- Промт: Генерация главы программной реализации.

Глава 4. Программная реализация системы DslTools

4.1. Введение

В данной главе рассматривается техническая реализация системы DslTools — программного комплекса, спроектированного как Unified Executable Parsing Environment (UEPE). Система предназначена для обеспечения полного цикла разработки предметно-ориентированных языков (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 с минут до миллисекунд, предоставляя разработчику немедленную обратную связь.

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.

4.2. Основные сценарии использования

Анализ функциональных требований [1] позволил выделить ключевые сценарии взаимодействия с системой, которые определяют архитектурный облик приложения. Жизненный цикл работы в UEPE не является линейным; он представляет собой набор взаимосвязанных процессов, инициируемых различными акторами. В основе проектирования лежит принцип бесшовности (seamlessness) — минимизации когнитивной и технической нагрузки при переключении между контекстами разработки грамматики и исполнения кода.

Сценарий разработчика (Creator): Реактивный цикл проектирования

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:

1. **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.
2. **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.[1]
3. **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.

Этот сценарий требует от системы высокой реактивности. Любая задержка свыше 100-200 мс нарушает ощущение "прямого манипулирования" языком.

Сценарий пользователя (TargetUser): Прозрачное исполнение

Для конечного пользователя (TargetUser) система DslTools выступает как "черный ящик" или IDE для конкретного DSL. Пользователь не должен знать о внутренней сложности генерации парсеров.

Сценарий работы:

1. **Написание кода:** Пользователь работает в редакторе, который предоставляет подсветку синтаксиса, динамически сгенерированную на основе текущей версии грамматики.
2. **Запуск (Execution):** По команде "RunDSL" [1] инициируется выполнение программы. Система скрывает этап инстанцирования интерпретатора. Для пользователя процесс выглядит как запуск скрипта: ввод данных -> нажатие кнопки -> получение результата в консоли вывода.
3. **Изоляция:** Критически важно, чтобы ошибки в пользовательском коде (например, бесконечный цикл) не приводили к падению всей среды разработки.

Сценарий эксперта (Strong Creator): Глубинная модификация ядра

Роль Strong Creator [1] 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.

1. **Подготовка модуля:** Эксперт пишет код кастомного токенизатора, реализующего интерфейс ITokenizer.
2. **Инъекция зависимости:** Через интерфейс настройки ядра эксперт загружает новый модуль.
3. **Горячая подмена:** Система DslTools должна корректно выгрузить стандартный модуль, загрузить новый, связать его с текущим парсером и перезапустить конвейер обработки, не разрывая пользовательскую сессию.

Ниже приведена таблица, сравнивающая требования к системе со стороны различных акторов, влияющие на программную реализацию.

Актор	**Ключевая потребность**	**Архитектурное решение**	**Влияние на Hot Reload**
Creator	Скорость итераций	In-memory генерация парсеров	Триггер на изменение EBNF
TargetUser	Стабильность исполнения	Изоляция инстансов интерпретатора	Прозрачный перезапуск сессии
StrongCreator	Гибкость алгоритмов	Интерфейсная абстракция (IoC)	Динамическая перелинковка DLL/модулей

4.3. Архитектура приложения

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

Модульная структура и абстракция конвейера

Обработка исходного кода на DSL проходит через серию трансформаций. Чтобы обеспечить возможность замены алгоритмов, каждый этап определяется не конкретной реализацией, а контрактом (интерфейсом).

1. **Лексический анализ (Tokenizer):**
   - *Интерфейс:* ITokenizer

    - *Функция:* Преобразует поток символов в поток токенов.
    - *Реализация по умолчанию:* RegexTokenizer, использующий скомпилированные регулярные выражения из блока TERMINALS грамматики.[1]
    - *Обоснование:* Использование интерфейса позволяет Strong Creator заменить regex-движок на конечный автомат (DFA) или рукописный лексер для оптимизации производительности или поддержки контекстно-зависимой лексики.
2. **Синтаксический анализ (Parser/CSTCreator):**
    - *Интерфейс:* IParser
    - *Функция:* Преобразует поток токенов в Конкретное Синтаксическое Дерево (CST), проверяя соответствие структуре правил (RULES).
    - *Реализация:* Динамический парсер (например, Recursive Descent или Packrat), конфигурируемый объектом грамматики на лету.
3. **Семантический анализ и трансляция (AST Translator):**
    - *Интерфейс:* IASTBuilder
    - *Функция:* Очищает CST от синтаксического шума (скобок, запятых, ключевых слов) и формирует Абстрактное Синтаксическое Дерево (AST), пригодное для исполнения паттерном Visitor.

Сущность GrammarObject

Центральным элементом, связывающим декларативное описание языка с исполняемым кодом, является GrammarObject. Анализ структуры файла описания грамматик (parse.py и pseco.rbnf [1]) показывает, что GrammarObject — это не просто структура данных, а активная сущность.

GrammarObject инкапсулирует:

- **Словарь терминалов:** Отображение имен токенов на скомпилированные объекты регулярных выражений (re.compile). Это критически важно для производительности, так как компиляция regex является дорогой операцией.
- **Граф правил:** Направленный граф, где узлами являются нетерминалы, а ребрами — отношения вложенности. Этот граф строится при инициализации GrammarObject и позволяет выявлять левую рекурсию и недостижимые правила.
- **Метаданные:** Версия языка, имя модуля, настройки генерации, извлекаемые из METABLOCK.

При изменении EBNF система не модифицирует существующий GrammarObject, а создает новый. Это обеспечивает атомарность изменений: система либо работает со старой версией грамматики, либо с новой, но никогда с промежуточным, несогласованным состоянием.

Механизм Hot Reload

Hot Reload (горячая перезагрузка) является ключевой функциональной возможностью

UEPE. Его реализация выходит за рамки простого перезапуска процесса и затрагивает управление памятью и состоянием приложения.

Алгоритм работы Hot Reload:

1. **Отслеживание изменений (Watchdog):** В контейнере Backend запущен фоновый сервис, подписывающийся на события файловой системы (file system events) в директории проекта.
2. **Событие изменения:** При детектировании изменения файла .rbnf или скомпилированного модуля ядра запускается таймер "дребезга" (debounce), чтобы предотвратить многократные пересборки при частых сохранениях.
3. **Пересборка контекста:**
    - Система парсит новый файл грамматики.
    - В случае успеха создается новый экземпляр GrammarObject.
    - Фабрика парсеров (ParserFactory) получает новый GrammarObject и подготавливает новый инстанс Pipeline.
4. **Атомарная подмена:**
    - Глобальный указатель на текущий активный интерпретатор блокируется мьютексом.
    - Ссылка подменяется на новый инстанс.
    - Мьютекс освобождается.
    - Старый инстанс помечается для сборки мусора (Garbage Collection).

Стратегия инстанцирования и чистота состояния

При реализации среды исполнения DSL критически важно обеспечить предсказуемость работы программы. В связи с этим принята стратегия **Isolated Re-instantiation** (изолированного пересоздания).

При каждом событии Reload (вызванном пользователем или системой) происходит полный сброс контекста исполнения.

- **Память:** Все переменные, объявленные в ходе выполнения предыдущей версии программы DSL (например, значения переменных $a, $b из примера [1]), уничтожаются.
- **Visitor:** Инстанс Visitor, хранящий состояние обхода дерева, пересоздается заново.

Это гарантирует, что побочные эффекты от выполнения кода на старой версии грамматики не повлияют на поведение кода в новой версии. Для пользователя это выглядит как "чистый старт" при каждом запуске.

Контейнеризация и инфраструктура

Для обеспечения безопасности и переносимости система DslTools развертывается с использованием технологии контейнеризации Docker.[1] Архитектура разделена на

независимые сервисы.

1. **UI Container:** Содержит статику веб-интерфейса (React/Vue/ScriptBook implementation) и легковесный веб-сервер (Nginx). Этот контейнер полностью изолирован от логики исполнения. Падение ядра не влияет на доступность интерфейса.
2. **Core Container (Backend):** Содержит среду выполнения (Python/Node.js), библиотеки парсинга и механизмы Hot Reload. Именно здесь живут инстансы Pipeline.
3. **Communication:** Взаимодействие между UI и Core происходит через REST API или WebSocket (для потоковой передачи логов), что позволяет разнести их даже на разные физические машины при необходимости.

Localhost Deployment: Для разработки система поставляется в виде docker-compose конфигурации, поднимающей всю среду на локальной машине (Localhost). Это позволяет исследователям использовать инструмент без доступа к интернету, имея при этом полностью настроенное окружение, идентичное продакшн-версии.

4.4. Пользовательский интерфейс

Пользовательский интерфейс (UI) DslTools реализован в парадигме "Language Workbench", что подразумевает интеграцию инструментов определения языка и инструментов его использования в едином рабочем пространстве. Визуальный дизайн базируется на принципах, заложенных в Microsoft ScriptBook [1], адаптированных для задач языкового инжиниринга.

Логика построения UI строится вокруг одновременного отображения причины (грамматики) и следствия (поведения программы). Интерфейс разделен на четыре основные функциональные зоны.

Панель мета-моделирования

Эта зона предназначена для актора Creator. Она представляет собой специализированный редактор кода для EBNF.

- *Функциональность:* Подсветка синтаксиса мета-языка (выделение TERMINALS, RULES), автодополнение имен нетерминалов, валидация ошибок "на лету".
- *Связь с ядром:* Каждое сохранение в этом редакторе триггерит отправку содержимого на Backend для обновления GrammarObject.

Панель прикладного программирования

Зона для актора TargetUser. Здесь осуществляется написание кода на целевом DSL.

- *Особенность:* Подсветка синтаксиса в этом редакторе является динамической. Она не зашита в код UI, а генерируется на основе метаданных, получаемых от

GrammarObject. Если разработчик добавит ключевое слово UNTIL, оно автоматически начнет подсвечиваться в этом редакторе без обновления страницы.

Интерактивное окно визуализации

Хотя в базовом дизайне [1] эта часть может быть представлена упрощенно, в программной реализации это полноценный интерактивный холст.

- *CST/AST:* Визуализация деревьев реализуется через графовые библиотеки (например, D3.js). Узлы дерева интерактивны: клик по узлу подсвечивает соответствующий участок кода в редакторе (bi-directional mapping).
- *Графы правил:* Визуализация графа зависимостей нетерминалов помогает Strong Creator оптимизировать структуру грамматики и устранять циклы.

Панель диагностических сообщений и логов

Консолидированная зона вывода информации.

- *Вкладка "Build":* Сообщения от генератора парсеров (ошибки грамматики, конфликты).
- *Вкладка "Run":* Стандартный вывод (stdout) исполняемой программы DSL.
- *Вкладка "System":* Логи Hot Reload и состояния контейнеров.

4.5. Пример задания грамматики целевого языка

Для демонстрации возможностей системы рассмотрим реализацию грамматики простого императивного языка, поддерживающего арифметические операции и управляющие конструкции, необходимые для реализации алгоритма Евклида. Формат описания соответствует спецификации pseco.rbnf.[1]

Листинг грамматики (EBNF)

Ниже приведен код грамматики, описывающий структуру программы, циклы WHILE, условные операторы IF и операции присваивания.

EBNF

```
GRAMMAR SimpleAlgorithmicLanguage {

  TERMINALS {
    NAME    ::= /[a-zA-Z_][a-zA-Z0-9_]*/ ;
    NUMBER  ::= /[0-9]+/ ;
```

```
    WS      ::= /\s+/ [skip] ;
  }

  KEYS {
     "WHILE", "DO", "END_WHILE",
     "IF", "THEN", "ELSE", "END_IF",
     "gets", "neq", "gt", "mod"
  }

  RULES {
    Program       ::= StatementList ;
    StatementList ::= { Statement } ;
    Statement     ::= ( Assignment | Loop | Conditional ) ;

    Assignment    ::= NAME "gets" Expression ;

    Loop          ::= "WHILE" Condition "DO"
                      StatementList
                      "END_WHILE" ;

    Conditional   ::= "IF" Condition "THEN"
                      StatementList

                      "END_IF" ;

    Condition     ::= Expression ( "neq" | "gt" ) Expression ;
    Expression    ::= Term { "mod" Term } ;
    Term          ::= NAME | NUMBER ;
  }

  AXIOM : Program ;
}
```

Сгенерированное дерево разбора (Rule Grammar)

На основе приведенной грамматики и входного потока токенов, модуль CSTCreator формирует иерархическую структуру данных. Для конструкции цикла (правило Loop) фрагмент текстового представления сгенерированного дерева будет выглядеть следующим образом:

```
LoopNode
├── KeyToken ("WHILE")
├── ConditionNode
```

```
│  ├── ExpressionNode (Left)
│  ├── OperatorToken ("neq")
│  └── ExpressionNode (Right)
├── KeyToken ("DO")
├── StatementListNode
│  ├── StatementNode (Assignment)
│  └──... (other statements)
└── KeyToken ("END_WHILE")
```

Это представление наглядно демонстрирует, как линейный код преобразуется в вложенную структуру, где ключевые слова служат маркерами границ блоков, а нетерминалы (ConditionNode, StatementListNode) инкапсулируют логику.

Использование паттерна Visitor

Для интерпретации построенного дерева используется поведенческий паттерн **Visitor** (Посетитель). Это архитектурное решение позволяет отделить алгоритм исполнения от структуры данных объектов дерева. В контексте DslTools это означает, что Strong Creator может изменить семантику языка (то, *как* выполняется код), не меняя парсер (то, *как* читается код).

При обходе дерева интерпретатор "посещает" каждый узел. Для узла LoopNode (описывающего цикл WHILE) логика посетителя заключается в следующем:

1. Вызвать метод visit для дочернего узла ConditionNode.
2. Если результат вычисления условия истинен, вызвать метод visit для узла тела цикла (StatementListNode).
3. Повторять шаги 1-2 до тех пор, пока условие истинно.
4. Игнорировать токены WHILE, DO, END_WHILE, так как они нужны только для синтаксиса, но не для исполнения.

Ниже приведен псевдокод реализации Visitor для данного правила:

Python

```
class InterpreterVisitor(NodeVisitor):
    def visit_LoopNode(self, node):
        # Получаем условие и тело цикла из структуры узла
        condition_node = node.children['Condition']
        body_node = node.children

        # Реализация семантики цикла WHILE
```

```
    # Мы вычисляем condition_node снова и снова на каждой итерации
    while self.visit(condition_node):
        self.visit(body_node)
```

4.6. Примеры программ на целевом языке

Для валидации реализованной грамматики и проверки корректности работы конвейера UEPE используется тестовая программа, реализующая алгоритм Евклида для нахождения наибольшего общего делителя (НОД).

Листинг программы (test.txt)

В приведенном примере используются операторы gets (присваивание), neq (не равно), gt (больше) и mod (остаток от деления), определенные в секции KEYS грамматики.[1]

WHILE $a \neq 0 & b \neq 0$ DO
IF $a > b$ THEN
$a \gets a \bmod b$
ELSE
$b \gets b \bmod a$
END_IF
END_WHILE
Примечание: В рамках текстового представления используются символы gets, neq, gt в соответствии с ASCII-ограничениями базового редактора кода.
Скорректированный под ASCII грамматику код:
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

Визуализация дерева разбора

После обработки данного кода конвейером DslTools, в окне визуализации отображается Абстрактное Синтаксическое Дерево (AST). В отличие от CST, AST очищено от ключевых слов DO, THEN, END_IF, оставляя только смысловую нагрузку.

Текстовая интерпретация визуализированного дерева для приведенной программы:

Program
└── StatementList
└── Loop (WHILE)
├── Condition: (a neq 0)

└── Body: StatementList
└── Conditional (IF)
├── Check: (a gt b)
├── Then-Block: Assignment
│ ├── Target: a
│ └── Value: (a mod b)
└── Else-Block: Assignment
├── Target: b
└── Value: (b mod a)

Данная структура подтверждает корректность вложенности оператора IF внутри цикла WHILE и правильную ассоциацию блоков THEN и ELSE с условием. Визуализация позволяет разработчику языка убедиться, что приоритет операций и границы блоков распознаны верно, завершая цикл верификации в сценарии Creator.

Источники

1. Lab3.tex