Валерий Антонов

Грамматическая машина. Том 26. GrammaLang: Практическое руководство

Грамматическая машина

Валерий Антонов

Грамматическая машина. Том 26. GrammaLang: Практическое руководство

«Автор»
2026

Антонов В.

Грамматическая машина. Том 26. GrammaLang: Практическое руководство / В. Антонов — «Автор», 2026 — (Грамматическая машина)

Эта книга — практическое руководство по GrammaLang версии 0.2, работающему языку программирования, где онтология является первоклассным гражданином. Вы научитесь анализировать философские тексты, восстанавливать онтологию бинарных файлов, проектировать ГМ-промпты для LLM и создавать агентов с операторным мышлением.Книга проведёт вас от философского фундамента через инженерную методологию Семантического реактора к исполнимой виртуальной машине. Реакторный режим — для демонтажа понятий, полифонический — для удержания неразрешимых противоречий. Grammar Change переключает между ними.Это приглашение к со-исполнению. ГМ — не технология, а способ мышления. Пересобирайте свою грамматику. Создавайте свои миры. Удерживайте сложность.

Содержание

Валерий Антонов
Грамматическая машина. Том 26. GrammaLang: Практическое руководство

Введение.

Эта книга — мост между философской идеей и работающей программой.

В 2025 году вышла серия книг «Грамматическая машина». В них был описан мета-инструмент для анализа того, как грамматика конституирует реальность. Четыре оператора — split, hold, transition, grammar_change. Онтологические типы — Substance, Modus, Boundary, TensionNode. Гибридная архитектура, где языковая модель служит мозгом, а MCP-инструменты — руками. Виртуальная машина с шестью компонентами. Язык GrammaLang, в котором онтология является частью кода.

Это была архитектурная спецификация — подробные чертежи самолёта. Был и картонный макет — Python-прототип, который генерировал музыку из философских понятий. Но сам самолёт ещё не летал.

Сейчас он летает. GrammaLang версии 0.2 — это работающая система. В ней есть Python-пайплайн с подключаемыми анализаторами и генераторами. Rust-ядро с детерминированной онтологией. NLP-анализатор на базе stanza, который разбирает философский текст, выделяет категории, находит грамматические маркеры противоречий, привязывает глаголы к субъектам. MIDI-генератор, превращающий онтологическую карту в звук. Визуализатор, рисующий граф субстанций и противоречий. Командная строка. Репозиторий промптов для языковых моделей.

Попробуйте прямо сейчас. Откройте в браузере:

text

https://pawelkaev.github.io/grammalang/interface.html

Это веб-интерфейс GrammaLang. Нажмите «Загрузить пример (Кант)», затем «Анализ» — и вы увидите, как система разбирает фрагмент «Критики чистого разума»: выделяет философские категории, находит противоречия, определяет модусы, строит граф и подсвечивает результаты прямо в тексте. Для полной работы интерфейсу нужен локальный сервер (инструкция — в главе 4), но демо-граф отображается и без него.

Примечание об иллюстрациях. Эта книга не содержит скриншотов. GrammaLang — визуальный инструмент, и описать граф словами непросто. Вместо картинок откройте интерфейс по ссылке выше и увидите всё вживую. Это лучше любого скриншота: граф будет ваш собственный, из вашего текста.

Эта книга — документация к этой системе. Она написана для трёх читателей.

Для философа, который хочет увидеть, как абстрактные категории становятся структурами данных, а противоречия — узлами графа. Для него написаны Часть I (философия и архитектура) и Часть VII (визуализация). Он узнает, что оператор hold — это не просто слово, а метод hold_contradiction в Rust-структуре OntologicalContext, и может запустить его и увидеть результат.

Для исследователя или студента, который хочет проанализировать философский текст новым инструментом. Для него — Часть II (установка и первый запуск), Часть IV (пайплайн и анализ), Часть V (промпты и LLM-интеграция). Через пять минут после установки он получит MIDI-файл из «Критики чистого разума» или увидит грамматическую разметку гегелевского абзаца.

Для разработчика, который хочет расширить систему: написать новый анализатор, генератор, MCP-инструмент или добавить оператор в ядро. Для него — Часть VIII (разработка и вклад) и Приложения с EBNF-грамматикой и схемами данных.

Книга построена по принципу «от простого к сложному». Сначала вы устанавливаете GrammaLang и запускаете первый анализ. Затем знакомитесь с архитектурой пайплайна. Затем — с онтологическими типами и операторами. Затем — с расширенными возможностями: NLP-анализом, промптами, визуализацией. В конце — со структурой кода и тем, как его модифицировать.

Каждое утверждение в этой книге проверяемо. Если написано, что hold_contradiction создаёт TensionNode и возвращает его идентификатор — вы можете открыть файл grammalang-core/src/ontology.rs, найти этот метод и увидеть его код. Если написано, что интеграционный тест защищает систему от регрессий — вы можете запустить pytest -v и увидеть зелёную строку.

Эта книга — приглашение. GrammaLang вырос из философского концепта в инженерный проект. Он не завершён — ему нужны новые анализаторы, новые генераторы, распределённая VM, интеграция с IDE. Если вы хотите участвовать — последняя глава расскажет, с чего начать.

Добро пожаловать в Грамматическую Машину.

О версии 0.2.

Важно. Данная книга описывает GrammaLang версии 0.2. Это работающая система, но не завершённая. То, что помечено как «планируется» или «в версии 0.3», в текущем коде отсутствует. В частности:

Команда grammalang run script.gl пока не реализована. Скрипты на языке GrammaLang можно писать, но их исполнение через VM появится в версии 0.3.

Полноценный MCP-клиент с динамической загрузкой инструментов — в планах.

Распределённая VM с параллельным исполнением ветвей split — в проекте.

WebSocket-трассировка для визуализатора реального времени — ожидается.

Это не ошибки и не незавершённые функции. Это дорожная карта. Если вы попытаетесь запустить то, чего ещё нет, вы не найдёте этого в коде — и это нормально.

Самый простой способ попробовать GrammaLang — открыть веб-интерфейс по ссылке выше. Он демонстрирует граф, разметку текста и статистику без установки. Для полного анализа потребуется локальный сервер (Глава 4).

Благодарности.

Эта система и эта книга — результат диалога. Философская основа заложена автором концепции «Грамматической машины». Архитектурная спецификация разработана и описана в серии книг. Инженерная реализация — от Python-прототипа до Rust-ядра, от CLI до визуализатора, от первого теста до NLP-анализатора — выполнена при участии Claude (Anthropic) в качестве архитектора-соавтора и инженера-реализатора.

Особая благодарность — сообществам разработчиков Rust, Python, stanza, PyO3, mido и D3.js, на чьих инструментах построен GrammaLang.

Обозначения.

В книге приняты следующие обозначения.

Команды для терминала набраны моноширинным шрифтом и начинаются с приглашения PS>:

text

PS> grammalang analyze "Свобода и необходимость" -o output.mid

Вводить нужно только текст после PS>, приглашение показано для ориентира (вы находитесь в PowerShell, в папке проекта).

Пути к файлам записываются относительно корня проекта. Запись src/grammalang/analyzers/grammar_analyzer.py означает: внутри папки grammalang откройте папку src, затем grammalang, затем analyzers, затем файл grammar_analyzer.py.

Имена программных сущностей (классов, методов, переменных) выделены моноширинным шрифтом в тексте: OntologicalContext, hold_contradiction, TensionStatus::Held. Философские термины, впервые вводимые в книге, выделены курсивом: *субстанция*, *модус*, *противоречие*.

Важные замечания помечены словом **Примечание**, предупреждения — **Внимание**.

Часть I. Философия и Архитектура.

Глава 1. Что такое Грамматическая Машина.

1.1. Идея: грамматика конституирует реальность.

Грамматическая Машина исходит из тезиса: реальность не дана нам непосредственно — она конституируется через грамматику. Грамматика не описывает уже существующий мир, а задаёт условия, при которых нечто может быть помыслено как существующее.

Этот тезис восходит к Канту: пространство и время суть априорные формы созерцания, а категории рассудка — априорные формы мышления. Грамматическая Машина переносит этот принцип на уровень языка: грамматика — это активный механизм, производящий онтологию. Разные грамматики производят разные реальности. Переключение между грамматиками — это смена самого мира, а не точки зрения на него.

GrammaLang — это инструмент для работы с этим процессом. Он не предписывает единственную правильную грамматику. Он позволяет удерживать несколько грамматик одновременно, переключаться между ними и отслеживать, какие онтологии они производят.

1.2. Четыре оператора.

Split (разделение). Берёт онтологический контекст и разделяет его на две или более независимых ветви. Каждая ветвь может развиваться по собственным законам. Философски это операция различения, фундаментальная для любого мышления.

Hold (удержание). Фиксирует противоречие, не разрешая его. Создаёт TensionNode — узел, в котором две субстанции удерживаются в напряжении. Чуждый классической логике и близкий к диалектике.

Transition (переход). Переносит контекст на другой уровень или в другую среду. Вызов внешнего инструмента, смена уровня анализа, передача контекста другому агенту.

Grammar_change (смена грамматики). Переключает режим работы VM. Реакторный режим — линейный анализ с разрешением противоречий. Полифонический — удержание множества противоречивых позиций.

Эти четыре оператора образуют алфавит Грамматической Машины.

1.3. Онтологические типы.

Substance (субстанция). Фундаментальная сущность. Имеет имя, идентификатор и энергию — число от 0 до 1, отражающее её активность в контексте.

Modus (модус). Способ действия или проявления субстанции. Всегда привязан к субстанции. В грамматическом анализе модусами становятся глаголы-предикаты.

Boundary (граница). Разделитель между контекстами. Определяет, что внутри и снаружи данного рассуждения.

TensionNode (узел противоречия). Центральный тип GrammaLang. Зафиксированное противоречие с жизненным циклом: held (удерживается), resolved (разрешено), escalated (эскалировано).

Подробная спецификация каждого типа с полями и методами приведена в Главе 6.

1.4. Два режима мышления.

Реакторный режим. Последовательный анализ. Противоречия разрешаются. Соответствует классическому рациональному мышлению.

Полифонический режим. Множественные противоречия удерживаются одновременно. Результат анализа — не ответ, а карта противоречий. Соответствует диалектическому мышлению.

Переключение между режимами — оператор grammar_change. Сама возможность переключения — ключевая особенность GrammaLang: язык рефлексирует собственный способ мышления.

1.5. Пример: антиномия свободы и необходимости.

Пользователь вводит текст: «Свобода и необходимость: антиномия чистого разума». Анализатор создаёт две субстанции: «Свобода» с энергией 0.8 и «Необходимость» с энергией 0.4. Затем hold создаёт TensionNode с причиной «antinomy» и статусом held.

Если VM в реакторном режиме, противоречие может разрешиться в пользу одной из сторон. Если в полифоническом — оба полюса удерживаются в напряжении.

В MIDI-генераторе это звучит так: субстанции становятся нотами (высокая для Свободы, низкая для Необходимости), противоречие — диссонансом, разрешение — мажорным аккордом. Визуализатор показывает два цветных шара и красную пунктирную линию между ними.

Глава 2. Архитектура GrammaLang.

2.1. Общий взгляд.

GrammaLang — экосистема из нескольких слоёв. Внешний слой — интерфейсы: CLI, веб-интерфейс, визуализатор. Средний слой — Python-пайплайн: анализаторы, генераторы, сервер. Внутренний слой — Rust-ядро: онтологический контекст, операторы, типы данных.

Разделение даёт три преимущества. Анализатор можно написать на Python или Rust — пайплайн работает с обоими. Генератор можно заменить: сегодня MIDI, завтра JSON-отчёт. Ядро можно переписать на другой язык — пока пайплайн вызывает те же методы, ничего не сломается.

2.2. Python-пайплайн.

Пайплайн состоит из трёх абстрактных классов и одного связующего. MaterialSource — источник текста. OntologicalAnalyzer — строитель онтологии. OutputGenerator — генератор выхода. GrammaPipeline — связующее звено.

Контракт пайплайна: компоненты не общаются друг с другом напрямую. Единственная точка обмена — OntologicalContext. Это делает систему тестируемой и расширяемой.

2.3. Rust-ядро.

Ядро состоит из трёх модулей. ontology.rs — структуры данных и методы. error.rs — система ошибок. trace.rs — буфер событий для визуализатора.

Для хранения субстанций используется BTreeMap, а не HashMap. Это гарантирует детерминированный порядок ключей и делает интеграционный тест стабильным.

Rust выбран не случайно. Система владения идеально ложится на семантику GrammaLang: split — перемещение контекста, hold — инвалидация мутабельных ссылок, grammar_change — смена типажа. Компилятор проверяет эти инварианты на этапе сборки.

2.4. Мост Rust ↔ Python.

Связь через PyO3. Функция analyze_text в Rust скомпилирована в .pyd, который Python импортирует как обычный модуль. RustAnalyzer — обёртка, вызывающая Rust-функцию и превращающая JSON в Pydantic-модель. С точки зрения пайплайна RustAnalyzer не отличается от KantianAnalyzer.

2.5. Сервер и визуализатор

HTTP-сервер слушает порт 8080, принимает POST-запросы, возвращает JSON с результатом анализа. Визуализатор — одностраничное приложение без зависимостей. Доступен онлайн по адресу https://pawelkaev.github.io/grammalang/interface.html.

2.6. Интеграционный тест как страховка.

Test Oracle прогоняет текст через всю систему и сравнивает MD5-хеш результата с эталоном. Если хеш совпадает — система работает правильно. Если изменился — что-то значимое поменялось, и разработчик должен обновить эталон осознанно.

2.7. CLI и промпты.

Командная строка даёт три команды: analyze, prompt-validate, prompt-run. Промпты — структурированные запросы к LLM. Хранятся как JSON-файлы в папке prompts/. Превращают языковую модель в ГМ-агента.

Часть II. Установка и Первый Запуск.

Глава 3. Установка.

3.0. Быстрый старт без установки.

Если вы хотите просто посмотреть, что такое GrammaLang, не устанавливая Python, Rust и NLP-модели, откройте в браузере:

text

https://pawelkaev.github.io/grammalang/interface.html

Нажмите «Загрузить пример (Кант)», затем «Анализ». Вы увидите граф, список субстанций и противоречий, а также размеченный текст с подсветкой — и всё это без установки. Единственное ограничение: для полного NLP-анализа нужен локальный сервер (инструкция ниже).

Если вы хотите полную версию с MIDI-генерацией, NLP-анализом и командной строкой — читайте дальше.

3.1. Что нужно иметь.

Python версии 3.10 или новее. Проверить: python --version. Если нет — скачайте с python.org, отметив «Add Python to PATH».

Rust версии 1.70 или новее. Установить: winget install Rustlang.Rustup. Проверить: rustc --version.

Visual Studio Build Tools 2022. Установить: winget install Microsoft.VisualStudio.2022.BuildTools. После установки обязательно перезагрузить компьютер.

Git — опционально, только для клонирования репозитория.

3.2. Получение проекта.

Скачайте ZIP-архив:

text

https://github.com/PawelKaev/grammalang/archive/refs/heads/main.zip

Автор проекта — Валерий Антонов. Репозиторий: https://github.com/PawelKaev/grammalang.

Извлеките архив на рабочий стол. Переименуйте папку grammalang-main в grammalang. Откройте её — внутри pyproject.toml, Makefile, interface.html, visualizer.html, папки src, grammalang-core, tests, prompts.

3.3. Открытие проекта.

Запустите VS Code. File → Open Folder → выберите папку grammalang. Откройте терминал: Terminal → New Terminal.

3.4. Виртуальное окружение Python

text
PS> python -m venv .venv
PS> .venv\Scripts\Activate.ps1
Если ошибка политики безопасности:
text
PS> Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser
3.5.
Сборка
Rust-
ядра.
text
PS> cd grammalang-core
PS> $env:PYO3_USE_ABI3_FORWARD_COMPATIBILITY = "1"
PS> cargo build --release
PS> Copy-Item -Path target\release\grammalang_core.dll -Destination ..\src\grammalang \rust_bridge\grammalang_core.pyd -Force
PS> cd ..

3.6. Установка Python-пакета

```
text
PS> pip install -e ".[dev]"
PS> python -c "import stanza; stanza.download('ru')"
```

3.7. Проверка установки.

text
PS> pytest -v
PS> grammalang --help
Обе команды должны выполниться без ошибок.
Глава 4. Первый анализ.

4.1. Команда analyze.

text

PS> grammalang analyze "Свобода и необходимость: антиномия чистого разума" -o my_first.mid

CLI создаёт пайплайн, анализатор строит онтологию, генератор превращает её в MIDI.

4.2. Что вы услышите

Две ноты: высокая (Свобода, энергия 0.8), низкая (Необходимость, 0.4). Затем диссонанс — TensionNode в статусе held. Если бы противоречие разрешилось — мажорный аккорд.

4.3. Веб-интерфейс.

Откройте https://pawelkaev.github.io/grammalang/interface.html. Для полной работы запустите сервер:

text

PS> python -c "from grammalang.server import run_server; run_server()"

Нажмите «Загрузить пример (Кант)», затем «Анализ». Увидите граф, карточки субстанций, список противоречий и размеченный текст.

4.4. Эксперименты с разными текстами.

Попробуйте: «Пространство и время как априорные формы», «Единство и множество в диалектике», «Дух и материя». Каждый текст даст разную музыку и разный граф.

4.5. NLP-анализ через веб-интерфейс

При запущенном сервере вставьте любой философский текст в поле ввода и нажмите «Анализ». Сервер вызовет GrammarAnalyzer, который разберёт каждое предложение через stanza.

4.6. За пределами звука.

MIDI — лишь одна из проекций онтологической карты. Тот же контекст можно экспортировать в JSON, визуализировать как граф, разметить в тексте или отправить в LLM через промпт.

Часть III. Язык GrammaLang.

Глава 5. Синтаксис GrammaLang.

5.1. Зачем GrammaLang свой синтаксис

У GrammaLang есть две стороны. Одна — программный интерфейс: Python-классы и Rust-методы, которые вызываются из кода. Другая — декларативный язык, на котором можно описывать онтологические структуры напрямую, без программирования.

Синтаксис GrammaLang минимален. Он не является языком общего назначения — в нём нет циклов, условных операторов или работы с файлами. Его единственная задача — объявлять онтологические сущности и применять к ним операторы Грамматической Машины. Это язык разметки реальности, а не язык программирования.

В текущей версии (0.2) парсер реализован как каркас в Rust-ядре и поддерживает базовый синтаксис. Полноценное исполнение скриптов GrammaLang появится в версии 0.3. Однако синтаксис уже определён и документирован — вы можете писать скрипты сейчас и тестировать их через Python-пайплайн.

5.2. Базовые элементы

Комментарии начинаются с двух косых черт и действуют до конца строки:

text

// Это комментарий

Идентификаторы — это имена субстанций, модусов, границ и узлов противоречий. Они могут содержать буквы (в том числе русские), цифры и знак подчёркивания. Регистр учитывается. Примеры допустимых идентификаторов: свобода, Freedom, вещь_в_себе, tension_01.

Числа с плавающей точкой используются для задания энергии субстанций. Диапазон: от 0.0 до 1.0. Примеры: 0.8, 0.0, 1.0, 0.35.

Строки заключаются в двойные кавычки и используются для указания причины противоречия: "antinomy", "диалектическое противоречие".

5.3. Объявление субстанции

Субстанция — фундаментальная сущность. Объявляется ключевым словом substance, за которым следуют идентификатор и блок с энергией:

text

substance свобода {

energy: 0.8

}

Идентификатор становится уникальным именем субстанции в онтологическом контексте. Энергия — число от 0 до 1, отражающее активность или вес субстанции в данном анализе.

Объявление субстанции не создаёт противоречий и не меняет существующие. Оно просто добавляет сущность в контекст. Если субстанция с таким идентификатором уже существует, новое объявление перезаписывает её.

Примеры:

text

substance разум {

energy: 0.9

}

```
substance материя {
energy: 0.3
}
substance пространство {
energy: 0.6
}
substance время {
energy: 0.7
}
```

5.4. Объявление модуса

Модус — способ действия или проявления субстанции. Объявляется ключевым словом modus, за которым следуют имя и значение:

```
text
modus разум полагает
modus разум мыслит
modus природа отвечает
```

Первое слово после modus — идентификатор субстанции, к которой привязывается модус. Второе — значение модуса (обычно глагол). Субстанция должна быть объявлена ранее; попытка привязать модус к несуществующей субстанции вызовет ошибку.

Модусы не имеют энергии и не участвуют в противоречиях напрямую. Они описывают, *как* субстанция действует, предоставляя дополнительный контекст для анализа.

5.5. Объявление границы

Граница разделяет контексты. Объявляется ключевым словом boundary, за которым следуют имя и два списка — inside и outside:

```
text
boundary кантовский_переход {
inside: [явление, созерцание, форма]
outside: [вещь_сама_по_себе, ноумен]
}
```

inside и outside — списки идентификаторов в квадратных скобках. Они указывают, какие субстанции находятся внутри границы, а какие снаружи. Идентификаторы в списках могут ссылаться на уже объявленные субстанции или на новые понятия.

Границы становятся особенно важны в полифоническом режиме, где контексты могут существовать параллельно.

5.6. Объявление противоречия

Узел противоречия (TensionNode) объявляется ключевым словом tension, за которым следуют идентификаторы двух полюсов и причина:

```
text
tension свобода vs необходимость reason "антиномия"
```

Ключевое слово vs разделяет полюса, reason предваряет строку с причиной противоречия. Обе субстанции должны существовать в контексте.

В отличие от оператора hold (который создаёт противоречие в статусе held), объявление tension просто фиксирует факт противоречия без указания статуса. Статус по умолчанию — held.

5.7. Операторы

Четыре оператора Грамматической Машины имеют прямой синтаксис в языке.

Hold — удержать противоречие:

```
text
hold tension_01
```

Идентификатор после hold — это существующий TensionNode. Оператор переводит его в статус held (если он ещё не в нём) и фиксирует.

Split — разделить контекст:

text
split {
// ветвь A
substance дух { energy: 0.9 }
} {
// ветвь B
substance материя { energy: 0.3 }
}

Каждая ветвь — это блок в фигурных скобках. Ветви выполняются в изолированных контекстах: изменения в одной ветви не видны в другой.

Resolve — разрешить противоречие:

text
resolve tension_01 with свобода

Ключевое слово with указывает, в пользу какого полюса разрешается противоречие. Полюс должен быть одной из субстанций, участвующих в TensionNode.

Grammar_change — сменить режим:

text
grammar_change polyphonic
grammar_change reactor

Допустимые значения: reactor (реакторный режим) и polyphonic (полифонический режим).

5.8. Полный пример скрипта

Вот скрипт на GrammaLang, анализирующий антиномию свободы и необходимости:

text
// Анализ антиномии чистого разума
substance свобода {
energy: 0.8
}
substance необходимость {
energy: 0.4
}
tension антиномия vs свобода vs необходимость reason "антиномия чистого разума"
modus разум полагает
modus разум мыслит
// Удерживаем противоречие в полифоническом режиме
grammar_change polyphonic
hold антиномия
// Затем переключаемся в реакторный и разрешаем
grammar_change reactor
resolve антиномия with свобода

Этот скрипт создаёт две субстанции, одно противоречие, два модуса, переключает режим, удерживает противоречие, снова переключает режим и разрешает противоречие. При исполнении он построит онтологический контекст, который затем можно экспортировать в JSON, MIDI или граф.

5.9. Формальная грамматика (EBNF)

Полная грамматика языка в расширенной форме Бэкуса-Наура приведена в Приложении Б. Здесь — её упрощённая версия для понимания структуры:

```
text
program = { statement } ;
statement = substance_decl | modus_decl | boundary_decl
| tension_decl | operation ;
substance_decl = "substance" ID "{" "energy" ":" NUMBER "}" ;
modus_decl = "modus" ID ID ;
boundary_decl = "boundary" ID "{" "inside" ":" "[" { ID "," } "]"
"outside" ":" "[" { ID "," } "]" "}" ;
tension_decl = "tension" ID "vs" ID "reason" STRING ;
operation = hold_op | split_op | resolve_op | grammar_change_op ;
hold_op = "hold" ID ;
split_op = "split" "{" { statement } "}" "{" { statement } "}" ;
resolve_op = "resolve" ID "with" ID ;
grammar_change_op = "grammar_change" ("reactor" | "polyphonic") ;
```

5.10. Развитие синтаксиса

В версии 0.3 планируется полноценный парсер на Rust, который будет принимать скрипты GrammaLang из командной строки:

```
text
PS> grammalang run script.gl
```

Также планируется поддержка выражений для энергии (вычисление вместо константы), условное создание противоречий и шаблоны грамматик — переиспользуемые блоки, которые можно включать в другие скрипты.

Глава 5.11. ГМ-диагностика: когда машина ломается.

Грамматическая машина — это не магический прибор, который всегда выдаёт правильный результат. Это инженерная система, и, как любая система, она может давать сбои. Диагностика этих сбоев — не второстепенная задача, а неотъемлемая часть работы с ГМ. Без неё аналитик рискует принять ложную кристаллизацию за истину, а ложное замыкание — за разрешение.

В этой главе мы рассмотрим четыре типа сбоев, соответствующих четырём операторам ГМ, и научимся их диагностировать. Каждый сбой имеет свои симптомы, свои причины и свои способы исправления.

5.11.1. Сбой Split: бесконечное ветвление.

Симптомы. Анализ не продвигается. Вместо того чтобы перейти к синтезу или кристаллизации, программа или агент создаёт всё новые и новые ветви. Онтологическая карта разрастается, но не становится глубже. TensionNode не создаются — или создаются, но не удерживаются, а просто множатся.

Причина. Оператор split применён без критерия остановки. Агент или программист не определил, когда ветвление должно прекратиться. Это похоже на рекурсию без базового случая: каждое новое различие порождает следующее, и процесс уходит в бесконечность.

Диагностика. Проверьте, задан ли критерий полноты для каждой ветви. В реакторном режиме критерий полноты — это завершение такта Облучения: все значимые элементы идентифицированы, все связи установлены. В полифоническом режиме — завершение такта Конденсации: все голоса выделены, все точки сгущения найдены.

Если критерий полноты не задан — задайте его. Если задан, но не соблюдается — проверьте, не слишком ли он строг (требует абсолютной полноты, которая недостижима) или не слишком ли слаб (позволяет завершить ветвь, не обработав существенные данные).

Исправление. Введите явный оператор collect или merge после split. Это не просто техническое действие — это онтологическое решение: мы заявляем, что ветви завершены и их результаты готовы к сборке. Если ветви принципиально не могут быть завершены (например, анализ бесконечного потока данных), используйте оператор throttle — ограничение количества ветвей или времени анализа.

5.11.2. Сбой Hold: вечное удержание.

Симптомы. TensionNode создаются, но никогда не разрешаются и не эскалируются. Онтологическая карта заполняется узлами напряжения, которые не влияют на дальнейший анализ. Агент или программа «застревает» в состоянии удержания, не переходя к следующим операторам.

Причина. Оператор hold применён без стратегии разрешения или эскалации. Это классическая ошибка, особенно в полифоническом режиме: аналитик или агент воспринимает «удержание» как разрешение ничего не делать. Но удержание — это активное действие: оно требует периодической проверки, не изменились ли условия, не появилась ли новая информация, которая позволяет разрешить противоречие.

Диагностика. Проверьте, задан ли у каждого TensionNode:

timeout — предельное время удержания;

escalate — условие, при котором удержание переходит в эскалацию;

review_interval — периодичность перепроверки условий.

Если ни одно из этих полей не задано — TensionNode «зависнет» навсегда. Если заданы, но не срабатывают — проверьте, не слишком ли жёсткие условия (никогда не наступают) или не слишком ли мягкие (срабатывают сразу, превращая удержание в разрешение).

Исправление. Для каждого TensionNode определите стратегию:

resolve_when — условие разрешения (например, поступление дополнительной информации);

escalate_after — время или событие, после которого узел передаётся на другой уровень;

periodic_review — регулярная проверка, не изменились ли условия.

В полифоническом режиме допустимо, чтобы некоторые TensionNode удерживались бесконечно — это философский выбор. Но этот выбор должен быть **осознанным**, а не случайным. Если вы удерживаете противоречие бесконечно, вы должны знать, почему, и это должно быть зафиксировано в reason.

5.11.3. Сбой Transition: потеря данных при подъёме.

Симптомы. При переходе с одного уровня абстракции на другой теряется существенная информация. На онтологическом уровне исчезают детали, которые были важны на лексическом. Или, наоборот, на онтологическом уровне появляются сущности, которые не имеют обоснования на нижних уровнях (галлюцинации онтологии).

Причина. Правило трансформации (via) не обеспечивает сохранение онтологической связности. Данные нижнего уровня переупаковываются в категории верхнего, но при этом часть информации отбрасывается как «несущественная». Ошибка в том, что критерий существенности определён неверно.

Диагностика. Проверьте цепочку происхождения каждой онтологической сущности. На онтологическом уровне каждая Substance должна иметь origin — ссылку на сущность или данные с нижнего уровня, из которых она была сконструирована. Если у Substance нет origin — это галлюцинация. Если origin есть, но не содержит всей необходимой информации — потеря данных.

Также проверьте, не нарушено ли правило соседства: переход возможен только между соседними уровнями. Если вы перешли с Lexical сразу на Ontological, минуя Syntactic, Semantic и Pragmatic, потеря данных неизбежна — потому что данные не были структурированы и интерпретированы на промежуточных уровнях.

Исправление. Используйте цепочки переходов вместо одного прыжка:

text

transition Lexical to Syntactic via group_into_structures

then to Semantic via interpret_as_values

then to Pragmatic via contextualize_as_actions

then to Ontological via reformulate_as_entities;

Каждый переход сохраняет origin в Modus целевой сущности. На онтологическом уровне можно проследить всю цепочку: от сущности до исходных токенов.

Если потеря данных неизбежна (например, при анализе больших объёмов, где детали не помещаются в онтологическую карту), используйте оператор summarize — он создаёт резюме, но не теряет ссылку на исходные данные.

5.11.4. Сбой Grammar Change: режимный джиттер.

Симптомы. Система переключается между реакторным и полифоническим режимами слишком часто. Онтологическая карта меняет интерпретацию TensionNode каждые несколько тактов. Анализ становится нестабильным: вчера TensionNode был held, сегодня — resolved, завтра — снова held, без явной причины.

Причина. Критерии переключения (grammar_change when) определены слишком чувствительно. Любое##ное изменение контекста вызывает смену режима. Это особенно опасно, когда система автоматически переключается на основе эвристик, которые не были тщательно откалиброваны.

Диагностика. Проверьте историю переключений. В ресурсах MCP хранится GrammarChangeRecord для каждого переключения. Если переключения происходят чаще, чем раз в N тактов (где N — средняя длительность такта), это джиттер.

Также проверьте, не являются ли переключения циклическими: Reactor → Polyphonic → Reactor → Polyphonic без содержательных изменений в онтологической карте. Это явный признак того, что критерии переключения конфликтуют друг с другом.

Исправление. Введите гистерезис: для переключения из реакторного в полифонический требуется, чтобы критерий сохранялся в течение K тактов (например, 3). Для обратного переключения — аналогично. Это предотвращает дребезг.

Также добавьте явный критерий стабильности: grammar_change when (plasma_duration > 3_cycles) and (not switching_back_immediately). Это гарантирует, что режим меняется только после того, как система «убедится», что изменение действительно необходимо.

5.11.5. Композитные сбои: когда всё ломается вместе.

На практике сбои редко происходят поодиночке. Бесконечное ветвление (Split) приводит к перегрузке онтологической карты, что провоцирует вечное удержание (Hold). Потеря данных при переходе (Transition) создаёт галлюцинации на онтологическом уровне, которые вызывают ложные срабатывания Grammar Change. Возникает каскадный сбой.

Диагностика каскадного сбоя. Проверьте три индикатора:

Рост онтологической карты: если количество Substance и TensionNode растёт экспоненциально, а не линейно — это признак бесконечного ветвления.

Старение TensionNode: если средний возраст TensionNode (время с момента создания) превышает ожидаемый — это признак вечного удержания.

Частота переключений: если переключений больше, чем тактов анализа — это признак джиттера.

Если все три индикатора положительны, система находится в каскадном сбое.

Исправление. Остановите анализ. Зафиксируйте состояние онтологической карты как аварийный дамп. Выполните обратную трассировку: найдите последний корректный такт (где все три индикатора были в норме) и восстановите состояние до него. Затем перезапустите анализ с более строгими критериями остановки, разрешения и переключения.

5.11.6. Профилактика: иммунная система онтологии.

Лучший способ справиться со сбоями — не допускать их. Иммунная система онтологии, описанная в Главе 4.6, является профилактическим механизмом. Но её можно усилить тремя дополнительными практиками.

Практика 1: Регулярная ревизия TensionNode. Периодически (каждые N тактов) проверяйте все TensionNode. Для каждого узла задайте вопрос: «Это противоречие всё ещё актуально? Появилась ли новая информация, которая позволяет его разрешить? Не пора ли его эскалировать?» Ревизия — это оператор hold в действии: не пассивное ожидание, а активное удержание.

Практика 2: Аудит переходов. После каждого transition проверяйте, не потеряны ли данные. Сравнивайте количество сущностей на исходном и целевом уровнях. Если на целевом уровне сущностей значительно меньше, чем на исходном — возможно, вы потеряли информацию. Аудит — это оператор transition в действии: не просто переход, а переход с проверкой.

Практика 3: Мониторинг режимной стабильности. Ведите журнал переключений. Если переключений больше, чем тактов, или они образуют циклы, сигнализируйте об этом. Мониторинг — это оператор grammar_change в действии: не просто переключение, а переключение с осознанием.

Конец ознакомительного фрагмента.