МИНОБРНАУКИ РОССИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«ВОРОНЕЖСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(ФГБОУ ВО «ВГУ»)

Факультет прикладной математики, информатики и механники
Кафедра программного обеспечения
и администрирования информационных систем

Разработка лексического и синтаксического анализаторов с целью подсветки синтаксиса и автодополнения исходного кода для предоставленного языка

Бакалаврская работа
Направление 02.03.03. Математическое обеспечение и администрирование информационных систем
Профиль Информационные системы и базы данных

Зав. кафедрой _____ д. ф-м. н. проф. М. А. Артёмов ___.___2019 г.
Обучающийся _____ А. С. Пахомов
Руководитель _____ преп. Н. В. Огаркова

Воронеж 2019

Содержание

Введение

Современная разработка програмного обеспечения стремительно движется в сторону более открытого и гибкого взаимодействия с пользователем системы (программист, аналитик и т.д.).

Одним из способов взаимодействия с пользователем является графический интерфейс. Такой подход позволяет скрыть детали работы приложения и предоставить возможность взаимодействовать с системой только с помошью графических элементов управления.

Другим инструментом взаимодействия пользователя с приложением являются предметно-ориентированные языки программирования (англ. Domain Specific Language, DSL). Основным отличием языка DSL от графического интерфейса является гибкость, которую DSL предоставляет пользователю приложения. С помощью языка DSL можно описать сценарии взаимодействия с системой.

Однако, у такого подхода существуют и недостатки. Например, для того, чтобы пользователь начал писать код на DSL, ему требуется время на изучение синтаксиса языка. Одним из решений является выбор среды разработки, которая позволяла бы писать код на языке DSL так же удобно, как и на языках общего применения.

В данной работе рассматривается язык GRIP DSL для предметной области класса DMP (Data Management Platform) с возможностью аналитики в реальном времени, разработкой которой занимается компания Gridfore. Язык GRIP DSL предназначен для разработки сценариев управления потоками данных, с которым могли бы взаимодействовать пользователи, не имеющие опыта разработки на языках программирования общего применения.

Целью работы является подсветка синтаксиса и автодополнение языка программирования GRIP DSL в интегрированной среде разработки IntelliJ IDEA.

Задачи работы:

1. Проанализировать существующие подходы к лексическому и синтаксическому анализу языков программирования с контекстно-свободной грамматикой и динамической компиляцией.
2. Разработать лексический и синтаксический анализаторы с целью подсветки синтаксиса и автодополнения исходного кода для языка GRIP DSL.

Областью исследования является лексический и синтаксический анализ, а также интеграция этих процессов с современной средой разработки.

Предметом исследования является подсветка и автодополнение синтаксиса Groovy DSL в Intellij IDEA.

Работа не включает в себя разработку полноценного плагина для IDE, самостоятельного компилятора и предметно-ориентированного языка.

1. Обзор и анализ существующих подходов

1.1. Основные понятия теории компиляции

1.1.1. Компилятор

Компилятор — это программа, которая принимает текст, написанный на одном языке — *исходном*, и транслирует (переводит) его в эквивалентный текст на другом языке — *целевом* [1]. Процесс компиляции можно разделить на две части.

Первая часть — *анализ* (выполняется в несколько фаз) разбивает исходную программу на составные части и накладывает на них грамматическую структуру. Затем эта структура используется для генерации промежуточного представления программы. Анализатор сообщает пользователю об ошибках, собирает информацию об исходной программе и сохраняет в *таблицу символов*. Таблица символов — это структура данных, которая используется компилятором для хранения информации о конструкциях исходной программы. Информация накапливается инкрементно в фазе анализа компилятора и используется фазой синтеза для генерации целевого кода.

Вторая часть — *синтез* (выполняется генератором кода), строит требуемую целевую программу на основе промежуточного представления и информации из таблицы символов.

Анализ и синтез делятся на несколько шагов, которые представлены на рис. 1.1.

Рис. 1.1. Схема взаимодействия фаз компилятора

Анализ:

– *Лексический анализ* или *сканирование*. Лексический анализатор читает поток символов, составляющих исходную программу, группирует их в значащие последовательности (лексемы), формирует для каждой лексемы специальные структуры данных, называемые токенами, и передает их синтаксическому анализатору.
– *Синтаксический анализ* или *разбор*. Синтаксический анализатор использует информацию из предыдущей фазы и на её основе строит синтаксическое дерево, в котором каждый внутренний узел представляет операцию, а дочерние узлы — аргументы этой операции.

– *Семантический анализ*. Данная фаза компиляции использует синтаксическое дерево и информацию из таблицы символов для проверки исходной программы на семантическую согласованность с определением языка. Важной частью семантического анализа является проверка и приведение типов.

Синтез:

– *Генерация промежуточного кода.* На этой фазе компилятор генерирует низкоуровневое промежуточное представление кода исходной программы, которое можно рассматривать как программу для абстрактной вычислительной машины.
– *Машинно-независимая оптимизация кода.* Фаза машинно-независимой оптимизации кода пытается улучшить промежуточный код. Например, заменить вызов метода на непосредственное выполнение тела метода в месте вызова.
– *Генерация кода.* В качестве исходных данных генератор кода получает промежуточное представление исходной программы и отображает его в целевой язык, как правило в ассемблерный код.
– *Машинно-зависимая оптимизация кода.* Фаза машинно-зависимой оптимизации кода улучшает код целевой машины, учитывая особенности архитектуры процессора.

Процесс анализа компиляции разделен на лексический, синтаксический и семантический анализ по ряду причин:

1. Упрощение разработки. Отделение лексического анализа от синтаксического позволяет упростить как минимум одну из фаз анализа. Например, удаление комментариев и пробельных символов лексическим анализатором значительно проще, чем включение работы с ними в синтаксический анализатор.
2. Увеличение переносимости компилятора. Например, для того, чтобы компилятор работал на разных операционных системах достаточно реализовать только часть генерации кода, вместо написания второго компилятора.
3. Разделение зон ответственности. Каждый компонент отвечает только за определённую часть функциональности. Таким образом

сокращается вероятность допущения ошибки, код каждого компонента становится легко модифицируемым.

1.1.2. Лексический анализатор

При рассмотрении лексического анализа используются три термина:

– *Токен* (от англ. *token* — знак, символ) представляет собой пару, состоящую из имени токена и необязательного атрибута. Имя токена — абстрактный символ, представляющий тип лексической единицы, например, конкретное ключевое слово или последовательность символов, составляющих идентификатор. Имена токенов являются входными символами, обрабатываемыми синтаксическим анализатором. Атрибут токена — строка или структура, объединяющая несколько блоков информации. Блоки информации представляют собой описание лексемы (строчка кода, значение), которая представляет токен. Выражение на языке Fortran `E=M*2` будет представленно в виде последовательности
⟨**id**, Указатель на запись в таблице символов для E⟩
⟨**assign_op**⟩
⟨**id**, Указатель на запись в таблице символов для M⟩
⟨**mult_op**⟩
⟨**number**, Целое значение 2⟩
Примеры токенов приведены в табл. 1.1.

Таблица 1.1. Примеры токенов

Токен	**Неформальное описание**	**Примеры лексем**
if	Символы `i`, `f`	`if`
else	Символы `e`, `l`, `s` `e`	`else`
comparison	<, >, <=, >=, ==, !=	<=, !=
id	Буква, за которой следуют буквы и цифры	`pi`, `score`, `D2`
number	Любая числовая константа	`3.14159`, `0`
literal	Все символы, заключенные в двойные кавычки, кроме самих кавычек	`"core dumped"`

– *Шаблон* — это описание вида, который может принимать лексема токена. В случае ключевого слова шаблон представляет собой последовательность символов, образующую это ключевое слово. Для некоторых токенов шаблон представляет более сложную структуру (например, регулярное выражение) [2].
– *Лексема* — последовательность символов исходной программы, которая соответствует шаблону токена и идентифицируется лексическим анализатором как экземпляр токена.

Основная задача лексического анализатора — чтение входных символов исходной программы, группировка их в лексемы и вывод последовательностей токенов для всех лексем исходной программы. Поток токенов пересылается синтаксическому анализатору для разбора. Лексический анализатор удаляет комментарии, пробельные символы, синхронизирует сообщения об ошибках и раскрывает макросы.

1.1.3. Формальное определение контекстно-свободной грамматики

Для работы синтаксического анализатора требуется описание грамматики языка программирования, которое задаётся с помощью контекстно-свободных грамматик (далее КС-грамматики).

Существует множество типов грамматик (грамматики типа 3, контекстно-свободные грамматики, контекстно-зависимые грамматики и грамматики без ограничений), с помощью которых можно описать различные языки. Для описания языков программирования используются КС-грамматики, потому что их разбор наиболее быстрый.

КС-грамматика используется для определения формального синтаксиса языка программирования. КС-грамматика естественным образом описывает иерархическую структуру множества конструкций языка программирования. Ниже приведён пример КС-грамматики, которая описывает выражение `switch` языка `Java` [3]:

$$
\begin{aligned}
SwitchStatement &\rightarrow \textbf{switch (}\ Expression\ \textbf{)}\ SwitchBlock \\
SwitchBlock &\rightarrow \textbf{\{}\ \{\ SwitchBlockStatements\ \}\ \{\ SwitchLabel\ \}\ \textbf{\}} \\
SwitchBlockStatements &\rightarrow SwitchLabels\ BlockStatements \\
SwitchLabels &\rightarrow SwitchLabel\ \{\ SwitchLabel\ \} \\
SwitchLabel &\rightarrow \textbf{case}\ ConstantExpression\ \textbf{:} \\
SwitchLabel &\rightarrow \textbf{case}\ EnumConstantName\ \textbf{:} \\
SwitchLabel &\rightarrow \textbf{default :} \\
EnumConstantName &\rightarrow Identifier \\
ConstantExpression &\rightarrow Expression
\end{aligned}
$$

КС-грамматика состоит из четырёх частей:

1. *Терминалы* — базовые символы, формирующие строки. Термин «Имя токена» является синонимом слова «Терминал». Пример терминала: `switch`, `case`, `(`, `)`. Другими словами, это листья дерева разбора, представленного на рис. 1.2.
2. *Нетерминалы* — синтаксические переменные, которые обозначают множества строк. В примере, приведённом выше, $Expression$, $SwitchBlock$, $SwitchLabel$ и др. являются нетерминалами. Эти множества строк, обозначаемые нетерминалами, помогают определить язык, порождаемый КС-грамматикой. Нетерминалы также налагают на язык иерархическую структуру, облегчающую синтаксический анализ и трансляцию.
3. *Стартовый символ* — один из нетерминалов, который обозначает множество строк, определяемых КС-грамматикой. По соглашению, продукции стартового символа указываются первыми (в примере, представленном выше, это $SwitchStatement$).
4. *Продукция* — способ, которым терминалы и нетерминалы объединяются в строки. Каждая продукция состоит из заголовка (левая часть), символа $\rightarrow$ и тела (правая часть). Тело может быть пустым или состоять из некоторого количества терминалов и нетерминалов.

1.1.4. Синтаксический анализатор

В процессе компиляции синтаксический анализатор получает строку токенов от лексического анализатора и проверяет, может ли эта строка имен токенов соответствовать грамматике исходного языка. Если программа написанна корректно, то синтаксический анализатор строит дерево разбора и передает его следующей части компилятора для дальнейшей обработки. В противном случае синтаксический анализатор сообщает обо всех выявленных ошибках и продолжает работу с оставшейся частью программы.

Имеется три основных типа синтаксических анализаторов КС-грамматик:

- *Универсальные методы разбора*: например, алгоритмы Кока-Янгера-Касами (Cocke-Younger-Kasami) и Эрли (Earley) [4]. Достоинство данного подхода — возможность работать с любой грамматикой. Недостаток — эти обобщённые методы слишком неэффективны для использования в промышленных компиляторах и для реализации подобных алгоритмов требуются тьюринг-полные языки программирования.
- *Восходящие методы разбора* (bottom-up): построение дерева разбора происходит снизу (от листьев) вверх (к корню). Поток токенов сканируется слева направо. Достоинством такого подхода является эффективный расход памяти (для реализации достаточно расширенного автомата с магазинной памятью) и близкая к линейной оценка работы.
- *Нисходящие методы разбора* (top-down): дерево разбора строится сверху (от корня) вниз (к листьям). Входной поток синтаксического анализатора, как и в восходящих методах, сканируется посимвольно слева направо. Такие методы используются для полностью ручной реализации, так как являются наиболее естественными для восприятия человеком.

Пример разбора для выражения $-(\mathbf{id} + \mathbf{id})$ показан на рис. 1.2.

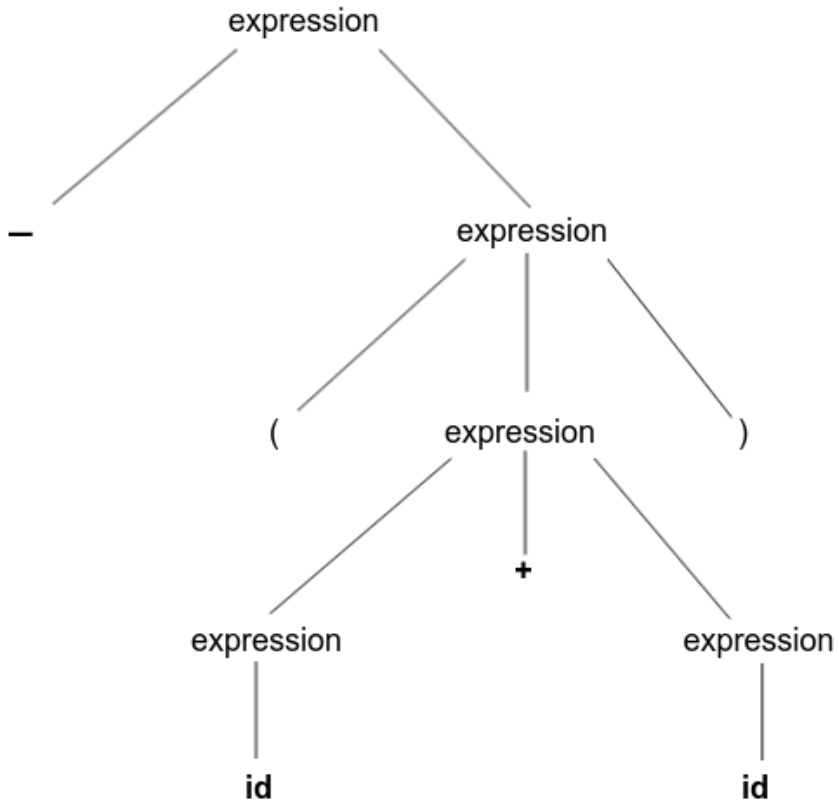

Рис. 1.2. Дерево разбора для выражения $-(\mathbf{id} + \mathbf{id})$

1.1.5. Семантический анализатор

Семантический анализатор использует синтаксическое дерево и информацию из таблицы символов для проверки исходной программы на семантическую согласованность с определением языка. Он также собирает информацию о типах и сохраняет ее в синтаксическом дереве или в таблице символов для последующего использования в процессе генерации промежуточного кода.

Важной частью семантического анализа является проверка типов, когда компилятор проверяет, имеет ли каждый оператор операнды соответствующего типа. Например, многие определения языков программирования требуют, чтобы индекс массива был целым неотрицательным числом. Компилятор должен сообщить об ошибке, если в качестве индекса массива используется число с плавающей точкой или целое отрицательное число [5].

1.1.6. Генератор лексических анализаторов Lex

Lex (в более поздних реализациях *Flex*) — программный инструмент, который позволяет определить лексический анализатор, указывая регулярные выражения для описания шаблонов токенов. Входные обозначения для Lex обычно называют *языком Lex*, а сам инструмент — *компилятором Lex*. Компилятор Lex преобразует входные шаблоны в диаграмму переходов [6].

На рис. 1.3 показанна схема использования генератора Lex.

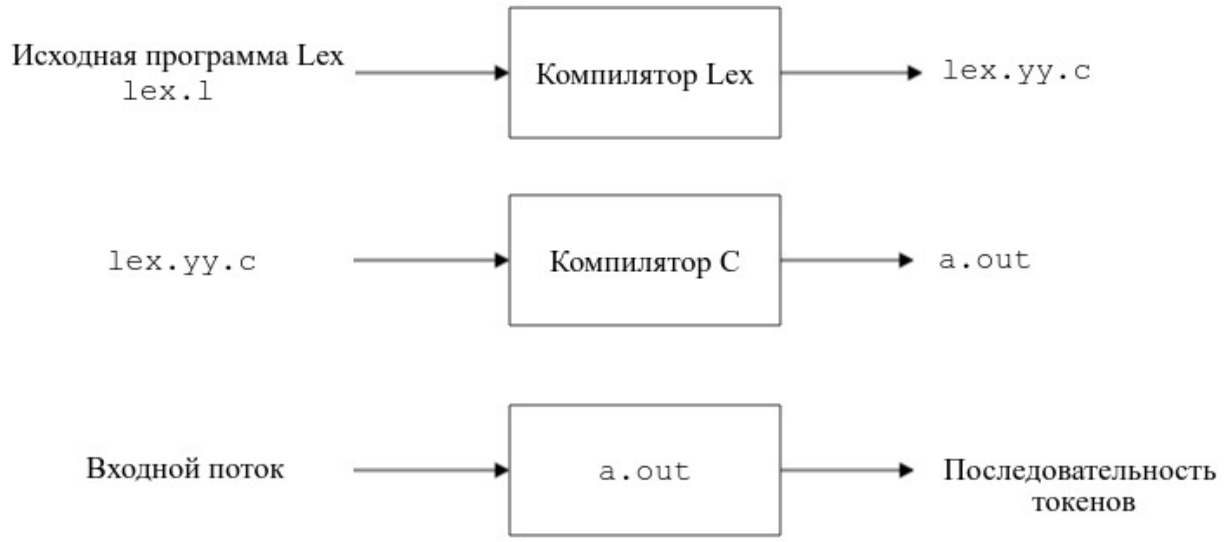

Рис. 1.3. Создание лексического анализатора с помощью Lex

Входной файл `lex.l` написан на языке Lex и описывает генерируемый лексический анализатор. Компилятор Lex преобразует `lex.l` в программу на языке программирования C в файле с именем `lex.yy.c`. Этот файл преобразуется компилятором C в файл с названием `a.out`. Результат работы компилятора C — работающий лексический анализатор, реализующий восходящий анализ, который может получать поток входных символов и выдавать поток токенов [1].

Следующим поколением лексического генератора Lex стал *Flex (Fast lex)*. Flex практически полностью совместим с Lex. Отличия Flex от Lex:

- может компилироваться в язык C++, что позволяет использовать объектно-ориентированные конструкции C++;
- генерирует более производительные лексические анализаторы;
- не имеет ограничений по размеру для таблиц символов (в отличие от Lex);

– небольшие синтаксические различия [6].

1.1.7. Генератор синтаксических анализаторов Yacc

Yacc — компьютерная программа, служащая стандартным генератором синтаксических анализаторов в Unix-системах. Название является акронимом *«Yet Another Compiler Compiler»* («Ещё один компилятор компиляторов») [7]. Yacc генерирует синтаксический анализатор на основе аналитической грамматики, описанной в нотации Бэкуса-Наура или контекстно-свободной грамматики. На выходе Yacc выдаёт код на языке программирования C.

Поскольку синтаксический анализатор, генерируемый с помощью Yacc, требует использования лексического анализатора, то часто он используется совместно с генератором лексических анализаторов, в большинстве случаев это Lex либо Flex.

Создание транслятора с использованием Yacc схематично показано на рис. 1.4.

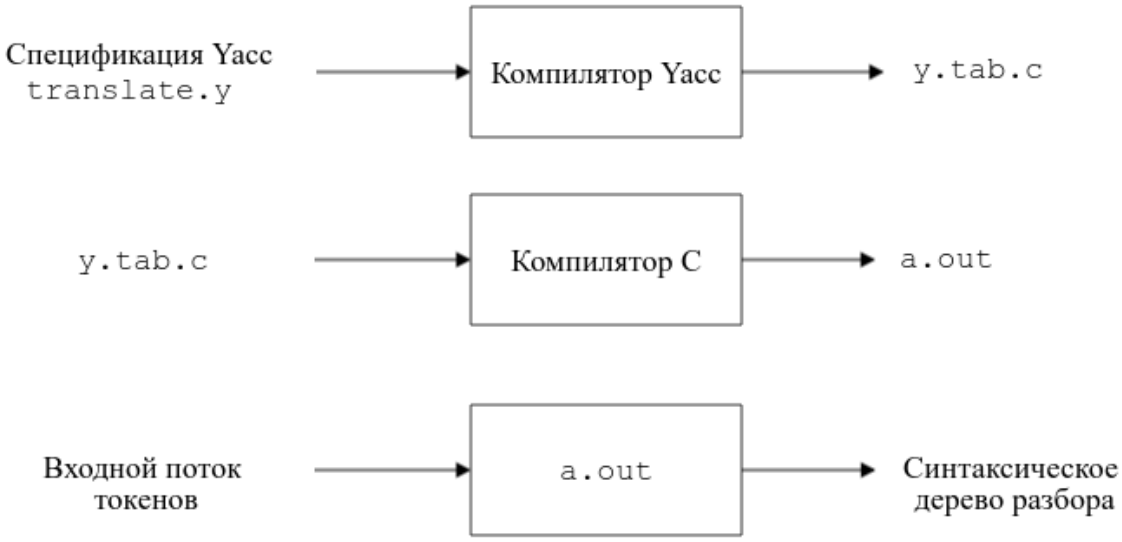

Рис. 1.4. Создание транслятора с помощью Yacc

Вначале создаётся файл, например, `translate.y`, содержащий Yacc-спецификацию разрабатываемого транслятора. Затем этот файл преобразуется в программу `t.tab.c` на языке C, которая является синтаксическим анализатором, который реализует восходящий тип разбора. На вход скомпилированной программе `a.out` поступает поток токенов, а результатом работы является синтаксическое дерево разбора [1].

GNU («GNU is Not Unix») альтернативой Yacc служит *Bison*, он практически полностью совместим с Yacc, за исключением небольших синтаксических дополнений и флагов запуска программы [6].

1.1.8. Универсальные генераторы анализаторов

Помимо инструментов Lex, Flex и Yacc, которые генерируют отдельно синтаксический и лексический анализаторы, существуют универсальные генераторы, способные автоматизировать создание лексического и синтаксического анализаторов одновременно.

ANTLR — генератор синтаксических и лексических анализаторов. Реализует алгоритмы нисходящего анализа. Обладает развитыми возможностями по оптимизации таблиц разбора. За счет этого достигается конкурентоспособность в быстродействии конечного продукта по сравнению с решениями, построенными на реализации восходящих алгоритмов анализа. На вход принимается описание КС-грамматики в расширенной форме Бекуса-Наура. На выход выдается код синтаксического и лексического анализаторов на языках Java или C++ и других. Данный генератор активно используется во многих программных продуктах, таких как среда разработки Eclipse, NetBeans, система баз данных Cassandra и другие.

Coco/R — генератор синтаксических и лексических анализаторов. Реализует алгоритмы нисходящего рекурсивного анализа. На вход подается описание контекстно-свободной грамматики в расширенной форме Бекуса-Наура. Результатом работы данного генератора является код синтаксического и лексического анализаторов на языках Java, C++ или других [8].

1.2. Предметно-ориентированные языки программирования

1.2.1. Определение предметно-ориентированного языка программирования

Предметно-ориентированный язык программирования (Domain-Specific Language, DSL) — язык программирования или исполняемая спецификация, которые предлагают выразительное и мощное решение конкретной предметной проблемы с помощью высокоуровневых абстракций и специализированных нотаций. Чаще всего такой язык программирования не является языком общего применения и предоставляет только

те конструкции, которые необходимы для решения задачи из конкретной предметной области (например, язык Yacc).

Альтернативой предметно-ориентированному языку может стать использование языков программирования общего назначения (объектно-ориентированных) вместе с библиотекой типов и функций, которые отвечают предметным потребностям [9].

1.2.2. Виды предметно-ориентированных языков программирования

Глобально предметно-ориентированные языки программирования делятся на две группы:

– внешние;
– внутренние.

Разработка внешних языков состоит из трёх этапов:

– определение семантической модели;
– определение синтаксической модели (абстрактный и конкретный синтаксис);
– определение правил трансформации (правила, по которым абстрактное представление транслируется в исполнимое).

Для генерации лексического и синтаксического анализатора внешних языков существуют готовые средства, например, связка программ Lex и Yacc, входящая в стандарт POSIX.

Достоинством внешних DSL является узкая специализация, что облегчает процесс решения предметных задач, а также гибкая базовая грамматика. Но у внешних языков существует и ряд недостатков. Например, среду разработки, которая поддерживает и облегчает написание сценариев на внешнем DSL, обычно разрабатывают либо с нуля, либо как дополнение к уже существующей современной интегрированной среде разработки(Integrated Development Environment, IDE). Также, наряду с лёгкостью решения предметных задач, внешние DSL практически никогда не подходят для решения задач в смежных областях.

Альтернативой внешним DSL являются внутренние. Внутренний DSL (Embedded Language) — это подмножество других языков программирования широкого применения [10]. Такой подход позволяет совместить выразительность и мощь предметно-ориентированного языка

вместе с возможностями языков широкого применения (Haskell, Python, Groovy, Scala, Kotlin, Ruby, C#, F#).

Выбирая современный язык программирования общего назначения как основу для создания внутреннего DSL, можно сразу получить готовый набор средств поддержки разработки — современные IDE, которые поддерживают базовый язык [11]. Подходы к созданию DSL, языки и инструментарий представлены на рис. 1.5.

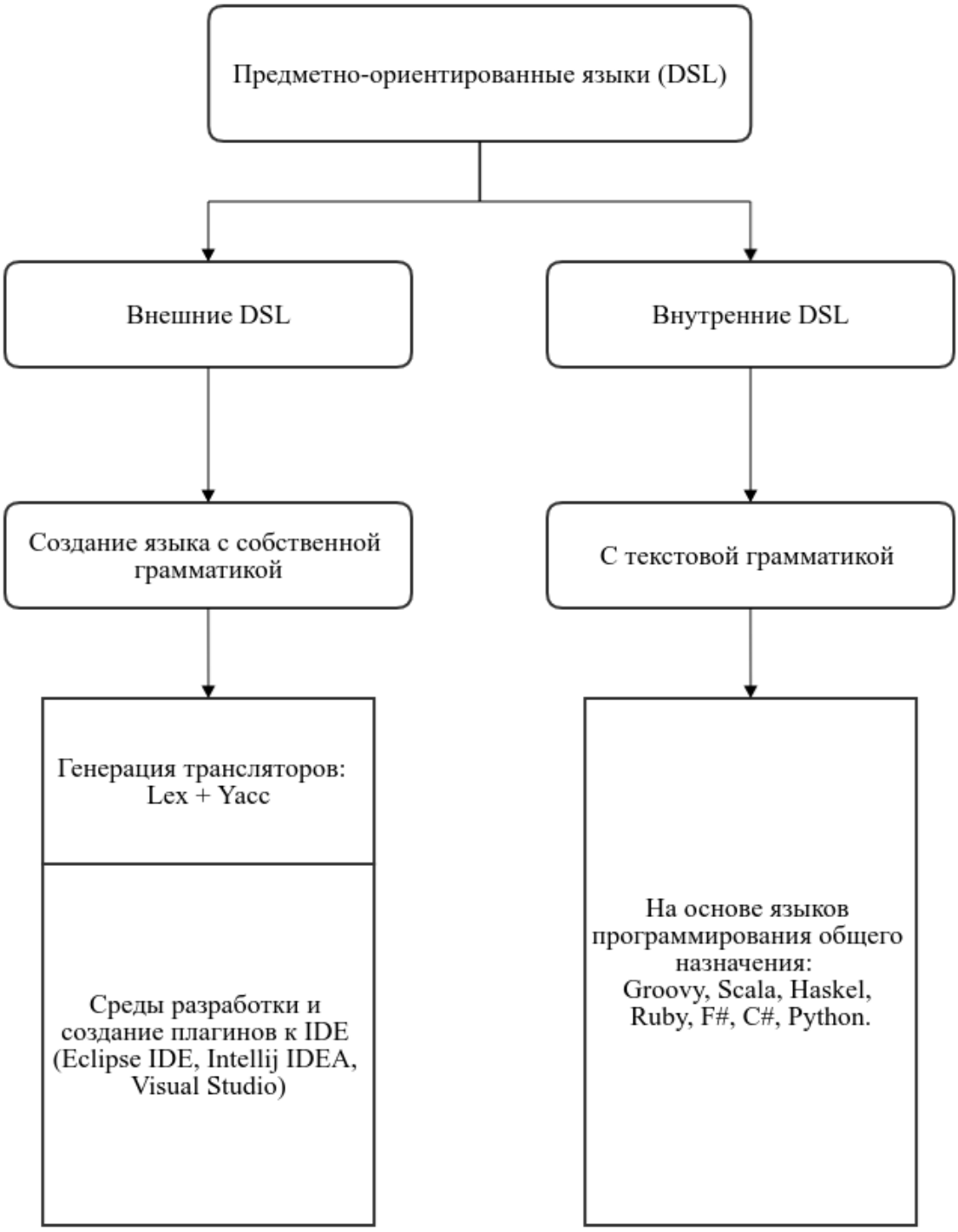

Рис. 1.5. Подходы к созданию внешних и внутренних DSL, языки и инструментарий создания и поддержки DSL

При разработке внутренних DSL возникают ситуации, когда IDE не может определить некоторые конструкции языка. Это может быть часть языка DSL, которая будет интерпретированна только во время выполнения программы (Groovy DSL). Или допустимый домен атрибута (переменной)

может зависеть от окружения, в котором находится содержащий его файл, что значительно усложняет статический анализ, производимый IDE.

1.3. Поддержка DSL в современных редакторах кода

1.3.1. Платформа IntelliJ

IntelliJ — платформа с открытым исходным кодом для создания интегрированных сред разработки, которые включают в себя:

- виртуальную файловую систему;
- поддержку визуального взаимодействия с пользователем;
- текстовый редактор;
- лексический и синтаксический анализ, с поддержкой различных языков программирования;
- инструменты для реализации навигации, авто-дополнения исходного кода, рефакторинга и т.д.;
- систему контроля версий;
- инструменты для отладки приложений;
- графическую поддержку запуска тестов.

Самым популярным продуктом, который построен на основе платформы IntelliJ является IntelliJ IDEA [12].

IntelliJ IDEA — интегрированная среда разработки программного обеспечения для многих языков программирования, в частности Java, JavaScript, Python, разработанная компанией JetBrains. Помимо работы с перечисленными языками, IntelliJ IDEA предоставляет обширный инструментарий для поддержки языков программирования, включая язык DSL.

1.3.2. Поддержка предметно-ориентированных языков программирования в IntelliJ IDE

Поддержка языка — процесс лексического, синтаксического и семантического анализа с целью навигации, автодополнения и подсветки синтаксиса исходного кода, написанного на поддерживаемом языке программирования.

Для реализации поддержки различных языков программирования платформа IntelliJ предоставляет возможность добавления собственных

лексических и синтаксических анализаторов с последующей интеграцией с IntelliJ.

Платформа IntelliJ преобразует исходный код в *PSI (Program Structure Interface)*. PSI — набор функциональных возможностей, который предназначен для анализа файлов, построения синтаксических моделей исходного кода и создания индексов. Функциональные возможности PSI включают в себя:

- быструю навигацию по файлам;
- подсветку синтаксиса;
- динамические проверки кода;
- исправление кода, включая обширный рефакторинг;
- автодополнение кода и д.р.

Поддержка языка DSL, в частности внутреннего, отличается от стандартной поддержки языка программирования общего назначения. Отличие заключается в том, что лексический и часть синтаксического анализов уже проведены для расширяемого языка программирования. Разработка модуля поддержи языка DSL сводится к синтаксическому анализу с использованием уже готовых компонентов (PSI).

Платформа IntelliJ предоставляет два способа интеграции подсветки языков DSL:

1. Отдельное расширение (плагин).
2. Groovy DSL (GDSL) — DSL для реализации подсветки и автодополнения скриптов на языке Groovy в среде разработки Intellij IDEA.

Рассматриваемый язык GRIP DSL написан на языке Groovy, поэтому для его подсветки необходимо использовать Groovy DSL.

2. Реализация GDSL-модуля

2.1. Постановка задачи

Разработать и интегрировать со средой разработки IntelliJ IDEA синтаксический анализатор (GDSL-модуль), который будет отвечать следующим функциональным требованиям:

1. Подсветка синтаксиса языка GRIP DSL.
2. Автодополнение переменных, объявленных на языке GRIP DSL.
3. Автоматическое определение типов переменных языка GRIP DSL.

2.2. Средства реализации

Для реализации модуля поддержки GRIP DSL использовались следующие средства:

- Intellij IDEA 2019.1;
- система контроля версий Git;
- язык программирования Groovy;
- GDSL(Groovy DSL) — DSL для реализации подсветки и автодополнения скриптов на языке Groovy в среде разработки Intellij IDEA.

2.3. Требования к программному и аппаратному обеспечению

Требования к аппаратному и программному обеспечению:

- RAM: 1 Гб минимум, 2 Гб рекомендовано;
- свободное место на диске: 300 Мб + не менее 1 Гб для кэша;
- минимальное разрешение экрана — 1024×768;
- JDK 8 и выше;
- Groovy 2.4 и выше;
- Intellij IDEA 9 и выше.

2.4. Реализация

2.4.1. Структура модуля

Модуль подсветки синтаксиса GRIP DSL реализован на языке GDSL (Groovy DSL) диалекте, который предназначен для подсветки синтаксиса языков DSL, написанных на Groovy, в интегрированной среде разработки IntelliJ IDEA.

Модуль состоит из двух частей:

– функция определения контекста работы модуля;
– функция разбора файла и определения грамматических конструкций, которые необходимо подсветить.

Определение контекста происходит посредством вызова специальной функции `context` с передачей параметров, которые определяют место, где должны подсвечиваться грамматические конструкции языка GRIP DSL, это могут быть:

– расширение файла;
– полное имя файла;
– грамматическая конструкция (блок кода, тело метода и т.д.).

В реализованном модуле контекст определяется как любое замыкание. Замыкание — это блок кода (или анонимная функция), который при выполнении имеет доступ к переменным того контекста, в котором он был объявлен. В то же время, замыкание является объектом, который может быть передан в другие методы, сохранен в переменных, и т.п.

Разбор файла и выделение новых полей и их типов реализованно в функции `contributor`, которая на вход принимает контекст исполнения (имя файла, место вызова), а на выходе неявно возвращает список новых полей и их типов, которые необходимо подсветить.

2.4.2. Интеграция модуля с IDE

Процесс взаимодействия IDE с модулем GDSL представлен на рис. 2.1.

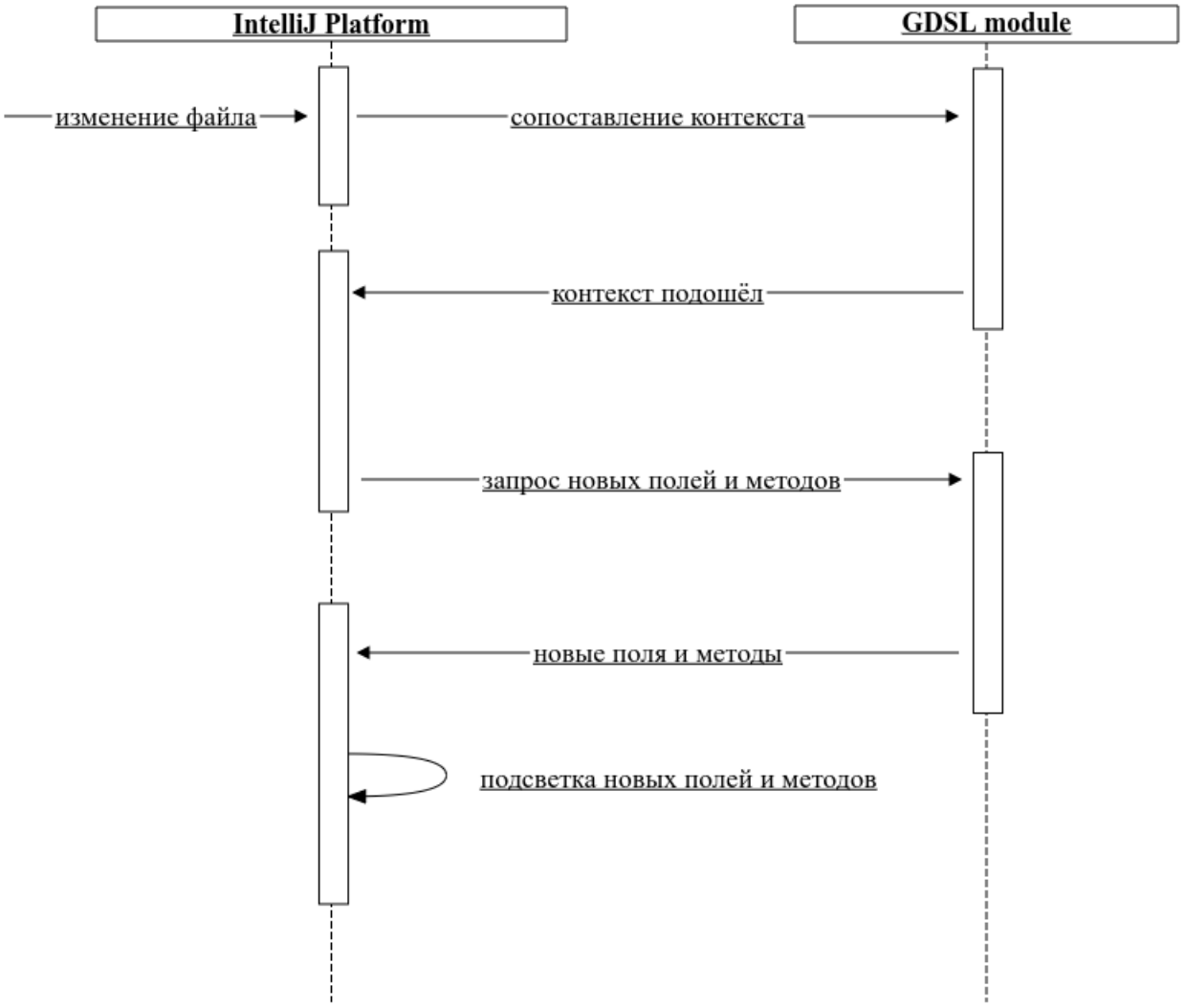

Рис. 2.1. Взаимодействие IDE с модулем GDSL

При каждом действии пользователя (нажатие на клавишу) IntelliJ IDEA обращается к GDSL-модулю, который находится в проекте, и передаёт ему контекст. GDSL-модуль сопоставляет переданный контекст исполнения со своим. Если контекст совпадает, то модуль вызывает функцию `contributor`, в которой происходит разбор файла и поиск новых грамматических конструкций (динамических полей), определенных в GRIP DSL. Результатом работы модуля является набор имен и типов полей, которые соответствуют синтаксису GRIP DSL. Среда IntelliJ IDEA подсвечивает эти поля и делает проверку типов.

2.5. Интерфейс пользователя

2.5.1. Подготовка окружения пользователя

Для того, чтобы среда IntelliJ IDEA начала взаимодействовать с GDSL-модулем, достаточно поместить его в директорию проекта как показанно на рис. 2.2. Конкретное местоположение модуля не имеет значение, главное чтобы оно было в области видимости среды (class path) IntelliJ IDEA. При обнаружении GDSL-модуля, среда сразу же включает его в список обработчиков синтаксического анализа.

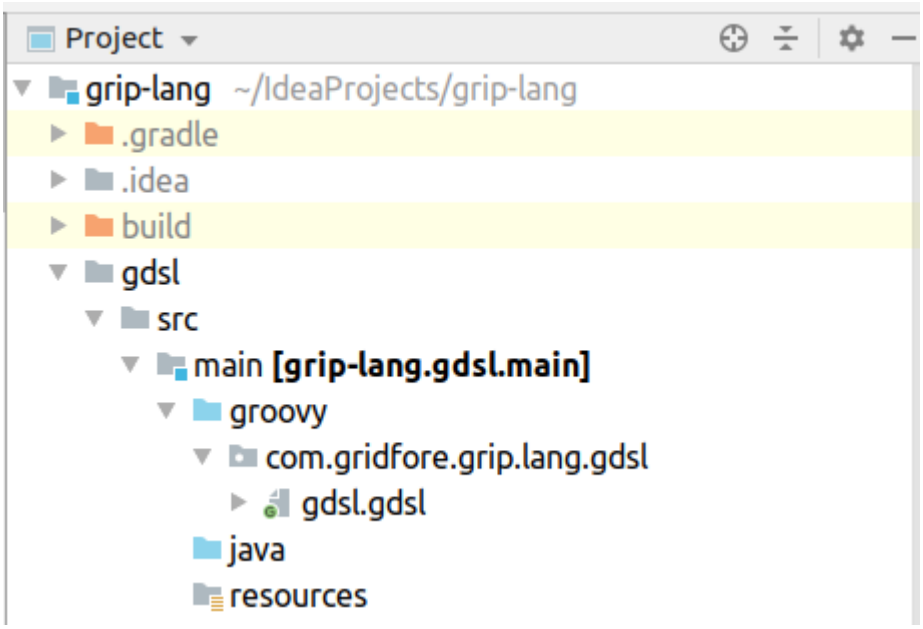

Рис. 2.2. Интеграция GDSL модуля с проектом пользователя

Сам же проект, согласно требованиям языка GRIP DSL, должен соблюдать структуру, которая представлена на рис. 2.3.

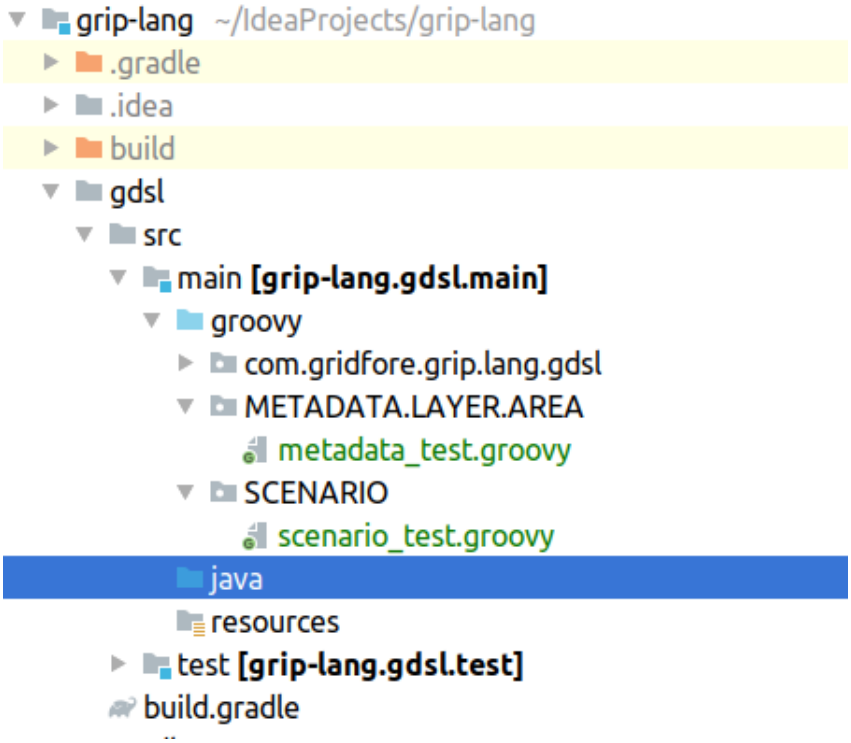

Рис. 2.3. Структура проекта пользователя

2.5.2. Определение метаданных

Метаданные — набор выражений на языке GRIP DSL, который определяет имена и типы полей, с которыми будет работать пользователь.

В пакете `metadata.layer.area` (в терминологии Data Warehousing (хранилище данных) *layer* — Data Source Layer (уровень источника данных), *area* — Staging Area (область загрузки) [13]) должен находиться один или несколько groovy скриптов, внутри которых определяются поля и их типы с помощью GRIP DSL, как показано на рис. 2.4

```
field name: 'CODE'          type Integer    description description: 'test CODE field'
field name: 'ID'            type Long       description description: 'test I_D field'
field name: 'NAME'          type String     description description: 'test NAME field'
field name: 'SURNAME'       type String     description description: 'test SURNAME field'
field name: 'ADDRESS'       type String     description description: 'test ADDRESS field'
```

Рис. 2.4. Определение метаданных

2.5.3. Поддержка метаданных в сценариях исполнения

Сценарий исполнения — это groovy-скрипт, написанный на диалекте GRIP DSL, в котором пользователь определяет интеграцию со множеством хранилищ данных, их обработку и сохранение (*Extract-Transform-Load, ETL*). Согласно GRIP DSL, сценарий исполнения должен находиться в пакете `scenario`.

В процессе написания сценария исполнения пользователь использует метаданные, описанные в пакете `metadata.layer.area` для более удобного и гибкого описания ETL процесса. Поддержка метаданных в сценарии исполнения показанна на рис. 2.5.

Рис. 2.5. Подсветка синтаксиса и автодополнение метаданных

2.6. План тестирования

Тест 1. Проверка отсутствия подсветки у неопределённых полей

Цель: проверить отсутствие подсветки полей, которые не являются членами скрипта метаданных.

Порядок выполнения:

1. Создать пустой скрипт определения метаданных.
2. Ввести название любой переменной в скрипте сценария исполнения, не определённой в контексте данного скрипта.
3. Убедиться, что редактор сообщил об ошибке.

Результат: тест считается пройденным, если редактор кода сообщил о том, что такой переменной не существует, как показанно на рис. 2.6.

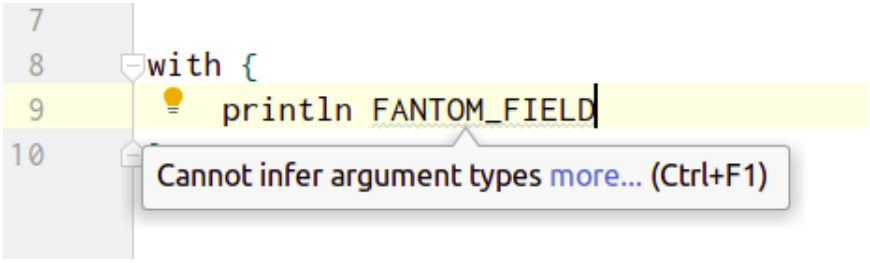

Рис. 2.6. Пример отсутствия подсветки у поля, не определённого в скрипте метаданных

Тест 2. Проверка подсветки полей

Цель: проверить подсветку поля, которое определенно в скрипте метаданных.

Порядок выполнения:

1. Определить поле `TEST_FIELD` типа `String` в скрипте метаданных.
2. Ввести название переменную `TEST_FIELD` в скрипте сценария исполнения.
3. Убедиться, что редактор подсветил переменную.

Результат: тест считается пройденным, если среда разработки определила поле как уже существующее, как показанно на рис. 2.7.

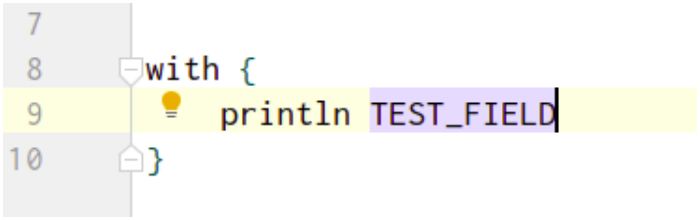

Рис. 2.7. Пример подсветки поля, определённого в скрипте метаданных

Тест 3. Проверка автодополнения

Цель: проверить автодополнение имени поля во время набора текста.

Порядок выполнения:

1. Определить поле `TEST_FIELD` типа `String` в скрипте метаданных.
2. Начать вводить название переменной `TEST_FIELD` в скрипте сценария исполнения.
3. Убедиться, что редактор предлагает авто-дополнение к имени переменной.

Результат: тест считается пройденным, если среда разработки предложила автодополнение поля в процессе написания его имени, как показанно на рис. 2.8.

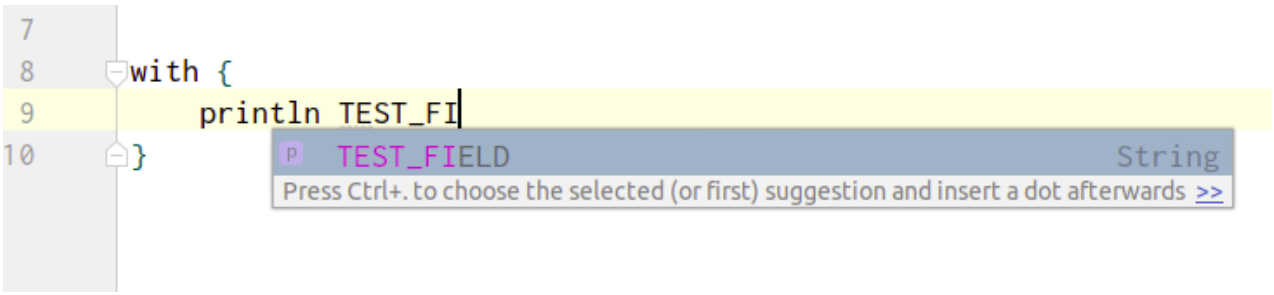

Рис. 2.8. Пример автодополнения поля, определённого в скрипте метаданных

Тест 4. Проверка определения типов

Цель: убедиться в корректной проверки типов.

Порядок выполнения:

1. Определить поле `TEST_FIELD` типа `String` в скрипте метаданных.
2. В скрипте сценария исполнения написать выражение `Integer i = TEST_FIELD + 1`.
3. Убедиться, что редактор сообщает о некорректности данного выражения из-за несоответствия типов.

Результат: тест считается пройденным если среда разработки указала на несоответствие типов, как показанно на рис. 2.9.

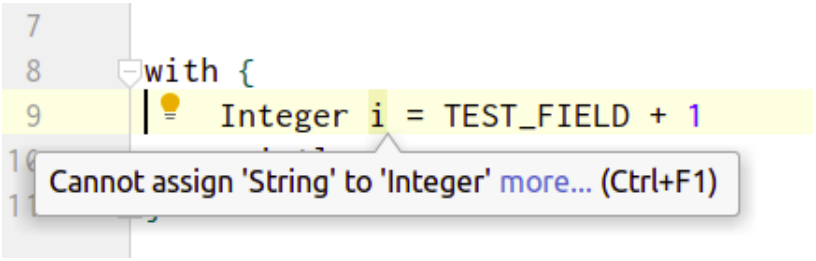

Рис. 2.9. Пример определения типа у поля, определённого в скрипте метаданных

Заключение

В результате проделанной работы был разработан GDSL-модуль, который отвечает следующим функциональным требованиям:

1. Подсветка синтаксиса языка GRIP DSL.
2. Автодополнение переменных, объявленных на языке GRIP DSL.
3. Автоматическое определение типов переменных языка GRIP DSL.

Модуль внедрен в коммерческий продукт «Реактивная аналитическая платформа Gridfore (Gridfore Reactive Intellijent Platform)».

Список литературы

1. Aho, A. V. Compilers: principles, techniques and tools / A. V. Aho. — Pearson Education India, 2003. — С. 109—160.
2. Регулярные выражения. — 2019. — URL: https://ru.wikipedia.org/wiki/Regular_expressions.
3. Java Language Specification. — 14.02.2019. — URL: https://docs.oracle.com/javase/specs/jls/se8/html/jls-14.html#jls-BlockStatements.
4. Earley, J. An efficient context-free parsing algorithm / J. Earley // Communications of the ACM. — 1983. — Т. 26, № 1. — С. 57—61.
5. Синтаксический и семантический анализ. — 25.01.2019. — URL: http://73-antitwilight.blogspot.com/2012/07/blog-post_19.html.
6. Lex & yacc / J. R. Levine [и др.]. — 1992.
7. Yacc: Yet another compiler-compiler / S. C. Johnson [и др.]. — 1975.
8. Создание методики разработки трансляторов предметно-ориентированных языков, на примере прототипа распределенного транслятора языка описания архитектуры. — 2017.
9. Deursen, A. V. Little languages: Little maintenance? / A. V. Deursen, P. Klint // Journal of Software Maintenance: Research and Practice. — 1998. — Т. 10, № 2. — С. 75—92.
10. Van Deursen, A. Domain-specific languages: An annotated bibliography / A. Van Deursen, P. Klint, J. Visser // ACM Sigplan Notices. — 2000. — Т. 35, № 6. — С. 26—36.
11. Ботов, Д. Обзор современных средств создания и поддержки предметно-ориентированных языков программирования / Д. Ботов // Вестник ЮУрГУ. Серия «Компьютерные технологии, управление, радиоэлектроника». — 2013.
12. IntelliJ IDEA. — 20.03.2019. — URL: https://ru.wikipedia.org/wiki/IntelliJ_IDEA.
13. Data Warehousing. — 25.03.2019. — URL: https://www.teradatapoint.com/data-warehousing-architecture.

Приложение А. Листинг кода

```
package com.gridfore.grip.lang.gdsl

import com.intellij.lang.ASTNode
import com.intellij.openapi.project.Project
import com.intellij.psi.JavaPsiFacade
import com.intellij.psi.PsiFile
import com.intellij.psi.PsiPackage
import com.intellij.psi.search.GlobalSearchScope

def ctx2 = context(scope: closureScope())

contributor(ctx2) {
    // XXX without this line the whole contributor stops
   working
    String type = psiType

    Project project = place.manager.project
    JavaPsiFacade jpf = JavaPsiFacade.getInstance(project)

    List<PsiFile> metadataFiles = getAllMetadataFiles(jpf,
   project)

    List<FieldExpression> fields = getFieldsExpressions(
   metadataFiles)

    fields.each { field ->
        property([
                name: field.fieldName,
                type: field.fieldType,
                doc : "from ${field.file} file".toString()
        ])
    }
}

private static synchronized List<FieldExpression>
   getFieldsExpressions(List<PsiFile> metadataFiles) {
    Map<PsiFile, List<FieldExpression.FieldExpressionBuilder>>
    map = [:]

    metadataFiles.collect { file ->
        List<ASTNode> nodes = file.node.getChildren()
```

```
        List<ASTNode> applicationExpressions = nodes.findAll {
    it.elementType.toString() == "APPLICATION_EXPRESSION" }
        List<ASTNode> fieldsExpressions =
    applicationExpressions.collect {
            getLeftChildOrSelf(getLeftChildOrSelf(it))
        }.findAll {
            getLeftChildOrSelf(getLeftChildOrSelf(
    getLeftChildOrSelf(it))).text == 'field'
        }

        map[file] = fieldsExpressions.collect {
            FieldExpression.FieldExpressionBuilder builder =
    FieldExpression.builder()
            builder.file = file
            ASTNode[] fieldNames = getLeftChildOrSelf(
    getLeftChildOrSelf(it)).getChildren()
            if (fieldNames.size() > 2) {
                builder.fieldName = fieldNames[2]?.text?.
    replaceAll("'", '')?.trim()
            }


            if (it.getChildren().size() > 2) {
                String fieldTypeName = it?.getChildren()[2]?.
    text
                if (fieldTypeName) {
                    builder.fieldType = findClassBySimpleName(
    fieldTypeName)
                }
            }

            builder
        }

    }

    map.collectMany { entry -> entry.value }
            .findAll { expr -> expr.fieldName != null }
            .collect { it.build() }
}

private static ASTNode getLeftChildOrSelf(ASTNode node) {
    if (node.getChildren().size() > 0) {
```

```
        node.getChildren()[0]
    } else {
        node
    }
}

private static synchronized Class<?> findClassBySimpleName(
   String fieldTypeName) {
    switch (fieldTypeName.trim()) {
        case "Long": Class.forName("java.lang.Long")
            break
        case "Integer": Class.forName("java.lang.Integer")
            break
        case "Short": Class.forName("java.lang.Short")
            break
        case "Byte": Class.forName("java.lang.Byte")
            break
        case "String": Class.forName("java.lang.String")
            break
        case "Timestamp": Class.forName("java.sql.Timestamp")
            break
        case "BigDecimal": Class.forName("java.math.BigDecimal
   ")
            break
        default: null
    }
}

private static synchronized List<PsiFile> getAllMetadataFiles(
   JavaPsiFacade jpf, Project project) {
    PsiPackage metadata = jpf.findPackage("METADATA")
    List<PsiPackage> packages = []
    metadata.subPackages.collect { packages.addAll(it.
   subPackages) }

    List<PsiFile> files = []
    packages.collect { files.addAll(it.getFiles(
   GlobalSearchScope.allScope(project))) }
    files
}

class FieldExpression {
    final String fieldName
```

```
    final Class fieldType
    final PsiFile file

    FieldExpression(final String fieldName,
                    final Class fieldType,
                    final PsiFile file) {
        this.fieldName = fieldName
        this.fieldType = fieldType
        this.file = file
    }

    static FieldExpressionBuilder builder() {
        new FieldExpressionBuilder()
    }

    static class FieldExpressionBuilder {
        String fieldName
        Class fieldType
        PsiFile file

        FieldExpression build() {
            new FieldExpression(fieldName, fieldType, file)
        }
    }
}
```