Архитектура и реализация интерпретаторов и компиляторов предметно-ориентированных языков (DSL): Глубокий анализ паттернов, структур данных и стратегий трансформации

Введение

Разработка предметно-ориентированных языков (Domain-Specific Languages, DSL) представляет собой одну из наиболее значимых и динамично развивающихся областей современной программной инженерии. В отличие от языков общего назначения (GPL), таких как Java или C++, DSL оптимизированы для решения узкого класса задач, предоставляя экспертам предметной области выразительные средства, абстрагирующие сложность вычислений и инфраструктуры. Однако реализация эффективного, расширяемого и поддерживаемого интерпретатора или компилятора для DSL требует строгого архитектурного подхода, выходящего за рамки простейших скриптовых решений. Современные требования к языковому инструментарию диктуют необходимость не просто исполнения кода, но и обеспечения глубокой интеграции с средами разработки (IDE), поддержки рефакторинга и взаимодействия с генеративными моделями искусственного интеллекта.

Данный отчет представляет собой исчерпывающее руководство по реализации стандартного конвейера обработки DSL. Мы детально рассмотрим этапы лексического анализа, синтаксического разбора с построением дерева конкретного синтаксиса (CST), трансляции в абстрактное синтаксическое дерево (AST) и последующего исполнения или компиляции. Особое внимание будет уделено обоснованию использования CST как промежуточного представления — шага, который часто ошибочно пропускают в угоду производительности, жертвуя функциональностью инструментария. Мы также проанализируем применение паттерна Посетитель (Visitor) для обхода структур данных, а также архитектурную эволюцию интерпретатора в компилятор через внедрение модулей генерации промежуточного представления (IR), оптимизации и JIT-компиляции.

Анализ опирается на современные исследования в области теории языков программирования, паттернов проектирования и интеграции больших языковых моделей (LLM) в процессы синтеза кода, демонстрируя, как классические подходы адаптируются

к требованиям современной экосистемы разработки.[1]

Часть I: Лексический анализ и Моделирование токенов

Любая реализация языка начинается с преобразования линейного потока символов в структурированное представление. Этот процесс, традиционно называемый лексическим анализом или токенизацией, является фундаментом для всего последующего пайплайна компиляции. В контексте DSL, который часто встраивается в другие языки или требует специфической обработки доменно-зависимых литералов, этот этап приобретает особые нюансы.

1.1. Теоретические основы и реализация лексера

Модуль токенизации, или лексер, служит первым барьером между неструктурированным исходным кодом и логикой интерпретатора. Его основная задача — сгруппировать символы входного потока в значимые последовательности, называемые лексемами, и классифицировать их, создавая токены. Стандартный подход к реализации лексера базируется на теории конечных автоматов (Finite Automata). Лексер обычно реализуется как детерминированный конечный автомат (DFA), который переходит из одного состояния в другое при чтении символов входного потока.

Спецификация токенов традиционно описывается с помощью расширенной формы Бэкуса-Наура (EBNF) или регулярных выражений. Например, идентификатор в DSL может быть описан правилом [a-zA-Z_][a-zA-Z0-9_]*. Современные генераторы парсеров и лексеров часто интегрируют определение лексики и грамматики в единый файл спецификации, что упрощает поддержку целостности языка.[4] При реализации лексера критически важно соблюдать принцип "жадного" сопоставления (maximal munch). Согласно этому принципу, лексер должен захватывать максимально длинную последовательность символов, соответствующую правилу. Это необходимо для разрешения неоднозначностей, например, между оператором сравнения == и оператором присваивания =. Без применения этого принципа лексер может ошибочно интерпретировать == как два отдельных знака равенства.[6]

1.2. Проблема предпросмотра (Lookahead) и неоднозначности

Эффективные лексеры часто требуют механизма предпросмотра (lookahead) на k символов (LL(k)), чтобы разрешить неоднозначность без необходимости отката (backtracking). В сложных DSL, где контекст может менять значение символа (например, > может быть оператором "больше" или закрывающей скобкой шаблона), лексер может потребовать взаимодействия с парсером для разрешения контекстной зависимости. Современные подходы к лексическому анализу часто используют ленивую токенизацию,

где токены генерируются по требованию парсера, что позволяет оптимизировать потребление памяти, особенно при обработке больших файлов исходного кода DSL.[7]

1.3. Обработка "Trivia": Ключ к поддержке IDE

В классических учебниках по компиляторам пробелы, комментарии и переводы строк часто рассматриваются как "шум", который должен быть отброшен на этапе токенизации. Однако, для современных промышленных DSL такой подход неприемлем. Элементы, не влияющие на семантику исполнения, но важные для восприятия кода человеком, называются "Trivia".

Сохранение Trivia критически важно по двум причинам:

1. **Fidelity (Точность воспроизведения):** Инструменты автоматического форматирования (pretty-printers) и рефакторинга должны уметь восстанавливать исходный код байт-в-байт, сохраняя авторское форматирование и комментарии. Если лексер отбрасывает комментарии, любой автоматический рефакторинг приведет к их потере.
2. **Документация кода:** В DSL комментарии часто несут семантическую нагрузку для человека (например, документационные комментарии к функциям).

Архитектурно это решается двумя способами: либо включением токенов пробелов и комментариев в основной поток токенов (что усложняет грамматику парсера), либо использованием "скрытых каналов" (hidden channels). В последнем случае, токены Trivia прикрепляются к ближайшему значимому токену в виде метаданных, не мешая основному процессу синтаксического анализа, но оставаясь доступными для инструментов анализа и IDE.[8]

Часть II: Дилемма структурного представления: CST против AST

После токенизации поток лексем поступает на вход синтаксическому анализатору (парсеру). Здесь архитектор DSL сталкивается с ключевой развилкой: строить ли сразу абстрактное синтаксическое дерево (AST) или сначала сформировать дерево конкретного синтаксиса (CST, также называемое Parse Tree).

2.1. Определение и роль CST

Дерево конкретного синтаксиса (CST) — это точное и полное иерархическое представление исходного кода, где каждый узел однозначно соответствует правилу грамматики (продукции EBNF). CST включает в себя абсолютно все токены исходного кода, включая синтаксический сахар, скобки, запятые, ключевые слова, и, в случае

реализации с поддержкой Trivia, комментарии и информацию о пробелах.

Существует устойчивое заблуждение, что построение CST является избыточным шагом, замедляющим интерпретацию и потребляющим лишнюю память. Действительно, для простейшего скриптового языка, запускаемого в режиме "fire and forget", CST может показаться лишним. Однако, анализ современных требований к экосистеме языков программирования показывает, что отказ от CST является преждевременной оптимизацией, влекущей серьезные функциональные ограничения и технический долг.[10]

2.2. Обоснование необходимости построения CST

Решение строить CST перед AST обосновывается тремя фундаментальными факторами: поддержкой инструментария (Tooling), возможностями рефакторинга и разделением ответственности архитектурных слоев.

Аргумент 1: Инструментарий и IDE (Tooling Support)

Современный DSL не существует в вакууме; пользователи ожидают поддержки автодополнения, подсветки синтаксиса, навигации к определению (Go to Definition) и отображения ошибок в реальном времени.

- **Позиционирование:** CST сохраняет полную информацию о позиционировании каждого элемента (start/end offset, line/column). Это критически важно для реализации протоколов языковых серверов (LSP), которые должны точно указывать место ошибки или область применения рефакторинга.[9]
- **Устойчивость к ошибкам (Error Recovery):** При редактировании кода в IDE программа большую часть времени находится в некорректном, незавершенном состоянии. Парсеры, ориентированные на построение AST, часто требуют валидной программы для создания корректного дерева. Напротив, CST-парсеры (например, на базе алгоритмов GLR или Tree-sitter) способны строить частичные деревья (Partial Trees) для некорректного кода, вставляя узлы-заглушки или узлы ошибок (Error Nodes). Это позволяет IDE продолжать предоставлять функции навигации и автодополнения даже в сломанном коде.[13]

Аргумент 2: Рефакторинг и трансформация исходного кода

Инструменты автоматического рефакторинга (например, переименование переменных, экстракция методов или миграция API) должны модифицировать код, сохраняя авторский стиль.

- **Lossless Modification (Модификация без потерь):** Если система строит только AST, информация о конкретном синтаксисе (например, были ли использованы скобки в выражении a + b или нет, если приоритет операций позволяет их опустить) теряется. При обратной генерации кода из AST форматирование пользователя будет уничтожено и заменено стандартным. CST позволяет производить локальные трансформации дерева и сериализовать его обратно в текст с сохранением всех

нетронутых частей файла.
- **Паттерны поиска:** Инструменты рефакторинга, такие как Bowler для Python, используют CST для поиска паттернов кода и их трансформации "на лету". Они полагаются на точную структуру CST для идентификации блоков кода, подлежащих изменению, не затрагивая окружающий контекст.[8]

Аргумент 3: Разделение ответственности (Decoupling)

Грамматика языка (описывающая CST) и его семантическая модель (описывающая AST) меняются с разной скоростью и по разным причинам.

- **Изоляция изменений:** CST жестко привязан к парсеру и EBNF-грамматике. AST привязан к бизнес-логике, системе типов и механизму исполнения. Наличие CST позволяет менять синтаксический сахар (например, добавить альтернативный синтаксис для циклов), не затрагивая структуру AST и логику интерпретатора. Это создает слой абстракции, защищающий ядро интерпретатора от поверхностных изменений в синтаксисе.[14]

В таблице ниже представлен сравнительный анализ двух подходов к представлению кода, подчеркивающий различия в их назначении и содержании.

Таблица 1: Сравнительный анализ CST и AST

Характеристика	CST (Concrete Syntax Tree)	AST (Abstract Syntax Tree)
Связь с грамматикой	Полное 1:1 соответствие правилам EBNF.	Отражает логическую и семантическую структуру программы.
Содержание	Все токены: ключевые слова, скобки, запятые, пробелы (опционально).	Только значимые данные: операторы, операнды, идентификаторы.
Детализация	Высокая, избыточная (Verbose). Содержит промежуточные нетерминалы.	Компактная, оптимизированная для обхода.
Основное назначение	Парсинг, подсветка синтаксиса, форматирование,	Анализ типов, оптимизация, генерация кода, интерпретация.

	рефакторинг, IDE.	
Устойчивость к ошибкам	Высокая (может содержать error nodes и незавершенные ветви).	Низкая (обычно требует синтаксически корректной программы).
Изменяемость	Часто меняется при изменении синтаксиса языка.	Более стабильна, меняется при изменении семантики.

2.3. Концепция Enriched CST (eCST)

В современной практике, особенно при создании универсальных инструментов анализа кода, используется концепция "обогащенного" CST (eCST). Обычный CST является прямым отражением грамматики, что делает его зависимым от конкретного языка. eCST добавляет слой абстракции в виде "универсальных узлов" (universal nodes). Например, узел LoopStatement в eCST может представлять как цикл for, так и цикл while, сохраняя при этом ссылки на конкретные токены исходного кода. Это позволяет создавать языко-независимые метрики и инструменты анализа, которые работают с eCST, абстрагируясь от синтаксических различий, но сохраняя возможность точной навигации по коду. Для реализации DSL это означает, что модуль построения CST может быть не просто пассивным хранилищем результатов парсинга, но и содержать логику первичной классификации конструкций.[11]

Часть III: Семантический подъем: Трансляция CST в AST

После построения и валидации CST необходимо преобразовать его в форму, пригодную для анализа типов, оптимизации и исполнения — AST. Этот процесс часто называют "desugaring" (удаление синтаксического сахара) или семантическим подъемом (semantic lifting).

3.1. Механизм трансляции

Трансляция представляет собой рекурсивный обход CST, в ходе которого создается новая структура данных — AST. На этом этапе происходит отбрасывание вспомогательных токенов (пунктуации, ключевых слов), которые выполнили свою роль при парсинге, и упрощение структур.

- **Упрощение иерархии:** В CST выражение (a + b) может быть представлено глубокой

иерархией узлов: Expression -> Term -> Factor -> ParenExpression, содержащий левую скобку, вложенное выражение и правую скобку. При трансляции в AST эта структура схлопывается в узел BinaryExpression с оператором + и двумя операндами. Скобки игнорируются, так как приоритет операций уже зафиксирован в структуре дерева (вложенности узлов).[17]

- **Нормализация:** Различные синтаксические конструкции, имеющие одинаковую семантику, могут быть приведены к единому узлу AST. Например, синтаксический сахар until (x) {... } может быть транслирован в узел AST WhileLoop, где условие инвертировано.

3.2. Разрешение имен (Name Resolution)

Часто именно на этапе трансляции CST в AST (или сразу после него, на первом проходе по AST) реализуется модуль разрешения имен. Это процесс связывания идентификаторов (использования переменных) с их определениями (декларациями).

- **Symbol Table:** В процессе трансляции строится таблица символов, которая отслеживает области видимости (Scopes).
- **Cross-referencing:** Ссылки в AST заменяются с простых строковых имен на прямые ссылки на узлы деклараций или записи в таблице символов. Это превращает AST из дерева в граф (так как появляются перекрестные ссылки), что значительно упрощает последующие этапы проверки типов и генерации кода.[12]

3.3. Паттерны реализации мапперов

Для реализации трансляции часто используются специализированные конвертеры (Mappers). В сложных экосистемах, таких как Eclipse Modeling Framework (EMF) или языковые верстаки (Language Workbenches), CST и AST определяются через формальные мета-модели. Трансляция в этом случае описывается как декларативный набор правил отображения (Model-to-Model Transformation) между мета-моделью CST и мета-моделью AST. В более простых, "ручных" реализациях DSL, это обычно реализуется методом toAst() в узлах CST или внешним классом-билдером AstBuilder, который обходит CST и конструирует узлы AST.[12]

Часть IV: Исполнение и Анализ: Паттерн Visitor

После того как AST построено, интерпретатор должен выполнять над ним различные операции: проверку типов, оптимизацию, генерацию кода или непосредственное исполнение (evaluation). Паттерн Visitor (Посетитель) является стандартом де-факто для реализации этих операций, решая фундаментальную проблему архитектуры компиляторов.

4.1. The Expression Problem и выбор паттерна

При разработке интерпретатора архитекторы сталкиваются с классической проблемой, известной как "Expression Problem". Она описывает сложность расширения системы в двух ортогональных измерениях:

1. **Типы данных:** Добавление новых узлов в AST (например, новой языковой конструкции TryCatch).
2. **Операции:** Добавление новых функций над AST (например, TypeCheck, Evaluate, PrettyPrint).

Существует два основных подхода к организации кода:

- **Объектно-ориентированный (Интерпретатор):** Логика операций вшивается внутрь классов узлов (например, каждый узел имеет метод eval()). В этом случае легко добавлять новые типы узлов (просто создав новый класс), но сложно добавлять новые операции (нужно модифицировать все существующие классы узлов).
- **Функциональный (Visitor):** Логика операций выносится в отдельные компоненты (Посетители). Легко добавлять новые операции (создав новый Visitor), но сложно добавлять новые типы узлов (так как нужно обновлять интерфейс Visitor и все его реализации).

Для DSL, где набор языковых конструкций (синтаксис) обычно стабилизируется быстрее, чем набор необходимых анализов и оптимизаций, подход Visitor является предпочтительным. Исследования, проведенные на примере языка Rascal, показывают, что Visitor превосходит классический паттерн Interpreter (встраивание логики в узлы) в вопросах поддерживаемости (Maintainability) и модульности. Visitor позволяет инкапсулировать сложную логику конкретного алгоритма (например, проверки типов) в одном классе, вместо того чтобы "размазывать" её по сотням классов AST.[18]

4.2. Механизм двойной диспетчеризации (Double Dispatch)

В языках программирования с одиночной диспетчеризацией (Java, C#, C++) метод выбирается только на основе типа объекта, у которого он вызван. Visitor реализует эмуляцию двойной диспетчеризации, позволяя выбрать нужный метод на основе двух типов: типа операции (Visitor) и типа узла (Node).

Механизм работает в два шага:

1. **node.accept(visitor):** Клиент вызывает метод accept у узла AST. Благодаря полиморфизму, управление передается в реализацию accept конкретного класса узла (например, BinaryExpression).
2. **visitor.visit(this):** Внутри метода accept узел вызывает метод visit у переданного объекта visitor, передавая себя (this) в качестве аргумента. Компилятор или рантайм выбирает перегруженный метод visit, соответствующий типу this (например, visit(BinaryExpression node)).

Это позволяет корректно маршрутизировать вызов к нужному блоку кода, который знает, как обработать конкретный узел конкретным алгоритмом.[20]

4.3. Управление состоянием и итеративные реализации

Важный аспект реализации Visitor в интерпретаторе — управление состоянием (State Management). Например, при интерпретации необходимо хранить текущие значения переменных.

- **Stateful Visitor:** Состояние (Environment) хранится в полях самого объекта Visitor. Это позволяет накапливать контекст при обходе дерева. Однако, это делает Visitor "одноразовым" или требующим сложного управления сбросом состояния при рекурсивных вызовах функций DSL.[20]
- **Context Passing:** Более чистый функциональный подход предполагает передачу контекста (таблицы символов, памяти) как аргумента в методы visit и возвращение обновленного контекста.

Проблема рекурсии: Классическая реализация Visitor использует рекурсию хост-языка (например, Java Stack). Для глубоких деревьев AST это может привести к ошибке переполнения стека (StackOverflowError). В промышленных интерпретаторах часто применяют **итеративный Visitor** или стиль "Trampoline". В этом случае метод visit не вызывает рекурсивно следующий accept, а возвращает специальный объект "Продолжение" (Continuation) или следующий узел для обработки. Главный цикл интерпретатора (Event Loop) получает этот узел и вызывает accept на нем. Это превращает процесс обхода дерева в плоский цикл, сохраняя при этом логическую структуру паттерна Visitor.

Часть V: Эволюция: От Интерпретатора к Компилятору

Превращение интерпретатора в компилятор — это процесс добавления новых стадий обработки, которые отделяют анализ программы от её немедленного исполнения. Простой обход AST (Tree-walking interpretation) страдает от накладных расходов на рекурсию, динамические проверки типов и отсутствие глобальных оптимизаций. Чтобы превратить интерпретатор в компилятор, необходимо внедрить ряд дополнительных модулей.

5.1. Модуль генерации промежуточного представления (IR Generator)

Первым шагом к компиляции является отказ от древовидной структуры в пользу

линейной. Деревья сложны для анализа потока данных и оптимизации.

- **Linearization (Линеаризация):** Преобразование AST в последовательность инструкций.
- **Intermediate Representation (IR):** Внедрение промежуточного представления, которое абстрагирует код от синтаксиса (как AST), но уже приближено к машине (как ассемблер).
    - **Three-Address Code (TAC):** Популярный формат IR, где инструкции имеют вид x = y op z. Это упрощает отображение на машинные инструкции.
    - **Control Flow Graph (CFG):** Построение графа потока управления, где узлами являются базовые блоки (последовательности инструкций без ветвлений), а ребрами — переходы (jump/branch). CFG необходим для любого серьезного анализа кода.[22]

5.2. Модуль SSA (Static Single Assignment)

Для реализации эффективных оптимизаций современные компиляторы преобразуют IR в форму статического единственного присваивания (SSA). В SSA каждой переменной значение присваивается ровно один раз. Если переменная изменяется в разных ветках кода (например, в if-else), в точке слияния веток вставляется специальная Phi-функция, которая выбирает нужное значение.

- **Значение для компилятора:** SSA кардинально упрощает анализ использования данных (Def-Use chains). Алгоритмам оптимизации не нужно сканировать весь код, чтобы узнать, где определена переменная — это гарантируется структурой SSA. Это делает такие оптимизации, как удаление мертвого кода (Dead Code Elimination) и распространение констант (Constant Propagation), быстрыми и надежными.[22]

5.3. Модуль Оптимизации (Optimizer)

На уровне IR подключаются модули оптимизации, не зависящие от целевой архитектуры (Machine-Independent Optimizations):

- **Constant Folding:** Вычисление константных выражений во время компиляции (например, замена 2 + 3 на 5).
- **Dead Code Elimination:** Удаление кода, который никогда не будет исполнен (анализ достижимости в CFG).
- **Loop Invariant Code Motion:** Вынос вычислений, не зависящих от переменных цикла, за пределы тела цикла.

5.4. Backend: Выбор инструкций и Распределение регистров

Этот слой отвечает за трансляцию абстрактного IR в конкретные инструкции целевой машины (Native Code) или виртуальной машины (Bytecode).

- **Instruction Selection (Выбор инструкций):** Модуль должен выбрать наилучшую

последовательность машинных инструкций для реализации операций IR. Традиционно используется метод **Tree Covering** (покрытие дерева), где фрагменты IR сопоставляются с шаблонами инструкций. Современные подходы используют SMT-солверы (Satisfiability Modulo Theories) для автоматического синтеза оптимальных последовательностей инструкций, доказывая их эквивалентность исходному IR.[25]

- **Register Allocation (Распределение регистров):** В AST-интерпретаторе переменные живут в памяти (хеш-мапе окружения). В компиляторе они должны быть отображены на ограниченный набор физических регистров процессора.
  - **Graph Coloring:** Классический подход моделирует проблему как раскраску графа интерференции переменных. Если две переменные "живы" одновременно, их узлы в графе соединяются ребром. Задача сводится к раскраске графа K цветами (где K — число регистров), чтобы смежные узлы имели разные цвета. Переменные, которые не удалось "раскрасить", вытесняются в стек (Spilling).[22]

5.5. Модуль JIT-компиляции (Just-In-Time)

Для достижения максимальной производительности, особенно для динамических DSL, добавляются модули JIT-компиляции.

- **Tracing JIT:** Модуль, который наблюдает за исполнением программы интерпретатором, записывает ("трассирует") последовательность операций в "горячих" циклах и компилирует этот линейный след в машинный код. Это позволяет игнорировать сложные ветвления, которые не выполняются в данном конкретном запуске.[27]
- **Partial Evaluation (Частичные вычисления):** Подход, популяризированный фреймворком GraalVM/Truffle. Разработчик пишет только AST-интерпретатор с определенными аннотациями. Во время исполнения специальный механизм анализирует AST и "схлопывает" его вместе с данными в оптимизированный машинный код, фактически автоматически генерируя JIT-компилятор из интерпретатора.[29]

Таблица 2: Эволюция архитектуры от Интерпретатора к Компилятору

Стадия	Ключевая структура данных	Механизм исполнения	Добавляемые модули
Интерпретатор (Tree-walker)	AST	Рекурсивный обход (Visitor)	Parser, Semantic Analyzer, Evaluator Visitor.

Bytecode VM	Линейный массив байткода	Цикл выборки инструкций (Dispatch Loop)	Bytecode Emitter (Compiler), Stack/Register VM Architecture.
AOT Компилятор	IR (SSA), CFG, Machine Code	Исполняемый бинарный файл (Native Binary)	IR Generator, SSA Transformer, Optimizer, Register Allocator, Code Emitter.
JIT Компилятор	Profiling Data, Trace Trees	Смешанное (Interpreter + Native Code)	Profiler (счетчики исполнений), Tracer, On-Stack Replacement (OSR), Deoptimizer.

Часть VI: Симбиоз DSL и Генеративного ИИ

В современной разработке DSL нельзя игнорировать роль больших языковых моделей (LLM). Стандартный подход к реализации DSL сегодня расширяется модулями интеграции с AI, где грамматика языка начинает играть новую роль.

6.1. Грамматика как ограничитель для LLM

Одной из главных проблем при генерации кода с помощью LLM являются "галлюцинации" — генерация синтаксически некорректного кода или выдуманных функций. Архитектура современного DSL может решать эту проблему через технику **Constrained Decoding** (Ограниченное декодирование).

- **Grammar Prompting:** EBNF грамматика DSL, используемая для генерации парсера, также подается на вход механизму инференса LLM.
- **Механизм действия:** В процессе генерации каждого токена LLM вычисляет вероятности всех слов в словаре. Модуль ограниченного декодирования использует текущее состояние парсера грамматики, чтобы наложить "маску" на эти вероятности. Вероятности всех токенов, которые недопустимы в данной позиции согласно грамматике DSL, принудительно обнуляются. Это математически гарантирует, что LLM будет генерировать только синтаксически валидный код.[2]

6.2. Synchromesh и семантическая валидация

Более продвинутые системы, такие как **Synchromesh**, идут дальше и используют CST/AST

не только для синтаксических, но и для семантических ограничений. Используя методы **Target Similarity Tuning**, система находит примеры кода, похожие на запрос пользователя, и использует их как few-shot примеры для LLM. Одновременно с этим, модуль **Constrained Semantic Decoding (CSD)** проверяет семантические правила (например, наличие переменных в области видимости или соответствие типов) "на лету" во время генерации, предотвращая появление кода, который парсится, но не компилируется.[31]

Таким образом, CST и грамматика DSL становятся не просто инструментом разбора входящего текста, но и "скелетом", направляющим генеративные способности нейросетей, обеспечивая надежность программного синтеза.

Заключение

Реализация интерпретатора для DSL — это комплексная инженерная задача, требующая баланса между простотой разработки, производительностью и качеством инструментария. Анализ стандартных подходов и современных исследований позволяет сделать следующие выводы:

1. **CST является обязательным элементом:** Стратегия прямого построения AST является устаревшей для языков, претендующих на использование в современных IDE. CST обеспечивает необходимую базу для LSP, рефакторинга и устойчивого парсинга.
2. **Visitor обеспечивает долгосрочную поддерживаемость:** Несмотря на кажущуюся сложность, паттерн Visitor предоставляет необходимую гибкость в добавлении новых анализов и фаз компиляции, превосходя классический паттерн Interpreter в модульности.
3. **Многоступенчатая эволюция:** Переход от интерпретатора к компилятору не требует полного переписывания, если архитектура модульная. Внедрение IR, SSA и распределения регистров позволяет плавно эволюционировать систему.
4. **Интеграция с AI:** Современная спецификация DSL должна рассматриваться как двойной актив: для генерации парсера и для управления генерацией кода LLM.

Следование этим архитектурным принципам позволяет создавать надежные, расширяемые и производительные языковые системы, готовые к интеграции в современные цепочки разработки ПО.

Источники

1. From a Natural to a Formal Language with DSL Assistant - arXiv, дата последнего обращения: декабря 28, 2025, https://arxiv.org/pdf/2408.09766
2. Grammar Prompting for Domain-Specific Language Generation with ..., дата последнего обращения: декабря 28, 2025,

https://proceedings.neurips.cc/paper_files/paper/2023/file/cd40d0d65bfebb894ccc9ea822b47fa8-Paper-Conference.pdf
3. CREATING LLM AGENTS FOR DOMAIN-SPECIFIC LANGUAGE ..., дата последнего обращения: декабря 28, 2025, https://stal.blogspot.com/2025/10/creating-llm-agents-for-domain-specific.html
4. Grammar induction - Wikipedia, дата последнего обращения: декабря 28, 2025, https://en.wikipedia.org/wiki/Grammar_induction
5. Prolog Coding Guidelines: Status and Tool Support - ResearchGate, дата последнего обращения: декабря 28, 2025, https://www.researchgate.net/publication/335869460_Prolog_Coding_Guidelines_Status_and_Tool_Support
6. Flexible and Efficient Grammar-Constrained Decoding - arXiv, дата последнего обращения: декабря 28, 2025, https://arxiv.org/pdf/2502.05111?
7. Using Grammar Masking to Ensure Syntactic Validity in LLM-based ..., дата последнего обращения: декабря 28, 2025, https://arxiv.org/html/2407.06146v1
8. Refactoring - Bowler, дата последнего обращения: декабря 28, 2025, https://pybowler.io/docs/basics-refactoring
9. Is treesitter worth it? : r/emacs - Reddit, дата последнего обращения: декабря 28, 2025, https://www.reddit.com/r/emacs/comments/16ey3hq/is_treesitter_worth_it/
10. How are individual lines of code and functions stored in a Concrete ..., дата последнего обращения: декабря 28, 2025, https://softwareengineering.stackexchange.com/questions/303766/how-are-individual-lines-of-code-and-functions-stored-in-a-concrete-syntax-tree
11. INTRODUCING ENRICHED CONCRETE SYNTAX TREE - arXiv, дата последнего обращения: декабря 28, 2025, https://arxiv.org/pdf/1310.0802
12. On CSTs, ASTs and Model-Driven Editors, дата последнего обращения: декабря 28, 2025, https://www.eclipse.org/lists//mdt-ocl.dev/pdftCzZojIJNH.pdf
13. Code to Tree: A Deep Dive into Michael Lee's AST Server for AI ..., дата последнего обращения: декабря 28, 2025, https://skywork.ai/skypage/en/code-tree-deep-dive-ai-engineers/1979027289129918464
14. AST-Transformation — DHParser 1.3.5 documentation, дата последнего обращения: декабря 28, 2025, https://dhparser.readthedocs.io/en/v1.3.5/manuals/03_transform.html
15. Declarative Syntax Tree Engineering* - NYU Computer Science, дата последнего обращения: декабря 28, 2025, https://cs.nyu.edu/media/publications/TR2007-905.pdf
16. (PDF) Introducing enriched concrete syntax trees - ResearchGate, дата последнего обращения: декабря 28, 2025, https://www.researchgate.net/publication/251565750_Introducing_enriched_concrete_syntax_trees
17. Building a Python compiler and interpreter – 03 visitor pattern, дата последнего обращения: декабря 28, 2025, https://mathspp.com/blog/building-a-python-compiler-and-interpreter-03-visitor

-pattern
18. A Case of Visitor versus Interpreter Pattern - CWI, дата последнего обращения: декабря 28, 2025, https://homepages.cwi.nl/~storm/publications/visitor.pdf
19. (PDF) A Case of Visitor versus Interpreter Pattern - ResearchGate, дата последнего обращения: декабря 28, 2025, https://www.researchgate.net/publication/220878226_A_Case_of_Visitor_versus_Interpreter_Pattern
20. Stateful visitor pattern? : r/Compilers - Reddit, дата последнего обращения: декабря 28, 2025, https://www.reddit.com/r/Compilers/comments/13mxfpp/stateful_visitor_pattern/
21. What is the difference between Strategy pattern and Visitor Pattern?, дата последнего обращения: декабря 28, 2025, https://stackoverflow.com/questions/8665295/what-is-the-difference-between-strategy-pattern-and-visitor-pattern
22. (PDF) SSA-Based Register Allocation with PBQP - ResearchGate, дата последнего обращения: декабря 28, 2025, https://www.researchgate.net/publication/221302576_SSA-Based_Register_Allocation_with_PBQP
23. CP25C04 Acd | PDF | Program Optimization | Parsing - Scribd, дата последнего обращения: декабря 28, 2025, https://www.scribd.com/document/957519480/CP25C04-ACD-1
24. Modular Compiler for the TinyC Language - DSpace, дата последнего обращения: декабря 28, 2025, https://dspace.cvut.cz/bitstream/handle/10467/108868/F8-DP-2023-Prokopic-Martin-thesis.pdf?sequence=-1&isAllowed=y
25. Synthesizing Instruction Selection Rewrite Rules from RTL using SMT, дата последнего обращения: декабря 28, 2025, http://theory.stanford.edu/~barrett/pubs/DDM+22.pdf
26. Compiling With Constraints | Hey There Buddo! - Philip Zucker, дата последнего обращения: декабря 28, 2025, https://www.philipzucker.com/compile_constraints/
27. Torchy: A Tracing JIT Compiler for PyTorch (Extended Version), дата последнего обращения: декабря 28, 2025, https://web.ist.utl.pt/nuno.lopes/pubs/torchy-cc23-extended.pdf
28. Tracing the meta-level: PyPy's tracing JIT compiler | Request PDF, дата последнего обращения: декабря 28, 2025, https://www.researchgate.net/publication/229422188_Tracing_the_meta-level_PyPy's_tracing_JIT_compiler
29. Surgical Precision JIT Compilers - Computer Science Purdue, дата последнего обращения: декабря 28, 2025, https://www.cs.purdue.edu/homes/rompf/papers/rompf-pldi14.pdf
30. Grammar-Constrained Decoding for Large Language Models, дата последнего обращения: декабря 28, 2025, https://saibo-creator.github.io/uploads/NexThink_talk_2024_06_18.pdf
31. Reliable code generation from pre-trained language models - arXiv, дата

последнего обращения: декабря 28, 2025, https://arxiv.org/abs/2201.11227
32. SYNCHROMESH: RELIABLE CODE GENERATION FROM PRE ..., дата последнего обращения: декабря 28, 2025, https://www.microsoft.com/en-us/research/wp-content/uploads/2022/01/csd_arxiv.pdf