

Министерство науки и высшего образования Российской Федерации  
Санкт-Петербургский политехнический университет Петра Великого  
Физико-механический институт  
Высшая школа прикладной математики и вычислительной физики

Работа допущена к защите

Руководитель ОП

\_\_\_\_\_ К.Н. Козлов

«\_\_\_» \_\_\_\_\_ 20\_\_ г.

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА**  
**МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ**

**Инструменты и методы эффективного конструирования**  
**языков предметных областей**

по направлению подготовки 01.04.02 «Прикладная математика и  
информатика»

Направленность 01.04.02\_02 «Математические методы анализа и  
визуализации данных»

Выполнил студент  
гр. 5040102/00201

Д.С.Демьянов

Руководитель  
профессор ВШПМиВФ,  
д.т.н., с.н.с.



Ф.А.Новиков

Санкт-Петербург

2022

**САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ  
ПЕТРА ВЕЛИКОГО**

**Физико-Механический институт  
Высшая Школа Прикладной математики и вычислительной физики**

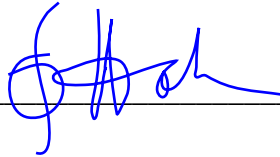
**УТВЕРЖДАЮ**

Руководитель ОП  
Козлов К.Н.  
«\_\_» \_\_\_\_\_ 20\_\_ г.

**ЗАДАНИЕ  
на выполнение выпускной квалификационной работы**

студенту Демьянову Дмитрию Сергеевичу, гр. 5040102/00201  
фамилия, имя, отчество (при наличии), номер группы

1. Тема работы: Инструменты и методы эффективного конструирования языков предметных областей.
2. Срок сдачи студентом законченной работы: 15 июня 2022г.
3. Исходные данные по работе:
  - 3.1. Исходный код инструмента для создания языка предметных областей, разработанного в рамках бакалаврской ВКР Орищенко А.О.
  - 3.2. Определение языка описания онтологий.
  - 3.3. Рекомендованные литературные источники
4. Содержание работы (перечень подлежащих разработке вопросов):
  - 4.1. Разработка инструмента для создания языка предметных областей и написания программы на нём удобного для использования в многопользовательском режиме
  - 4.2. Практика применения инструмента студентами в рамках курса по грамматикам и автоматам.
  - 4.3. Создание языка описания онтологий, преобразующего пакеты сущностей и отношений в документы json формата.
5. Перечень графического материала (с указанием обязательных чертежей): UML диаграммы для языка описания онтологий.
6. Дата выдачи задания: 01 декабря 2021г.

Руководитель ВКР \_\_\_\_\_ д.т.н. Новиков Ф.А.  
(подпись) 

Задание принял к исполнению 01 декабря 2021г.

Студент: \_\_\_\_\_ Демьянов Д.С.  
(подпись)

## РЕФЕРАТ

На 61 с., 33 рисунков, 1 таблица.

КЛЮЧЕВЫЕ СЛОВА: ЯЗЫКИ ПРЕДМЕТНЫХ ОБЛАСТЕЙ, ГРАММАТИКИ, АВТОМАТЫ, ДИСКРЕТНАЯ МАТЕМАТИКА, ОНТОЛОГИИ, UML.

Тема выпускной квалификационной работы: «Инструменты и методы эффективного конструирования языков предметных областей».

Данная работа посвящена разработке инструмента для создания языков предметных областей и практике применения разработанного инструмента в учебном процессе.

Задачи, которые решались в ходе работы:

1. Разработка инструмента для создания языков предметных областей и написания программы на нём удобного для использования в многопользовательском режиме.
2. Тестирование разработанного инструмента.
3. Практика применения студентами и поддержка инструмента в рамках курса «Грамматики и автоматы».

В результате предложен метод взаимодействия инструментов для обработки синтаксиса и семантики для удобного описания языков предметных областей. В качестве инструмента для обработки синтаксиса был взят ANTLR4, для описания семантики был взят язык CIAOv2. Разработан инструмент для создания языков предметных областей и написания программы на нём. Проведена демонстрация возможностей и практической применимости полученной системы посредством тестирования и использования на практике студентами в рамках курса «Грамматики и автоматы».

## **ABSTRACT**

61 pages, 33 figures, 1 table.

**KEYWORDS:** DOMAIN-SPECIFIC LANGUAGES, GRAMMAR, AUTOMATA, DISCRETE MATHEMATICS, ONTOLOGIES, UML.

The subject of the graduate qualification work: «Tools and methods for the effective design of domain specific languages».

This given work is devoted to the development of a tool for creating domain-specific languages and the practice of using the developed tool in the educational process.

The research set the following goals:

1. Develop a tool for creating domain-specific languages and writing a program using it. Tool should be convenient for use in multi-user mode.
2. Test the developed tool.
3. Practice using of the tool by students and support it in scope of the course "Grammars and automata".

As a result, a method of interaction of tools for processing syntax and semantics was proposed. ANTLR4 was taken as a tool for syntax processing and CIAOv2 was taken to describe the semantics. A tool for creating domain languages and writing a program on it was developed. Capabilities and practical applicability demonstration of the resulting system was carried out by testing and using it in practice by students as part of the course "Grammars and Automata".

## Оглавление

Введение.....	7
Глава 1. Обзор.....	9
1. Описание DSL.....	9
1.1    Описание синтаксиса .....	10
1.2    Описание семантики .....	12
2. Обзор инструментальных средств.....	13
2.1    Meta Programming System.....	13
2.2    Rational Rose.....	17
3. Задачи для исследования.....	18
Глава 2. Реализация инструментального средства DSL-editor.....	19
1. Описание синтаксиса .....	19
2. Описание семантики .....	21
2.1    Мотивация.....	22
2.2    Визуализация описанных автоматов .....	23
2.3    Генерация кода по CIAOv2.....	25
3. Архитектура и дизайн решения .....	30
Глава 3. Применение инструмента.....	34
1. Тестирование: генерация деревьев на языке DOT.....	34
2. Применение на практике - язык описания онтологий.....	38
2.1    Онтологии дискретной математики .....	39
2.2    Постановка задачи для реализации в DSL-editor.....	42
2.3    Результаты применения .....	43
3. Расширенные возможности.....	47
3.1    Итерация Цейтина .....	47
3.2    Дополнительные обработчики .....	48
3.3    Авторизация.....	50
4. Раскрутка ANTLR .....	51
Заключение .....	52
Список использованных источников .....	53
Приложение 1 .....	56
Приложение 2 .....	57
Приложение 3 .....	58
Приложение 4 .....	61

## Список иллюстраций

1. Синтаксическая диаграмма “оператор цикла с предусловием” .....	12
2. Пример кода в проекционном редакторе .....	14
3. Традиционный и MPS подход к способу задания DSL .....	15
4. Пользовательский интерфейс Rational Rose .....	18
5. Процесс генерации дерева разбора .....	20
6. Нотация диаграммы переходов на примере счетного триггера .....	24
7. Синтаксическая диаграмма языка CIAOv2 .....	26
8. Программа на CIAOv2 для реализации счетного триггера .....	28
9. Шаблон кода на Python3 для реализации счетного триггера .....	29
10. Процесс генерации диаграммы и заготовки программы .....	30
11. Диаграмма потоков данных .....	31
12. Интерфейс основного компонента приложения .....	32
13. Визуализация методов API .....	34
14. Грамматика языка описания деревьев .....	35
15. Пример программы на языке описания деревьев с ошибкой .....	35
16. Пример вывода при синтаксической ошибке .....	36
17. Пример корректной программы на языке описания деревьев .....	36
18. Пример построенного дерева разбора .....	36
19. Диаграмма конечных автоматов для языка генерация деревьев .....	37
20. Мета модель онтологии. Общая структура .....	39
21. Мета модель онтологии. Сущности. ....	40
22. Мета модель онтологии. Атрибуты и операции. ....	40
23. Мета модель онтологии. Отношения. ....	41
24. Синтаксическая диаграмма для языка описания онтологий .....	44
25. Пример программы на DSL .....	45
26. Дерево разбора для языка описания онтологий .....	45
27. Диаграмма конечных автоматов для языка описания онтологий .....	47
28. Улучшенная модель обработки потока данных для DSL-editor .....	49
29. Кнопка загрузки обработчиков .....	50
30. Форма загрузки обработчиков .....	50
31. Страница авторизации пользователя .....	51
32. Страница со списком проектов пользователя .....	51
33. Фрагмент дерева разбора для раскрутки ANTLR .....	52

## Список таблиц

1. Грамматика, порождающая язык описания онтологий .....	43
--	----

## Введение

Создание программного обеспечения необходимо во множестве отраслей промышленности и экономики. Разработка приложений связана с решением задач в различных предметных областях, например, таких как хранение данных или проектирование интерфейса пользователя. Для разработки приложений часто пользуются языками программирования общего назначения, например, Java, C, C++ и т.д. Такие языки рассчитаны для решения достаточно большого класса задач, однако они не являются оптимальными для каждой конкретной задачи, особенно в случае, когда она нестандартна и имеет много особенностей. Также при программировании с помощью языков общего назначения возникает шаблонный код, который будет повторяться из раза в раз и который не несет в себе никакой специфической логики. Программисту приходится писать этот код каждый раз, как правило, занимая ненужное место, загромождая программу и отвлекая внимание от бизнес-логики.

В отличие от языков программирования общего назначения, языки предметных областей (Domain-Specific Language — DSL) — это языки программирования с более высоким уровнем абстракции, которые отражают специфику решаемых с их помощью задач [1] и позволяют формулировать решение целевой задачи в терминах предметной области. Такие языки оперируют понятиями и правилами из определенной предметной области. Преимущества использования DSL заключаются в существенном снижении денежных и временных затрат на разработку. В некоторых случаях, благодаря снижению сложности, разрабатывать ПО могут специалисты, поверхностно знающие программирование, но хорошо знающие предметную область. Подход, в основе которого лежит идея создания DSL-языка, специально разработанного под поставленную задачу, называется языково-ориентированное программирование (Language-oriented programming — LOP)

При всех достоинствах DSL у них есть большой недостаток – сложность разработки. Если языки общего назначения позволяют создавать программы безотносительно предметной области, то в случае

DSL для каждой предметной области, а иногда и для каждой конкретной задачи приходится создавать свой предметно-ориентированный язык. Если предметная область достаточно проста и язык несложен, то построить транслятор для этого языка также будет несложно. В целом для предметной области создание и реализация предметно ориентированного языка потребуют больших усилий, несмотря на то, что в настоящее время существуют генераторы лексического и синтаксического анализаторов, и другие инструменты для разработки компиляторов, облегчающие работу программиста-разработчика.

Развитие в области языково-ориентированного программирования привело к созданию языковых инструментальных средств [3], под которыми понимается набор программ, предоставляющих разработчику возможности по созданию и использованию предметно-ориентированных языков. Одна из задач такого языкового инструментария – упрощение процесса разработки DSL при создании, освоении предметно ориентированных языков, применении их в процессе обучения и использовании в бизнес-приложениях.

*Целью работы* является разработка программного решения инструментальной поддержки конструирования языков предметных областей и применение его в процессе обучения.



# Глава 1. Обзор

## 1. Описание DSL

Предметно-ориентированный язык — это язык программирования с ограниченными выразительными возможностями, ориентированный на решение задач в конкретной предметной области. Сильная связь предметно-ориентированных языков с предметной областью позволяет разработчику оперировать понятиями, близкими к этой предметной области, формулировать прикладные задачи в более естественном виде, тем самым заметно упрощая написание сложных программ.

DSL, как и любой другой язык программирования, является средством общения между программистом и компьютером.

Язык программирования, как и любой формальный язык — это множество цепочек символов над некоторым алфавитом. Но кроме алфавита язык предусматривает и задание правил построения допустимых цепочек, поскольку обычно далеко не все цепочки над заданным алфавитом принадлежат языку.

Специалисты по языкам программирования обычно выделяют следующие составляющие описания языка [4]:

1. синтаксис — совокупность правил, которые определяют, как сформированы предложения языка. Синтаксис задаётся с помощью правил, которые описывают структуру предложений с помощью грамматических конструкций языка.
2. семантика — правила и условия, определяющие соотношения между элементами языка и их смысловыми значениями, а также интерпретацию содержательного значения синтаксических конструкций языка.

Таким образом, для описания DSL требуется задать синтаксис и семантику. Далее разберем основные подходы к формализации синтаксиса и семантики языка.

## 1.1 Описание синтаксиса

Использование грамматики в задании синтаксиса обеспечивает значительное преимущество разработчикам языков программирования и создателям компиляторов [5]. Контекстно-свободная грамматика частный случай грамматики, у которой левые части всех продукций являются одиночными нетерминалами (объектами, обозначающими какую-либо сущность языка и не имеющими конкретного символического значения) [19]. Она задает точное и при этом простое для понимания определение языка программирования. Также хорошо построенная грамматика придает языку программирования структуру, которая способствует трансляции исходной программы в корректный код и выявлению ошибок.

Один из наиболее распространенных способов описания синтаксиса языка – это форма Бэкуса-Наура (БНФ) [14]. Этот способ был разработан для описания Алгола-60, однако, в дальнейшем он использован для многих других предметно-ориентированных языков. Существует также расширенная форма Бэкуса-Наура (РБНФ). Главное преимущество РБНФ перед БНФ — возможность описывать простые повторяющиеся конструкции неопределённой длины (списки, строки, последовательности и так далее) без рекурсивных правил, что делает запись в РБНФ одновременно и короче, и удобнее для восприятия человеком. Естественной платой за преимущества РБНФ перед БНФ является увеличение сложности автоматической интерпретации его описаний.

Стандартная запись расширенной формы Бэкуса-Наура состоит из следующих частей [6]:

- терминальные символы. Такие символы существуют в виде предопределенных идентификаторов или последовательностей символов в кавычках;
- нетерминальные символы. Для этих символов существуют правила замены левой части на правую. Такие символы обычно составляют

грамматические конструкции языка программирования (операторы, выражения и прочие);

- правила грамматики. Правило состоит из двух частей: нетерминал в левой части и выражение – в правой. Правила сопоставляют нетерминальным символам некоторые выражения.

НЕТЕРМИНАЛ := ВЫРАЖЕНИЕ

- выражение конкатенации. Такое выражение обозначает, что нетерминал в левой части правила соответствует объединению двух символов, указанных в конкатенации через запятую или пробел.  $A=B, C$
- выражение выбора. Такое выражение обозначает, что нетерминал в левой части правила соответствует или одному, или другому символу, указанному в выборе.

$A=B \mid C$

- выражение условия. Такое выражение обозначает, что нетерминал в левой части правила либо пустой, либо соответствует символу в правой части.

$A=[B]$

- выражение повторения. Такое выражение обозначает, что нетерминал в левой части правила либо пустой, либо соответствует любому количеству конкатенаций подряд идущих символов из правой части.

$A=\{B\}$

- выражение группировки. Такое выражение используется для формирования более сложных выражений.

$A=(A \ B)$

Итоговая грамматика – список правил. Каждое правило последовательно определяет все нетерминальные символы грамматики. Таким образом, каждый нетерминальный символ сводится к комбинации терминальных символов путём последовательной замены нетерминальных символов в правой части правила. РБНФ никак не регламентирует последовательность записи правил.

Однако, такие правила могут быть заданы программным инструментом, который генерирует синтаксический анализатор по заданной грамматике, который преобразует входные данные (как правило, текст) в структурированный формат (как правило, дерево разбора).

Задать синтаксис языка можно также через синтаксические диаграммы [7].

Синтаксическая диаграмма — это направленный граф с одним входным ребром и одним выходным ребром и помеченными вершинами. Представление грамматики состоит из набора синтаксических диаграмм. Каждая диаграмма определяет нетерминал. Существует главная диаграмма, которая определяет язык следующим образом: чтобы принадлежать языку, слово должно описывать путь в главной диаграмме. Каждая диаграмма имеет точку входа и конечную точку. Диаграмма описывает возможные пути между этими двумя точками, проходя через другие нетерминалы и терминалы. При этом терминалы представлены круглыми ячейками, а нетерминалы — прямоугольными. Таким образом, цепочка пометок при вершинах на любом пути от входного ребра к выходному — это цепочка языка, задаваемого синтаксической диаграммой. Пример синтаксической диаграммы можно рассмотреть на рисунке 1.

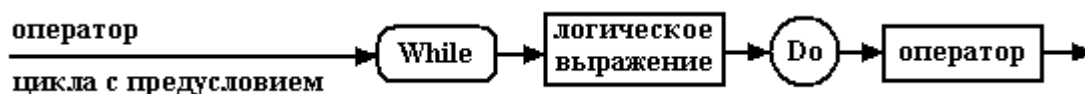


Рисунок 1. Синтаксическая диаграмма "оператор цикла с предусловием"

## 1.2 Описание семантики

В работе [9] рассматриваются три основных подхода к формализации семантики:

1. Операционная семантика специфицирует поведение языка программирования, определяя для него абстрактную машину. Состояние машины представляет собой значение текущего состояния, а поведение ее

определяется функцией перехода, которая для каждого состояния либо указывает следующее состояние, произведя шаг вычисления (упрощения), либо объявляет машину остановившейся.

2. Денотационная семантика рассматривает значение с более абстрактной точки зрения: в качестве значения принимается не последовательность машинных состояний, а некоторый математический объект, например, число или функция. Построение денотационной семантики для языка состоит в нахождении некоторого набора семантических доменов («типов данных» — областей определения функций), а также определении функции интерпретации, которая ставит элементы этих доменов в соответствие выражениям.

3. Аксиоматическая семантика предполагает более прямой подход к этим законам: вместо того, чтобы сначала определить поведение программ (с помощью операционной или денотационной семантики), а затем строить законы, соответствующие этому поведению, аксиоматические методы принимают сами законы в качестве определения языка. Значение выражения — это то, что о нем можно доказать.

В качестве основного подхода описания семантики для разрабатываемого в работе инструмента задания DSL был выбран операционный подход, манипулирующий автоматами.

## **2. Обзор инструментальных средств**

В данном разделе рассмотрим программные продукты, которые позволяют проектировать DSL.

### **2.1 Meta Programming System**

это языковая рабочая среда с открытым исходным кодом, ориентированная на предметно-ориентированные языки. В настоящее время она применяется в различных областях, от электротехники и интеллектуального анализа данных до страховой отрасли и государственных организаций. В ней конечным пользователям DSL предлагается взаимодействовать с абстрактным

синтаксическим деревом (АСД), которое редактируется проекционным редактором в текстовой манере. Пример проекции для подсчета скидки клиента, встроенной в исходный код программы на языке Java показан на рисунке 2.



Рисунок 2. Пример кода в проекционном редакторе

Для описания языка используются «аспекты», для каждого из которых существует собственный язык, заточенный под проблемы, которые возникают в аспекте:

1. *структурный аспект* - определяет виды узлов (называемые концептами), которые можно использовать в пользовательских моделях. Каждый узел в программе(модели) относится к своему концепту. Концепты определяют, какие свойства, дочерние узлы и ссылки могут иметь узлы. Концепты могут расширять другие концепты и реализовывать интерфейсы ConceptInterfaces.
2. *ограничительный аспект* - ограничивает отношения между узлами, а также допустимые значения свойств за пределами правил, определенных в структуре.
3. *аспект-редактор* - вместо определения синтаксического анализатора, который переводил бы код из редактируемой формы (то есть текста) в древовидную структуру, которой мог бы манипулировать компьютер, MPS предлагает концепцию проекционного редактора, который позволяет

пользователю редактировать AST напрямую. Аспект редактора позволяет разработчикам языков создавать пользовательский интерфейс для редактирования своих концептов.

4. *аспект-генератор* - модели, написанные на одном или нескольких языках, в конечном итоге преобразуются в исполняемый код на каком-либо целевом языке и платформе общего назначения, таких как Java. Правила трансляции концептов и их надлежащий порядок определены в аспекте-генераторе.

5. прочие аспекты, выполняющие задачи проверки типов, детектирования недостижимого кода, тестирования, миграций между версиями языка, отладки, автодополнения в IDE.

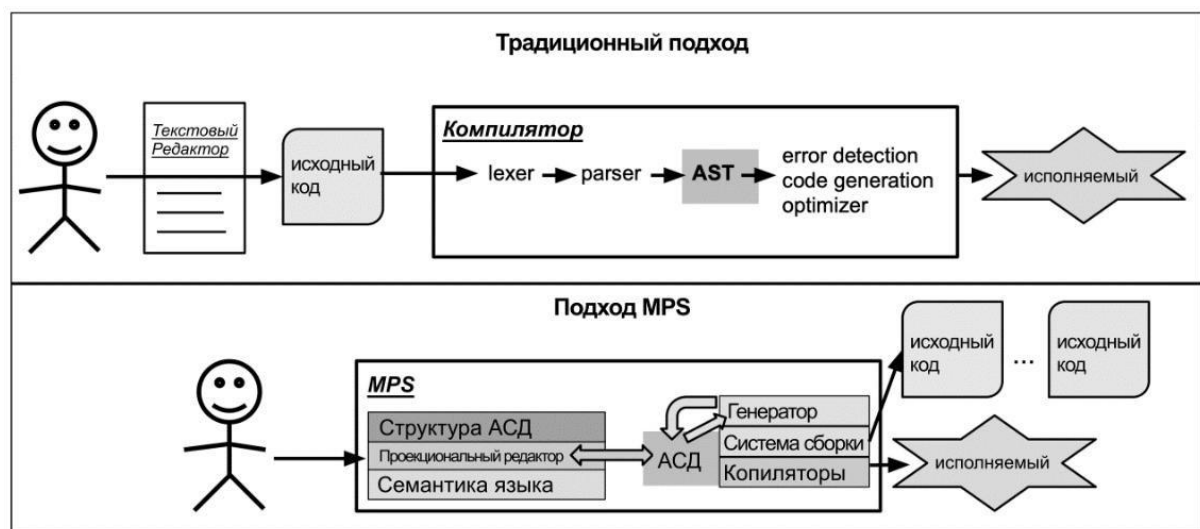


Рисунок 3. Традиционный и MPS подход к способу задания DSL

Чтобы разработать или расширить программист последовательно описывает модель языка и необходимые аспекты этого языка, например, систему типов, поведение в редакторе. Далее описывает, как модель нашего языка компилируется в текст или в модель другого языка. И наконец собираем наш язык в плагин для IntelliJ IDEA и использует его. Все переходы из одной модели в другую делаются автоматически. В итоге получается готовый скомпилированный код.

По сравнению с традиционным подходом (рисунок 3), когда программа представляется в виде текста и далее с помощью лексического и

синтаксического анализаторов трансформируется в АСД, подход MPS, работая с АСД напрямую, обладает следующими преимуществами:

1. Возможность комбинировать различные языки в одной программе без риска неоднозначности парсинга конкретного синтаксиса.
2. Представление элементов языка не только в виде текстовых примитивов, но и формул, таблиц, диаграмм, графиков, что упрощает восприятие и позволяет сосредоточиться на предметной области.
3. Интегрированная со средой разработки поддержка автодополнения, навигация, подсветка синтаксиса и ошибок, инспекции кода, инструменты рефакторинга. Для реализации не требуется знать теорию синтаксических анализаторов.
4. Упрощённая разработка расширений уже существующих языков. Расширение компонентов компилятора для сложных языков требует, как наличие опыта и особых умений, из-за риска случайно ввести неоднозначности в грамматику языка, так и знание внутреннего устройства конкретного компилятора. Расширение же языка с помощью MPS состоит из определения новых структур языка.

При этом есть и ряд минусов:

1. Поддерживается только платформа IntelliJ IDEA.
2. Крутая кривая обучения [17]: требуется усвоить большое количество абстракций, прежде получится написать язык.
3. Стандартный проекционный редактор, который автоматически генерируется для каждой структуры (концепта) языка при отсутствии аспекта редактора неудобен в использовании. [22]
4. Структурные правила описывают как выглядит абстрактное синтаксическое дерево, на какие узлы оно разбито. Но отсутствует информация о конкретных характеристиках написанного кода: о пробелах, разрывах строки и прочем форматировании — которая может быть важна для реализуемого DSL.



## 2.2 Rational Rose

IBM Rational Rose - популярное средство визуального моделирования, которое считается популярным среди средств визуального проектирования приложений [20]. Этот продукт входит в состав пакета IBM Rational Suite и предназначен для моделирования программных систем с использованием широкого круга инструментальных средств и платформ.

Являясь простым и мощным решением для визуальной разработки информационных систем любого класса, Rational Rose позволяет создавать, изменять и проверять корректность модели. Rational Rose объединяет команду разработчиков на базе универсального языка моделирования UML, который определяет стандартную графическую символику для описания архитектуры ПО. Любые участники проекта - аналитики, специалисты по моделированию, разработчики и другие - могут использовать модели, построенные в Rational Rose, для большей эффективности создания конечного продукта.

Продукт Rational Rose в отличие от подобных средств проектирования способен проектировать системы любой сложности, то есть инструментальный программы допускает как высокоуровневое (абстрактное) представление (например, схема автоматизации предприятия), так и низкоуровневое проектирование (интерфейс программы, схема базы данных, частичное описание классов). Вся мощь программы базируется всего на 7 диаграммах, которые в зависимости от ситуации способны описывать различные действия[21].

Однако стоит отметить, что Rose позволит упростить разработку сложных классов посредством выразительных возможностей по графическому представлению классов и их взаимоотношений. Иначе говоря, этот инструмент сможет сгенерировать только код классов и их связей на определенном языке программирования. Исполняемый код генерироваться не будет. Пользовательский интерфейс средства Rational Rose изображен на рисунке 4.

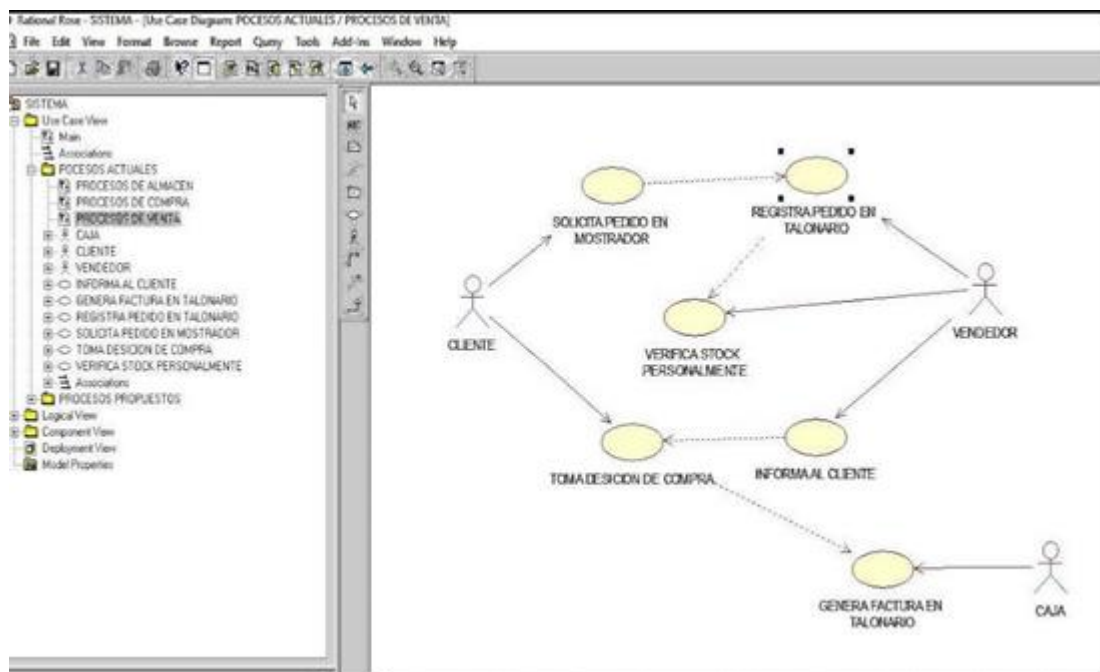


Рисунок 4. Пользовательский интерфейс Rational Rose

### 3. Задачи для исследования

В результате, целью этой работы является создание программного решения инструментальной поддержки конструирования языков предметных областей, удобной для использования студентами.

Чтобы достигнуть поставленной цели, были поставлены следующие задачи:

1. Предложить метод описания семантики на основе автоматного программирования
2. Разработать программное решение, которое позволит: задавать синтаксис предметно-ориентированного языка с помощью формальной грамматики задавать семантику предметно-ориентированного языка с помощью методики, разработанной в пункте 1.
3. Провести тестирование, разработав конкретные языки предметных областей с использованием инструмента.
4. Провести поддержку инструмента в ходе применения его студентами и улучшить инструмента по результатам поддержки.

## **Глава 2. Реализация инструментального средства DSL-editor**

Глава посвящена изложению подхода к описанию синтаксиса DSL с помощью формальной грамматики, семантики DSL — с помощью модели взаимодействия автоматных объектов. Также рассмотрена архитектура и дизайн программного решения инструментальной поддержки DSL.

### **1. Описание синтаксиса**

Описать синтаксис входного языка пользователю предлагается с помощью инструмента ANTLR4 (ANother Tool for Language Recognition) [16]. ANTLR4 — это мощный генератор синтаксических анализаторов для чтения, обработки, выполнения или перевода структурированного текста или двоичных файлов, позволяющий автоматически создавать программы лексического и синтаксического анализа на одном из целевых языков программирования (C++, Java, C#, Python). Помимо того, что он поддерживает РБНФ входных грамматик, инструмент обладает всеми преимуществами LL-техники генерации парсеров: простое отслеживание ошибок вывода, простое осуществление “нагрузки” анализа различными семантическими конструкциями и функциями [17].

ANTLR4 представляет собой набор возможностей, состоящий из двух частей:

1. генератор анализаторов — приложение, которое получает на вход описание грамматики и генерирует код для лексического и синтаксического анализатора.
2. библиотека времени выполнения, которая используется для создания конечной программы. Эта библиотека содержит базовые классы для анализаторов, а также классы, управляющие потоками символов, обрабатывающие ошибки разбора, генерирующие выходной код на основе шаблонов и многое другое.

В данной работе [18] автор утверждает, что ANTLR4 достаточно прост для понимания новичка в области конструирования языков программирования, так как этот инструмент тщательно документирован, для него существует

много успешных примеров реализации DSL, а также доступны инструменты разработчика, такие как интегрированные плагины среды разработки и собственная среда разработки AnlrWorks.

Так как представленный в работе инструмент разрабатывается для студентов, то аргументы выше стали основополагающими в выборе средства для описания синтаксиса. Кроме того, DSL, создающиеся студентами в рамках курсовой работы, в основном простые, то есть их синтаксис в большинстве случаев регулярный и не имеет неоднозначностей в своем строении.

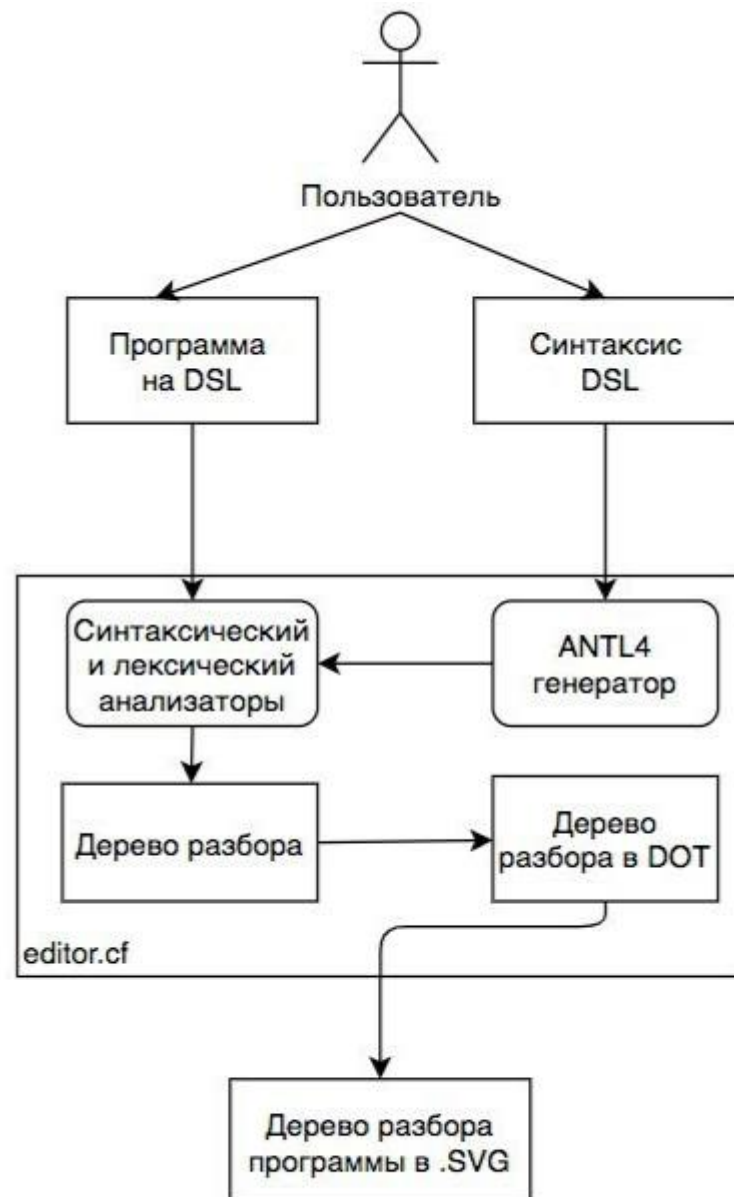


Рисунок 5. Процесс генерации дерева разбора

Создатель языка описывает синтаксис в расширенной форме Бэкуса-Наура с помощью инструмента ANTRL4. Далее принимая на вход текст входной программы, разработанный инструмент может отобразить пользователю дерево разбора программы. Дерево разбора программы является удобным и наглядным способом отладки синтаксиса программы.

Процесс создания дерева разбора начинается с генерации лексического и синтаксического анализаторов (рисунок 5). После этого система строит дерево разбора программы. Дерево разбора конвертируется из внутреннего представления ANTLR4 в нотацию языка описания графов и диаграмм DOT

Далее с помощью пакета утилит по автоматической визуализации графов Graphviz (Graph Visualization Software [24]) формируется векторное изображение в SVG формате. Данный файл может быть просмотрен и скачан пользователем. SVG-формат был выбран из-за таких очевидных преимуществ, как:

1. возможность редактирования. То есть создатель языка при желании, может изменить, например, цвет или расположение узлов дерева в полученном файле.
2. возможность масштабируемости. Так как в общем случае деревья разбора получаются громоздкими для больших программ, важным фактором будет выступать увеличение любой части SVG-изображения без потери качества.

Если при формировании дерева разбора возникли исключительные ситуации, то они будут транслированы пользователю с указанием этапа обработки программы, на котором возникла ошибка. Также по возможности будет указан номер проблемной строки или символа, если ошибка возникла при определении грамматики или при анализе кода программы. Отображение ошибок и неточностей при разработке DSL в обучающих инструментах создает комфортные условия для быстрого и эффективного обучения.

## **2. Описание семантики**

Описать семантику пользователю предлагается автоматным подходом.

## 2.1 Мотивация

Автоматное программирование — парадигма программирования, в рамках которой программные системы представляются в виде конечных автоматов [19]. Суть автоматного программирования состоит в выявлении набора состояний описываемого процесса, условий перехода между этими состояниями и выявлении действий, которые должны выполняться в том или ином состоянии. Иначе говоря, автоматный объект реагирует на входное воздействие не только сменой своего состояния, но и формированием определенных значений или действий на выходе.

Развитием парадигмы автоматного программирования стала модель автоматного объекта [11]. Концепция автоматного объекта расширяет возможности обычного автомата и позволяет ему общаться с внешним миром через интерфейсы взаимодействия. Автоматный объект обеспечивает входное и выходное взаимодействие. При этом в соответствии с принципом Б. Мейера ], операции интерфейса любого объекта разделяются на запросы, доставляющие значения и не меняющие состояния объекта, и команды, меняющие состояние объекта, но не доставляющие значений. Таким образом, на основе этих двух факторов автоматный объект может взаимодействовать с окружающим миром четырьмя различными способами:

1. событие (входное взаимодействие меняющее состояние автомата)
2. сторожевые условия (входное взаимодействие доставляющее значение в автомат)
3. эффекты (выходное взаимодействие меняющее состояние внешнего мира)
4. текущее состояние (выходное взаимодействие доставляющее значение во внешний мир)

Предоставляя такие возможности автоматным объектам их можно связывать их друг с другом, тем самым получая модель взаимодействующих автоматных объектов.

Описанная выше технология легла в основу создания языка описания взаимодействия автоматных объектов CIAO (Cooperative Interaction of

Automata Objects) [13]. Идеи данной работы легли в основу конструирования схожего графического языка, но с некоторыми изменениями:

1. стирается грань между событием и сторожевым условием, данные операции заменяются на предоставляемый интерфейс. В графической нотации предоставляемый интерфейс выглядит как «шарик» или «леденец»
2. стирается грань между эффектом и текущим состоянием, данные операции заменяются на требуемый интерфейс. В графической нотации требуемый интерфейс выглядит как «сокет»
3. появляется раздел внутренних операций, которые используются только внутри автомата и не имеют завязки на внешний мир

## **2.2 Визуализация описанных автоматов**

Пример автоматного объекта в виде изображения на языке CIAOv2 можно увидеть на рисунке 6. Каждый автоматный объект содержит:

1. Название. Обязательный параметр. Является идентификатором автоматного объекта
2. Раздел переменных. Необязательный раздел. Содержит переменные, используемые и изменяемые в течение жизни автоматного объекта
3. Раздел требуемого интерфейса. Необязательный раздел. В данном примере `isItLightOutside()` сторожевое условие, а `turnOnLamp()` и `turnOffLamp()` — эффекты.
4. Раздел предоставляемого интерфейса. Необязательный раздел. В данном примере `isButtonPressed()` — текущее состояние, а `buttonPress(mode)` — событие. Данные методы не представлены на диаграмме переходов, так как предназначены для вызова из внешнего контекста
5. Раздел внутреннего интерфейса. Необязательный раздел. В данном примере `printInfo(text)` внутренняя операция, которая может вызываться несколько раз в течение жизни автоматного объекта
6. Начальное состояние. Обязательный параметр. Всегда единственно

7. Конечное состояние. Необязательный параметр. Необязательно единственно
8. Переходы, ведущие из состояний в состояния. Обязательно имеют начальное и конечное состояние Состояния. Необязательные параметры. Но обязательно присутствие хотя бы одного состояния — начального
9. Условие перехода. Необязательный параметр. Условие должно возвращать истину или ложь. Также возможен переход по условию «else», если переход по всем другим условиям не сработал.
10. Действие при переходе. Необязательный параметр. Операция, которая выполняется после проверки условия в момент перехода в следующее состояние. Может возвращать значение, однако оно будет игнорироваться.

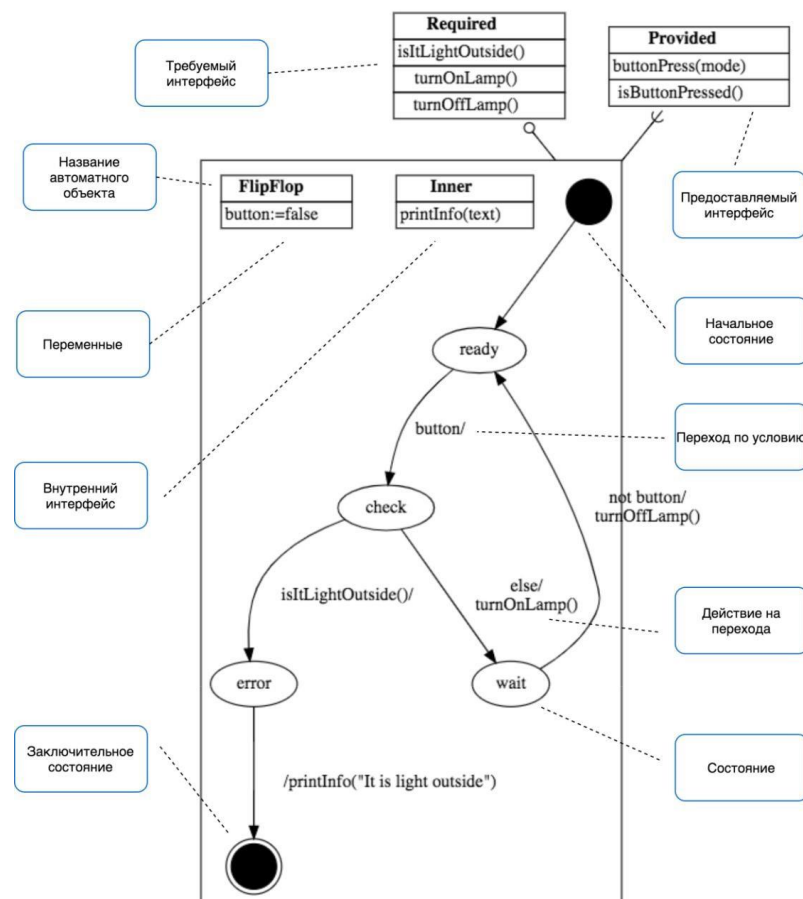


Рисунок 6. Нотация диаграммы переходов на примере счетного триггера



## **2.3 Генерация кода по CIAOv2**

CIAOv2 — текстовый язык описания автоматных объектов, разработанный в рамках работы [8]. На рисунке 7 приводится описание синтаксиса языка в виде синтаксических диаграмм.



Существует следующий ряд контекстных условий программы на языке CIAOv2, которые запускаются после прохода синтаксического анализатора:

1. наличие входного состояния
2. единственность входного состояния
3. отсутствие одинаковых имен переменных и интерфейсов
4. количество переменных в описании интерфейса совпадает с количеством переменных в вызове
5. начальное состояние не может быть концом перехода
6. заключительное состояние не может быть началом перехода
7. на множестве переходов из состояния А в состояние В есть не более одного условия «else»
8. «else» зарезервированное слово (переменные с таким именем запрещены)

После того, как исходный текст программы прошел все вышеописанные проверки, он может быть транслирован:

1. в диаграмму графа переходов состояний на языке DOT
2. в заготовку кода программы на языке Python3

Пример сформированного векторного изображения по коду в DOT представлено на рисунке 8.

Ниже представлен сгенерированный шаблон кода на Python3, построенный для реализации счетного триггера.

```
1 FlipFlop
2 VAR
3 button := false
4
5 REQUIRED
6 isItLightOutside()
7 turnOnLamp()
8 turnOffLamp()
9
10 PROVIDED
11 buttonPress(mode)
12 isButtonPressed()
13
14 INNER
15 printInfo(text)
16
17 STATE
18 entry -> / -> ready
19
```

Рисунок 8. Программа на CIAOv2 для реализации счетного триггера

```

1 class FlipFlop:
2     # VARS
3     def __init__(self):
4         self.button = False
5         self.__state__ = "entry"
6
7     # REQUIRED
8     # uses
9     # isItLightOutside()
10    # turnOnLamp()
11    # turnOffLamp()
12
13    # INNER
14    def __printInfo(self, text):
15        pass
16
17    # PROVIDED
18    def buttonPress(self, mode):
19        pass
20
21    def isButtonPressed(self):
22        pass
23
24    def run(self):
25
26        while (True):
27            if self.__state__ == "exit":
28                break
29            if self.__state__ == "entry":
30                if True:
31                    self.__state__ = "ready"
32                    continue
33            if self.__state__ == "ready":
34                if self.button:
35                    self.__state__ = "check"
36                    continue
37            if self.__state__ == "check":

```

Рисунок 9. Шаблон кода на Python3 для реализации счетного триггера

Фрагмент кода представляет собой класс, где есть:

1. Конструктор, который инициализирует переменные и присваивает служебной переменной `__state__` начальное состояние

2. Заготовки для описания частных методов (раздел INNER) и публичных методов (раздел PROVIDED). Раздел REQUIRED в коде представлен в виде комментария (#uses), так как это требуемый интерфейс от внешней системы
3. Основной ход программы в виде бесконечного цикла, условий выбора следующего состояния и действий при переходе.

В общем случае процесс работы программного решения для описания семантики на языке CIAOv2 и трансляции кода с языка CIAOv2 в код программы на Python3 или диаграмму переходов представлен на рисунке 10.

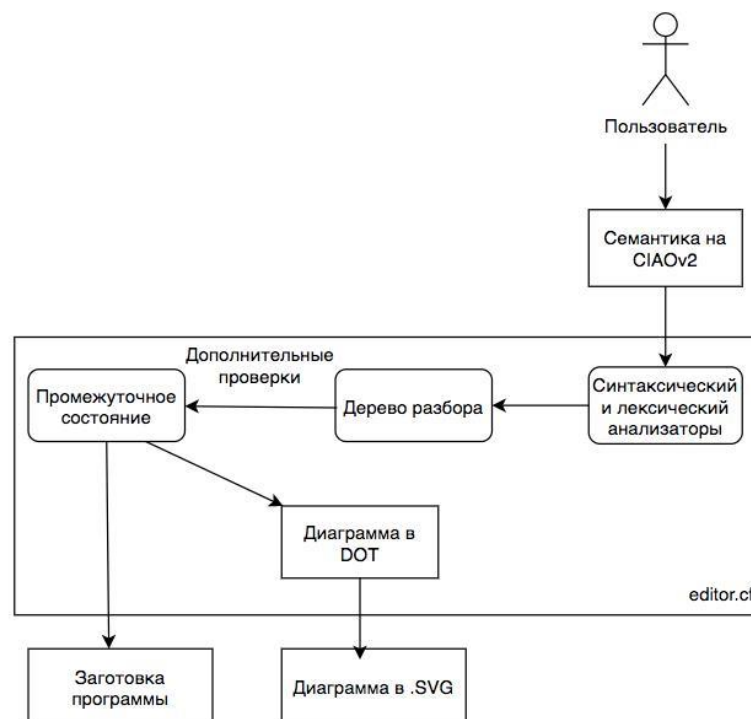


Рисунок 10. Процесс генерации диаграммы и заготовки программы

### 3. Архитектура и дизайн решения

В рамках решения поставленной задачи был разработан программный модуль, который представляет собой веб-приложение со следующими возможностями для пользователя:

1. описания синтаксиса языка на ANTLR4 описания семантики языка на CIAOv2
2. скачивание SVG-файла с деревом разбора программы на основе введенного синтаксиса
3. скачивание SVG-файла с семантикой языка в виде диаграммы переходов

4. скачивание .py-файла с заготовкой программы на языке Python3 для описания семантики

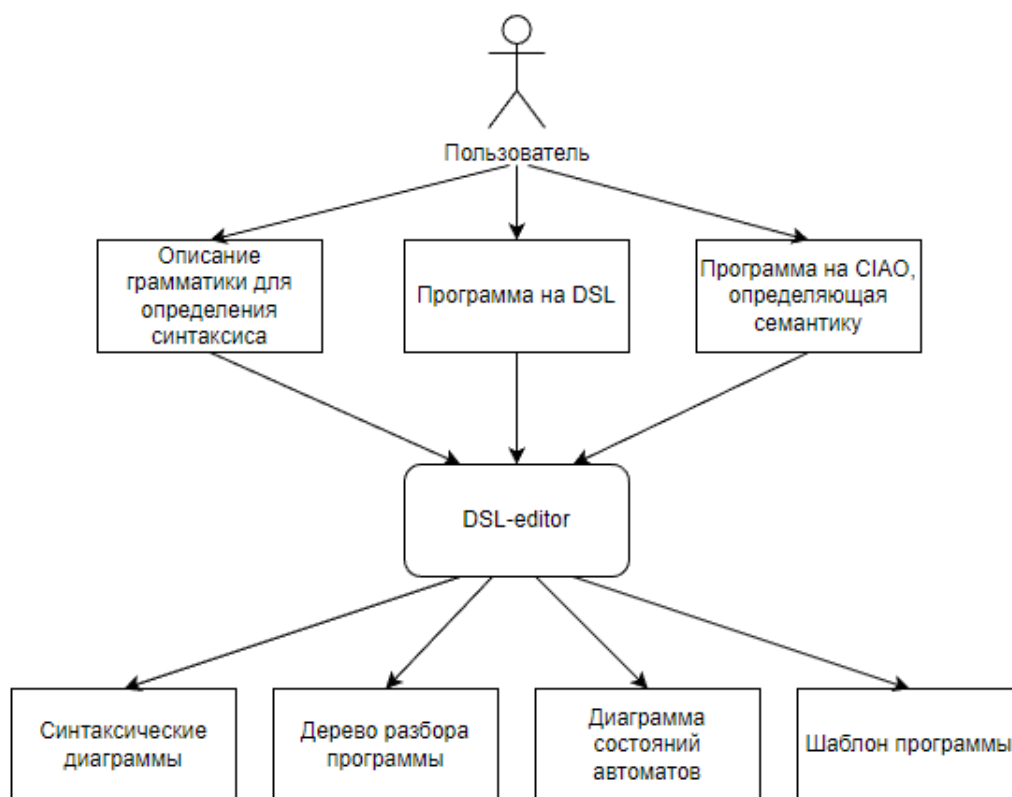


Рисунок 11. Диаграмма потоков данных

Диаграмма потоков данных изображена на рисунке 11. На момент написания работы инструмент доступен по данному адресу: <http://dsleditorapp.hopto.org/>

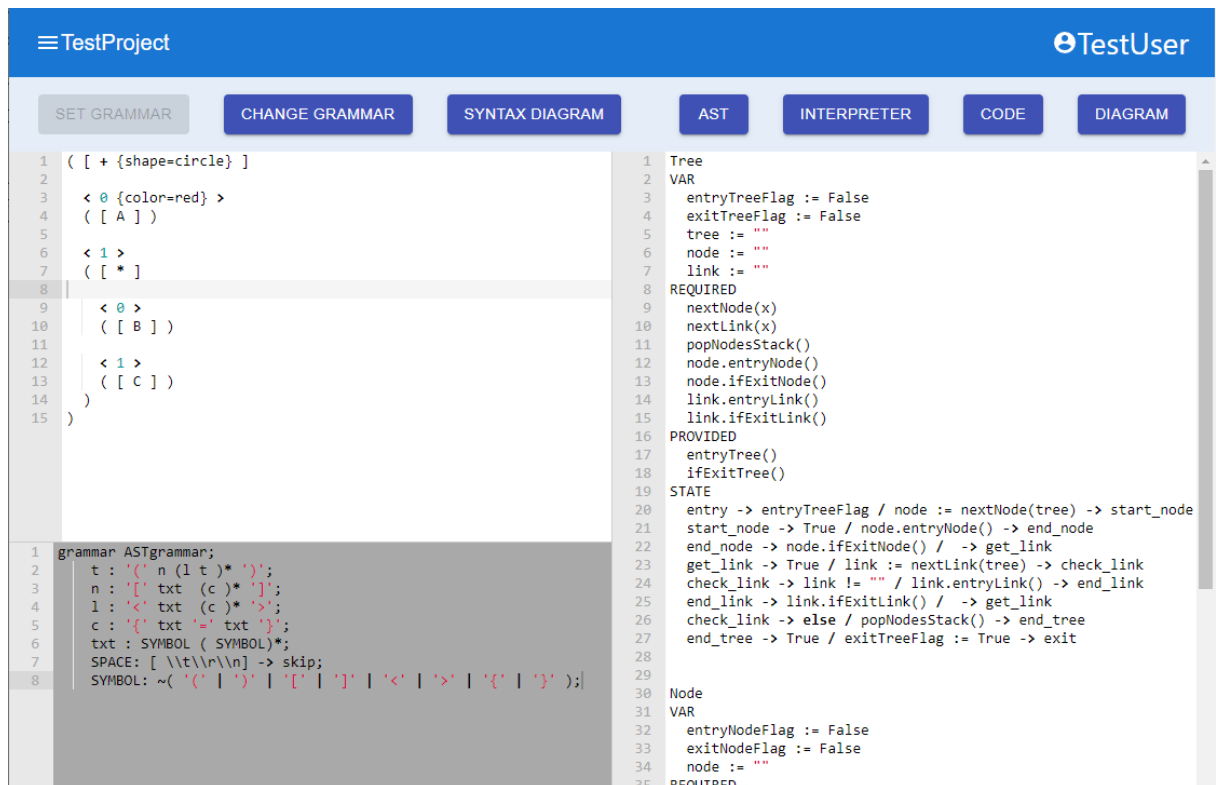


Рисунок 12. Интерфейс основного компонента приложения

Интерфейс веб-приложение DSL-editor представлен на рисунке 12. В левом верхнем окне пользователь указывает исходный текст программы, в левом нижнем окне — синтаксис программы, в правом верхнем — семантику. Также у пользователя есть три кнопки внизу экрана, с помощью которых он может скачать дерево разбора программы, диаграмму состояний и заготовку программы.

Фронтальная часть приложения написана с помощью технологии ReactJS [25]. ReactJS — разработанный компаниями Facebook и Instagram фреймворк с открытым исходным кодом для создания пользовательских интерфейсов. ReactJS позволяет разработчикам создавать крупные веб-приложения, при этом отрисовка элементов будет происходить только в том случае, если они были затронуты изменением данных.

Основная концепция создания React-приложений — разбивка на компоненты. Компоненты позволяют разделить интерфейс на независимые, повторно используемые части, и работать с каждой из них отдельно. Компоненты, которые были созданы во время работы над тем или иным



проектом, не имеют дополнительных зависимостей. Таким образом, компонентно-ориентированный подход дает возможность переиспользовать код и превращают React-разработку в непрерывный процесс улучшения.

В качестве компонента для отображения и редактирования кода используется AceEditor [26]. AceEditor — онлайн-редактор исходного кода с подсветкой синтаксиса и темами, написанный на Javascript.

Основное преимущество этого редактора, что он может быть встроен в любую веб-страницу и приложение JavaScript.

AceEditor поддерживает изменение подсветки синтаксиса для некоторых предопределённых режимов. В рамках данной работы для языка CIAOv2 был написан собственный режим подсветки синтаксиса и выделения таких ключевых слов как VAR, REQUIRED, PROVIDED, INNER и STATE.

Серверная часть приложения была создана с помощью веб-фреймворка Flask[27]. Flask представляет собой легковесный инструмент для создания веб-приложений. Например, в нем отсутствуют встроенный интерфейс администратора, собственный ORM (Object-Relational Mapping) – технология связывания таблиц реляционной базы данных с объектами языка программирования, система кеширования и другие возможности, которые предоставляют высокоуровневые веб-фреймворки. Таким образом Flask является минималистичным каркасом, дающим самые базовые возможности разработчику. Данный факт и послужил причиной выбора его в качестве веб-сервера, так как в данном проекте от серверной части требуется поддержка минимальной функциональности.

В результате разработки серверной части было создан API (рисунок – программный интерфейс приложения – со следующими возможностями:

1. запрос на создание дерева разбора /ast
2. запрос на создание диаграммы состояний /diagram запрос на создание заготовки кода /code

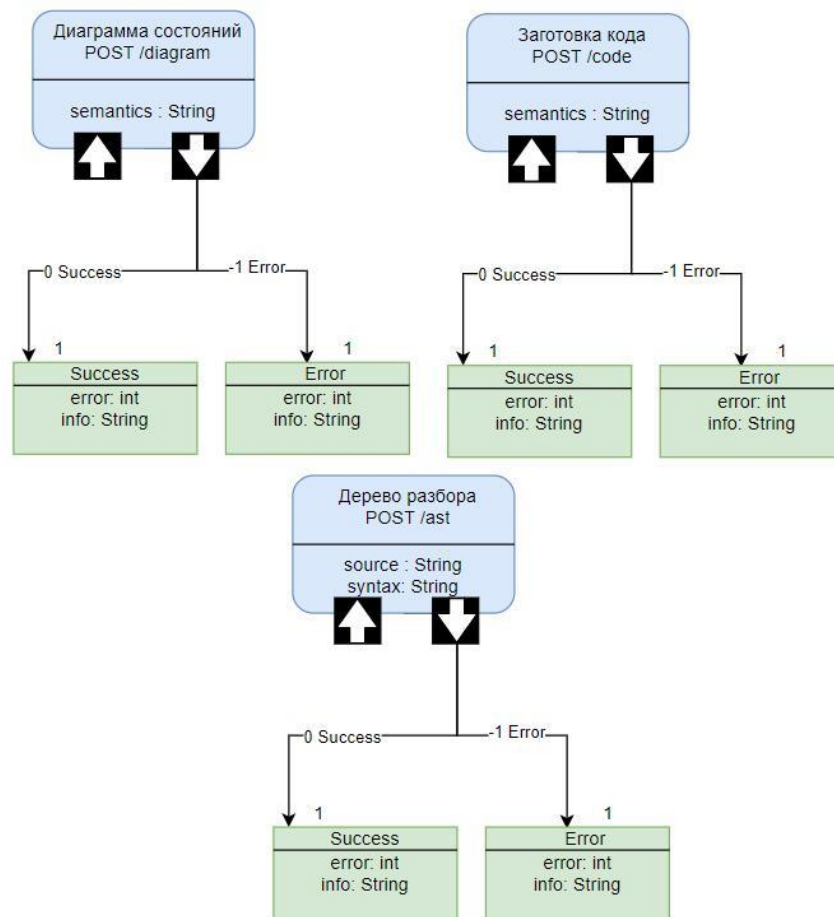


Рисунок 13. Визуализация методов API

Фронтальная часть программного комплекса в зависимости от действий пользователя отправляет необходимые запросы серверу. На основе полученных данных (поля «error» в ответе сервера) фронтальная часть отображает пользователю или сообщение об ошибке или перенаправляет его на страницу для просмотра и скачивания запрашиваемой информации.

## Глава 3. Применение инструмента

### 1. Тестирование: генерация деревьев на языке DOT

Для тестирования был взят DSL, разработанный в рамках работы [8]. Задача этого DSL: по описанию дерева на языке порождаемом заданной грамматикой, сгенерировать код на языке представления графов DOT.

```

1 grammar ASTgrammar;
2   t : '(' n (l t)* ')';
3   n : '[' txt (c)* ']';
4   l : '<' txt (c)* '>';
5   c : '{' txt '=' txt '}';
6   txt : SYMBOL (SYMBOL)*;
7   SPACE: [ \t\r\n ] -> skip;
8   SYMBOL: ~( '(' | ')' | '[' | ']' | '<' | '>' | '{' | '}' );

```

Рисунок 14. Грамматика языка описания деревьев

В такой грамматике дерево представляет собой структуру, заключенную в круглые скобки (), состоящую из узла в квадратных скобках [], и возможно поддеревьев, соединённых с корнем связями в угловых скобках <>. У каждого узла или связи должно быть непустое название. За названием может идти один или несколько комментариев, заключенных в фигурные скобки «{}».

Комментарии используются для того, чтобы указывать атрибуты отрисовки такие, как цвет связи или форма узла дерева. Атрибуты имеют синтаксис name=value, где name и value произвольные строки, не содержащие =. Любые комментарии, удовлетворяющие данному синтаксису, будут интерпретированы как атрибуты.

В качестве имён узлов и связей могут использоваться любые наборы символов, за исключением “()[]<>{}”.

После того, как грамматика создана, пользователь может попробовать “запустить” ее на примерах реальных программ. Для этого в инструменте он может ввести синтаксис и пример входного языка и получить дерево разбора. Если программа не будет удовлетворять грамматике, то пользователь получит сообщение об ошибке.

Например, для программы вида:

```

1 (
2   [{shape}]
3   <0{color=blue}> ([A])
4   <1{color=red}> ([A])
5 )

```

Рисунок 15. Пример программы на языке описания деревьев с ошибкой

Пользователь увидит на экране:

Exception: line 2:10:mismatched input '}' expecting '='

Рисунок 16. Пример вывода при синтаксической ошибке

Сообщение об ошибке будет указывать пользователю на номер строки и столбца, где возникла ошибка. В данном случае проблема возникла из-за того, что запись “{shape}” не удовлетворяет правилу грамматики “c : '{' txt '=' txt '}'”.

Пример корректной программы:

```
1 (
2   [{shape=box}]
3   <0{color=blue}>
4   ([A])
5 )
```

Рисунок 17. Пример корректной программы на языке описания деревьев

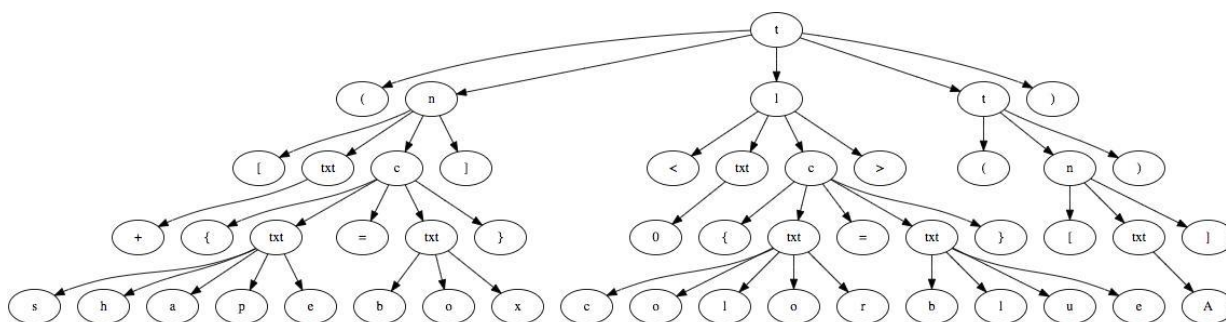


Рисунок 18. Пример построенного дерева разбора

Далее переходим к описанию семантики в автоматном стиле. В данном примере семантика должна реализовать обход по дереву разбора. Обход по дереву в автоматном стиле можно задать разными способами.

Диаграмма переходов для выбранного описания семантики представлена на рисунке 19.

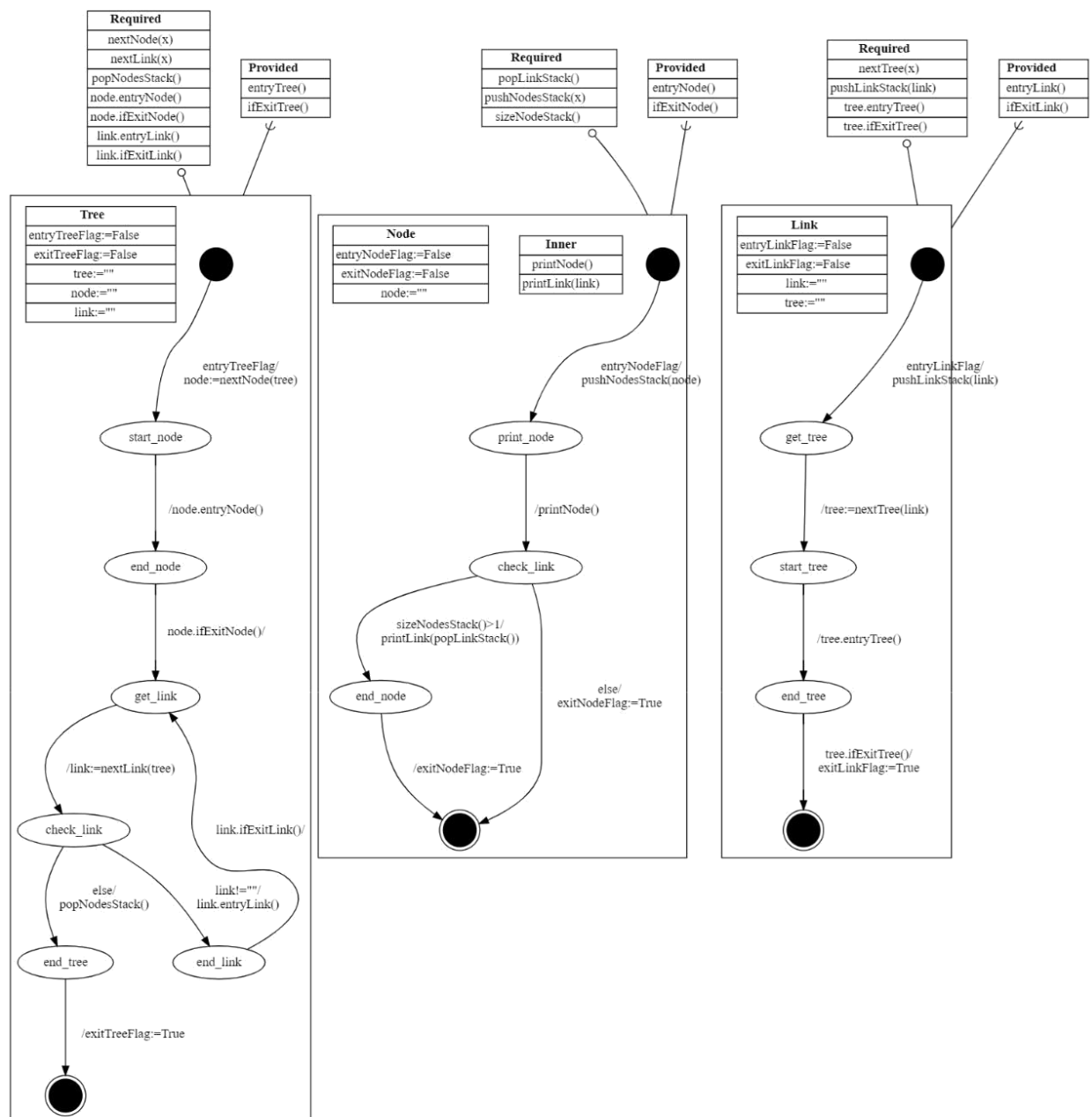


Рисунок 19. Диаграмма конечных автоматов для языка генерация деревьев

Рассмотрим автоматный объект Tree. Из диаграммы видно, что данный объект предоставляет два метода интерфейса: entryTree() и ifExitTree(). При этом метод entryTree() будет использован другими объектами для изменения локальной переменной entryTreeFlag, ifExitTree() будет использован внешними объектами для запроса значения exitTreeFlag.

Заметим, что переход из начального состояния в состояние start\_node происходит только в том случае, если переменная entryTreeFlag выставлена в True. Так как изначально данная переменная выставлена в False (в разделе

VAR), то переход в состояние `start_node` будет выполнен только в том случае, когда внешняя среда изменит значение переменной `entryTreeFlag` с `False` на `True`. Иначе говоря, когда дереву будет дана команда начать обход.

Далее при переходе из состояния `start_node` в `end_node` автоматный объект дерева взаимодействует с узлом через предоставляемый интерфейс, а именно через метод `entryNode()`. Тем самым дерево передает команду узлу, что можно начинать обход. Таким образом дерево производит «эффект», а узел принимает «событие» в терминах модели взаимодействующих автоматных объектов. Далее дерево переходит в ожидание, то есть сможет перейти в следующее состояние только в том случае, когда «сторожевое условие» `ifExitNode()` будет верным, то есть, когда узел закончит свой обход. Обход каждой дуги при этом будет аналогичен обходу узла: передаем дуге команду, что можно начинать, а затем ждем, что дуга закончит выполнение. При этом важно заметить, что исходное дерево запускает обход только в связанной с ним дуге и ничего “не знает” про то, что под каждой дугой есть дочернее поддерево.

Перед переходом в заключительное состояние автоматный объект выставляет значение `exitTreeFlag` в `True`. Таким образом автоматный объект сообщает о том, что он закончил обход.

## **2. Применение на практике - язык описания онтологий**

Инструмент DSL-editor использовался в рамках курса «Грамматики и автоматы», где под руководством Фёдора Александровича Новикова в Санкт-Петербургском политехническом университете Петра Великого (СПбПУ) студентам предлагается получить расширенные знания в теории конечных автоматов и грамматик, а также применить полученные навыки в создании своего языка программирования в предметно-ориентированном подходе.

Студентам было предложено создать язык описания онтологий с помощью DSL-editor.

## 2.1 Онтологии дискретной математики

Онтология – это агрегация множества сущностей и множества отношений между ними (рисунок 20). Сущности и отношения определяются в пакетах композиционно и импортируются в онтологии. Пакет – это композиция множества сущностей и множества отношений между ними. Отношения, сущности и пакеты определяются уникальными для своего типа именами (рисунок 21). Пакеты могут пересекаться и дополнять друг друга – это приравнивается к необходимости произвести их объединение в один с проверкой корректности.

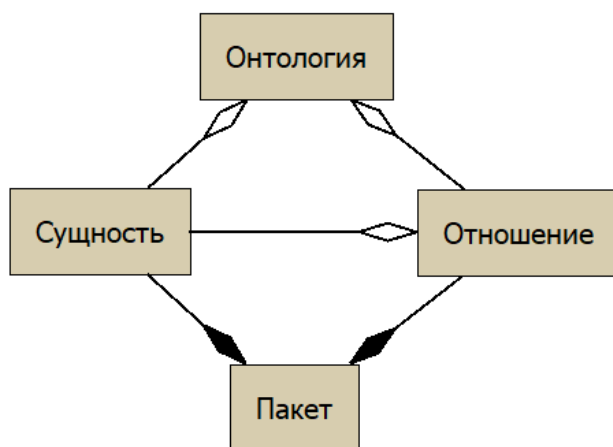


Рисунок 20. Мета модель онтологии. Общая структура.

Отношения, сущности, пакеты и онтологии являются контейнерными элементами и образуют пространство имен. Все имена элементов одного типа в пространстве имен должны быть уникальными.

Сущности – это множество объектов, самым востребованным из которых является класс (рисунок 21). Также к сущностям относятся перечисление и тип с сохранением UML семантики. Отдельно стоит сказать, что в случае онтологий тип – это вспомогательное понятие, которое применяется в тех позициях, где необходимо указать множество объектов не как основной элемент онтологии, а в другой роли. Три типа считаются переопределенными и встроенными: «Bool», «String», «Number».

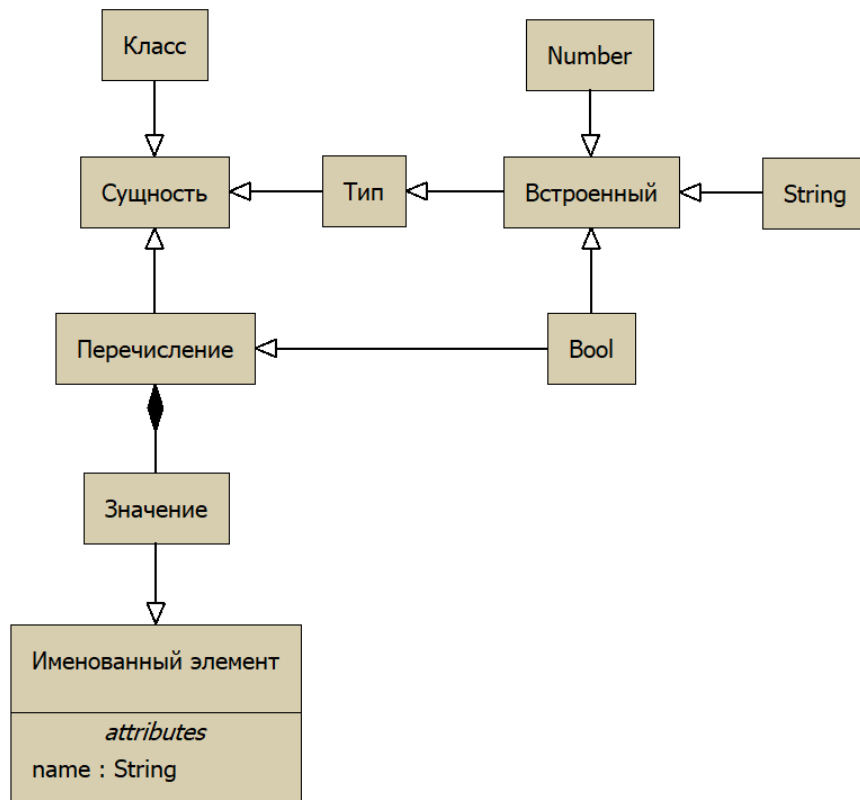


Рисунок 21. Мета модель онтологии. Сущности.

Классы и некоторые отношения могут иметь атрибуты, классы могут иметь операции (рис. 22). Атрибуты и операции являются частными случаями сущности и отношения соответственно и входят в свои контейнеры композиционно. Атрибуты могут иметь тип, у операций могут быть указаны имена и типы аргументов и результатов.

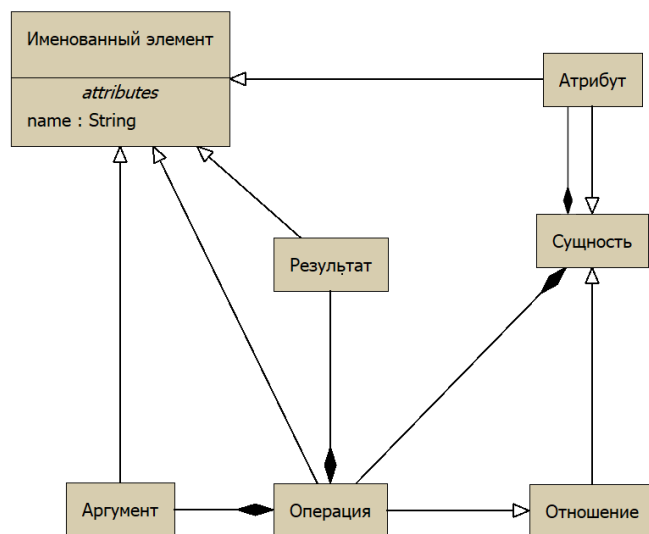


Рисунок 22. Мета модель онтологии. Атрибуты и операции.



Отношение (рис. 23) – это множество кортежей объектов. Все отношения считаются бинарными, если не сказано иного. Обобщение используется, чтобы показать иерархию, бывает N:1, является антисимметрическим, ациклическим, может иметь атрибуты. Композиция и агрегация используются, чтобы показать отношение часть-целое с той разницей, что при композиции целое бессмысленно без частного. Агрегация может быть N:N, композиция N:1, оба отношения антисимметричны, имеют кратность, композиция также является ациклическим отношением. Отношение зависимости может использоваться для представления определений математических терминов, является антисимметрически, ациклическим. Реализация используется, чтобы показать, что объект является воплощением другого абстрактного понятия. Отношение является антисимметрическим, ациклическим, может иметь атрибуты. Оно наиболее редкое, но и ему иногда находится применение. Ассоциация – это наиболее общее понятие, с помощью которого можно описать любое симметричное отношение между парой сущностей, имеет кратность, может иметь атрибуты. Также для спецификации алгоритмов используется отношение действия. Оно может быть N:N.

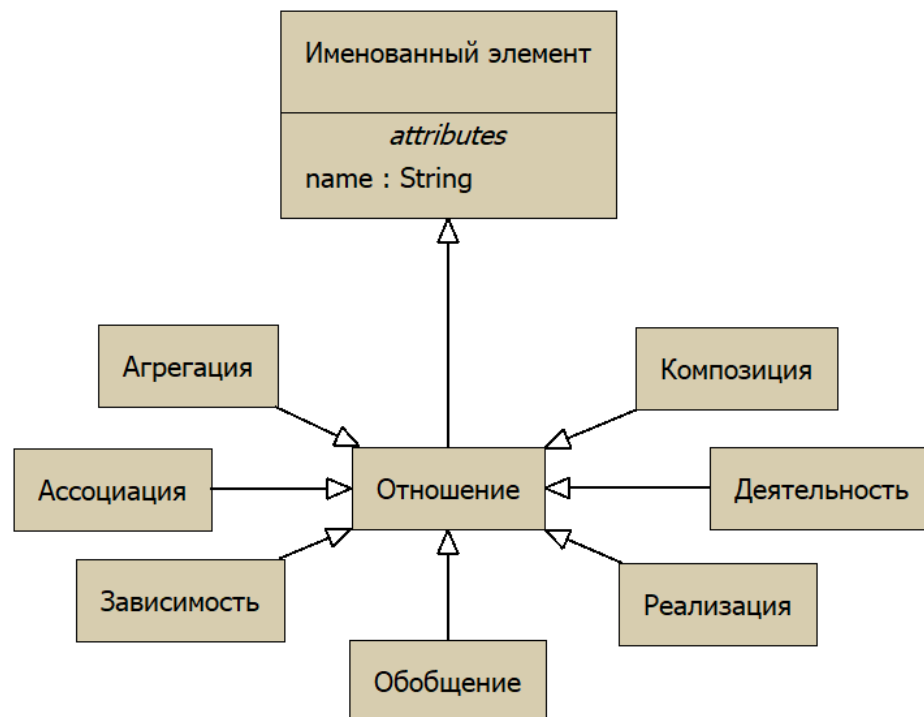


Рисунок 23. Мета модель онтологии. Отношения.

Кроме того, все сущности и отношения могут иметь один или несколько связанных с ними комментариев.

## 2.2 Постановка задачи для реализации в DSL-editor

Для описания языка онтологий используется формальная порождающая грамматика [6] (далее просто грамматика), представленная таблицей 1. В ней с помощью черного и синего цвета обозначены нетерминальные и терминальные символы соответственно, «*Ont*» – основной символ, а каждая строчка представляет одно правило вывода, помеченное номером. Красным цветом обозначены метасимволы со следующей семантикой:

1. Фигурные скобки **{ }** – повторение конструкции (1 и более раз).
2. Квадратные скобки **[ ]** – необязательная конструкция (0 или 1 раз).
3. Вертикальная черта **/** – разделение альтернатив («или»).

1	<i>Ont</i>	<b>{P}</b>
2	<i>P</i>	<i>package</i> name { <b>[Types][Class][Enum][Rel][Comm]</b> }
3	<i>Types</i>	<i>types</i> { <b>[{ typeName , } typeName ]</b> }
4	<i>Class</i>	<i>class</i> name { <b>[{Attr}][Oper]</b> }
5	<i>Enum</i>	<i>enum</i> name { <b>[{string , } string ]</b> }
6	<i>Rel</i>	<b>{Act Assoc Gen Aggr Comp Dep Impl}</b>
7	<i>Comm</i>	<i>comm</i> name string ;
8	<i>Act</i>	<i>act</i> name { <b>{Arg}{Res}</b> }
9	<i>Assoc</i>	<i>assoc</i> name { MArg MRes <b>[{Attr}]</b> }
10	<i>Gen</i>	<i>gen</i> name { <b>{Arg}</b> Res <b>[{Attr}]</b> }
11	<i>Aggr</i>	<i>aggr</i> name { <b>{MArg}{MRes}</b> }
12	<i>Comp</i>	<i>comp</i> name { <b>{MArg}</b> MRes }
13	<i>Dep</i>	<i>dep</i> name { Arg Res }
14	<i>Impl</i>	<i>impl</i> name { <b>{Arg}</b> Res <b>[{Attr}]</b> }
15	<i>Attr</i>	<i>attr</i> <b>[typeName]</b> name ;
16	<i>Oper</i>	<i>oper</i> name { <b>[{Arg}][{Res}]</b> }

17	<i>Arg</i>	<i>arg</i> [ <i>typeName</i> ] <i>name</i> ;
18	<i>Res</i>	<i>res</i> [ <i>typeName</i> ] <i>name</i> ;
19	<i>MArg</i>	<i>arg</i> [ <i>typeName</i> ] <i>name</i> [ <i>Mult</i> ];
20	<i>MRes</i>	<i>res</i> [ <i>typeName</i> ] <i>name</i> [ <i>Mult</i> ];
21	<i>Mult</i>	[ 0   1   *   0..1   1..* ]
22	<i>name</i>	< <i>syms</i> >
23	<i>string</i>	" <i>syms</i> "
24	<i>typeName</i>	' <i>syms</i> '

Таблица 1. Грамматика, порождающая язык описания онтологий

Здесь «syms» – любая последовательность символов русского алфавита и символов таблицы ASCII. Таким образом, лексика двуязычная.

Для хранения онтологий студентами был выбран формат JSON. Выбор обоснован тем, что несмотря на фундаментальное различие в построении (JSON является иерархическим, а UML нет), описанная далее структура позволяет в полной мере сохранять описываемые онтологии. Кроме того, данный формат можно использовать в сочетании с подавляющим большинством языков программирования, что существенно увеличивает практическую ценность модели.

Задачей поставленной к реализации в инструменте была разработка языка предметных областей на основе представленной грамматики. На выходе программа должна получать JSON представление онтологии.

При этом конечная программа должна корректно обрабатывать: семантические ошибки, очевидные в рамках одного пакета до операций объединения.

## 2.3 Результаты применения

Полученный язык представлен контекстно-свободной грамматикой в регулярной форме, что позволяет воспользоваться инструментом DSL-editor для получения синтаксического анализатора на основе данной грамматики.

Формат языка предполагается свободным (пробелы, отступы и переносы не учитываются).

В приложении 1 приводится листинг грамматики, разработанной студентом в ходе создания языка. На рисунке 24 представлен фрагмент синтаксической диаграммы, полученной с помощью инструмента на основе разработанной грамматики.

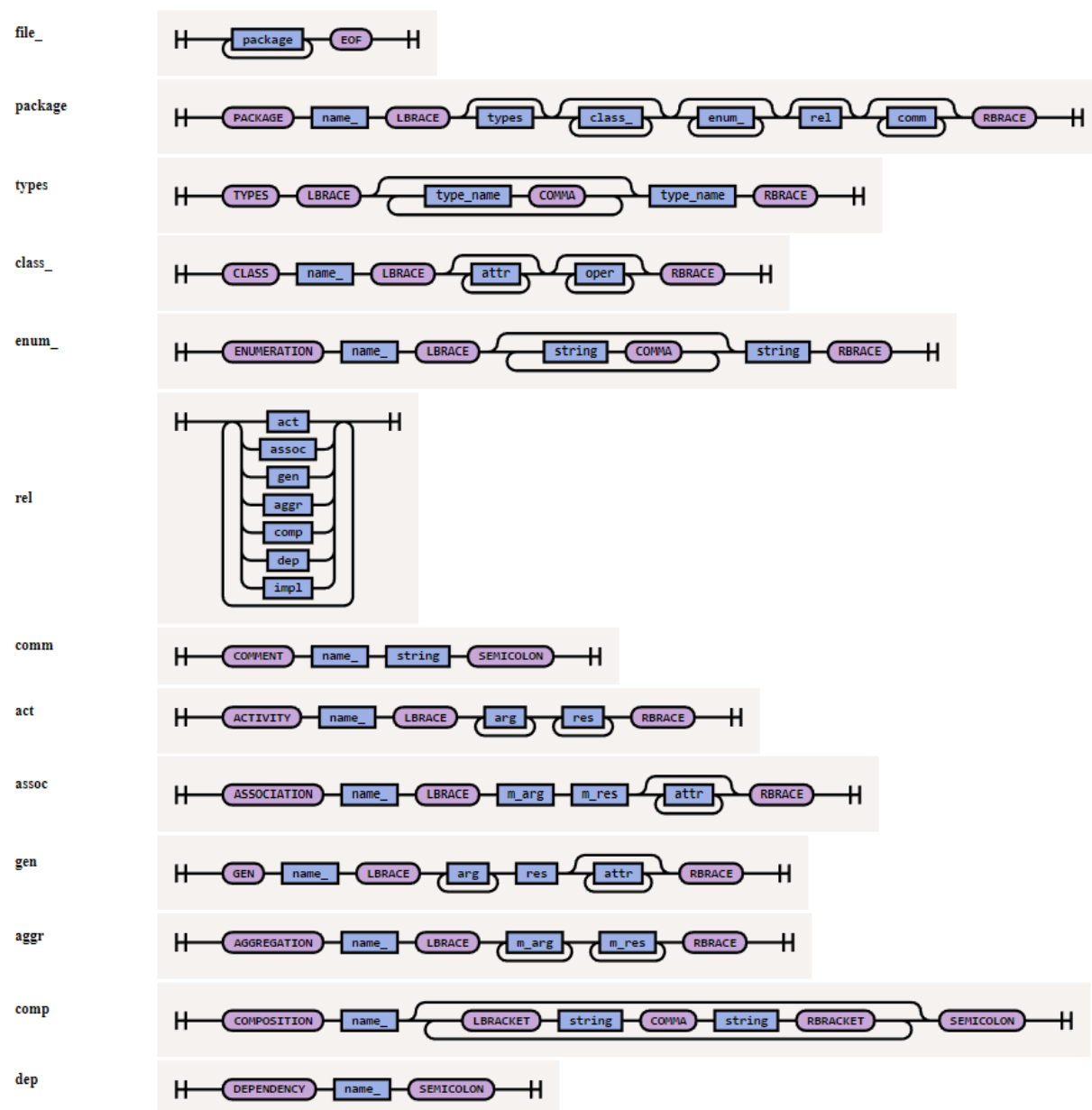


Рисунок 24. Синтаксическая диаграмма для языка описания онтологий

На рисунке 26 представлен пример дерева разбора, полученного обработкой входных данных с помощью синтаксического анализатора,

который был сгенерирован с помощью DSL-editor для программы, описанной на рисунке 25.

```
1 package <Булевы функции> {  
2   types {  
3     'функция'  
4   }  
5   class <Формула> {}  
6   enum <E2> {  
7     "0",  
8     "1"  
9   }  
10 }
```

Рисунок 25. Пример программы на DSL

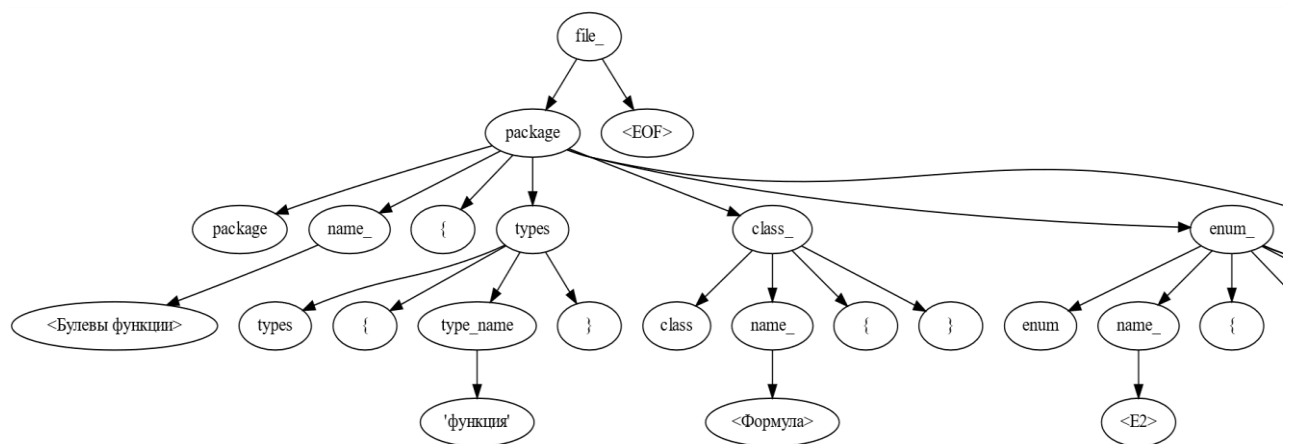


Рисунок 26. Дерево разбора для языка описания онтологий

Следующим шагом необходимо было описать семантику в автоматном стиле. В данном примере семантика должна реализовать обход по дереву разбора. Обход по дереву в автоматном стиле можно задать разными способами.

Создадим для каждой сущности свой автоматный объект. Внутри каждого объекта опишем стратегию обхода конкретно этого элемента. Иначе говоря, семантика для описания класса “не знает” как нужно обрабатывать его атрибуты и операции. Однако, каждый автоматный объект зависит от обхода других элементов и может принимать сигналы о том, что нужно начать свой обход, или давать сигналы другим объектам о том, что им нужно начать обход. Например, объект, описывающий очередной класс в пакете, будет находиться в ожидании, пока не удостоверится, что очередной атрибут не был обработан.

Иначе говоря, автоматные объекты должны взаимодействовать друг с другом, чтобы передавать команды дочерним объектам и ожидать информации от дочерних объектов.

Дополнительно был придуман автоматный объект List, помогающий обработать множественные объекты. Во время обработки класса запускается обработка объекта List, внутри которого обрабатываются все атрибуты класса и т.д.

Фрагмент диаграммы переходов для такого описания семантики представлена на рисунке 27.

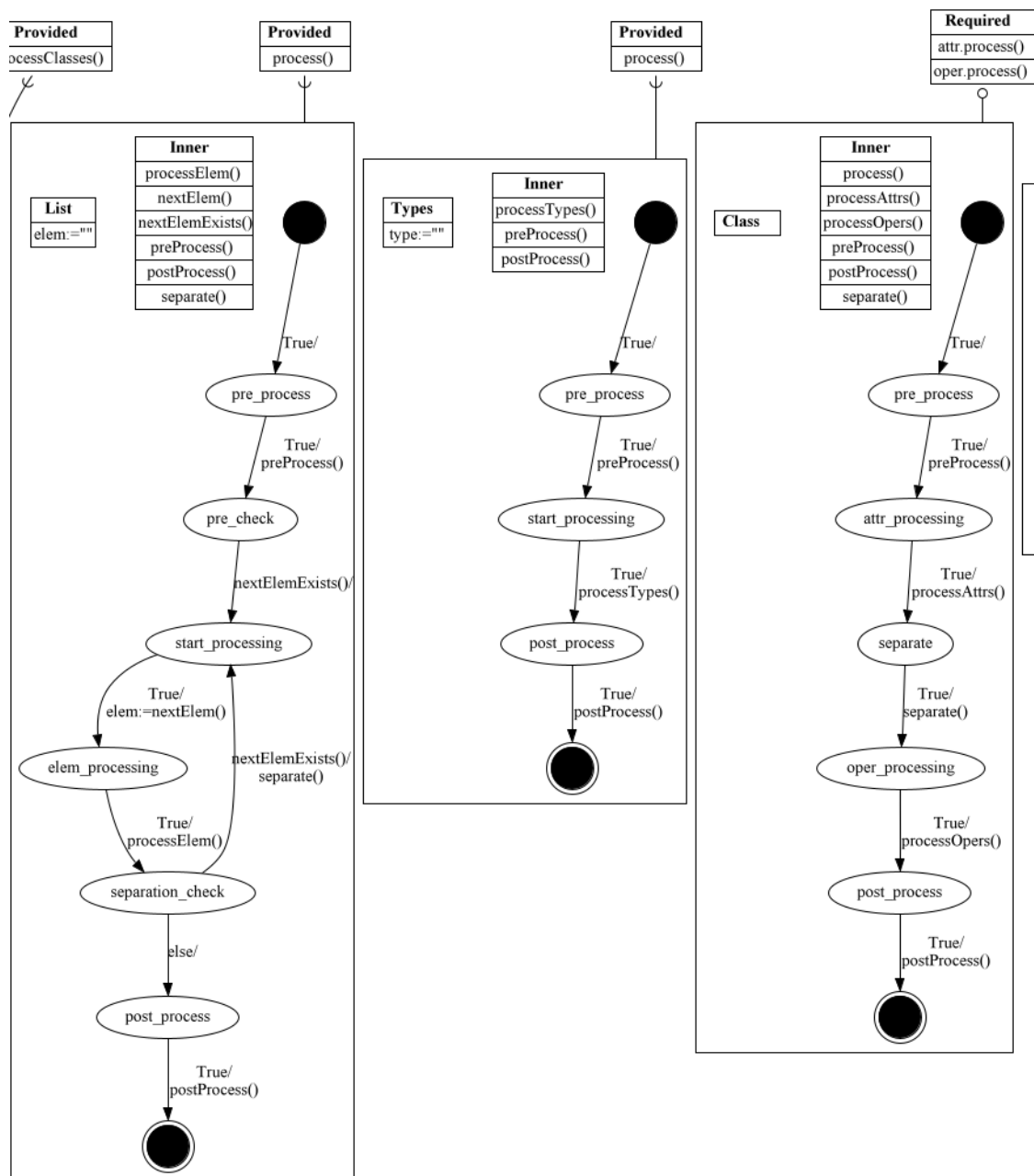


Рисунок 27. Диаграмма конечных автоматов для языка описания онтологий

В приложении 2 приведен листинг Python класса для обработки сущностей классов, сгенерированного инструментом, дописанного студентом.

В приложении 3 приведен пример JSON-объекта, полученного с помощью разработанного DSL и Код программы для которого был получен данный JSON.

### 3. Расширенные возможности

Во время помощи студентам с разработкой языка описания онтологий был выявлен ряд минусов разработанного инструмента. Основные из них:

- Устаревший язык описания грамматик в ANTLR4
- Отсутствие возможности работы с семантикой на этапе работы с синтаксисом
- Отсутствие возможности преобразования выходных данных в разные форматы.
- Сложность многопользовательской работы с инструментом
- Исходные данные DSL не сохраняются при выходе со страницы или перезагрузке страницы

Для повышения качества использования инструмента был предложен и частично реализован ряд улучшений.

#### 3.1 Итерация Цейтина

Вводится итерация с разделителем или обобщённая итерация  $\#$  - итерация Цейтина, которая не расширяет множество регулярных слов и может быть определена через одноместную операцию Клина (\*) как  $P\#Q = (P, Q)^*, P$ . Это частично решает задачу минимизации регулярного выражения по числу вхождений символов из объединённого алфавита символов и делает язык более современным. Для реализации добавлен пред обработчик входного потока символов, который разбивает входной текст программы на слова и меняет  $P\#Q$  на  $(P, Q)^*, P$ .

Благодаря этому улучшению в описании грамматики можно заменить строку.

```
types: TYPES LBRACE (type_name COMMA)* type_name RBRACE;
```

на

```
types: TYPES LBRACE type_name#COMMA RBRACE;
```

### 3.2 Дополнительные обработчики

Для решения проблем отсутствия возможности работы с семантикой на этапе работы с синтаксисом и отсутствия возможности преобразования выходных данных в разные форматы было предложено добавить обработчики данных, встраиваемые в текущий пайплайн:

1. Обработчик входных данных – получает на вход бинарные данные и преобразует их в текст программы, соответствующий заданной грамматике. Такой обработчик позволяет описывать программу не только в виде текста в но и текста в формате WYSIWIG, Markdown, набора голосовых команд, жестов или графики.

2. Обработчик дерева синтаксического разбора – получает на вход дерево разбора, построенное ANTLR, обходит дерево, и возвращает новое дерево или выбрасывает исключение. С его помощью можно дополнять дерево синтаксического разбора новыми данными или, например, обрабатывать семантические ошибки.

В случае языка описания онтологий одно из требований – корректная обработка семантических ошибок в рамках одного пакета до операций объединения.

Одна из таких ошибок – дублирование имён различных сущностей. Такую ошибку можно корректно обработать на этапе обхода дерева разбора двумя способами:

- построить таблицу имён и проверять, если в таблице уже содержится имя – возвращать ошибку
- добавлять к имени сущности тип сущности

Оба способа можно реализовать внутри обработчика дерева синтаксического разбора



3. Обработчик выходных данных – получает на вход данные, сгенерированные в результате выполнения работы программы, и преобразует их в другие форматы (выполнение, изображение, текст, звуковая запись).

При этом предполагается, что эти обработчики пользователь будет реализовывать сам и загружать в проект. Модель работы с данными приведена на рисунке 28.

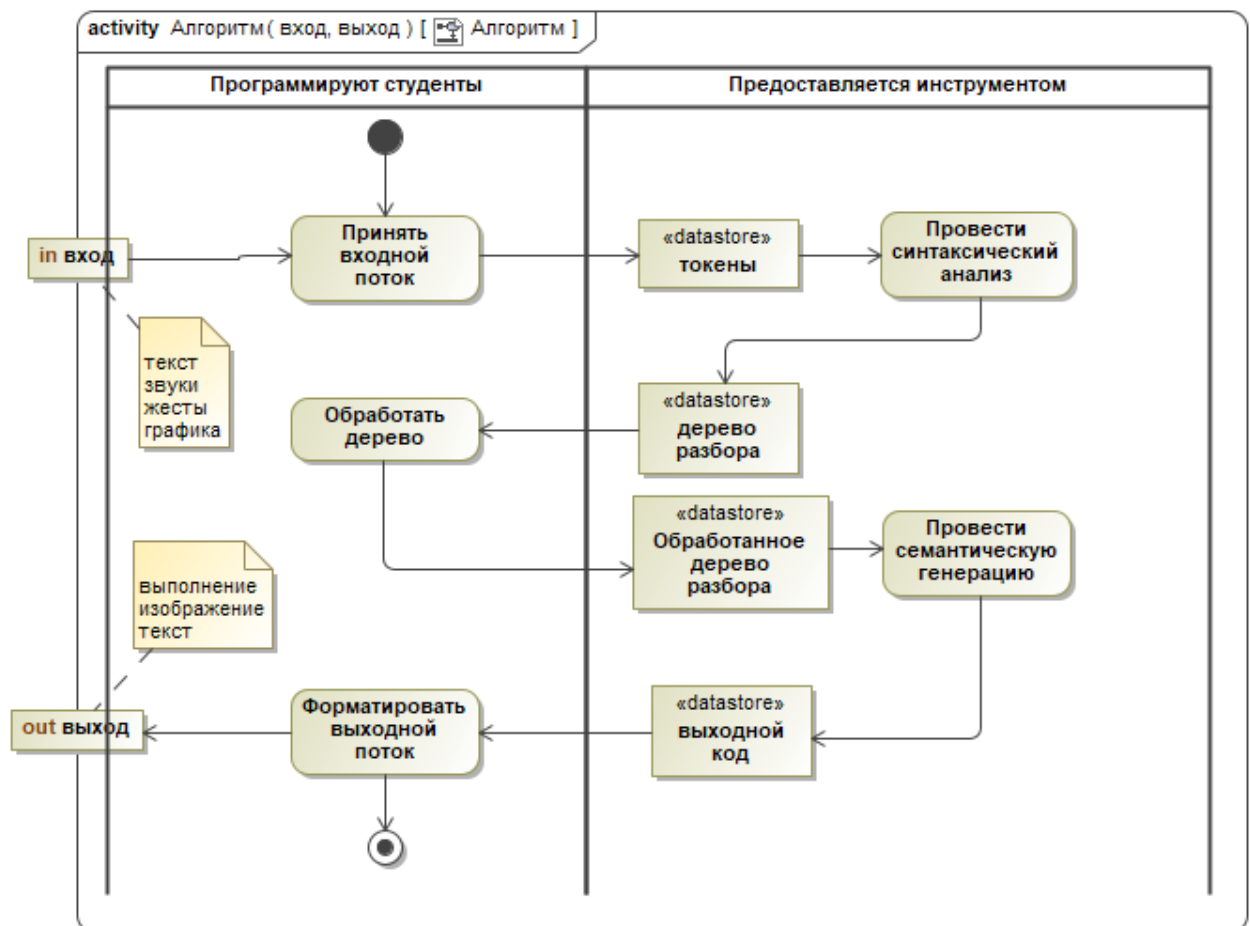


Рисунок 28. Улучшенная модель обработки потока данных для DSL-editor

Для загрузки кода обработчиков в пользовательский интерфейс была добавлена кнопка загрузки файла с кодом обработчиков.

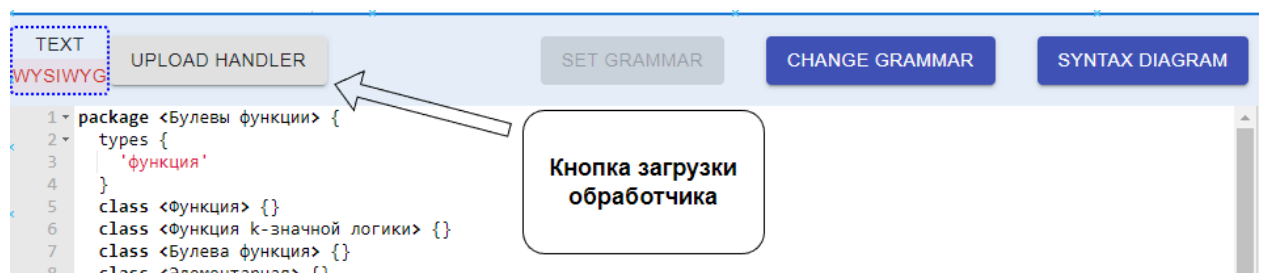


Рисунок 29. Кнопка загрузки обработчиков

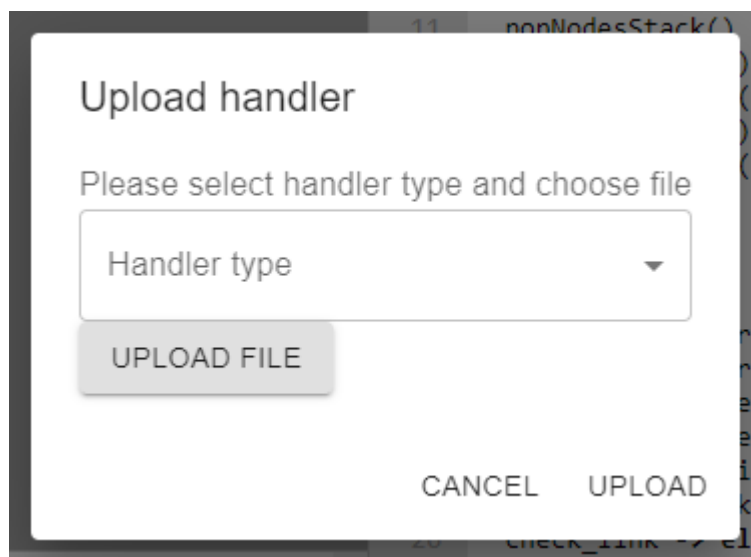


Рисунок 30. Форма загрузки обработчиков

### 3.3 Авторизация

Для решения проблем с многопользовательской работой и отсутствием возможности сохранить исходные данные DSL было принято решение добавить в инструмент возможность авторизации пользователя, а также работу над несколькими проектами.

На рисунках 31 и 32 приведены добавленные страницы для авторизации и выбора/создания проекта.

Исходные коды проектов автоматически сохраняются при любом их изменении и загружаются при перезагрузке.

В виду ограниченных ресурсов текущее решение включает в себя лишь макет авторизации без хранения и проверки паролей.

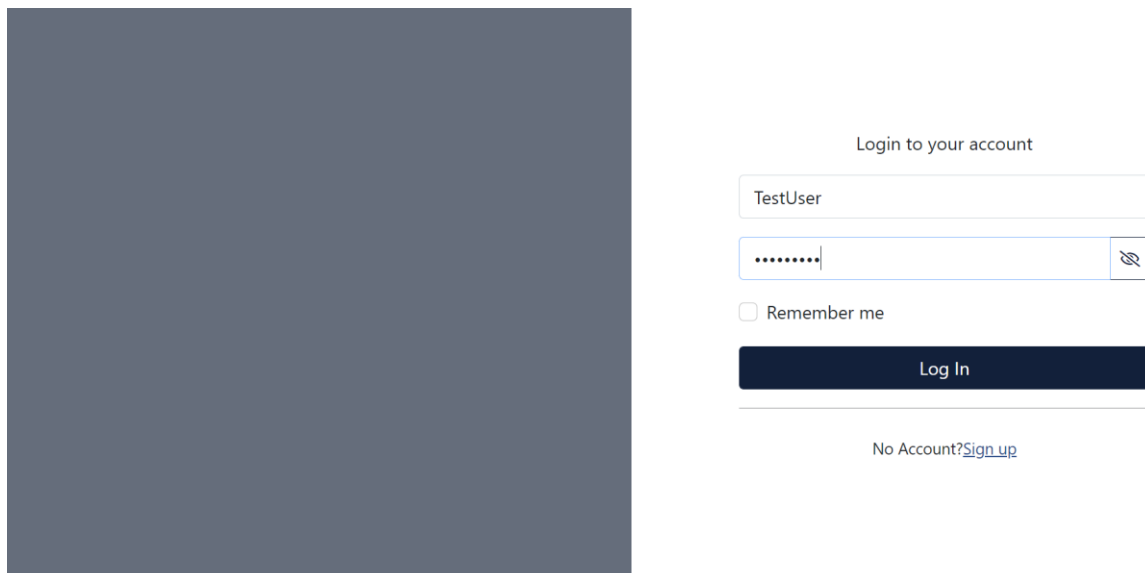


Рисунок 31. Страница авторизации пользователя

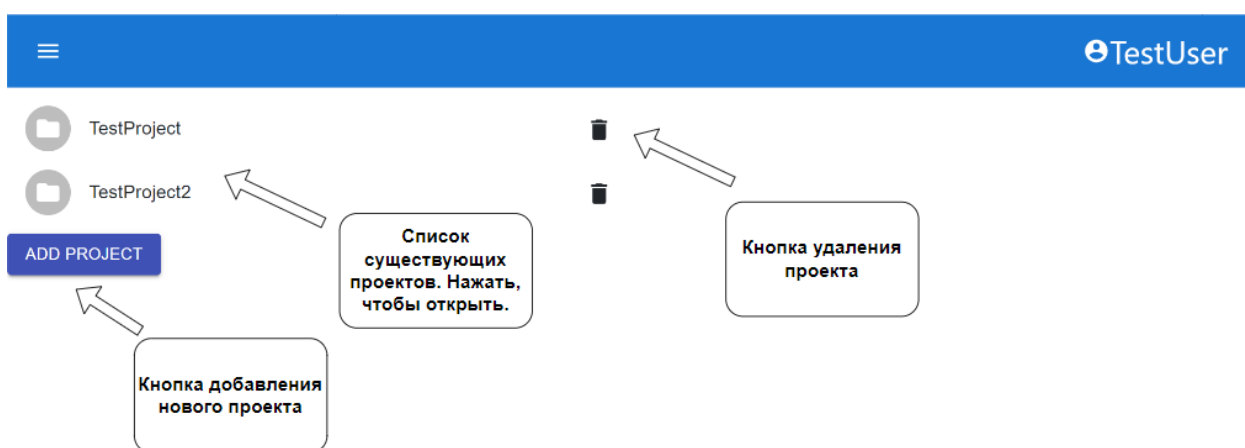


Рисунок 32. Страница со списком проектов пользователя

## 4. Раскрутка ANTLR

Для демонстрации работоспособности и применимости инструмента традиционно применяют метод раскрутки: если на вход инструмента подать модель инструмента, спроектированную в этом инструменте, то на выходе мы получим тот же инструмент [15].

В качестве первого приближения к возможной раскрутке языка можно построить дерево синтаксического разбора для описания грамматики на языке, который принимает на вход ANTLR. Для этого была составлена грамматика, определяющую язык описания грамматик ANTLR (приложение 4) и получено

дерево разбора для грамматики, определяющей синтаксис языка генерации деревьев (рисунок 33).

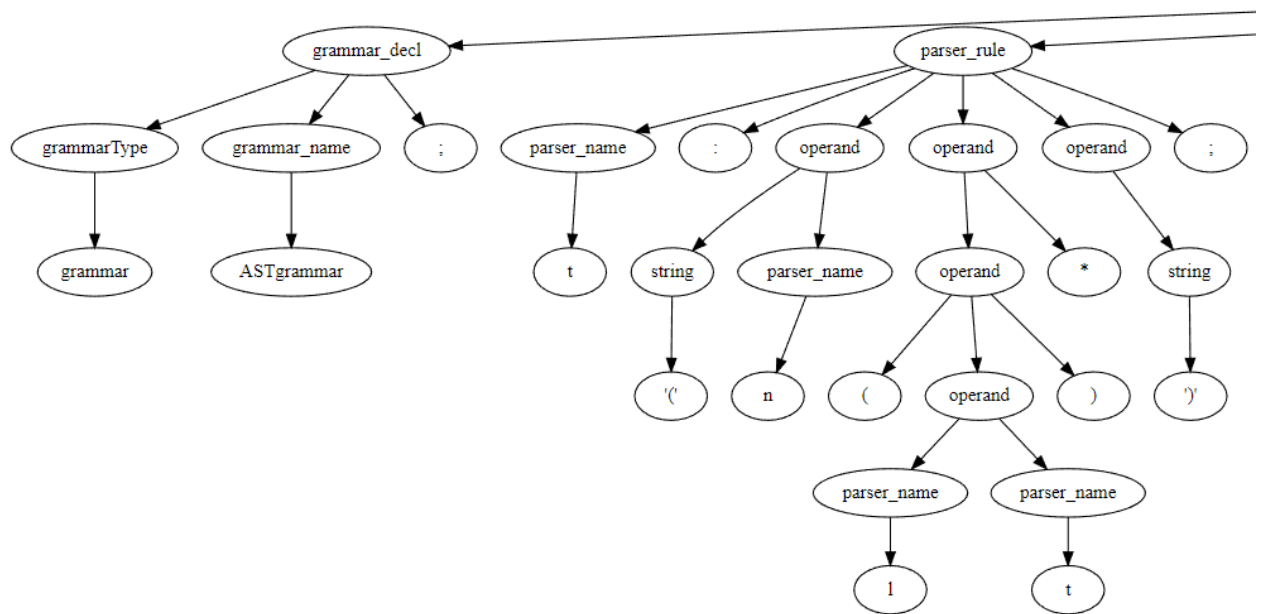


Рисунок 33. Фрагмент дерева разбора для раскрутки ANTLR

## Заключение

В результате работы над поставленными задачами были достигнуты следующие результаты:

1. Предложен метод взаимодействия инструментов для обработки синтаксиса(ANTLR4) и семантики(CIAOv2) для удобного описания языка предметных областей. На его основе был создан программный инструмент предоставляющий следующие возможности:

- Отображать синтаксическую диаграмму грамматики, определяющей язык DSL
- Строить и отображать дерево разбора программы, базируясь на введенном синтаксисе.
- Отображать семантику языка языв виде диаграммы переходов

- Генерировать шаблон исходного кода на языке Python, описывающий семантику языка
2. Изучена предметная область – инструменты и методы построения языков предметных областей
  3. Проведено тестирования инструмента
  4. Проведена поддержка инструмента в ходе курса «Грамматики и автоматы» Санкт-Петербургского политехнического университета Петра Великого, в ходе которой были выявлены и доработаны слабые стороны приложения.

### **Список использованных источников**

1. Фаулер М. Предметно-ориентированные языки программирования. // М.: Вильямс, 2011.
2. Ward M. P., Language Oriented Programming // October 1994
3. Fauler M., Language Workbenches: The Killer-App for Domain Specific Languages? [Электронный ресурс] – URL: <https://www.martinfowler.com/articles/languageWorkbench.html> (дата обращения 14.05.2022)
4. Lämmel R., Software Languages: Syntax, Semantics, and Metaprogramming // 2006
5. Ахо А. В., Сети Р., Лам М. С, Компиляторы: принципы, технологии и инструменты // 1986
6. Feynman R., EBNF: A Notation to Describe Syntax // 2016
7. Карпов Ю. Г. Теория автоматов. // СПб.: Питер, 2002.
8. Орищенко А.О. Методы и инструменты эффективного конструирования языков предметных областей // Санкт-Петербург, 2020
9. Лавров С.С., Программирование. Математические основы, средства, теория // БВХ-Петербург, 2001
10. Федорченко Л.Н. Регуляризация контекстно-свободных грамматик на основе эквивалентных преобразований

- синтаксических граф-схем [Электронный ресурс] – URL:  
[http://conference.spiiras.nw.ru/seminar\\_ICT/20080523Fedorchenko.pdf](http://conference.spiiras.nw.ru/seminar_ICT/20080523Fedorchenko.pdf)  
(дата обращения 29.04.2022)
11. Atiskov A. Y., Novikov F. A., Fedorchenko L. N. et al., *Ontology-Based Analysis of Cryptography Standards and Possibilities of Their Harmonization* // *Theory and Practice of Cryptography Solutions for Secure Information Systems*, 2013.
  12. Meyer, B. *Object-Oriented Software Construction* // Prentice-Hall, 2000
  13. Афанасьева И.В. Метод проектирования и реализации параллельных реагирующих систем // Диссертация на соискание ученой степени кандидата технических наук, 2018
  14. Ахо А. В., Сети Р., Лам М. С, *Компиляторы: принципы, технологии и инструменты* // 1986
  15. Новиков Ф.А., *Визуальное конструирование программ* // *Информационно-управляющие системы* № 6, 2005
  16. ANother Tool for Language Recognition [Электронный ресурс] – URL: <https://wwwantlr.org/> (дата обращения 15.05.2022)
  17. Anzanello M. *Learning curve models and applications: literature review and research directions* // *International Journal of Industrial Ergonomics*, 2011
  18. Фаулер М. *Предметно-ориентированные языки программирования.* // М.: Вильямс, 2011.
  19. Мотвани Раджив, Хопкрофт Джон Э., Ульман Джеффри Д, *Введение в теорию автоматов, языков и вычислений* // М.: Вильямс, 2008.
  20. Обзор IBM Rational [Электронный ресурс] – URL: <https://www.ibm.com/developerworks/ru/rational/> (дата обращения 21.04.2022)
  21. Новичков А.Н, *Rational Rose для разработчиков и ради разработчиков* // 2000
  22. Vysoky P., *Grammar to JetBrains MPS Convertor* // Prague 2016
  23. Emden R., *Drawing graphs with dot* // 2015
  24. Welcome to Graphviz [Электронный ресурс] – URL: <https://www.graphviz.org/> (дата обращения 29.04.2022)

25. JavaScript-библиотека для создания пользовательских интерфейсов [Электронный ресурс] – URL: <https://ru.reactjs.org/> (дата обращения 29.04.2022)

26. AceEditor [Электронный ресурс] – URL: <https://ace.c9.io/> (дата обращения 29.04.2022)

27. Flask's documentation [Электронный ресурс] – URL: <https://flask.palletsprojects.com/en/1.1.x/> (дата обращения 31.04.2022)

# Приложение 1

## Грамматика языка описания онтологий, записанная в ANTLR4

**grammar** ontology;

```
file_           : package+ EOF;
package         : PACKAGE name_ LBRACE types? class_* enum_* rel? comm*
RBRACE;
types           : TYPES LBRACE (type_name COMMA)* type_name RBRACE;
class_          : CLASS name_ LBRACE attr* oper* RBRACE;
enum_           : ENUMERATION name_ LBRACE (string COMMA)* string RBRACE;
rel_            : (act | assoc | gen | aggr | comp | dep | impl)+;
comm            : COMMENT name_ string SEMICOLON;
act             : ACTIVITY name_ LBRACE arg+ res+ RBRACE;
assoc           : ASSOCIATION name_ LBRACE m_arg m_res attr* RBRACE;
gen             : GEN name_ LBRACE arg+ res attr* RBRACE;
aggr            : AGGREGATION name_ LBRACE m_arg+ m_res+ RBRACE;
comp            : COMPOSITION name_ (LBRACKET string COMMA string
RBRACKET)* SEMICOLON;
dep             : DEPENDENCY name_ SEMICOLON;
impl            : IMPLEMENTATION name_ SEMICOLON;
attr            : ATTRIBUTES type_name name_ SEMICOLON;
oper            : OPERATIONS (string COMMA)* string SEMICOLON;
arg             : ARGUMENT type_name? name_ SEMICOLON;
res             : RESULT type_name? name_ SEMICOLON;
m_arg           : ARGUMENT type_name? name_ mult? SEMICOLON;
m_res           : RESULT type_name? name_ mult? SEMICOLON;
mult            : LSBRACE mult_cnt RSBRACE;
in_             : INPUT (name_ COMMA)* name_ SEMICOLON;
out_            : OUTPUT (name_ COMMA)* name_ SEMICOLON;
enum_literals   : LITERALS (string COMMA)* string SEMICOLON;
type_name       : TYPE_NAME;
name_           : NAME_;
mult_cnt        : MULT;
string          : STRING;

MULT            : ('0' | '1' | '*' | '0..1' | '1..*');
NAME_           : LTBRACE SYM RTBRACE;
TYPE_NAME       : APOSTROPHE SYM APOSTROPHE;
STRING          : DQUOTE SYM DQUOTE;
PACKAGE         : 'package';
TYPES           : 'types';
CLASS           : 'class';
ENUMERATION     : 'enum';
ACTIVITY        : 'act';
ASSOCIATION     : 'assoc';
GEN             : 'gen';
ATTRIBUTES      : 'attr';
OPERATIONS      : 'operations';
AGGREGATION     : 'aggr';
COMPOSITION     : 'composition';
DEPENDENCY      : 'dependency';
IMPLEMENTATION   : 'implementation';
INPUT           : 'input';
OUTPUT          : 'output';
LITERALS        : 'literals';
COMMENT         : 'comm';
ARGUMENT        : 'arg';
RESULT          : 'res';
LBRACKET        : '(';
```



```

RBRACKET      : ')';
LBRACE        : '{';
RBRACE        : '}';
LSBRACE       : '[';
RSBRACE       : ']';
LTBRACE       : '<';
RTBRACE       : '>';
DQUOTE        : '"';
COLON         : ':';
COMMA         : ',';
APOSTROPHE    : '\'';
SEMICOLON     : ';';
fragment SYM : ('\u0041..\u0044F' | 'A'..'Z' | 'a'..'z' | '0'..'9' |
'-' | '_' | '.')*;

SPACE         : [ \t\r\n] -> skip;

```

## Приложение 2

Python класс для обработки сущностей классов, сгенерированный инструментом, дописанный студентом.

```

class Class:

    #VARS
    def __init__(self, class_container, file):
        self.exitClassesFlag = False
        self.__state__ = "entry"

        self.class_container = class_container
        self.file = file

    #INNER
    #PROVIDED
    def __process(self):
        self.run()

    def __processAttrs(self):
        List('attr', self.class_container.attrs, self.file).run()

    def __processOpsers(self):
        List('oper', self.class_container.opsers, self.file).run()

    def __preProcess(self):
        print('{', file=self.file, end='')
        print('\\"name\\": \"' + self.class_container.name + '\",',
file=self.file, end='')

    def __postProcess(self):
        print('}', file=self.file, end='')

    def __separate(self):
        print(',', file=self.file, end='')

    def run(self):
        while (True):
            if self.__state__ == "exit":
                break
            if self.__state__ == "entry":

```

```

        if True:
            self.__state__ = "pre_process"
            continue
    if self.__state__ == "pre_process":
        if True:
            self.__preProcess()
            self.__state__ = "attr_processing"
            continue
    if self.__state__ == "attr_processing":
        if True:
            self.__processAttrs()
            self.__state__ = "separate"
            continue
    if self.__state__ == "separate":
        if True:
            self.__separate()
            self.__state__ = "oper_processing"
            continue
    if self.__state__ == "oper_processing":
        if True:
            self.__processOps()
            self.__state__ = "post_process"
            continue
    if self.__state__ == "post_process":
        if True:
            self.__postProcess()
            self.__state__ = "exit"
            continue

```

## Приложение 3

Пример программы на разработанном языке описания онтологий и JSON-объекта полученного при запуске программы.

```

package <Булевы функции> {
    types {
        'функция'
    }
    class <Функция> {}
    class <Функция k-значной логики> {}
    class <Булева функция> {}
    class <Элементарная> {}
    class <Симметричная> {}
    class <Самодвойственная> {}
    class <Формула> {}
    enum <E2> {
        "0",
        "1"
    }
    gen <Булевы функции gen 1> {
        arg 'функция' <Функция k-значной логики>;
        res <Функция>;
    }
    gen <Булевы функции gen 2> {
        arg 'функция' <Булева функция>;
        res 'функция' <Функция k-значной логики>;
    }
    gen <Булевы функции gen 3> {
        arg 'функция' <Булева функция>;
        res <Функция>;
    }
    gen <Булевы функции gen 4> {

```

```

        arg 'функция' <Элементарная>;
        res 'функция' <Булева функция>;
    }
    gen <Булевы функции gen 5> {
        arg 'функция' <Симметричная>;
        res 'функция' <Булева функция>;
    }
    gen <Булевы функции gen 6> {
        arg 'функция' <Самодвойственная>;
        res 'функция' <Булева функция>;
    }
    assoc <Реализует> {
        arg <Формула> [*];
        res 'функция' <Булева функция> [1];
    }
    act <Принцип двойственности> {
        arg <Формула>;
        res <Формула>;
    }
    act <Эквивалентные преобразования> {
        arg <Формула>;
        res <Формула>;
    }
    act <Алгоритм интерпретации> {
        arg <Формула>;
        res 'функция' <Булева функция>;
    }
    comm <Булева функция> "Истинностные значения. Существенные и фиктивные
переменные";
}

```

```

{
    "name": "Булевы функции",
    "types": [
        "функция"
    ],
    "class": [
        {
            "name": "Функция",
            "attr": [],
            "oper": []
        },
        {
            "name": "Функция k-значной логики",
            "attr": [],
            "oper": []
        },
        {
            "name": "Булева функция",
            "attr": [],
            "oper": []
        },
        {
            "name": "Элементарная",
            "attr": [],
            "oper": []
        },
        {
            "name": "Симметричная",
            "attr": [],
            "oper": []
        },
    ],
}

```

```

        "name": "Самодвойственная",
        "attr": [],
        "oper": []
    },
    {
        "name": "Формула",
        "attr": [],
        "oper": []
    }
],
"rel": [
    {
        "type": "act",
        "name": "Принцип двойственности",
        "arg": [
            {
                "name": "Формула"
            }
        ],
        "res": [
            {
                "name": "Формула"
            }
        ]
    },
    {
        "type": "act",
        "name": "Эквивалентные преобразования",
        "arg": [
            {
                "name": "Формула"
            }
        ],
        "res": [
            {
                "name": "Формула"
            }
        ]
    },
    {
        "type": "act",
        "name": "Алгоритм интерпретации",
        "arg": [
            {
                "name": "Формула"
            }
        ],
        "res": [
            {
                "type": "функция",
                "name": "Булева функция"
            }
        ]
    }
]
]
}

```

## Приложение 4

Грамматика, описывающая язык описания грамматик, который принимает на вход ANTLR4

```
grammar ANTLRgrammar;

grammar_ : grammar_decl (lexer_command | lexer_rule | parser_rule )+;
grammar_decl : grammarType grammar_name SEMI;
lexer_command : lexer_name COLON alternative ARROW command SEMI;
lexer_rule : lexer_name COLON operand+ SEMI;
parser_rule : parser_name COLON operand+ SEMI;
operand : LBRACE operand RBRACE
        | operand POST_OPERATION
        | PRED_OPERATION operand
        | (string | lexer_name | parser_name)+;
grammarType : GRAMMAR;
grammar_name : (NAME | LEXER_NAME | PARSEER_NAME);
lexer_name : LEXER_NAME;
parser_name : PARSEER_NAME;
lexer_command_ : LEXER_COMMAND;
string : STRING;
command : COMMAND;
alternative: ALTERNATIVE;

GRAMMAR : 'grammar';
COMMAND: ('skip' | 'more');
ALTERNATIVE: LSBRACE AL_SYM RSBRACE;
STRING : QUOTE SYM QUOTE;
LEXER NAME: BIG LETTER SYM NAME;
PARSEER NAME: SMALL LETTER SYM NAME;
NAME : (BIG LETTER | SMALL LETTER) SYM;
SEMI : ';';
COLON : ':';
QUOTE : '\'';
POST_OPERATION : POST_OPERATIONF;
PRED_OPERATION : PRED_OPERATIONF;

fragment BIG LETTER : 'A'..'Z';
fragment SMALL LETTER : 'a'..'z';
LSBRACE : '[';
RSBRACE : ']';
fragment AL_SYM : ('A'..'Z' | '\\t' | '\\r' | '\\n' | ' ' | 'a'..'z' |
'0'..'9' | '_' | '-' | '.' | '(' | ')' | '{' | '}' | '[' | ']' | '<' | '>' |
'=' ) *;
fragment SYM_NAME : ('A'..'Z' | '\\t' | '\\r' | '\\n' | 'a'..'z' | '0'..'9' |
'_') *;
fragment SYM : ('A'..'Z' | '\\t' | '\\r' | '\\n' | 'a'..'z' | '0'..'9' | '_' | '-'
| '.' | '(' | ')' | '{' | '}' | '[' | ']' | '<' | '>' | '=' ) *;
fragment POST_OPERATIONF : '*' | '+' | '?';
fragment PRED_OPERATIONF : '~';
LBRACE : '(';
RBRACE : ')';

ARROW : '->';

SPACE: [ \t\r\n] -> skip;
```