Глава 3. Программная реализация инструментальной среды DslTools

3.1 Введение в реализацию Unified Executable Parsing Environment (UEPE)

Разработка современных предметно-ориентированных языков (DSL) требует перехода от статических компиляторных цепочек к динамическим средам исполнения, обеспечивающим мгновенную обратную связь. В рамках проекта DslTools была реализована концепция Unified Executable Parsing Environment (UEPE) — унифицированной исполняемой среды парсинга. Данная глава посвящена детальному описанию программной архитектуры, сценариев использования, пользовательского интерфейса и алгоритмического ядра системы. Особое внимание уделяется реализации механизмов семантического анализа на базе паттерна Visitor, адаптированного для трансляции псевдокода (PSECO) в издательскую систему LaTeX.

Реализация системы базируется на гибридном подходе, сочетающем детерминированные алгоритмы синтаксического анализа (RBNF) с гибкой архитектурой интерпретации, позволяющей модифицировать грамматику и семантику языка «на лету» (Hot Reload).

3.2 Основные сценарии использования

Проектирование взаимодействия с системой DslTools строилось на основе анализа ролевой модели, выделенной в ходе предварительного этапа исследования. Архитектура системы поддерживает работу четырех ключевых акторов, каждый из которых взаимодействует с платформой на различном уровне абстракции: от конечного пользователя до архитектора ядра. Жизненный цикл работы в системе не является линейным, а представляет собой итеративный процесс, замыкающий цикл «разработка — верификация — исполнение».

3.2.1 Сценарий разработчика DSL (Creator)

Роль разработчика (Creator) является центральной в экосистеме DslTools. Традиционный процесс создания языка программирования сопряжен с длительной цепочкой действий: правка грамматики (например, в Yacc/Bison), генерация исходного кода парсера, компиляция проекта, сборка бинарных файлов и запуск тестов. В DslTools этот цикл радикально оптимизирован.

Жизненный цикл разработчика строится следующим образом:

1. **Формализация синтаксиса:** Разработчик взаимодействует с панелью

мета-моделирования, вводя правила грамматики в формате EBNF (Extended Backus-Naur Form). Система осуществляет валидацию синтаксиса EBNF в реальном времени, подсвечивая структурные ошибки (например, использование неопределенных нетерминалов).

2. **Автоматическая сборка (Hot Reload):** Ключевой особенностью является отсутствие явного этапа компиляции среды. При сохранении изменений (или по срабатыванию таймера простоя, debounce) система инициирует фоновый процесс пересборки графа интерпретатора. В отличие от классических компиляторов, DslTools не создает физические исполняемые файлы на диске для каждой итерации. Вместо этого происходит обновление графа состояний GrammarObject в памяти активного контейнера.
3. **Верификация через визуализацию:** Немедленно после пересборки обновляется интерактивное окно визуализации CST (Concrete Syntax Tree) и AST (Abstract Syntax Tree). Разработчик может загрузить тестовый пример кода (например, алгоритм Евклида [1]) и наблюдать, как изменение в правиле (например, изменение приоритета операторов или введение новой конструкции LOOP) мгновенно отражается на иерархической структуре дерева. Это позволяет выявлять логические ошибки в грамматике, такие как левая рекурсия или неоднозначность разбора, на самых ранних этапах.[1]

3.2.2 Сценарий пользователя DSL (TargetUser)

Для конечного пользователя (TargetUser) система DslTools выступает как интегрированная среда разработки (IDE) для конкретного DSL. Пользователь абстрагирован от внутренней сложности генерации парсеров.

Поток выполнения задач пользователем:

1. **Инициализация среды:** При входе в систему пользователь выбирает целевой язык из списка доступных (например, SimpleMathDSL или PSECO). Система подгружает соответствующий конфигурационный профиль, включающий лексер, парсер и набор визиторов.
2. **Написание прикладного кода:** Пользователь работает в редакторе кода с поддержкой синтаксической подсветки, специфичной для выбранного DSL.
3. **Исполнение (RunDSL):** По команде "Run" инициируется выполнение программы. Запрос передается на Backend, где происходит интерпретация AST. Результат выполнения (консольный вывод, сгенерированный файл или график) возвращается на клиент. Важно отметить, что для пользователя этот процесс выглядит как запуск обычного скрипта, несмотря на то, что интерпретатор мог быть обновлен разработчиком секунду назад.[1]

3.2.3 Сценарий эксперта (Strong Creator)

Роль Strong Creator предъявляет наиболее жесткие требования к архитектуре, требуя

возможности вмешательства в работу конвейера (Pipeline) обработки кода на низком уровне. Это выходит за рамки декларативного описания грамматик.

Модификация алгоритмов ядра:

- **Подмена токенизатора:** Если стандартный регулярный лексер (RegexTokenizer) не справляется со сложной логикой (например, вложенные комментарии специфического формата или контекстно-зависимые лексемы, как в Python), эксперт может инъецировать собственный модуль токенизации, реализующий интерфейс ITokenizer.
- **Кастомизация построения AST:** Эксперт может переопределить логику трансляции CST в AST (IASTBuilder), внедряя специфические оптимизации или дополнительные проверки типов на этапе построения дерева.
- **Управление инстанцированием:** Система DslTools позволяет эксперту настраивать стратегию жизненного цикла интерпретатора — от создания нового изолята на каждый запрос (Stateless) до поддержания персистентного состояния сессии (Stateful).

В таблице 3.1 представлено сравнение возможностей различных ролей в системе.

Возможность	**BaseUser**	**TargetUser**	**Creator**	**Strong Creator**
Запуск программ DSL	+	+	+	+
Просмотр результатов	+	+	+	+
Редактирование EBNF	-	-	+	+
Настройка Visitor	-	-	+	+
Визуализация AST	-	-	+	+
Модификация ядра (Core)	-	-	-	+

Доступ к Docker-конфигам	-	-	-	+

*Таблица 3.1: Матрица доступа к функциональным возможностям системы DslTools *

3.3 Архитектура приложения

Архитектура DslTools спроектирована как модульная распределенная система, следующая принципам слабой связности (loose coupling) и высокой когезии (high cohesion). Это необходимое условие для реализации требований экспертной модификации ядра и бесперебойной работы механизма Hot Reload. В основе архитектуры лежит паттерн «Конвейер» (Pipeline), где каждый этап обработки данных инкапсулирован в отдельный компонент, скрытый за абстрактным интерфейсом.

3.3.1 Модульная структура конвейера (Pipeline)

Обработка исходного кода на DSL проходит через серию трансформаций, каждая из которых выполняется независимым модулем. Абстракция через интерфейсы позволяет заменять реализации модулей без остановки системы.

1. Лексический анализ (ITokenizer):
   Первый этап конвейера отвечает за преобразование потока символов в поток токенов. Реализация по умолчанию (RegexTokenizer) использует скомпилированные регулярные выражения, извлекаемые из блока TERMINALS грамматики. Однако архитектура допускает подключение рукописных лексеров для языков со сложной лексикой (например, поддерживающих интерполяцию строк). Токенизатор генерирует объекты Token, содержащие тип, значение и позицию в исходном тексте для детальной диагностики ошибок.1
2. Синтаксический анализ (IParser):
   Модуль парсинга преобразует поток токенов в Конкретное Синтаксическое Дерево (CST). Используется динамический парсер (RBNFParser), который конфигурируется объектом GrammarObject во время выполнения. Это позволяет системе поддерживать произвольные КС-грамматики без перекомпиляции самого ядра парсера. Важной особенностью является использование алгоритмов, устойчивых к ошибкам, что позволяет строить частичные деревья даже при наличии синтаксических ошибок в коде пользователя.1
3. Семантический анализ и трансляция (IASTBuilder):
   На этом этапе «сырое» CST очищается от синтаксического шума (скобок, запятых, ключевых слов) и преобразуется в Абстрактное Синтаксическое Дерево (AST). Здесь происходит связывание идентификаторов (Name Resolution) и первичная

валидация типов. Результатом работы является структура данных, готовая к обходу паттерном Visitor. Интерфейс IASTBuilder позволяет Strong Creator-у внедрять кастомные стратегии свертки дерева.1

3.3.2 Сущность GrammarObject

Центральным элементом, связывающим декларативное описание языка (EBNF) с исполняемым кодом, является GrammarObject. Анализ файла parse.py показывает, что это не пассивная структура данных, а активный объект-фабрика.

GrammarObject инкапсулирует:

- **Словарь терминалов (terminals):** Отображение имен токенов на скомпилированные объекты регулярных выражений (re.compile), что критично для производительности лексического анализа.
- **Граф правил (rules):** Направленный граф, представляющий структуру нетерминалов. Каждое правило (Rule) содержит левую часть (имя нетерминала) и правую часть (RuleElement), которая может быть последовательностью, альтернативой, итерацией или опциональным блоком.
- **Метаданные:** Версия языка, имя модуля, настройки генерации, извлекаемые из блока METABLOCK грамматики.
- **Таблица символов (symbol_table):** Хранилище глобальных определений, доступных в рамках грамматики.

При изменении EBNF-описания система создает новый экземпляр GrammarObject, валидирует его (проверка на недостижимые символы, циклические зависимости) и только при успешной валидации атомарно подменяет ссылку в активном парсере. Это обеспечивает транзакционную целостность обновления языка.

3.3.3 Механизм Hot Reload и стратегия инстанцирования

Реализация «горячей» перезагрузки (Hot Reload) является критической для сценария быстрой итерации разработки (Edit-Run). Механизм работает следующим образом:

1. **Мониторинг (Watchdog):** Фоновый сервис отслеживает изменения в файлах грамматики (.rbnf) и скриптах визиторов (.py).
2. **Дебаунсинг:** Для предотвращения избыточных пересборок при активном наборе текста используется таймер задержки.
3. **Изолированное пересоздание (Isolated Re-instantiation):** При обнаружении изменений система не пытается "допатчить" существующий интерпретатор. Вместо этого применяется стратегия полного сброса контекста:
    - Создается новый EvalContext.
    - Инстанцируется новый набор визиторов, зарегистрированных в EvalRegistry.
    - Пересоздается GrammarObject.
      Это гарантирует «чистоту» состояния (State Purity), исключая побочные

эффекты от предыдущих версий грамматики (например, остаточные данные в статических переменных или кэшах).

3.3.4 Контейнеризация и инфраструктура

Для обеспечения безопасности, масштабируемости и переносимости, DslTools развертывается с использованием технологии контейнеризации Docker. Архитектура разделена на микросервисы:

- **UI Container:** Содержит статику веб-интерфейса (HTML/CSS/JS) и легковесный веб-сервер Nginx. Этот контейнер полностью изолирован от логики исполнения, что гарантирует доступность интерфейса даже при падении вычислительного ядра (например, из-за бесконечной рекурсии в пользовательском DSL).
- **Core Container (Backend):** Содержит среду выполнения Python, библиотеки парсинга (DSLTools), компиляторы и механизмы Hot Reload. Взаимодействие между UI и Core осуществляется через REST API.
- **Localhost Deployment:** Для разработки и отладки самой системы предусмотрен режим запуска на localhost, где контейнеры поднимаются через Docker Compose, обеспечивая единое сетевое пространство для всех компонентов.

3.4 Пользовательский интерфейс

Интерфейс системы DslTools реализован в парадигме «Language Workbench» (верстак языкового инженера), объединяя инструменты разработки метамодели и прикладного программирования в едином оконном пространстве. Дизайн интерфейса вдохновлен решениями Microsoft ScriptBook, обеспечивая интуитивно понятное взаимодействие.

3.4.1 Функциональные зоны

Интерфейс четко зонирован для поддержки различных этапов работы:

1. Панель мета-моделирования (Meta-modeling Pane):
   Расположена, как правило, в левой части экрана (или на отдельной вкладке в режиме Creator). Представляет собой редактор кода с подсветкой синтаксиса для EBNF/RBNF. Редактор поддерживает автодополнение для ключевых слов грамматики (TERMINAL, RULES, AXIOM) и валидацию структуры правил «на лету».
2. Панель прикладного программирования (Target Code Editor):
   Центральная рабочая область для пользователя DSL (TargetUser). Здесь пишется код на целевом языке (например, PSECO). Редактор динамически подгружает правила подсветки синтаксиса, сгенерированные на основе текущей грамматики, что является уникальной фичей DslTools.
3. Интерактивное окно визуализации (Visualization Pane):
   Критически важный инструмент для отладки. Здесь отображается графическое представление CST и AST.

- **CST View:** Показывает полное дерево разбора со всеми токенами.
- AST View: Показывает упрощенное дерево, готовое к интерпретации.
  Узлы дерева интерактивны: при клике на узел подсвечивается соответствующий участок кода в редакторе (двусторонний маппинг).

4. Панель диагностических сообщений и логов (Log/Diagnostics Pane):
   Нижняя панель, выводящая результаты сборки интерпретатора, ошибки парсинга и вывод (stdout) исполняемой программы. В режиме Strong Creator здесь также отображаются внутренние логи работы конвейера (трейсы токенизатора и переходов автомата парсера).

3.5 Пример задания грамматики целевого языка (PSECO)

Для демонстрации возможностей системы была выбрана грамматика языка PSECO (Pseudocode for Educational Computational Operations). Этот язык предназначен для описания алгоритмов и их последующей трансляции в LaTeX.

3.5.1 Листинг грамматики (EBNF)

Ниже представлено полное описание грамматики PSECO в формате RBNF, используемом в DslTools. Описание включает определение терминалов, ключевых слов и правил вывода.[1]

EBNF

```
# Описание метаданных языка
METABLOCK {
  name = "PSECO"
  version = "1.0"
}

# Определение лексических токенов (Терминалов)
TERMINALS:
  name      ::= '[A-Za-z_][A-Za-z0-9_]*';
  number    ::= '[0-9]+';
  string    ::= '"[^"]*"';
  whitespace ::= '\s+';
  # Специальные символы для операторов
```

```
  operator   ::= '[:=\#;()\{\}\[\],.<>+\-*\/]';

# Ключевые слова
KEYS:
  'IF'; 'THEN'; 'ELSE'; 'END_IF';
  'WHILE'; 'DO'; 'END_WHILE';
  'INPUT'; 'OUTPUT';
  'gets'; 'neq'; 'mod'; # Текстовые операторы
  '('; ')'; ';';

# Список нетерминалов
NONTERMINALS:
  Program; Block; Statement;
  Assignment; Conditional; Loop;
  Input; Output; Expression;
  Variable;

# Аксиома грамматики (точка входа)
AXIOM:
  Program

# Правила вывода (Синтаксис)
RULES:
  # Программа состоит из имени и тела (блока инструкций)
  Program ::= name '{' Block '}';

  # Блок - это последовательность инструкций
  Block ::= { Statement };

  # Инструкция может быть присваиванием, условием, циклом или вводом/выводом
  Statement ::= ( Assignment | Conditional | Loop | Input | Output ) ';';

  # Присваивание: переменная gets выражение
  Assignment ::= Variable 'gets' Expression;

  # Условный оператор
  Conditional ::= 'IF' Expression 'THEN' Block 'END_IF';

  # Цикл While
  Loop ::= 'WHILE' Expression 'DO' Block 'END_WHILE';

  # Ввод и Вывод
  Input ::= 'INPUT' Variable;
```

```
  Output ::= 'OUTPUT' Expression;

  # Выражения (упрощенно для примера)
  Expression ::= Variable | number | Variable 'mod' Variable | Variable 'neq' number | Variable '>' Variable;

  # Переменная - это идентификатор
  Variable ::= name;
```

3.5.2 Пример сгенерированного дерева разбора (Концептуальная модель)

Рассмотрим, как система строит дерево разбора для правила Program. При обработке входного текста парсер DslTools создает иерархическую структуру объектов ASTNode. Для входной строки EuclidAlgo {... }:

1. Корневой узел типа Program.
2. Первый дочерний узел: Терминал name со значением "EuclidAlgo".
3. Второй дочерний узел: Токен {.
4. Третий дочерний узел: Нетерминал Block (содержащий список Statement).
5. Четвертый дочерний узел: Токен }.

Эта структура полностью соответствует описанию в правиле Program ::= name '{' Block '}'. Важно отметить, что в AST (в отличие от CST) токены {, } и ; могут быть отброшены визитором, если они не несут семантической нагрузки для дальнейшей трансляции, оставляя только значащие узлы: Имя программы и список инструкций блока.

3.5.3 Описание применения паттерна Visitor для трансляции

Для трансляции полученного дерева PSECO в целевой формат (LaTeX) используется паттерн Visitor в его модификации Attribute Evaluation, принятой в DslTools.
Вместо монолитного класса-визитора с методами visitProgram, visitBlock, архитектура DslTools предлагает создавать отдельные классы-оценщики (Evaluator) для каждого типа узла AST. Каждый такой класс реализует интерфейс ASTNode.IAttrEval и метод __call__.1
Логика работы:

- Для узла Program визитор создает окружение документа LaTeX (\begin{algorithm}...).
- Для узла Loop визитор генерирует конструкцию \While{$condition$}... \EndWhile.
- Для узла Assignment визитор преобразует оператор gets в стрелку \gets LaTeX.

Регистрация визиторов происходит через центральный реестр EvalRegistry, который связывает строковое имя нетерминала (например, "Assignment") с экземпляром соответствующего класса-оценщика. Это позволяет подменять логику генерации для отдельных узлов без изменения всего транслятора.

3.6 Реализация паттерна Visitor (Python-код)

В данном разделе приводится полный исходный код модулей визиторов, реализующих трансляцию PSECO в LaTeX. Код представлен в виде tcblisting, демонстрирующем интеграцию с системой типов DslTools. Реализация базируется на архитектурных паттернах, выявленных в файле rbnfgo.py.[1]

Ключевые особенности реализации:

1. **Наследование от ASTNode.IAttrEval:** Все классы реализуют единый интерфейс вызова.
2. **Контекст (EvalContext):** Используется для передачи состояния (например, уровня отступа или настроек генерации), хотя в данном простом случае генерация в основном контекстно-независимая.
3. **Реестр (EvalRegistry):** Явное связывание грамматических правил с кодом Python.
4. **Обработка операторов:** Словарь LATEX_OPS маппит текстовые операторы PSECO (gets, neq) в математические символы LaTeX (\gets, \neq).

\begin{tcblisting}{
colback=gray!5,
colframe=gray!50,
listing only,
title=\textbf{Листинг 3.1: Реализация Visitor для трансляции PSECO в LaTeX},
fonttitle=\bfseries,
listing options={
language=Python,
basicstyle=\ttfamily\small,
keywordstyle=\color{blue}\bfseries,
commentstyle=\color{green!60!black},
stringstyle=\color{red},
showstringspaces=false,
breaklines=true,
tabsize=4,
numbers=left
}
}
import logging
from typing import List, Any, Dict

Импорт базовых моделей из ядра

DslTools

from DSLTools.models.ast import EvalRegistry, EvalContext
from DSLTools.models import ASTNode

--- Карта отображения операторов PSECO в LaTeX ---

LATEX_OPS: Dict[str, str] = {
"gets": r"\gets", # Присваивание
"neq": r"\neq", # Не равно
"eq": r"=",
"gt": r">",
"lt": r"<",
"mod": r"\bmod", # Деление по модулю
"and": r"\land",
"or": r"\lor"
}
class ProgramEval(ASTNode.IAttrEval):
"""
Визитор для корневого узла 'Program'.
Оборачивает код в окружение algorithm.
Синтаксис: Program ::= name '{' Block '}'
"""
def call(self, value: str, children: List, context: Any) -> str:
# children - name (Terminal)
# children1 - '{'
# 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

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)
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: Variable
# children1: 'gets' (Token)
# children: Expression




    lhs = children.evaluated(context)

    # Получаем LaTeX-символ для оператора из карты
    op_token = children.value
    op_latex = LATEX_OPS.get(op_token, r"\gets")

    rhs = children.evaluated(context)

    # Оборачиваем в \State и математический режим $...$
    return f"\\State ${lhs} {op_latex} {rhs}$"

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.evaluated(context)
    if_body = children.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.value == 'ELSE':
        else_body = children.evaluated(context)
        latex += f"\\Else\n    {else_body}\n"

    latex += r"\EndIf"
    return latex

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.1evaluated(context)
body = children.evaluated(context)

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

```
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)

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.evaluated(context)
return value
```

--- Регистрация визиторов в системе DslTools ---

```
def register_pseco_translators():
"""
Функция регистрации, вызываемая ядром при загрузке языка.
```

Связывает имена нетерминалов RBNF с классами визиторов.
"""
EvalRegistry.clear() # Очистка старых обработчиков

# Регистрация структурных блоков
EvalRegistry.register(ASTNode.Type.NONTERMINAL, "Program", ProgramEval())
EvalRegistry.register(ASTNode.Type.NONTERMINAL, "Block", SequenceEval())

# Регистрация управляющих конструкций
EvalRegistry.register(ASTNode.Type.NONTERMINAL, "Assignment", AssignmentEval())
EvalRegistry.register(ASTNode.Type.NONTERMINAL, "Conditional", ConditionalEval())
EvalRegistry.register(ASTNode.Type.NONTERMINAL, "Loop", LoopEval())

# Регистрация выражений и ввода/вывода
EvalRegistry.register(ASTNode.Type.NONTERMINAL, "Expression", ExpressionEval())
EvalRegistry.register(ASTNode.Type.NONTERMINAL, "Variable", ElementEval())

# Регистрация токенов (если они приходят как отдельные узлы)
EvalRegistry.register(ASTNode.Type.TOKEN, "name", ElementEval())
EvalRegistry.register(ASTNode.Type.TOKEN, "number", ElementEval())

# Регистрация операторов
for op in LATEX_OPS.keys():
    EvalRegistry.register(ASTNode.Type.TOKEN, op, ExpressionEval())

if name == "main":
register_pseco_translators()
print("PSECO-LaTeX Translators registered successfully.")
\end{tcblisting}

3.7 Примеры программ на целевом языке

В качестве эталонного примера для демонстрации работы транслятора выбран алгоритм Евклида для нахождения наибольшего общего делителя (НОД). Пример иллюстрирует использование циклов WHILE, условных переходов IF-ELSE и арифметических операций по модулю.

3.7.1 Исходный код (test.txt)

Ниже приведен листинг программы на языке PSECO, содержащийся в файле test.txt.[1]

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

3.7.2 Дерево разбора и результат трансляции

При обработке данного кода парсер строит AST, где корневым узлом является Block (или Statement list, в зависимости от того, обернут ли код в Program). Узел WHILE содержит условие a neq 0 & b neq 0 и тело цикла. Тело цикла содержит узел IF, который ветвится на две ветки Assignment.

После применения разработанных визиторов (Листинг 3.1) система генерирует следующий LaTeX-код:

Фрагмент кода

```
\begin{algorithm}
\caption{Euclid}
\begin{algorithmic}
\While{$a \neq 0 \land b \neq 0$}
   \If{$a > b$}
      \State $a \gets a \bmod b$
   \Else
      \State $b \gets b \bmod a$
   \EndIf
\EndWhile
\end{algorithmic}
\end{algorithm}
```

Визуальный результат рендеринга этого кода в PDF-документе будет выглядеть как стандартный, профессионально сверстанный алгоритм, что подтверждает корректность работы реализованной подсистемы трансляции. Использование \gets вместо текстового gets и \bmod вместо mod обеспечивает соответствие стандартам математической нотации.

3.8 Заключение

Реализованный модуль трансляции для языка PSECO демонстрирует эффективность архитектурного подхода DslTools. Использование реестра визиторов (EvalRegistry) позволило полностью отделить логику генерации LaTeX от описания грамматики, обеспечив модульность системы. Механизм Hot Reload позволяет вносить изменения в логику визиторов (например, изменить форматирование \If на \elf) без перезагрузки всей среды, что критически важно для эффективной работы эксперта (Strong Creator). Контейнеризация компонентов гарантирует стабильность работы пользовательского интерфейса даже при ошибках в логике трансляции.

Источники

1. Lab3.tex