Санкт-Петербургский политехнический университет Петра Великого

Физико-механический институт

Курсовая работа

«Разработка метода генерации интерпретаторов DSL с использованием графовых моделей и LLM»

по дисциплине «Автоматизация научных исследований»

Выполнил
студент гр. № 5040102/50201
Соломатов А.Д.

Преподаватель:
Новиков Ф.А.

Санкт-Петербург

2025 г.

Аннотация

Объем: 126 слов.
Ключевые слова: DSL, LLM, UEPE, HOT RELOAD, AST, VISITOR PATTERN, AI GENERATION.
Исследование посвящено автоматизации проектирования инструментария для предметно-ориентированных языков (Domain-Specific Languages, DSL). Цель работы заключается в создании метода генерации интерпретаторов, минимизирующего ручное написание программного кода для реализации семантики. Автором предложен гибридный подход, объединяющий детерминированный синтаксический анализ с возможностями больших языковых моделей (Large Language Models, LLM) для формирования семантических обработчиков на базе паттерна Visitor. Оригинальность метода состоит в использовании контекстных подсказок внутри грамматических правил для управления синтезом логики обхода деревьев разбора.

Результатом исследования стала программная среда DslTools, поддерживающая концепцию единого исполняемого окружения (Unified Executable Parsing Environment, UEPE) с механизмом «горячего» обновления грамматик. Разработанный инструментарий позволяет значительно ускорить прототипирование специализированных языков и снизить требования к квалификации разработчиков в области теории компиляторов. Интеграция LLM в процесс генерации кода признана эффективным средством автоматизации рутинных операций при сохранении строгого контроля над структурой данных.

Abstract

Word count: 129
Keywords: DSL, LLM, UEPE, HOT RELOAD, AST, VISITOR PATTERN, AI GENERATION.
This research focuses on automating the development of tooling for Domain-Specific Languages (DSLs). The study aims to develop a method for generating interpreters that minimizes manual coding for implementing language semantics. The author proposes a hybrid approach, integrating deterministic syntactic analysis with the capabilities of Large Language Models (LLMs) to create semantic handlers based on the Visitor pattern. The method's novelty lies in utilizing contextual prompts embedded within grammar rules to direct the synthesis of parse tree traversal logic.

The study resulted in the DslTools software environment, which supports the Unified Executable Parsing Environment (UEPE) concept and features a "hot reload" mechanism for grammars. The developed toolkit significantly accelerates the prototyping of specialized languages and reduces the expertise required in compiler theory. The integration of LLMs into the code generation process is demonstrated to be an effective means of automating routine operations while maintaining strict control over data structures

Оглавление

1 Введение

Современный этап развития информационных технологий характеризуется активным внедрением языковых инструментов в самые разнообразные сферы: от систем промышленной автоматизации и компьютерной безопасности до анализа биоинформатических данных. Рост сложности программных комплексов и увеличение ассортимента вычислительных устройств диктуют потребность в создании специализированных предметно-ориентированных языков (DSL). На сегодняшний день насчитывается более 10 тысяч языков, применяемых не только в общесистемном программном обеспечении, но и в узких прикладных областях, что делает задачу автоматизации их разработки критически важной.

Традиционно процесс создания языковых процессоров опирается на принципы синтаксического управления, где управляющие структуры описываются с помощью контекстно-свободных грамматик в формах Бэкуса-Наура (БНФ) или их расширений. Использование формальных методов на всех этапах проектирования позволяет существенно сократить трудозатраты на фазу синтаксического анализа, которая может занимать до 25% ресурсов при разработке языкового процессора. Однако переход к динамическим средам и потребность в ускорении цикла «разработка-исполнение» требуют новых подходов, выходящих за рамки классической статической компиляции.

В рамках данной работы предлагается метод генерации интерпретаторов DSL, реализованный в инструментальной среде **DslTools**. Система ориентирована на создание единого исполняемого окружения (UEPE)(*Unified executable parsing environment*), поддерживающего динамическую интерпретацию и «горячую» загрузку грамматик, что обеспечивает мгновенную обратную связь при проектировании языка.

1.1 Актуальность

Актуальность исследования обусловлена необходимостью создания гибких, способных конкурировать с такими системами, как JetBrains MPS или Xtext, но с меньшим порогом вхождения и более высокой скоростью прототипирования. В условиях рыночной экономики и частого выпуска обновленных моделей вычислительных систем, потребность в инструментах, поддерживающих концепцию *edit-run* без промежуточных стадий сборки, становится приоритетной для индустрии программного обеспечения.

1.2 Цель и задачи работы

Целью работы является разработка и формализация метода генерации интерпретаторов DSL на основе графовых моделей грамматик и технологий больших языковых моделей (LLM). Для достижения цели решаются следующие задачи:

1. Разработка модели представления грамматики.
2. Формализация метода автоматической генерации семантических обработчиков на базе паттерна *Visitor*.
3. Реализация прототипа системы **DslTools** как интерпретируемой среды исполнения.
4. Исследование влияния пользовательских аннотаций на точность генерации программного кода.

1.3 Объект исследования

Процесс проектирования и реализации предметно-ориентированных языков (Domain-Specific Languages, DSL).

1.4 Предмет исследования

Методы автоматизированной генерации интерпретаторов DSL на основе моделей грамматик и технологий больших языковых моделей (Large Language Models, LLM).

1.5 Научная новизна

Научная новизна заключается в интеграции детерминированного синтаксического разбора с интеллектуальной генерацией семантики. Впервые предложено использование пользовательских (подсказок) внутри грамматических правил для управления процессом генерации логики *Visitor-хендлеров* с помощью LLM. Это позволяет автоматизировать написание наиболее трудоемкой части интерпретатора, сохраняя при этом строгий контроль над синтаксической структурой данных.

1.6 Практическая значимость

Практическая значимость работы заключается в возможности применения системы **DslTools** для быстрого прототипирования DSL в учебном процессе. Предложенный подход позволяет сократить сроки разработки специализированных языков и повысить надежность ПО за счет исключения ручного кодирования рутинных операций по обходу деревьев разбора.

2 Обзор литературы

Для глубокого понимания механизмов автоматизированной генерации DSL необходимо рассмотреть фундаментальные концепции, лежащие в основе языковой инженерии в контексте модельно-ориентированного проектирования (Model-Driven Engineering, MDE).

2.1 Различия GPL и DSL

В современной разработке программного обеспечения выделяют два основных класса языков: языки общего назначения (GPL) и предметно-ориентированные языки (DSL). Ключевым различием между ними является уровень абстракции и выразительной мощности. Как отмечается в работе [1], DSL характеризуются сниженной выразительностью в обмен на высокую специализацию, что позволяет сократить «семантический разрыв» (semantic gap) между экспертными знаниями в предметной области и программной реализацией.

В качестве примера DSL можно привести сервис TidalCycles [2], предназначенную для лайв-кодинга музыки:

```
-- Mini-notation worksheet, number one!

-- Play a "kick" sound (the first one in the folder)
d1 $ sound "kick"

-- Play a different sound from the "kick" folder (the fourth one, counting from zero)
d1 $ sound "kick:3"

-- Play a kick - snare loop. Notice two sounds fit in the same time as one did above
d1 $ sound "kick snare"

-- The more you add, the faster it goes - the 'cycle' stays constant
d1 $ sound "kick snare kick snare"

d1 $ sound "kick snare kick snare kurt hi lo hi lo"

-- Again, we can pick sounds with : and a number

d1 $ sound "cpu:0 cpu:2 cpu:4 cpu:6 cpu:0 cpu:2 cpu:6 cpu:8"

-- If they're all from the same folder, it's easier to pattern
-- the sounds using a separate "n" pattern, like this:
d1 $ n "0 2 4 6 0 2 6 8" # sound "cpu"
```

Figure 1: Пример программы на языке сервиса

Таким образом, ограничение выразительной мощности в DSL выступает не как недостаток, а как инструмент повышения производительности в конкретном домене.

2.2 Формализмы описания и синтаксический анализ

2.2.1 Теория РБНФ (EBNF)

Зачастую формальное описание синтаксиса базируется на расширенной форме Бэкуса — Наура (EBNF), которая в рамках парадигмы Unified Executable Parsing Environment (UEPE) определяет структуру всех допустимых конструкций языка. Роль РБНФ

заключается в декларативном задании правил, на основе которых строятся модель целевых грамматик. Согласно [3], необходимыми компонентами описания являются:

- **Множество терминалов (Terminal Symbols)**: элементарные символы или последовательности (литералы, идентификаторы), не подлежащие дальнейшему делению.
- **Множество нетерминалов (Non-terminal Symbols)**: синтаксические категории, определяющие иерархическую структуру языка.
- **Ключевые слова (Keywords)**: зарезервированные терминальные последовательности, определяющие семантические границы конструкций.
- **Правила вывода (Rules)** дописать

2.2.2 Конвейер анализа

Процесс трансформации исходного кода представляет собой многоуровневый конвейер, обеспечивающий переход от текстового представления к исполняемым структурам.

Tokenization (Лексический анализ) Лексический анализ выполняется токенезатором, который группирует поток входных символов в значимые лексемы (lexemes) и сопоставляет их с категориями — токенами (tokens). Часто в архитектуре реализация токенезатора предполагает формирование объекта `TokenStream`. Важной особенностью является обработка «trivia» — пробельных символов и комментариев. Согласно принципам Fidelity (верности исходному коду), эти данные игнорируются основным разбором, но сохраняются в потоке для поддержания контекста при рефакторинге и диагностике, что подтверждается в [3].

CST Generation (Построение дерева разбора) Дерево конкретного синтаксиса (Concrete Syntax Tree, CST) определяется как структура, полностью иерархически отражающая правила грамматики и включающая все без исключения токены (включая вспомогательную пунктуацию). Использование CST необходимо для сохранения полной информации о синтаксисе, что критически важно для инструментов статического анализа и IDE-сервисов. Сравнительные характеристики CST и AST:

Table 1: Сравнение характеристик CST и AST

Характеристика	**CST**	**AST**
Состав	Полный набор токенов (вкл. пунктуацию)	Только семантически значимые узлы
Структура	Соответствует правилам EBNF 1:1	Оптимизирована для вычислений
Назначение	Сохранение контекста, рефакторинг	Исполнение, оптимизация

AST Translation (Трансляция в абстрактное дерево) Трансляция CST в абстрактное синтаксическое дерево (Abstract Syntax Tree, AST) заключается в очистке модели от вспомогательных узлов. Ключевым этапом здесь является разрешение имен (Name Resolution) — процесс связывания идентификаторов с их определениями, что превращает дерево в граф связей. Реализация мапперов поддерживает как автоматическое преобразование

на основе метамоделей, так и ручное описание трансформаций для сложных доменных правил [4].

Роль паттерна Visitor Для разделения структуры данных AST и алгоритмов их обработки применяется паттерн *Visitor* [5], [6]. Это позволяет инкапсулировать логику статического анализа (Static Analysis) и интерпретации (Evaluation) в отдельных модулях, не изменяя классы узлов дерева. Использование данного паттерна обеспечивает расширяемость системы: добавление новой функциональности требует лишь реализации нового «посетителя», что минимизирует архитектурную связность.

От Интерпретатора к Компилятору Переход от интерпретации к компиляции осуществляется через расширение конвейера. Путем добавления модулей статического анализа, оптимизации (например, на основе SSA) и кодогенерации, среда исполнения трансформируется в полноценный компилятор. Этот процесс, как отмечается в [7] и [8], базируется на многофазной обработке AST, где каждая фаза выполняет специализированную трансформацию модели в сторону снижения уровня абстракции.

2.3 Применение LLM в метамоделировании и языковой инженерии

Интеграция LLM начинается задолго до написания первой строки парсера. Современные подходы внедряют генеративный ИИ на этапе концептуализации языка и создания его метамодели.

2.3.1 Генерация формальных спецификаций из неструктурированных требований

Традиционно создание метамодели (например, в Ecore для EMF или мета-схемы JSON) требует от архитектора трансляции концептуальной модели предметной области в строгие формализмы: классы, атрибуты и отношения. Данный процесс характеризуется высокой архитектурной сложностью и риском семантических искажений при использовании инструментов моделирования, таких как Eclipse MPS или EMF. Современные исследования ([9], [10]) предлагают подход *LLM-assisted Metamodeling*, структурированный следующим образом:

- *Механизм*: Системный инженер подает на вход модели описание требований на естественном языке. На основе этого контекста LLM формирует иерархию сущностей и их взаимосвязей.
- *Результат*: Генерация валидных артефактов метамоделирования (например, Ecore XML или XMI), определяющих базовые доменные классы, их атрибутивный состав и типы связей (ассоциация, композиция).
- *Итеративность*: Реализация итеративного цикла проектирования, где сгенерированная модель визуализируется (например, через PlantUML) для экспертной оценки. Обратная связь инкорпорируется в последующие запросы для уточнения формальной спецификации.

2.3.2 От естественного языка к структуре

Данная область применения LLM ориентирована на трансляцию намерений пользователя (Natural Language) непосредственно в код DSL или структурированные вызовы API, фактически заменяя собой традиционные фазы лексического и синтаксического анализа. Однако прямая генерация (NL2DSL) в проприетарных доменах ограничена проблемой

галлюцинаций, когда модель, не знакомая со спецификацией конкретного DSL, подменяет конструкции языка аналогичными из GPL [11].

Для объективной оценки качества интеграции на данном этапе применяются следующие метрики:

- *Unparsed Rate*: Доля программ, не прошедших синтаксическую верификацию детерминированным парсером; показатель корректности усвоения синтаксиса.
- *Hallucination Rate*: Доля синтаксически верных программ, использующих несуществующие в данном домене идентификаторы или семантические конструкции.
- *Average Similarity*: Степень структурного соответствия между сгенерированным и эталонным AST, рассчитываемая через поиск наибольшей общей подпоследовательности (LCSS).

Стратегии RAG для DSL: Контекстная осведомленность Для преодоления ограничений, связанных с отсутствием доменных данных в обучающей выборке, применяется подход Retrieval-Augmented Generation (RAG). Эффективность системы повышается за счет специализированной подготовки контекста:

- *AST-Based Chunking*: Использование грамматически-ориентированного разбиения кодовой базы (например, через Tree-sitter) вместо строкового сегментирования. Это гарантирует сохранение семантической целостности блоков, подаваемых в контекстное окно модели [12].
- *Семантический поиск по коду*: Применение эмбеддингов, обученных на парах «документация-код», что позволяет извлекать наиболее релевантные синтаксические паттерны DSL по текстовому запросу пользователя [13].
- *Результаты*: Использование RAG в режиме few-shot обучения позволяет достичь качества генерации, сопоставимого с результатами специализированного дообучения (Fine-tuning), при значительно меньших вычислительных затратах [11].

Grammar-Constrained Decoding (GCD): Принудительная валидность Наиболее строгим методом обеспечения корректности является ограничение процесса декодирования на основе формальной грамматики. В этом режиме LLM функционирует в рамках детерминированного конечного автомата, заданного синтаксисом DSL.

- *Механизм*: На каждом шаге авторегрессионной генерации алгоритм сопоставляет возможные токены с правилами EBNF. Токены, нарушающие синтаксис, маскируются, что исключает вероятность генерации некорректных цепочек [14].
- *Инструментарий*: Подход реализуется посредством библиотек управления генерацией (например, SynCode, guidance) или специализированных фреймворков типа YieldLang [15].
- *Grammar Prompting*: Метод передачи грамматики непосредственно в промпт, позволяющий модели адаптироваться к новому DSL *in-context*. Это критически важно для семантического парсинга в динамически меняющихся средах.

2.3.3 Работа с AST и семантикой

Автоматическая генерация инфраструктуры Visitor Применение LLM позволяет автоматизировать создание объемного шаблонного кода (boilerplate), характерного для паттерна Visitor, обеспечивая синхронизацию между метамоделью и кодом интерпретатора.

- *Генерация иерархии*: Автоматическое построение классов узлов AST (например, на языке Python с использованием dataclasses) и соответствующих интерфейсов Visitor на основе спецификации языка [15].
- *Реализация логики обхода*: Генерация тел методов `visit` для прикладных задач, таких как Type Checker или статический анализатор. Модель способна синтезировать правила вывода типов на основе текстового описания семантики.
- *Минимизация архитектурных издержек*: Решение проблемы выражения (Expression Problem) — при изменении структуры AST модель способна мгновенно регенерировать все зависимые компоненты визитора, сохраняя целостность системы.

LLM как семантический анализатор (Semantic Analyzer) Интеграция LLM в конвейер анализа позволяет дополнить классические алгоритмы «мягкими» проверками, основанными на вероятностном выявлении логических аномалий в коде.

- *Поиск логических аномалий*: Анализ AST на предмет трудноформализуемых ошибок, таких как потенциальные утечки данных в лог-файлы или нарушение бизнес-логики, не описываемой строгой системой типов.
- *AST Enrichment*: Использование LLM для вероятностного восстановления типов или заполнения атрибутов в разреженных деревьях, где входная информация недостаточна для работы стандартного семантического анализатора [16].
- *Интеллектуальный линтинг*: Генерация расширенных отчетов о качестве кода на основе обхода узлов AST, что позволяет использовать LLM как продвинутую систему диагностики в среде разработки.

Нейро-символический поиск в пространстве программ В задачах автоматического синтеза программ (Program Synthesis) гибридный подход позволяет преодолеть проблему комбинаторного взрыва в широких пространствах поиска.

- *LLM как эвристическая функция*: Модель выступает в роли генератора гипотез (proposals), предлагая наиболее вероятные последовательности операторов DSL, соответствующие заданным примерам ввода-вывода [17].
- *Символьная верификация*: Предложенные LLM конструкции используются для направления точных методов поиска (например, A* или DFS), которые отсекают невалидные ветви с использованием формальных методов проверки.
- *Эффективность*: Нейро-символическая архитектура демонстрирует превосходство над чистыми LLM в задачах, требующих точных вычислений (ARC, FlashFill), обеспечивая при этом гибкость в выборе стратегий синтеза.

2.4 Вывод по главе

Проведенный анализ подтверждает, что использование «чистых» LLM для генерации интерпретаторов DSL ограничено проблемой низкой точности и отсутствием гарантий корректности. Решением является переход к гибридным системами, которые объединяют гибкость нейросетевого вывода с жесткими ограничениями, накладываемыми формальными моделями грамматик в среде UEPE.

3 Разработка программного обеспечения.

В данной главе рассматривается техническая реализация системы DslTools — программного комплекса, спроектированного как UEPE. Система предназначена для обеспечения полного цикла разработки DSL, начиная от формального описания грамматики и заканчивая исполнением целевого кода. Ключевой инженерной задачей при проектировании системы являлось создание архитектуры, способной поддерживать динамическое изменение правил языка "на лету" без необходимости перезапуска среды выполнения. Этот подход, реализованный через механизм Hot Reload, позволяет сократить время итерации разработки языка. Программная реализация базируется на строгом разделении ответственности между компонентами, модульности ядра и контейнеризации инфраструктуры, что обеспечивает масштабируемость и надежность системы при работе с пользовательским кодом, который по определению может быть нестабильным.

3.1 Основные сценарии использования

Жизненный цикл работы в UEPE не является линейным; он представляет собой набор взаимосвязанных процессов, инициируемых различными акторами. В основе проектирования лежит принцип бесшовности — минимизации когнитивной и технической нагрузки при переключении между контекстами разработки грамматики и исполнения кода.

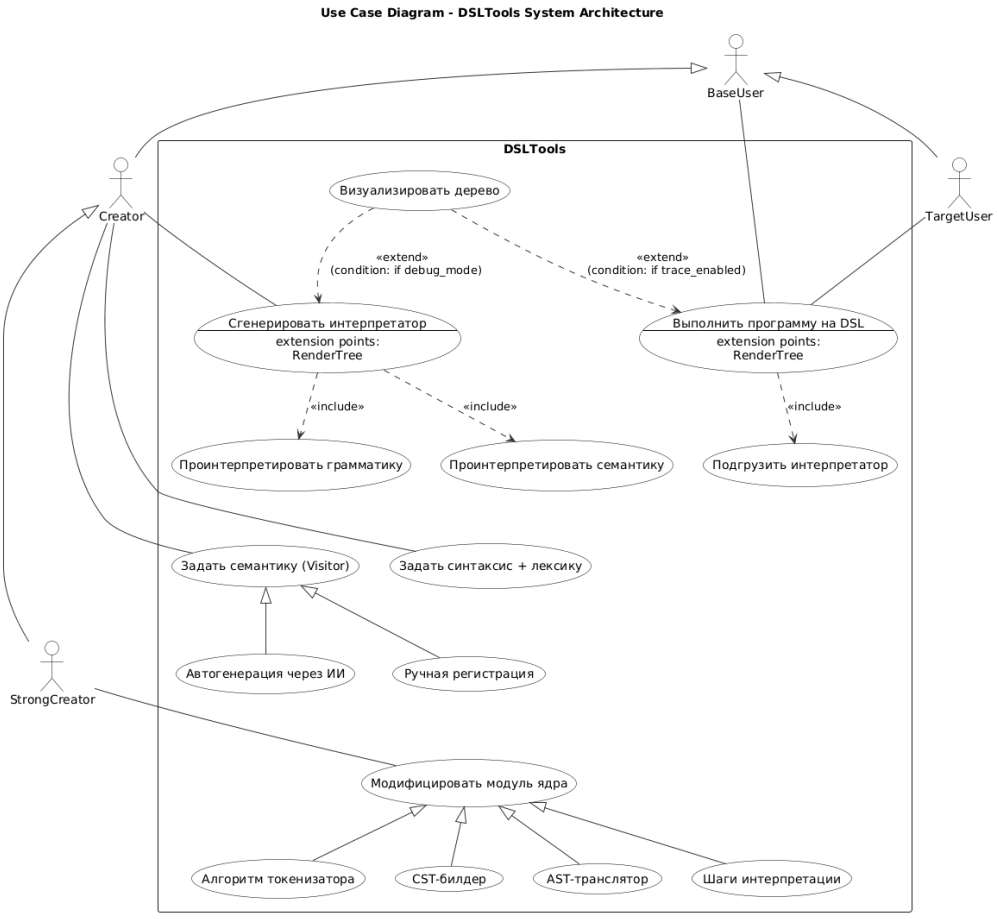

Figure 2: Диаграмма использования для системы DSLTools.

3.1.1 Сценарий разработчика (Creator)

Роль разработчика (Creator) является центральной в экосистеме DslTools. Процесс создания языка традиционно связан с длительной цепочкой действий: правка грамматики, генерация парсера, компиляция, сборка проекта, запуск тестов. В DslTools этот цикл сжат до единственного действия — сохранения конфигурации. Жизненный цикл разработчика строится следующим образом:

- **Описание грамматики**: Разработчик взаимодействует с панелью мета-моделирования, вводя правила в формате EBNF (Extended Backus-Naur Form). Система осуществляет валидацию синтаксиса EBNF в реальном времени.
- **Автоматическая сборка (Hot Reload)**: При сохранении изменений (или по таймеру простоя) система инициирует фоновый процесс пересборки интерпретатора. В отличие от классических компиляторов, DslTools не создает физические исполняемые файлы на диске для каждого изменения. Вместо этого происходит обновление графа состояний в памяти активного контейнера
- **Верификация**: Немедленно после пересборки обновляется визуализация деревьев разбора (CST/AST). Разработчик может видеть, как изменение в правиле (например, изменение приоритета операторов) мгновенно отражается на структуре дерева для тестового примера.

3.1.2 Сценарий пользователя (TargetUser)

Для конечного пользователя (TargetUser) система DslTools выступает как "черный ящик" или IDE для конкретного DSL. Пользователь не должен знать о внутренней сложности генерации парсеров, а использует готовый интерпретатор / компилятор полученный от разработчика. По команде "RunDSL" 1 инициируется выполнение программы.

3.1.3 Сценарий эксперта (Strong Creator)

Роль Strong Creator предъявляет наиболее жесткие требования к архитектуре. Эксперт имеет возможность вмешиваться в работу конвейера (Pipeline) обработки кода. Это не просто изменение правил грамматики, но и подмена алгоритмов. Пример сценария:

- Эксперт решает, что стандартный лексер на основе регулярных выражений не справляется с вложенными комментариями специфического формата.
- *Подготовка модуля*: Эксперт пишет код кастомного токенизатора, реализующего интерфейс ITokenizer. Инъекция зависимости: Через интерфейс настройки ядра эксперт загружает новый модуль.
- *Горячая подмена*: Система DslTools должна корректно выгрузить стандартный модуль, загрузить новый, связать его с текущим парсером и перезапустить конвейер обработки, не разрывая пользовательскую сессию.

3.2 Архитектура приложения

Архитектура DslTools спроектирована как модульная система с слабой связностью (loose coupling), что является необходимым условием для реализации требований расширенной модификации ядра (Strong Creator) и работы механизма Hot Reload. В основе архитектуры лежит паттерн «Конвейер» (Pipeline), где каждый этап обработки данных инкапсулирован в отдельный компонент, скрытый за абстрактным интерфейсом.

3.2.1 Модульная структура и абстракция конвейера

Каждый модуль конвейера определяется через абстрактный контракт (интерфейс), что позволяет производить бесшовную подмену алгоритмов:

- **Лексический анализ (ITokenizer):** Преобразует поток символов в поток токенов. Реализация по умолчанию (`RegexTokenizer`) использует скомпилированные регулярные выражения из блока `TERMINALS`.
- **Синтаксический анализ (IParser):** Преобразует поток токенов в Конкретное Синтаксическое Дерево (CST), проверяя соответствие структуре правил (`RULES`).
- **Семантический анализ и трансляция (IASTBuilder):** Очищает CST от синтаксического шума и формирует Абстрактное Синтаксическое Дерево (AST), пригодное для исполнения паттерном *Visitor*.

3.2.2 Описание грамматики. Объект GrammarObject

Центральным элементом, связывающим декларативное описание языка с исполняемым кодом, является `GrammarObject`. Это активная сущность, которая инкапсулирует:

- **Словарь терминалов:** Отображение имен токенов на скомпилированные объекты регулярных выражений (`re.compile`), что критично для производительности.
- **Граф правил:** Направленный граф, позволяющий выявлять левую рекурсию и недостижимые правила на этапе инициализации.
- **Метаданные:** Версия языка, имя модуля и настройки генерации, извлекаемые из `METABLOCK`.

Система обеспечивает атомарность изменений: при редактировании EBNF старый объект не модифицируется, а заменяется новым, что исключает нахождение системы в несогласованном состоянии.

3.2.3 Механизм Hot Reload

Реализация Hot Reload в UEPE включает следующие этапы:

1. **Watchdog:** Фоновый сервис отслеживает изменения в файлах проекта.
2. **Debounce:** Таймер «дребезга» предотвращает избыточные пересборки при частых сохранениях.
3. **Атомарная подмена:** Глобальный указатель на активный интерпретатор блокируется мьютексом, после чего ссылка подменяется на новый инстанс Pipeline, а старый объект помечается для сборки мусора.

3.2.4 Стратегия инстанцирования и чистота состояния

В системе принята стратегия *Isolated Re-instantiation* (изолированное пересоздание). При каждом событии перезагрузки происходит полный сброс контекста исполнения:

- **Память:** Все переменные предыдущей сессии уничтожаются.
- **Visitor:** Инстанс обходчика дерева создается заново.

Это гарантирует, что побочные эффекты от выполнения кода на старой версии грамматики не повлияют на поведение системы после обновления.

3.2.5 Контейнеризация и инфраструктура

Для обеспечения безопасности и переносимости DslTools развертывается с использованием Docker и разделена на независимые сервисы:

- **UI Container:** Содержит статику веб-интерфейса и сервер Nginx. Изолирован от логики исполнения, поэтому падение ядра не влияет на доступность интерфейса.
- **Core Container (Backend):** Содержит среду выполнения, библиотеки парсинга и механизмы Hot Reload.
- **Communication:** Взаимодействие между компонентами осуществляется через REST API.

3.2.6 Модуль автоматической генерации семантических обработчиков

Выбор базовой большой языковой модели (LLM) является определяющим архитектурным решением при создании инструментов автоматизированной разработки DSL. Анализ актуальных открытых решений на 2024–2025 годы показывает, что современные модели практически сравнялись по качеству генерации кода с проприетарными системами, демонстрируя точность на уровне 70–80% в специализированных тестах.

Для реализации системы DslTools был выбран гибридный подход, сочетающий возможности облачных и локальных вычислений. В качестве основной внешней модели для сложных задач синтеза и трансляции требований выбран DeepSeek API (модели Coder V2/V3), обладающий развитыми способностями к рассуждению и пониманию контекста репозитория. Для обеспечения оперативной локальной разработки и высокой скорости инференса без обращения к сети выбрана модель семейства Qwen2.5-Coder (в вариациях 7B или 32B), которая показывает наилучшее соотношение производительности и требований к видеопамяти.

Сравнительные характеристики моделей, послужившие обоснованием для их выбора, приведены в таблице 2.

Table 2: Рекомендованные модели для генерации инфраструктуры DSL

Сценарий	**Модель**	**Размер**	**Комментарий**
Максимальное качество	DeepSeek-Coder-V2	236B	Высочайший уровень интеллекта, подходит для сложных иерархий.
Баланс ресурсов	Qwen2.5-Coder-32B	32B	Оптимальный выбор для работы на одной мощной видеокарте.
Локальная работа	Qwen2.5-Coder-7B	7B	Работает на мобильных устройствах с высокой скоростью.
Юридическая чистота	StarCoder2-15B	15B	Обучена исключительно на лицензионно чистом коде.

3.2.7 Контейнеризация и инфраструктура

Система развертывается с помощью Docker и разделена на независимые сервисы:

- **UI Container:** Содержит статику веб-интерфейса и веб-сервер Nginx.
- **Core Container (Backend):** Содержит среду выполнения, библиотеки парсинга и механизмы Hot Reload.

Взаимодействие между компонентами происходит через REST API или WebSocket, что позволяет разносить их на разные машины.

3.3 Пользовательский интерфейс

Интерфейс реализован в парадигме «Language Workbench», объединяя инструменты разработки и использования языка в едином пространстве.

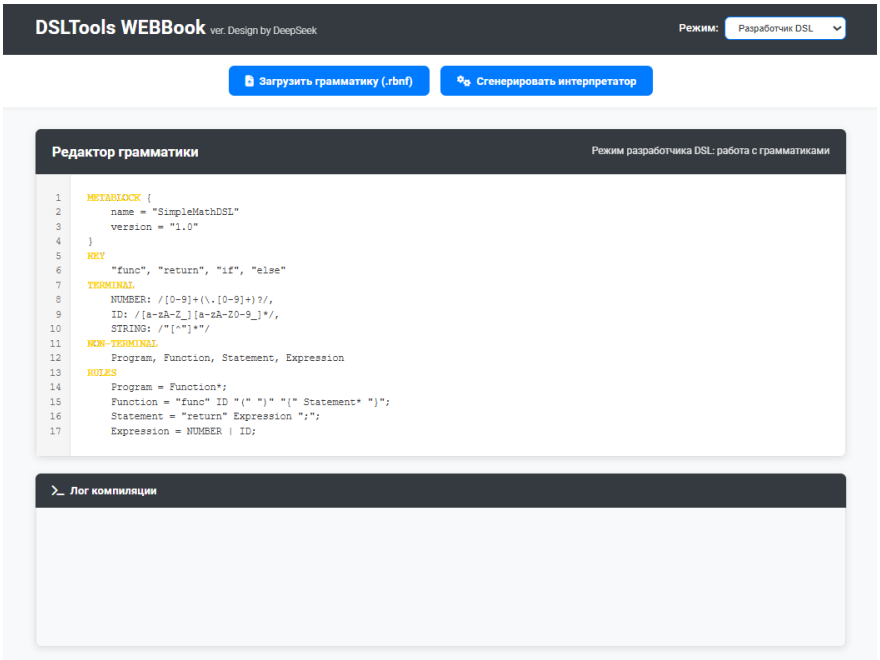

Figure 3: Интерфейс разработчика.

- **Панель мета-моделирования:** Редактор кода для EBNF с подсветкой синтаксиса и валидацией правил в реальном времени.
- **Окно визуализации:** Интерактивный холст для отображения CST/AST и графов правил с поддержкой двустороннего маппинга (клик по узлу подсвечивает код) (не представлен в текущей версии).
- **Панель диагностики:** Вкладки для вывода логов сборки, результатов выполнения программы и системных сообщений.

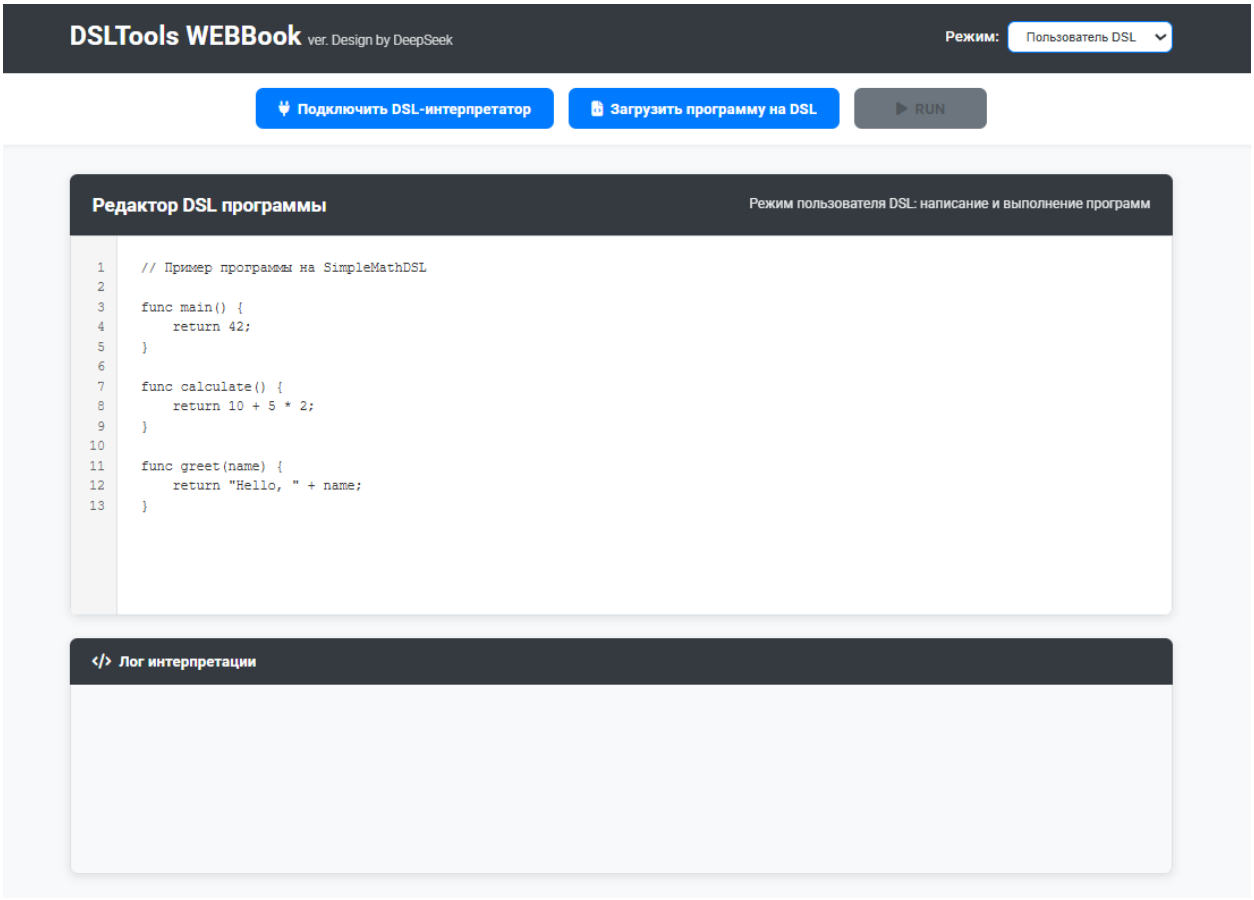

Figure 4: Интерфейс пользователя. На иллюстрации представлен редактор для написания кода на целевом DSL с динамической подсветкой синтаксиса.

3.4 Пример задания грамматики целевого языка

Для демонстрации используется грамматика PSECO переводящая алгоритмы на псевдокоде в валидный .tex код.

Описание грамматики PSECO

```
TERMINALS:
exprtext      ::= '\$[^\$]*\$';
commenttext   ::= '\s*//[^\n]*';
name          ::= '[A-Za-z_][A-Za-z0-9_]*';
number        ::= '[0-9]+';
symbol        ::= '[:=#;(){}\[\]|.,<>+^]';
text          ::= '"[^"]*"';
whitespace    ::= '\s+';
programname   ::= '"[^"]*\n"';

KEYS:
'IF';
'THEN';
'ELSE';
'ELSEIF';
'END_IF';
```

```
'FOR';
'DO';
'END_FOR';
'REPEAT';
'UNTIL';
'WHILE';
'END_WHILE';
'FUNC';
'END_FUNC';
'PROC';
'END_PROC';
'ITER';
'END_ITER';
'INPUT';
'OUTPUT';
'GOTO';
'ARRAY';
'STRUCT';
'SELECT';
'YIELD';
'RETURN';
'NEXT';
'FROM';
'TO';
'IN';
'OF';
'(';
')';
',';
'=';

NONTERMINALS:
Program; Block; Expression; FuncCall; Arguments; Parameters;
Range; Set; Variable; Identifier; Number; Text; Assignment;
Conditional; Loop; LC; LA; LB; LS; FuncDecl; ProcDecl; IterDecl;
Input; Output; Comment; Goto; ArrayDecl; StructDecl; FieldDecl;
Select; Yield; Return; NextFor;

AXIOM:
Program

RULES:
Program        ::= name { Block };
Block          ::= Assignment | Conditional | Expression | Loop | FuncCall |
  FuncDecl | ProcDecl | IterDecl | Input | Output | Comment | Goto |
  ArrayDecl | StructDecl | Select | Return | Yield | NextFor;
Expression     ::= exprtext;
FuncCall       ::= Identifier '(' [ Arguments ] ')';
Arguments      ::= Expression { ',' Expression };
Parameters     ::= Variable { ',' Variable };
Range          ::= 'FROM' Number 'TO' Number;
Set            ::= Identifier | Range;
```

```
Variable       ::= Identifier;
Identifier     ::= name | exprtext;
Number         ::= number;
Text           ::= text;
Assignment     ::= Variable '=' Expression;
Conditional    ::= 'IF' Expression 'THEN' Block [ { 'ELSEIF' Expression 'THEN
  ' Block } ] [ 'ELSE' Block ] 'END_IF';
Loop           ::= LC | LA | LB | LS;
LC             ::= 'FOR' Variable Range 'DO' Block 'END_FOR';
LA             ::= 'REPEAT' Block 'UNTIL' Expression;
LB             ::= 'WHILE' Expression 'DO' Block 'END_WHILE';
LS             ::= 'FOR' Variable 'IN' Set 'DO' Block 'END_FOR';
FuncDecl       ::= 'FUNC' Identifier '(' [ Parameters ] ')' Block Return '
  END_FUNC';
ProcDecl       ::= 'PROC' Identifier '(' [ Parameters ] ')' Block 'END_PROC';
IterDecl       ::= 'ITER' Identifier '(' [ Parameters ] ')' Block Yield '
  END_ITER';
Input          ::= 'INPUT' Expression;
Output         ::= 'OUTPUT' Expression;
Comment        ::= commenttext;
Goto           ::= 'GOTO' Identifier;
ArrayDecl      ::= 'ARRAY' Expression 'OF' Identifier | 'ARRAY' Expression '
  OF' Expression;
StructDecl     ::= 'STRUCT' Identifier '{' { FieldDecl } '}';
FieldDecl      ::= Identifier ':' Identifier | Identifier ':' Statement;
Select         ::= 'SELECT' Variable 'IN' Set;
Yield          ::= 'YIELD' Expression;
Return         ::= 'RETURN' Expression;
NextFor        ::= 'NEXT' 'FOR' Variable;
```

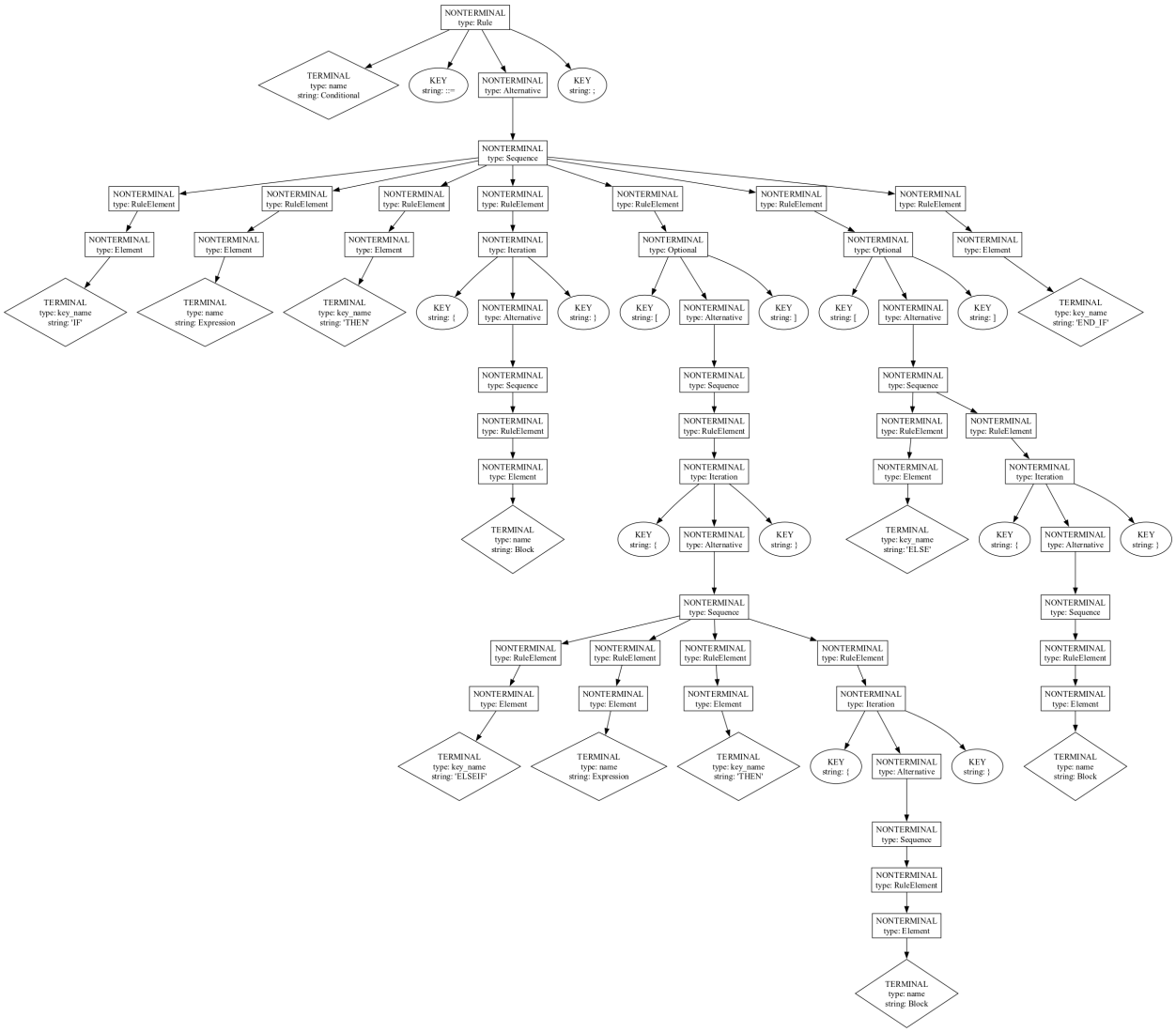

Figure 5: Пример дерева разбора правила Conditional

3.5 Примеры программ на целевом языке

Ниже приведен код алгоритма Евклида для нахождения НОД, адаптированный под ASCII-символы редактора:

Listing 1: Тестовая программа на DSL

```
WHILE a neq 0 DO
IF a gt b THEN
a gets a mod b
ELSE
b gets b mod a
END_IF
END_WHILE
```

После обработки данного кода в окне визуализации отображается AST.

Сам же пример имеет конечный вид:

while $a \neq 0 \land b \neq 0$ **do**
  if $a > b$ **then**
    $a \leftarrow a \bmod b$
  else

$b \leftarrow b \bmod a$
end if
end while

3.6 Примеры семантических обработчиков (сгенерированные)

Визитор для корневого узла 'Program'

```
class ProgramEval(ASTNode.IAttrEval):
 """ Оборачивает код в окружение algorithm. Синтаксис: Program ::= name '{'
  Block '}' """
 def __call__(self, value: str, children: List, context: Any) -> str:

   # children - name (Terminal)
   # children - '{'
   # children - Block (NonTerminal)

    program_name = children.value
    # Рекурсивный вызов визитора для блока тела( программы)
    body_code = children.evaluated(context)

    latex_out = (
    f"\\begin{{algorithm}}\n"
    f"\\caption{{{program_name}}}\n"
    f"\\begin{{algorithmic}}\n"
    f"{body_code}\n"
    f"\\end{{algorithmic}}\n"
    f"\\end{{algorithm}}"
    )
    return latex_out
```

Визитор для последовательности инструкций 'SequenceEval'

```
class SequenceEval(ASTNode.IAttrEval):
Визитор""" для последовательности инструкций (Block).
Обходит всех детей и склеивает результаты через перевод строки."""
def __call__(self, value: str, children: List, context: Any) -> str:
  results = []
  for child in children:
  # Вызов evaluated() автоматически находит нужный визитор в реестре
  res = child.evaluated(context)
  if res:
  results.append(res)
  return "\n".join(results)
```

Визитор для операции присваивания 'AssignmentEval'

```
class AssignmentEval(ASTNode.IAttrEval):
```

```
Визитор""" для операции присваивания.
Синтаксис: Variable 'gets' Expression
Генерация: \State $Variable \gets Expression$"""
def __call__(self, value: str, children: List, context: Any) -> str:
  # children[0]: Variable
  # children[1]: 'gets' (Token)
  # children[2]: Expression

  lhs = children[0].evaluated(context)
  # Получаем LaTeXсимвол- для оператора из карты
  op_token = children[1].value
  op_latex = LATEX_OPS.get(op_token, r"\gets")
  rhs = children[2].evaluated(context)
  # Оборачиваем в \State и математический режим $...$
  return f"\\State ${lhs} {op_latex} {rhs}$"
```

Визитор для условного оператора 'ConditionalEval'

```
class ConditionalEval(ASTNode.IAttrEval):
Визитор""" для условного оператора IF.
Синтаксис: 'IF' Expression 'THEN' Block 'END_IF'"""
def __call__(self, value: str, children: List, context: Any) -> str:
  # Индексы детей зависят от грамматики.
  # 0: IF, 1: Expr, 2: THEN, 3: Block (If-Body)
  condition = children[1].evaluated(context)
  if_body = children[3].evaluated(context)

  latex = f"\\If{{${condition}$}}\n"
  # Добавляем отступы для читаемости исходника LaTeX
  latex += f"    {if_body}\n"

  # Проверка наличия блока ELSE если( детей больше 5)
  # Структура с ELSE: ... ELSE Block END_IF всего( 6+ частей)
  if len(children) > 5 and children[4].value == 'ELSE':
  else_body = children[5].evaluated(context)
  latex += f"\\Else\n    {else_body}\n"

  latex += r"\EndIf"
  return latex
```

Визитор для цикла 'LoopEval'

```
class LoopEval(ASTNode.IAttrEval):
Визитор""" для цикла WHILE.
Синтаксис: 'WHILE' Expression 'DO' Block 'END_WHILE'"""
def __call__(self, value: str, children: List, context: Any) -> str:
  condition = children[1].evaluated(context)
  body = children[3].evaluated(context)

  latex = f"\\While{{${condition}$}}\n"
```

```
  latex += f"    {body}\n"
  latex += r"\EndWhile"
  return latex
```

Визитор для выражений 'ExpressionEval'

```
class ExpressionEval(ASTNode.IAttrEval):
Визитор""" для выражений.
Обрабатывает переменные, числа и вложенные операции."""
  def __call__(self, value: str, children: List, context: Any) -> str:
    # Если это листовой узел нет( детей), возвращаем значение
    if not children:
    # Если значение - оператор например( 'mod'), заменяем на LaTeX
    if value in LATEX_OPS:
    return LATEX_OPS[value]
    return value

    # Если это составное выражение (Variable op Variable)
    parts = []
    for child in children:
    val = child.evaluated(context)
    parts.append(val)

    # Соединяем части пробелом
    return " ".join(parts)
```

Базовый визитор для терминалов 'ElementEval'

```
class ElementEval(ASTNode.IAttrEval):
Базовый""" визитор для терминалов и идентификаторов.
Просто возвращает значение."""
  def __call__(self, value: str, children: List, context: Any) -> str:
  if children:
  return children[0].evaluated(context)
  return value
```

4 Сравнительный анализ

В данном разделе проводится сопоставление системы DslTools с промышленными стандартами и альтернативными подходами к проектированию языков. Анализ базируется на трех парадигмах: проекционной (JetBrains MPS), парсерной (Eclipse Xtext) и генеративной (Pure LLM).

4.1 Критерии и методология оценки

Для сравнения выбраны следующие метрики:

- **Порог вхождения:** сложность освоения мета-языков и API фреймворка.

- **Скорость итерации:** время от изменения грамматики до возможности исполнения кода (цикл Edit-Run).
- **Детерминированность:** гарантия синтаксической корректности генерируемых артефактов.

4.2 Сопоставление подходов

- **JetBrains MPS** обеспечивает высокую гибкость за счет работы напрямую с AST, что исключает ошибки парсинга. Однако система требует значительных ресурсов и глубокого обучения специфическим концепциям моделирования.
- **Eclipse Xtext** предлагает стандартный текстовый подход, удобный для интеграции с системами контроля версий. Главным ограничением является необходимость полной перегенерации и компиляции парсера при каждом изменении грамматики, что замедляет прототипирование.
- **Генеративный подход (Pure LLM)** минимизирует порог вхождения, но не обладает детерминизмом. Вероятностная природа моделей приводит к синтаксическим галлюцинациям, что делает невозможным их использование как самостоятельных парсеров в инженерных задачах.

4.3 Результаты оценки DslTools

Система DslTools реализует гибридную модель: детерминированный парсер на базе интерпретируемой грамматики сочетается с интеллектуальной генерацией семантических обработчиков (Visitor). Это позволяет достичь мгновенной обратной связи через механизм Hot Reload. Сводные данные приведены в таблице 3.

Table 3: Сравнительный анализ инструментов разработки DSL

Критерий	**JetBrains MPS**	**Eclipse Xtext**	**DslTools**
Парадигма	Проекционная	Парсерная	Гибридная
Реализация семантики	Ручная (Java)	Ручная (Xtend)	Автоматическая (LLM)
Цикл Edit-Run	Средний	Медленный	Мгновенный
Порог вхождения	Высокий	Средний	Низкий
Гарантия синтаксиса	100%	100%	100%

5 Заключение

В работе представлен программный комплекс DslTools, реализующий концепцию Unified Executable Parsing Environment (UEPE). В ходе исследования подтверждена гипотеза о возможности ускорения разработки интерпретаторов DSL за счет интеграции LLM в процесс генерации кода обхода деревьев разбора.

5.1 Итоги работы

- Реализована модель `GrammarObject`, поддерживающая динамическое обновление правил языка «на лету» без пересборки системы.

- Формализован алгоритм синтеза Visitor-хендлеров на языке Python с использованием контекстных подсказок для LLM.
- Создан рабочий прототип среды исполнения (Language Workbench) в контейнеризированном виде.

5.2 Перспективы развития

Дальнейшие исследования будут сосредоточены на количественной оценке влияния пользовательских аннотаций на точность генерации семантики (метрика Unparsed Rate). Также планируется внедрение локальных квантованных моделей семейства Qwen2.5 для обеспечения полной автономности среды и снижения задержек при генерации кода.

6 Источники

[1] Itemis. Large Language Models for Domain-Specific Language Generation. — Access mode: https://medium.com/itemis/large-language-models-for-domain-specific-language-generation-how-to-train-your-d

[2] TidalCycles. — Access mode: https://tidalcycles.org.

[3] Prokopič Martin. Architecture and Implementation of DSL Interpreters : Master's thesis ; CTU Prague. — 2023.

[4] Gane A. et al. Grammar-Constrained Decoding for Large Language Models // arXiv preprint. — 2023.

[5] Kuipers T., Visser J. A Case of Visitor versus Interpreter Pattern // CWI. — Access mode: https://homepages.cwi.nl/~storm/publications/visitor.pdf.

[6] Kuipers T., Visser J. A Case of Visitor versus Interpreter Pattern // ResearchGate. — Access mode: https://www.researchgate.net/publication/220878226_A_Case_of_Visitor_versus_Interpreter_Pattern.

[7] Rompf Tiark et al. Surgical Precision JIT Compilers // PLDI. — Purdue University.

[8] Zucker Philip. Compiling With Constraints. — Access mode: https://www.philipzucker.com/.

[9] Title of the paper 2503.22587 // arXiv preprint arXiv:2503.22587. — 2025. — Access mode: https://arxiv.org/html/2503.22587v1.

[10] Title of the paper 2503.05449 // arXiv preprint arXiv:2503.05449. — 2025. — Access mode: https://arxiv.org/pdf/2503.05449.

[11] Comparing approaches for robust NL to DSL generation // arXiv preprint arXiv:2408.08335. — 2024. — Access mode: https://arxiv.org/html/2408.08335v1 (online; accessed: 2025-12-28).

[12] VxRL. Enhancing LLM Code Generation with RAG and AST-Based Chunking. — Access mode: https://vxrl.medium.com/enhancing-llm-code-generation-with-rag-and-ast-based-chunking-5b81902ae9fc (online; accessed: 2025-12-28).

[13] A Comparative Study of DSL Code Generation: Fine-Tuning // arXiv preprint arXiv:2407.02742. — 2024. — Access mode: https://arxiv.org/html/2407.02742v1 (online; accessed: 2025-12-28).

[14] Grammar Prompting for Domain-Specific Language Generation // NeurIPS. — 2023. — Access mode: https://proceedings.neurips.cc/paper_files/paper/2023/file/cd40d0d65bfebb894ccc9ea822b47fa8-Paper-Conference.pdf (online; accessed: 2025-12-28).

[15] Guiding Large Language Models to Generate Computer-Parsable Code // arXiv preprint arXiv:2404.05499. — 2024. — Access mode: https://arxiv.org/pdf/2404.05499 (online; accessed: 2025-12-28).

[16] An LLM-based Semantic Enhancement Framework for POI // arXiv preprint arXiv:2502.10038. — 2025. — Access mode: https://arxiv.org/html/2502.10038v2 (online; accessed: 2025-12-28).

[17] NSA: Neuro-symbolic ARC Challenge // arXiv preprint arXiv:2501.04424. — 2025. — Access mode: https://arxiv.org/html/2501.04424v1 (online; accessed: 2025-12-28).