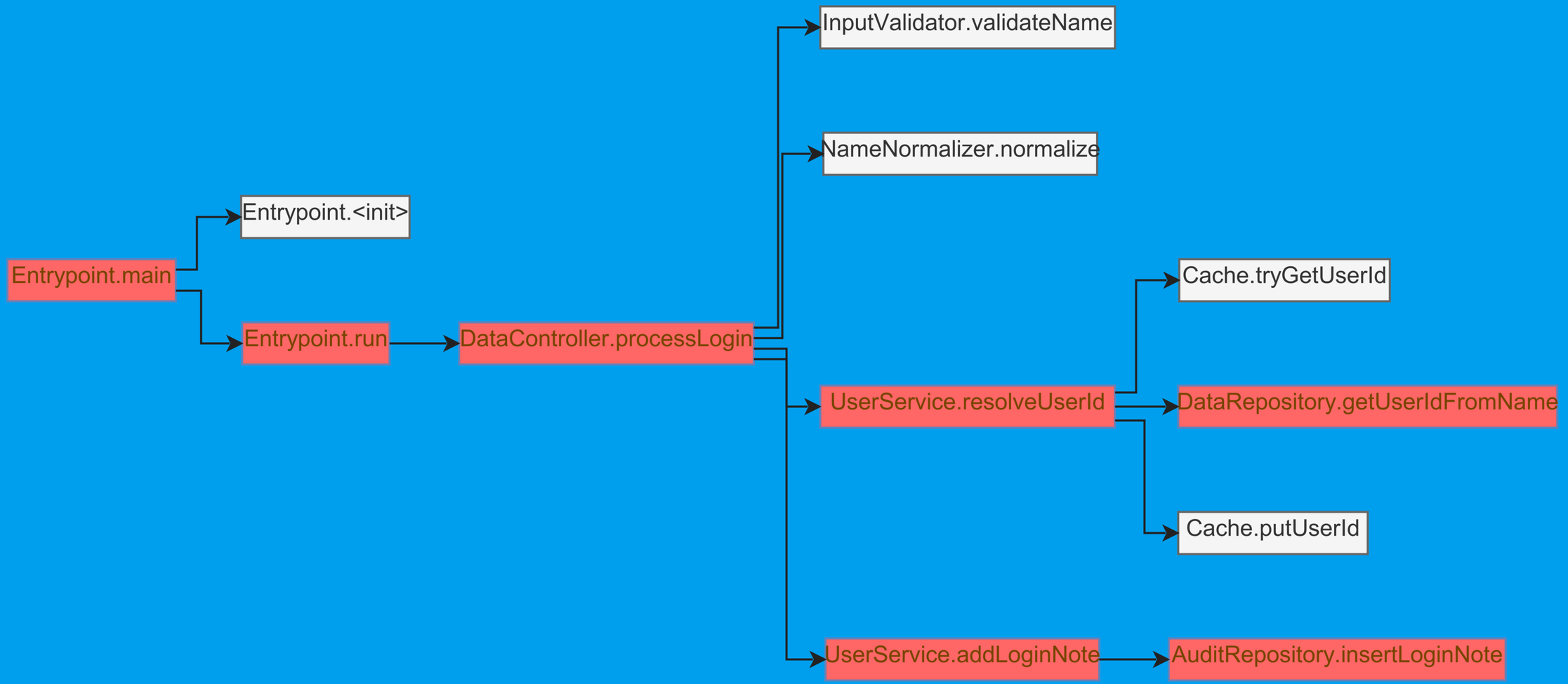
Граф вызовов


Entrypoint.<init>
Entrypoint.main
Entrypoint.run
DataController.processLogin


InputValidator.validateName
NameNormalizer.normalize
DataController.processLogin
Cache.tryGetUserId
UserService.resolveUserId
DataRepository.getUserIdFromName
Cache.putUserId
UserService.addLoginNote
AuditRepository.insertLoginNote

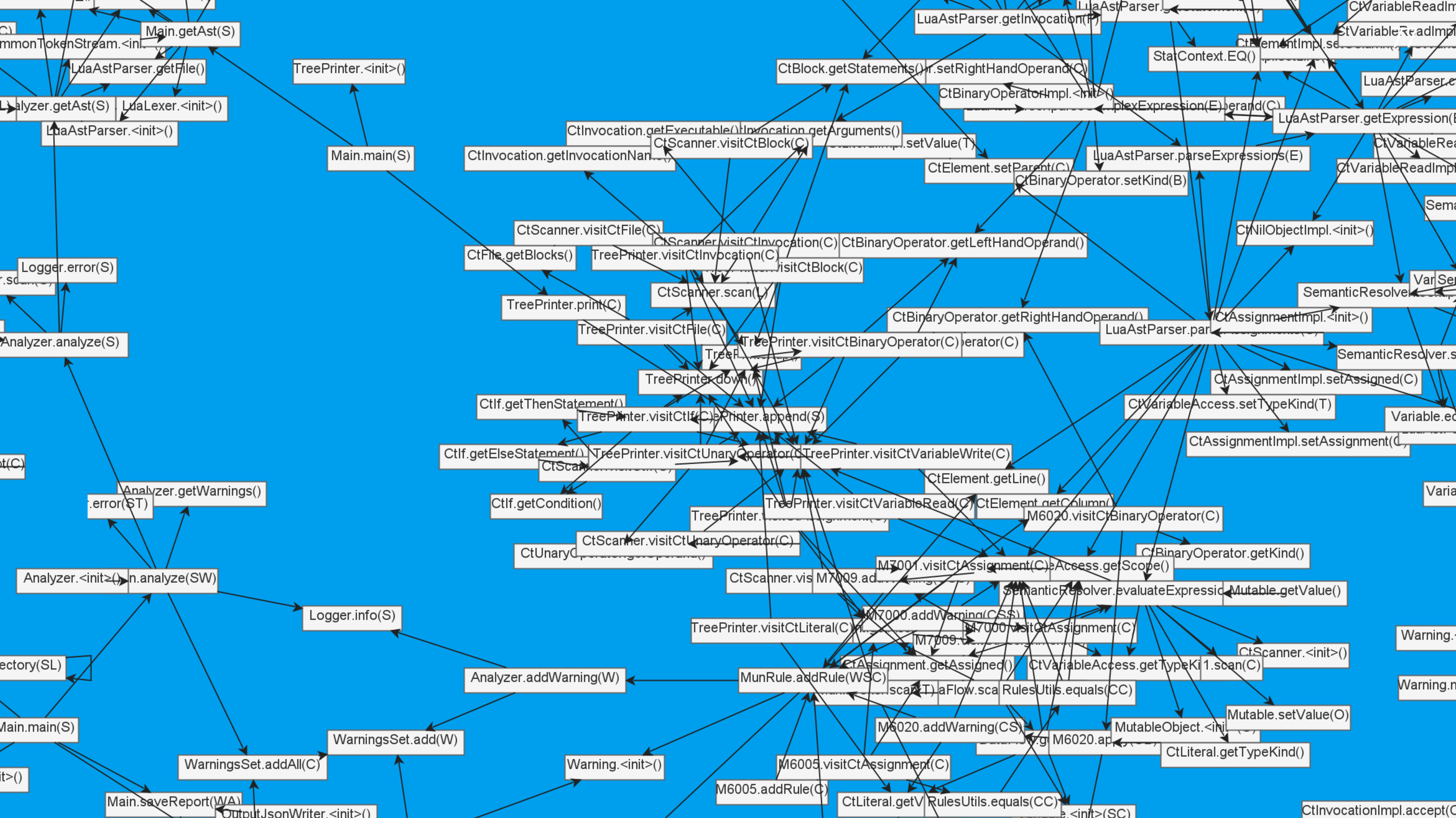

InputValidator.validateName
NameNormalizer.normalize
Login
Cache.tryGetUserId
UserService.resolveUserId
DataRepository.getUserIdFromName
Cache.putUserId
UserService.addLoginNote
AuditRepository.insertLoginNote

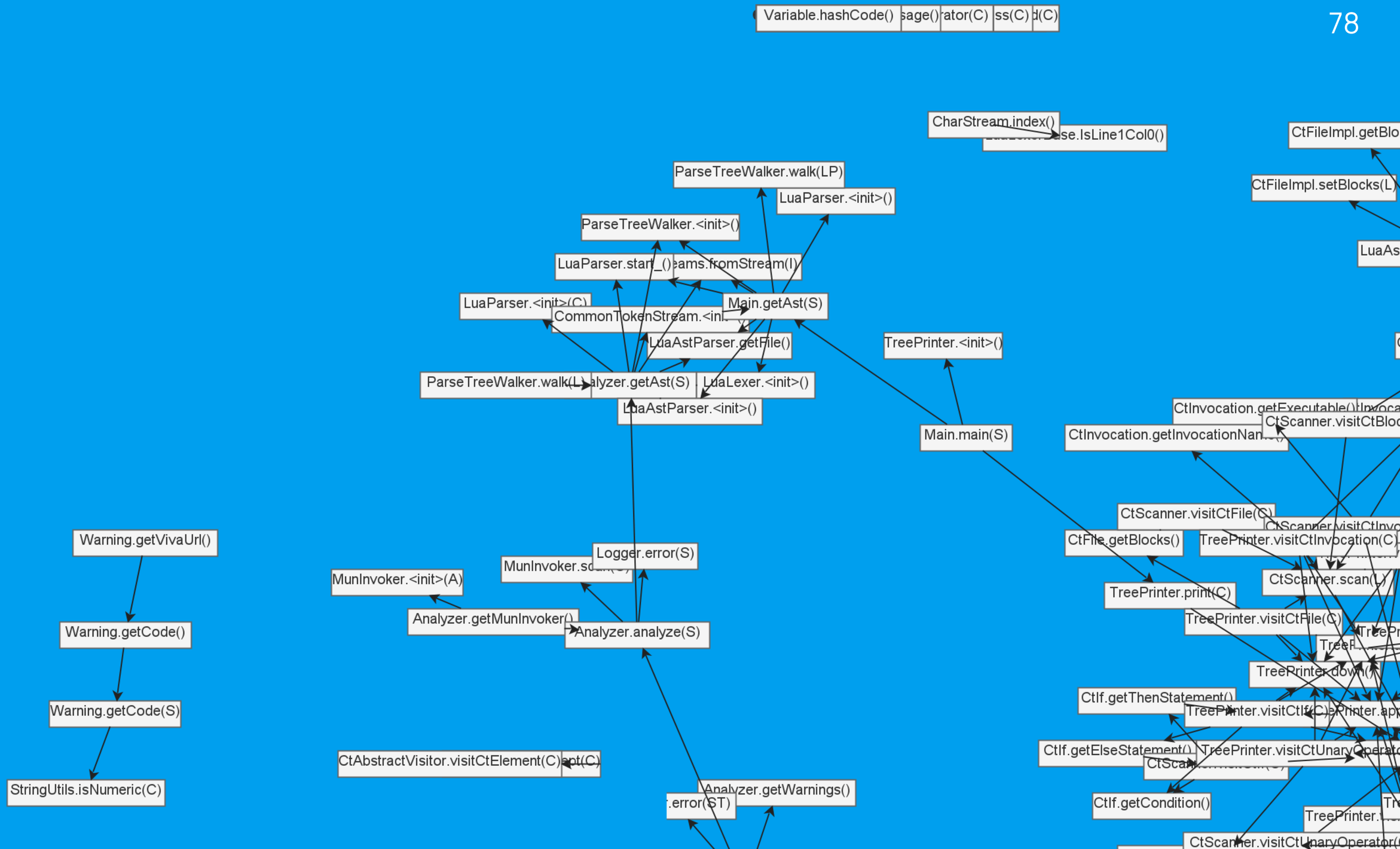

Entrypoint.main
Entrypoint.<init>
Entrypoint.run
DataController.processLogin
InputValidator.validateName
NameNormalizer.normalize
UserService.resolveUserId
Cache.tryGetUserId
DataRepository.getUserIdFromName
Cache.putUserId
UserService.addLoginNote
AuditRepository.insertLoginNote


Entrypoint.main
Entrypoint.<init>
Entrypoint.run
DataController.processLogin
InputValidator.validateName
NameNormalizer.normalize
UserService.resolveUserId
Cache.tryGetUserId
DataRepository.getUserIdFromName
Cache.putUserId
UserService.addLoginNote
AuditRepository.insertLoginNote



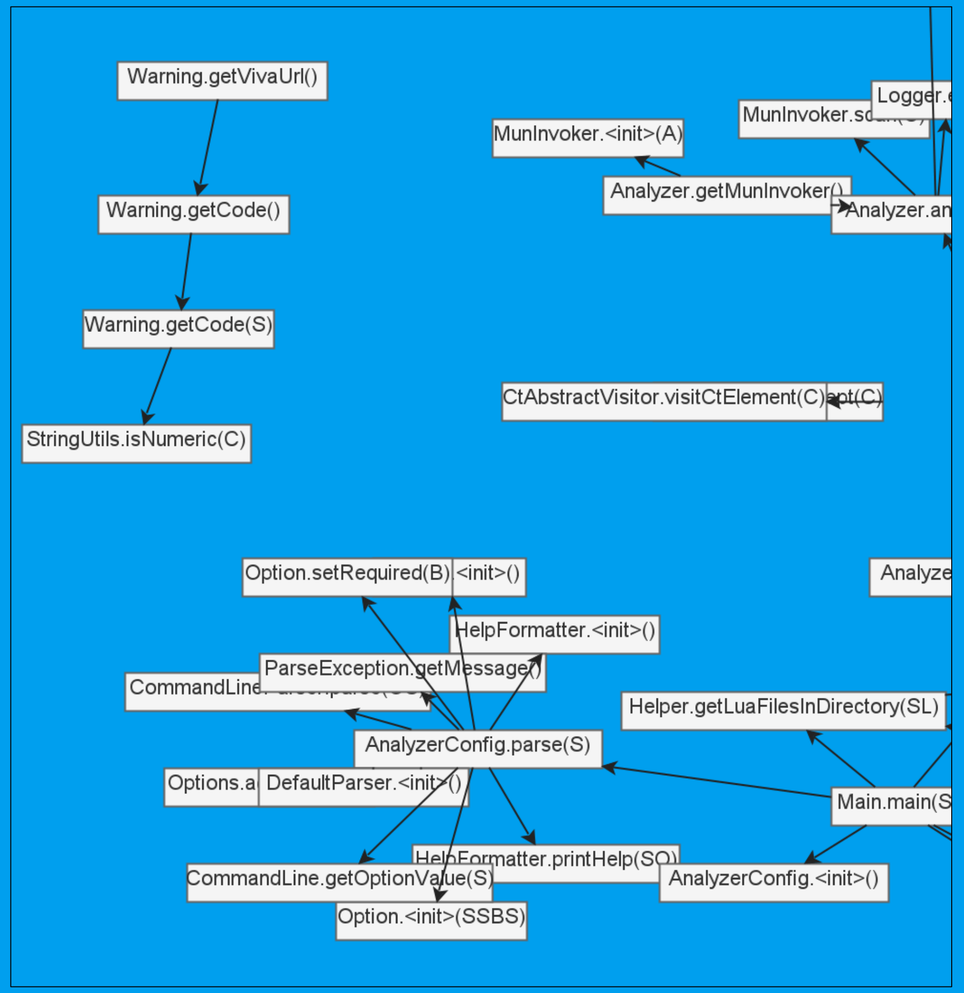
Граф вызовов


Main.getAst(S)
LuaAstParser.getFile()
TreePrinter.<init>()
LuaLexer.<init>()
LuaAstParser.<init>()
Main.main(S)
Logger.error(S)
Analyzer.analyze(S)
CtScanner.visitCtFile(C)
CtScanner.visitCtInvocation(C)
CtFile.getBlocks()
TreePrinter.visitCtInvocation(C)
CtScanner.visitCtBlock(C)
CtScanner.scan(L)
TreePrinter.print(C)
TreePrinter.visitCtFile(C)
TreePrinter.visitCtBinaryOperator(C)
TreePrinter.down(I)
CtIf.getThenStatement()
TreePrinter.visitCtIf(C)
TreePrinter.append(S)
CtIf.getElseStatement()
TreePrinter.visitCtUnaryOperator(C)
TreePrinter.visitCtVariableWrite(C)
CtIf.getCondition()
TreePrinter.visitCtVariableRead(C)
CtUnaryOperator.getOperand()
CtScanner.visitCtUnaryOperator(C)
CtBlock.getStatements()
CtInvocation.getExecutable()
CtInvocation.getArguments()
CtInvocation.getInvocationName()
CtElement.setParent(C)
CtBinaryOperator.setKind(B)
CtBinaryOperator.getLeftHandOperand()
CtBinaryOperator.getRightHandOperand()
CtBinaryOperatorImpl.<init>()
CtAssignmentImpl.<init>()
CtAssignmentImpl.setAssigned(C)
CtAssignmentImpl.setAssignment(C)
CtVariableAccess.setTypeKind(T)
CtNilObjectImpl.<init>()
CtElement.getLine()
CtElement.getColumn()
LuaAstParser.getExpression(E)
LuaAstParser.parseExpressions(E)
LuaAstParser.getInvocation(P)
StarContext.EQ()
SemanticResolver.evaluateExpressio
CtBinaryOperator.getKind()
CtVariableAccess.getScope()
Mutable.getValue()
Mutable.setValue(O)
MutableObject.<ini
CtLiteral.getTypeKind()
M6020.visitCtBinaryOperator(C)
M6020.addWarning(CS)
M6020.ap
M6005.visitCtAssignment(C)
M6005.addRule(C)
CtLiteral.getV
RulesUtils.equals(CC)
M7001.visitCtAssignment(C)
M7000.addWarning(CSS)
M7000.visitCtAssignment(C)
M7009.
CtScanner.vis
M7009.ad
CtScanner.<init>()
CtVariableAccess.getTypeKi
1.scan(C)
CtAssignment.getAssigned()
MunRule.addRule(WSC)
aFlow.sca
RulesUtils.equals(CC)
Analyzer.addWarning(W)
Logger.info(S)
Analyzer.getWarnings()
Analyzer.<init>()
n.analyze(SW)
WarningsSet.add(W)
WarningsSet.addAll(C)
Warning.<init>()
Main.saveReport(WA)
OutputJsonWriter.<init>()
Main.main(S)
ectory(SL)
TreePrinter.visitCtLiteral(C)
CtInvocationImpl.accept(C)

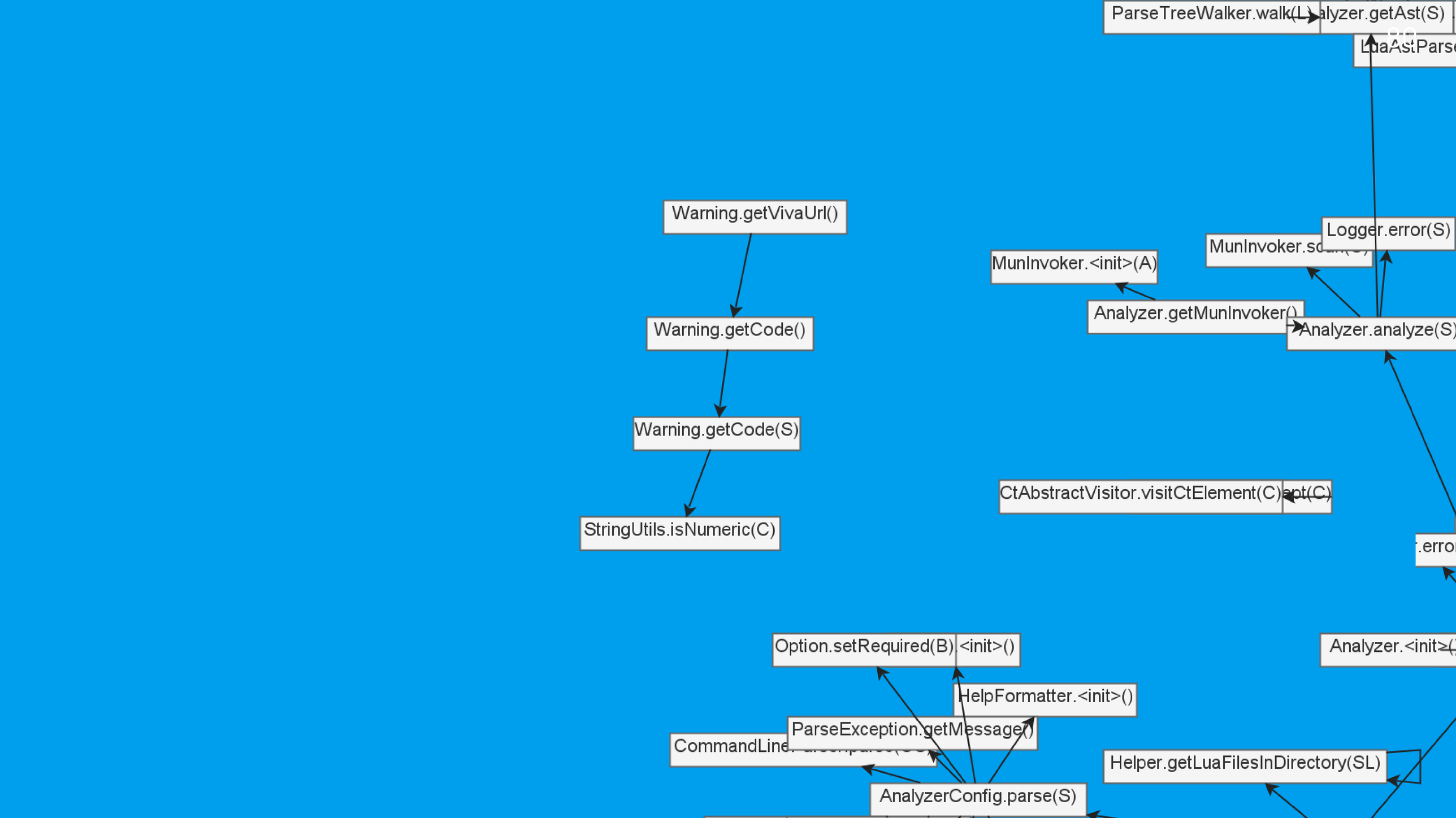

Variable.hashCode()
CharStream.index()
CtFileImpl.getBlo
CtFileImpl.setBlocks(L
ParseTreeWalker.walk(LP)
LuaParser.<init>()
ParseTreeWalker.<init>()
LuaParser.start_()
ams.fromStream(I)
LuaParser.<init>(C)
Main.getAst(S)
CommonTokenStream.<in
LuaAstParser.getFile()
TreePrinter.<init>()
ParseTreeWalker.walk(L
alyzer.getAst(S)
LuaLexer.<init>()
LuaAstParser.<init>()
Main.main(S)
CtInvocation.getExecutable()
CtScanner.visitCtBloc
CtInvocation.getInvocationNa
CtScanner.visitCtFile(C)
CtFile.getBlocks()
TreePrinter.visitCtInvocation(C)
Warning.getVivaUrl()
Logger.error(S)
MunInvoker.<init>(A)
CtScanner.scan(L)
TreePrinter.print(C)
Analyzer.getMunInvoker()
Analyzer.analyze(S)
TreePrinter.visitCtFile(C)
TreePrinter.down()
Warning.getCode()
CtIf.getThenStatement()
Warning.getCode(S)
CtIf.getElseStatement()
CtAbstractVisitor.visitCtElement(C)
Analyzer.getWarnings()
StringUtils.isNumeric(C)
CtIf.getCondition()

Что можно сделать с графом вызовов?

- Компилятор может не класть не вызываемые методы в сборку
- Можно делать эвристики для поиска code smell
  - Поиск god методов
  - Рекурсии



PVS-Studio


ParseTreeWalker.walk(L
alyzer.getAst(S)
LuaAstParse
Warning.getVivaUrl()
Logger.error(S)
MunInvoker.scan(S)
MunInvoker.<init>(A)
Analyzer.getMunInvoker()
Analyzer.analyze(S)
Warning.getCode()
Warning.getCode(S)
CtAbstractVisitor.visitCtElement(C)
apt(C)
StringUtils.isNumeric(C)
.erro
Option.setRequired(B)
<init>()
Analyzer.<init>(
HelpFormatter.<init>()
ParseException.getMessage()
CommandLine
Helper.getLuaFilesInDirectory(SL)
AnalyzerConfig.parse(S)

Что имеем

- AST: можем работать с кодом
- CFG: видим пути исполнения кода
- DU-цепи: знаем какие данные текут по этим путям
- При помощи графа вызовов понятно, кто и кого зовёт

- Теперь у нас есть всё для поиска инъекций

Мы засунули граф в граф

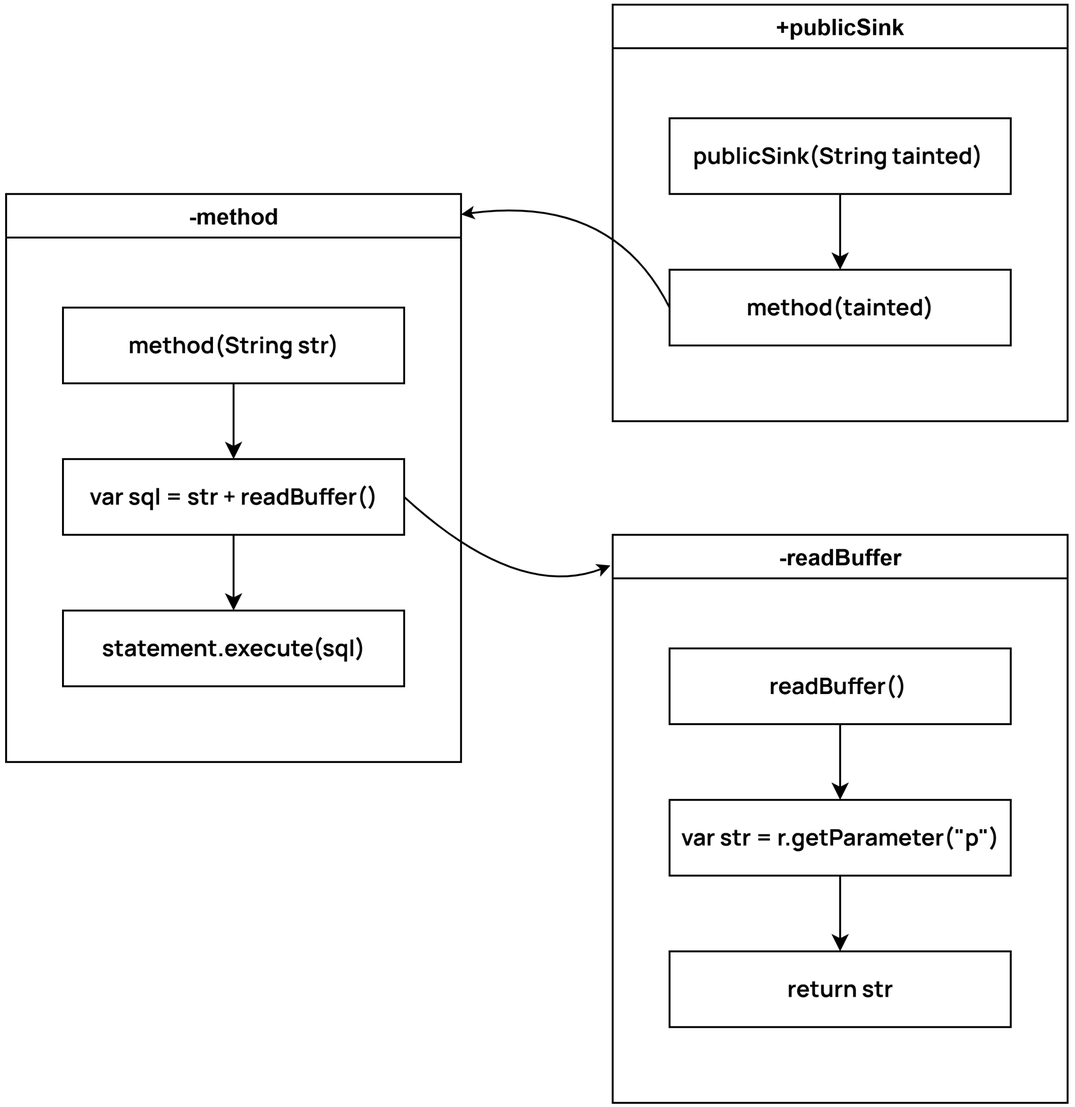


Чтобы вы могли обходить граф, пока обходите граф

Как найти уязвимость?

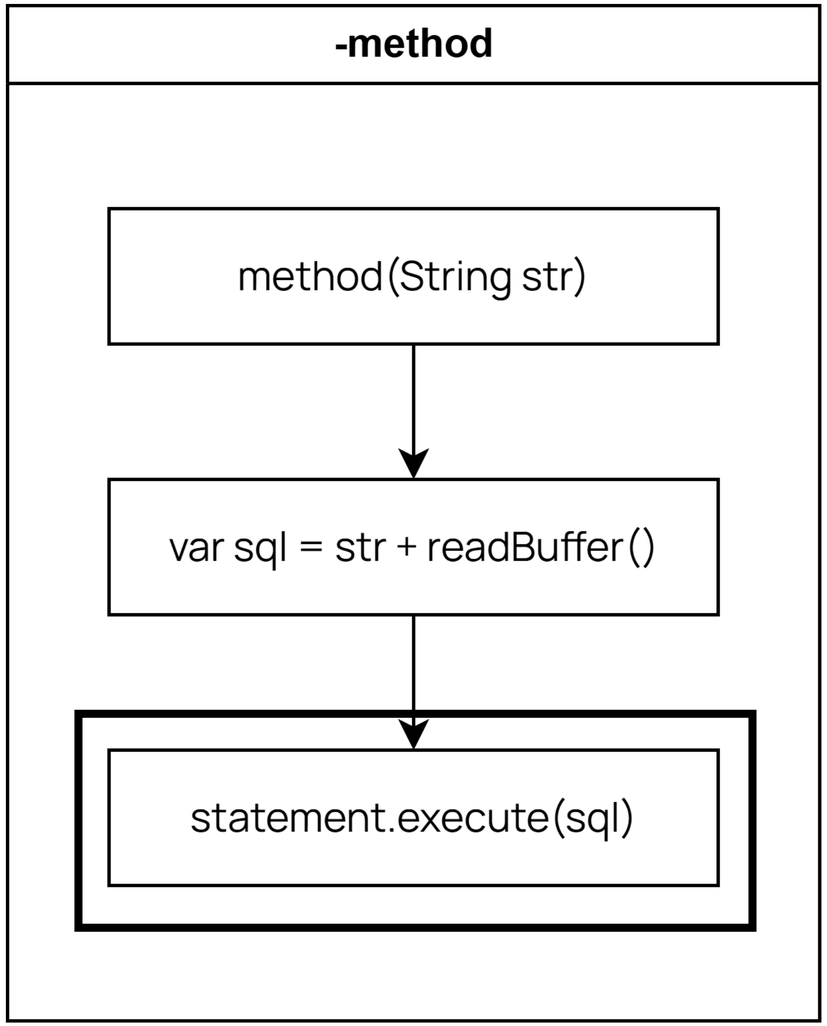
Как найти уязвимость

1. Сначала **обходим** снизу-вверх **метод**, от **стока**
2. При нахождении **вызова** метода **добавляем его в очередь**
3. Когда дошли до **начала** метода, добавляем в очередь **все методы**, которые **его вызвали**



Как найти инъекцию

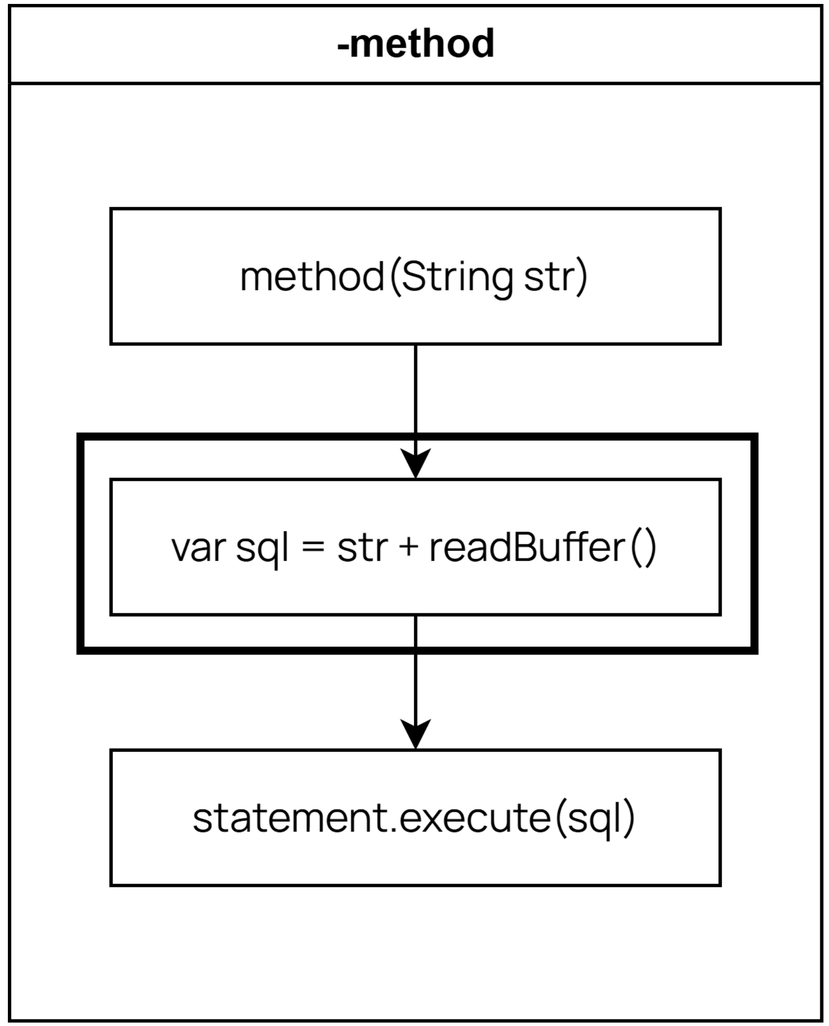
1. Начинаем со стока (`execute`)
2. Идём вверх по цепи для `sql`
3. Попадаем в определение `sql`, видим зависимость от `readBuffer`
4. Добавляем `readBuffer` в очередь



Очередь: `readBuffer`

Как найти инъекцию

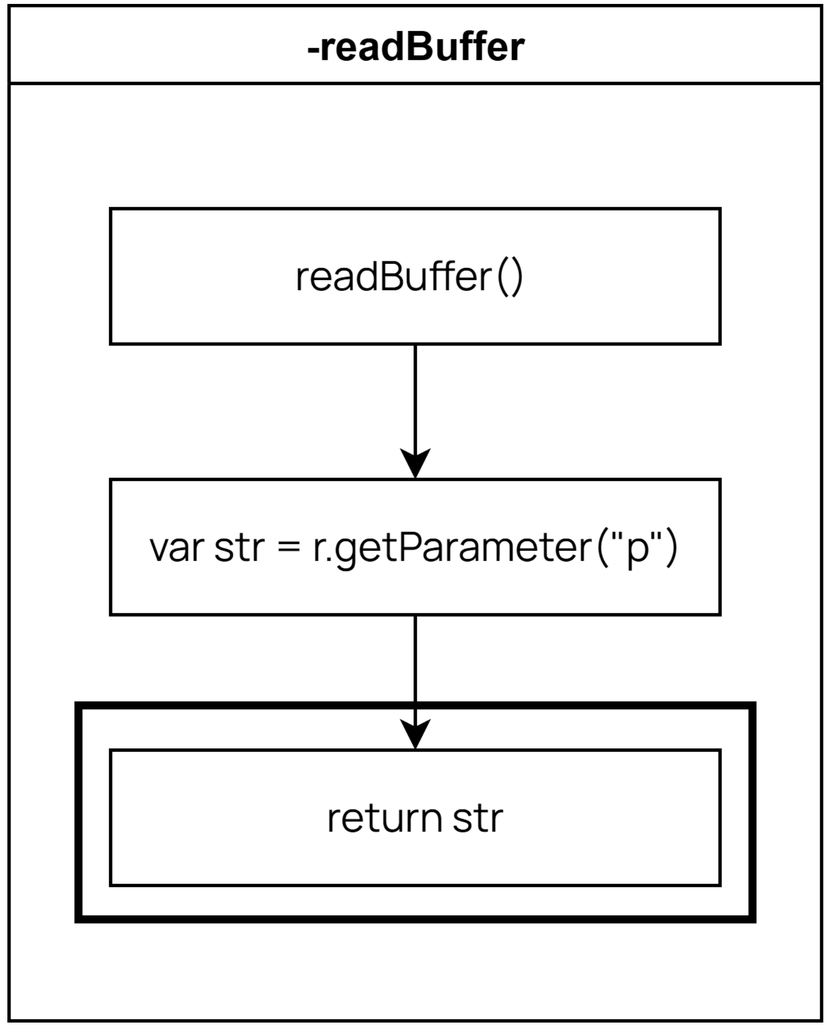
5. Теперь идём по цепи для `str`
6. Попадаем в начало приватного метода
7. Call graph говорит, что `method` вызывается из `publicSink`



Очередь: `readBuffer` `publicSink`

Как найти инъекцию

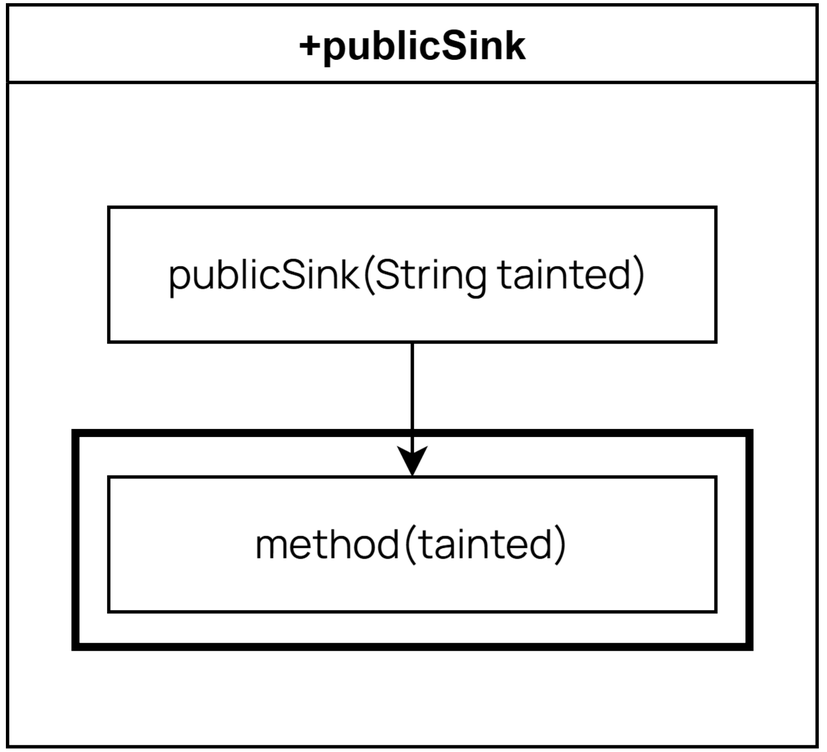
8. Переходим в `readBuffer`
9. Начинаем обходить снизу, идём по цепи для `str`
10. Видим, что `str` зависит от **источника** заражённых данных `getParameter`



Очередь: `publicSink`

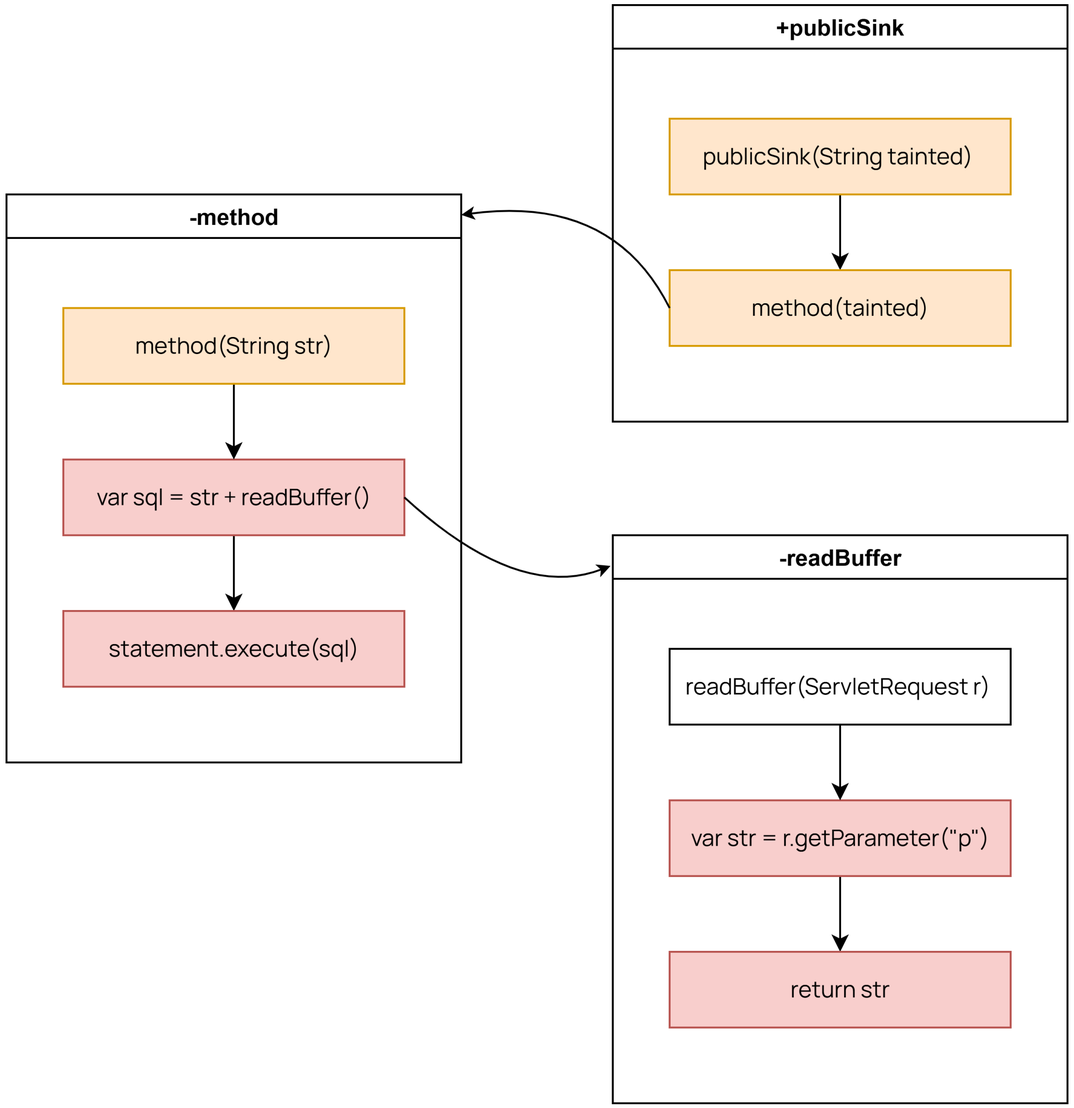
Как найти инъекцию

11. Переходим в `publicSink`
12. Начинаем обходить снизу, идём по цепи для `tainted`
13. Видим, что `tainted` это параметр **публичного** метода, т.е. это второй **источник**



Очередь:

Как найти уязвимость



Итоги

Если захочется что-то сделать самому

- Большую часть анализа кода можно проводить на AST:
  - Поиск code smell-ов, нарушение кодстайла
  - Подсветка синтаксиса, навигация по коду
  - Часть статического анализа (нарушения паттернов)
- Более продвинутый анализ (CFG, CG, DU/IR)
  - Анализ потока данных в статическом анализе
  - Поиск уязвимостей в коде
- Code Property Graph, Sea of Nodes
  - Оптимизации в компиляторе
  - Максимально глубокий статический анализ



SAST

- Требует продвинутых технологий анализа
- Но взамен имеет свои сильные стороны:
  - Таким же образом находит другие типы уязвимостей: SSRF, Path Traversal, XSS и пр.
  - Можно ловить уязвимости на графах ещё до запуска приложения
- Не забывайте, что лучше использовать параметризированные запросы :)

Q/A

Сделай свой проект
чистым и безопасным
вместе с PVS-Studio

PVS-Studio

