Министерство науки и высшего образования Российской Федерации Федеральное государственное бюджетное образовательное учреждение высшего образования «Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)» (МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Теоретическая информатика и компьютерные технологии»

ОТЧЕТ ПО ПРЕДДИПЛОМНОЙ ПРАКТИКЕ:

Студент _____

(Фамилия, Имя, Отчество)

Группа _____

Тип практики _____

Название предприятия _____

Студент И У 9-81Б _____ Козочкин М. С.
(Группа) (Подпись, дата) (И.О. Фамилия)

Рекомендуемая оценка _____

Руководитель НИР _____ Коновалов А. В.
(Подпись, дата) (И.О. Фамилия)

Оценка _____

2025 г.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Современные веб-приложения для разработки и обучения программированию требуют редакторы кода с поддержкой синтаксической подсветки. Качественная раскраска не только улучшает читаемость кода, но и помогает быстрее находить синтаксические ошибки, что особенно важно для образовательных платформ.

Веб-сервер поддержки учебного процесса кафедры ИУ9 МГТУ им. Н.Э. Баумана [1] предоставляет студентам возможность решать и проверять решения задач по программированию непосредственно в браузере. Однако существующий редактор кода на сервере имеет ограниченные возможности по раскраске синтаксиса в разрезе работы с большими текстами и потенциала для развития. Основная проблема заключается в том, что традиционные подходы к раскраске синтаксиса в веб-окружении требуют полного перерасчета подсветки при каждом изменении текста.

Целью данной работы является разработка и реализация эффективного алгоритма раскраски синтаксиса для веб-редактора.

В рамках работы был разработан подход, основанный на комбинации автомата Мили для лексического анализа и структуры данных Rope для эффективного стилями в тексте. Это позволило достичь логарифмической сложности операций обновления при сохранении простоты конфигурации через регулярные выражения.

1. Обзор предметной области и постановка задачи

1.1. Проблематика раскраски синтаксиса в веб-редакторах

Раскраска синтаксиса является базовой функциональностью современных редакторов кода. В контексте веб-приложений эта задача усложняется рядом специфических ограничений:

- ограничения производительности JavaScript. В отличие от нативных приложений, веб-редакторы выполняются в однопоточной среде JavaScript, где длительные вычисления блокируют пользовательский интерфейс. Это делает критически важным минимизацию времени, затрачиваемого на анализ и раскраску кода;
- высокая стоимость манипуляций с DOM. Обновление визуального представления текста в браузере требует изменения DOM-дерева, что является одной из самых затратных операций;
- необходимость инкрементальности. Пользователи ожидают мгновенной реакции на ввод текста. Это означает, что система должна обновлять только те части подсветки, которые действительно изменились, а не пересчитывать всю раскраску заново.

Существует три основных подхода к реализации раскраски синтаксиса в веб-редакторах:

- подход на основе регулярных выражений (используется в простых редакторах);
- на основе полного синтаксического анализа (примеры: Monaco Editor [2], CodeMirror6 [3]);
- на основе конечных автоматов (пример: TextMate grammars [4]);

1. 2. Требования к разрабатываемому решению

На основе анализа предметной области и специфики целевого применения были сформулированы требования к системе раскраски синтаксиса. Функциональные:

- поддержка раскраски для языков программирования, используемых в учебном процессе: C, C++, Go, Python (частично), Java, Scheme, Refal

(частично), Markdown (частично), Plaintext (частично);

- корректная обработка основных лексических элементов: ключевые слова, идентификаторы, числа, строки, комментарии, операторы;
- инкрементальное обновление подсветки при редактировании текста;
- возможность легкого добавления поддержки новых языков через конфигурационные файлы.

Нефункциональные:

- время обновления подсветки при вводе символа должно быть достаточно малым для обеспечения плавной работы интерфейса (60 кадров в секунду, что соответствует времени отрисовки одного кадра не более 16.67 миллисекунд);
- минимальное потребление памяти по сравнению с хранением простого текста;
- простота интеграции в существующие веб-приложения;
- отсутствие внешних зависимостей от сторонних библиотек раскраски.

1. 3. Постановка задачи

Требуется разработать и реализовать систему раскраски синтаксиса для веб-редактора кода, которая:

- использует эффективные алгоритмы и структуры данных для обеспечения высокой производительности как при начальной раскраске, так и при инкрементальных обновлениях;
- предоставляет простой декларативный способ описания правил раскраски для различных языков программирования через регулярные выражения;
- минимизирует количество операций с DOM за счет обновления только действительно изменившихся фрагментов текста;
- масштабируется для работы с большими файлами без существенной деградации производительности;
- легко интегрируется в веб-приложения как самостоятельный компонент с минимальными зависимостями.

Для решения поставленной задачи предлагается использовать комбинацию:

- автомата Мили — для эффективного лексического анализа на основе объединенного ДКА, построенного из регулярных выражений;
- структуры данных Rope — для представления текста и ассоциированных стилей с поддержкой эффективных локальных модификаций.

Такой подход позволит достичь оптимального баланса между производительностью, точностью раскраски и простотой конфигурации.

2. Разработка

2. 1. Анализ требований и выбор подхода

Проектирование системы требует анализа специфических ограничений браузерного окружения и потребностей целевой аудитории. Основной задачей является необходимость обработки больших объемов текста без блокировки пользовательского интерфейса при сохранении раскраски и простоты конфигурации.

Специфика образовательной платформы накладывает дополнительные требования. Студенты часто работают с кодом, содержащим синтаксические ошибки, неполные конструкции и нестандартное форматирование. Система должна быть устойчива к таким ситуациям.

В то же время, сложные системы парсинга текста в сложные структуры типа AST (Abstract Syntax Tree), используемые в профессиональных IDE, избыточны для образовательных задач и требуют значительных ресурсов на разработку и поддержку. Кроме того, многие современные решения полагаются на внешние сервисы языкового анализа (Language Server Protocol), что создает дополнительные зависимости.

Оптимальным решением является комбинация классических алгоритмов лексического анализа с современными структурами данных, обеспечивающими эффективные локальные модификации. Такой подход позволяет достичь баланса между производительностью, точностью и простотой реализации.

2. 2. Теоретические основы проектируемой системы

2. 2. 1. Процесс лексического анализа

В основе системы лежит модель лексического анализа, базирующаяся на теории регулярных языков. Каждый тип лексемы языка программирования описывается регулярным выражением, задающим множество строк, которые могут быть классифицированы как данный тип токена. Совокупность всех правил формирует лексическую спецификацию языка.

Формально, лексическая спецификация представляется как множество

пар (r, s), где r — регулярное выражение, а s — стиль оформления. Задача лексического анализатора состоит в нахождении покрытия входной строки w непересекающимися подстроками $w = w_1 w_2 \ldots w_n$ такими, что каждая подстрока w принадлежит языку некоторого регулярного выражения из спецификации. При этом должен соблюдаться принцип максимального захвата и приоритета правил при конфликтах.

Ключевой задачей является эффективное разбиение входного текста на последовательность токенов с одновременным определением их типов. Традиционный подход с последовательной проверкой каждого регулярного выражения имеет квадратичную сложность в худшем случае. Вместо этого предлагается построение объединенного конечного автомата, способного распознавать все типы токенов за один проход по w.

Процесс построения такого автомата включает несколько этапов. Сначала каждое регулярное выражение преобразуется в недетерминированный конечный автомат (НКА) используя алгоритм рекурсивного спуска. Важной особенностью этого процесса является сохранение структурной информации о том, какому исходному выражению (стилю) соответствует каждое конечное состояние.

Далее все построенные НКА объединяются в единый автомат путем добавления нового начального состояния с ε-переходами к начальным состояниям отдельных автоматов. Полученный объединенный НКА способен распознавать объединение всех языков, но его недетерминированная природа делает прямое использование неэффективным.

Следующим шагом является детерминизация с использованием алгоритма построения подмножеств (subset construction). Каждое состояние результирующего ДКА соответствует множеству состояний исходного НКА, достижимых по одной и той же входной последовательности. При этом сохраняется информация о том, какие конечные состояния НКА входят в каждое состояние ДКА, что позволяет определить, какие типы токенов могут быть распознаны в данной позиции.

2. 2. 2. Выбор модели автомата и обработка конфликтов

Для реализации лексического анализатора была выбрана модель автомата Мили, генерирующего выходные сигналы на переходах между состояниями. Это решение обусловлено естественным соответствием между процессом токенизации и работой такого автомата: выходной сигнал (тип токена) генерируется именно в момент завершения распознавания лексемы, что упрощает определение границ токенов и их классификацию.

Преобразование ДКА в автомат Мили требует обработку ситуаций, когда одно состояние ДКА содержит несколько конечных состояний исходных НКА. Такие конфликты разрешаются на основе системы приоритетов, учитывающей порядок правил в спецификации.

В отличие от классического ДКА, требующего дополнительной логики для отслеживания начала и конца токенов, автомат Мили инкапсулирует эту функциональность в своей структуре. Каждый переход, ведущий в состояние, которое было конечным в исходном ДКА, генерирует сигнал о распознанном токене. При этом важно обеспечить корректную обработку ситуаций, когда за распознанным токеном сразу следует начало нового без разделителей.

Автомат должен корректно различать последовательности символов, которые могут интерпретироваться как один составной токен или несколько отдельных. Это достигается правильным построением приоритетов и использованием контекстной информации о предыдущих переходах.

2. 2. 3. Хранения текста программы

Традиционные строковые представления в JavaScript являются неизменяемыми, что приводит к необходимости полного копирования при любой модификации. Для текста размером n символов операции вставки или удаления имеют сложность $O(n)$.

Анализ альтернативных структур данных показал преимущества использования Rope [5] — сбалансированного двоичного дерева для представления строк. В классической реализации Rope каждый узел дерева содержит либо небольшую строку (в листовых узлах), либо ссылки на

дочерние узлы с информацией о суммарной длине левого поддерева. Это обеспечивает логарифмическую сложность основных операций, таких как вставка или удаление подстроки, при сохранении эффективного использования памяти.

Для задачи раскраски синтаксиса каждый листовой узел расширяется для хранения не только текстового фрагмента, но и информации о его стиле. Поддерживается инвариант: весь текст в одном листовом узле имеет единообразное оформление. Это позволяет эффективно хранить информацию о раскраске без дублирования данных о стиле для каждого символа.

Важным аспектом проектирования является выбор оптимального размера листовых узлов. Слишком маленькие узлы приводят к избыточной фрагментации и увеличению высоты дерева, что негативно влияет на производительность. Слишком большие узлы снижают эффективность локальных операций.

Балансировка дерева осуществляется путем использования алгоритма построения красно-черного дерева. Использование именно этого типа дерева подходит под задачу раскраски, поскольку поиск по дереву и его изменение в процессе редактирования кода программы происходит с примерно равной регулярностью. Альтернатива – АВЛ-дерево, позволяющее держать дерево в более сбалансированном виде, но при этом тратя сильно больше ресурсов на перебалансировку, что будет негативно сказываться на общей производительности при частых изменениях дерева

2. 3. Архитектура

2. 3. 1. Используемые принципы

Архитектура системы строится на принципах модульности, разделения ответственности и минимизации связности между компонентами. Каждый модуль инкапсулирует определенную функциональность и взаимодействует с другими через интерфейсы. Это обеспечивает возможность независимой разработки, тестирования и модификации отдельных частей системы.

Применяется шаблон проектирования "наблюдатель" для организации

взаимодействия между компонентами Rope и визуализации. Структура данных Rope выступает в роли субъекта, уведомляющего компоненту визуализации об изменениях в дереве. Это позволяет отделить логику управления данными от логики их визуализации и обеспечивает расширяемость системы.

2. 3. 2. Компоненты системы

Система организована в виде трех основных подсистем с четко определенными границами и интерфейсами взаимодействия.

Подсистема лексического анализа представляет собой изолированный модуль, отвечающий за все аспекты работы с формальными языками. Она включает компоненты для парсинга регулярных выражений, построения и оптимизации автоматов, а также выполнения токенизации.

Внутренняя организация подсистемы следует принципу разделения этапов обработки. Компонент построения автоматов работает только на этапе инициализации или при изменении конфигурации. Компонент токенизации оптимизирован для многократного выполнения и использует предварительно построенные структуры данных.

Подсистема хранения текста реализует адаптированную структуру Rope и обеспечивает эффективные операции над текстом и ассоциированными стилями. Она предоставляет высокоуровневый API для работы с текстом, скрывая детали сложного древовидного представления. Ключевой функцией является отслеживание изменений и определение минимальных областей, требующих перерасчета раскраски.

Подсистема интеграции с пользовательским интерфейсом выполняет роль координатора, связывающего внутреннее представление с визуальным отображением в браузере. Она отвечает за трансляцию событий пользовательского ввода в операции над структурой данных, оптимизацию обновлений DOM и управление вспомогательными элементами интерфейса, такими как курсор и выделение.

2. 4. Проектирование алгоритма инкрементальной раскраски

2. 4. 1. Теоретические основы

Инкрементальный алгоритм раскраски основывается на паринципе локальности изменений и минимизации объема повторных вычислений. Теоретической основой является наблюдение, что в контексте лексического анализа изменение в позиции i может повлиять только на токены, начинающиеся не ранее позиции $i - k$, где k — максимальная длина токена в языке. Для большинства языков программирования k является небольшой константой, что позволяет ограничить область перерасчета.

Процесс инкрементального обновления начинается с определения начальной области перерасчета. Для этого используется структура дерева Rope. Алгоритм сопоставляет с областью изменения текста набор листьев дерева, содержащих измененный текст

После определения непосредственно затронутых узлов выполняется их лексический анализ с использованием автомата Мили. При этом важно правильно инициализировать состояние автомата. Если предыдущий узел заканчивается внутри многострочной конструкции (например, внутри комментария), автомат должен начать работу в соответствующем состоянии, а не в начальном.

Результатом анализа является новая последовательность токенов для обработанных узлов. Если эта последовательность отличается от предыдущей по количеству токенов или их границам, может потребоваться реструктуризация дерева — разделение узлов для поддержания инварианта о единообразном стиле внутри узла.

После обработки непосредственно измененных узлов запускается механизм проверки распространения изменений. Основная идея заключается в сравнении состояния автомата на границах узлов до и после изменения. Если состояние изменилось, это означает, что изменение может повлиять на последующие узлы.

Важным аспектом является обработка обратного распространения. Хотя в большинстве случаев изменения распространяются только вперед по тексту, существуют ситуации, когда необходимо перепроверить предыдущие узлы.

Для гарантии завершения алгоритма вводится механизм обнаружения стабилизации. Если обработка узла не приводит к изменению его токенизации и конечного состояния автомата, распространение в данном направлении прекращается. Учитывая конечность числа узлов и монотонность распространения изменений, алгоритм гарантированно завершается.

3. Реализация

Реализация системы раскраски синтаксиса представляет собой комплекс взаимосвязанных модулей, каждый из которых отвечает за определенный аспект функциональности. В данном разделе детально рассматриваются ключевые компоненты системы, их внутреннее устройство, структуры данных и механизмы взаимодействия.

3. 1. Построение конечных автоматов

3. 1. 1. Реализация недетерминированного конечного автомата

Модуль `nfa.js` содержит реализацию недетерминированного конечного автомата, являющегося основой для построения лексического анализатора. Класс `NFA` инкапсулирует структуру автомата и предоставляет методы для его построения и манипулирования.

Основными компонентами класса `NFA` являются множество состояний, таблица переходов, реализованная через вложенные `Map` структуры для удобства отладки, начальное состояние и множество принимающих состояний. Каждое состояние автомата представлено объектом с уникальным идентификатором и опциональной информацией о связанном с ним стиле токена.

Построение НКА из регулярного выражения выполняется алгоритмом рекурсивного спуска. Парсер поддерживает следующий набор операций регулярных выражений:

- альтернатива `|` — выбор между несколькими вариантами;
- группировка `()` — управление приоритетом операций;
- конкатенация — последовательное соединение элементов (неявная операция);
- замыкание Клини `*` — ноль или более повторений;
- плюс `+` — одно или более повторений;
- знак вопроса `?` — ноль или одно повторение;
- множества в квадратных скобках `[abc]` — любой символ из множества;
- диапазоны символов `[a-z]`, `[0-9]`, `[A-Z]`, `[…-…]`;

- отрицание множества `[^0-9]` — любой символ, кроме указанных;
- комбинированные классы `[a-zA-Z0-9_]`;
- `\d` — любая цифра (эквивалент `[0-9]`);
- `\w` — словесный символ (буквы, цифры и подчеркивание);
- `\s` — пробельный символ (пробел, табуляция, перенос строки);
- точка `.` — любой символ;
- экранирование — обратный слеш `\` для экранирования специальных символов.

Для каждой операции создается соответствующая структура состояний и переходов: простые символы преобразуются в автоматы из двух состояний с одним переходом, конкатенация реализуется соединением конечного состояния первого автомата с начальным состоянием второго через эпсилон-переход, альтернатива создает новые начальное и конечное состояния с эпсилон-переходами к подавтоматам, квантификаторы добавляют циклические эпсилон-переходы для реализации повторений.

3. 1. 2. Объединение автоматов и детерминизация

Статический метод `NFA.union` используется для объединения нескольких НКА в единый автомат при построении лексического анализатора. Метод создает новое начальное состояние с эпсилон-переходами ко всем начальным состояниям объединяемых автоматов, сохраняя при этом информацию о стилях и приоритетах токенов. Парсер включает проверки корректности выражения, выбрасывая исключения при обнаружении пустых групп, пустых альтернатив или пустых символьных классов, что обеспечивает раннее обнаружение ошибок в конфигурации правил раскраски.

Детерминизация НКА выполняется алгоритмом построения подмножеств, реализованным в модуле `dfa.js`. Класс `DFA` представляет детерминированный конечный автомат с аналогичной НКА структурой, но с важным отличием — из каждого состояния по каждому символу существует не более одного перехода.

Алгоритм детерминизации работает следующим образом. Начальным

состоянием ДКА становится эпсилон-замыкание начального состояния НКА. Для каждого необработанного состояния ДКА, представляющего множество состояний НКА, и каждого символа алфавита вычисляется множество достижимых состояний НКА. Если полученное множество непусто, создается новое состояние ДКА и соответствующий переход. Процесс продолжается до обработки всех достижимых состояний.

3. 1. 3. Реализация автомата Мили

Центральным компонентом системы лексического анализа является автомат Мили, реализованный в модуле `mealy.js`. В отличие от обычного конечного автомата, автомат Мили генерирует выходные сигналы на переходах между состояниями, что отлично подходит для задачи раскраски синтаксиса.

Класс `MealyMachine` содержит следующие ключевые компоненты: массив всех состояний `allStates`, упорядоченный для удобства отладки. Таблица переходов `transitions`, представленная как вложенный объект, где первый уровень — исходное состояние, второй — входной символ, значение — целевое состояние. Начальное состояние `startState`, из которого начинается обработка каждой новой лексемы. Объект принимающих состояний `acceptStates`, сопоставляющий каждому принимающему состоянию соответствующий стиль токена.

Преобразование ДКА в автомат Мили выполняется статическим методом `MealyMachine.fromDFA`. Процесс включает копирование всех состояний и переходов из ДКА, сохранение информации о принимающих состояниях и связанных с ними стилях, установку начального состояния. Важной особенностью является сохранение информации о приоритетах стилей при наличии конфликтов.

3. 2. Структура данных Rope и управление деревом

Реализация структуры Rope в модуле `tree.js` представляет собой сложную иерархию классов, обеспечивающую эффективное хранение и модификацию текста с ассоциированной информацией о стилях.

Абстрактный базовый класс `AbstractNode` определяет общий интерфейс для всех узлов дерева. Класс содержит ссылку на автомат Мили, используемый для анализа текста, поле `signal` для хранения сигнала раскраски, генерируемого узлом, абстрактные методы `isLeaf()`, `getText()`, `leaves()`, которые должны быть реализованы в наследниках.

Важной особенностью базового класса является метод `log`, обеспечивающий структурированное логирование операций над деревом. Система логирования позволяет отслеживать последовательность вызовов методов, изменения состояний узлов и распространение сигналов, что критически важно для отладки сложных сценариев редактирования.

Класс `InternalNode` представляет внутренние узлы дерева. Каждый внутренний узел содержит ссылки на левое и правое поддеревья, массив функции действия `A`, вычисляемый на основе функций действия дочерних узлов, текущий сигнал, определяемый сигналами дочерних узлов, стартовое состояние, наследуемое от левого дочернего узла.

Класс `LeafNode` представляет листовые узлы, непосредственно содержащие текст. Помимо общих для всех узлов полей, листовые узлы содержат текстовое содержимое, стиль раскраски, применяемый к тексту узла, callback-функцию для уведомления внешних компонентов об изменениях, флаг `isPseudo` для обозначения специальных псевдо-узлов.

Функция действия `A` является ключевой оптимизацией, позволяющей эффективно вычислять состояния автомата без повторного прогона текста. Для каждого узла массив `A` содержит информацию о конечном состоянии автомата после обработки текста узла, начиная из каждого возможного начального состояния.

Функция `buildA` вычисляет массив A для заданной строки. Для пустой строки функция возвращает тождественное отображение. Для односимвольной строки выполняется один шаг автомата из каждого состояния. Для более длинных строк выполняется полный прогон автомата.

Функция `calculateA` выполняет композицию двух функций действия. Для каждого начального состояния вычисляется промежуточное состояние после обработки первой строки, затем конечное состояние после обработки второй строки из промежуточного состояния. Эта операция имеет линейную сложность относительно числа состояний автомата.

3. 2. 1. Механизм сигналов и распространение стилей

Класс `Signal` инкапсулирует информацию о стиле, который должен быть применен к лексеме. Сигналы генерируются узлами, которые определяют завершение лексемы, и распространяются в обратном направлении по дереву.

Метод `applyStyleFromSignal` в классе `InternalNode` реализует логику распространения сигналов. Если правый дочерний узел существует, сигнал передается ему. Если правый узел сам генерирует сигнал, это означает, что он является границей между лексемами, и сигнал не распространяется на левое поддерево. В противном случае сигнал передается и левому дочернему узлу.

В листовых узлах метод `applyStyleFromSignal` непосредственно устанавливает стиль узла и вызывает callback-функцию для уведомления о изменении. Это позволяет внешним компонентам, таким как редактор, немедленно обновить визуальное представление.

Метод `onTextChange` в классе `LeafNode` является центральной точкой обработки изменений текста. При изменении текста узла выполняется анализ нового текста с учетом текущего стартового состояния узла.

Функция `calculateFragments` определяет, как новый текст должен быть разбит на лексемы. Функция возвращает массив фрагментов, каждый из которых представляет отдельную лексему или часть лексемы. В зависимости от количества фрагментов применяется различная логика обработки. Если фрагментов нет, текст является продолжением предыдущей лексемы. Узел не генерирует сигнал, обновляется только функция действия. Если фрагмент один, текст завершает предыдущую лексему и начинает новую. Узел переходит в начальное состояние автомата и генерирует сигнал для раскраски

предыдущей лексемы. Если фрагментов несколько, текст содержит несколько лексем. Узел заменяется поддеревом, содержащим отдельные узлы для каждого фрагмента.

3. 2. 2. Уведомление об изменениях и каскадное обновление

Механизм уведомлений обеспечивает согласованность дерева при локальных изменениях. Метод `onChildChanged` вызывается дочерним узлом при изменении его состояния. Родительский узел анализирует характер изменения и определяет, требуется ли дальнейшее распространение уведомления вверх по дереву.

Для левого дочернего узла проверяется изменение конечного состояния. Если оно изменилось, правому дочернему узлу передается новое стартовое состояние через метод `applyNewStartState`. Для правого дочернего узла проверяется изменение сигнала. Если появился новый сигнал, он распространяется на левое поддерево.

После обработки изменений дочерних узлов внутренний узел пересчитывает собственные функцию действия и сигнал. Если эти значения изменились, уведомление передается родительскому узлу, обеспечивая каскадное обновление дерева.

3. 2. 3. Ограничение размера листовых узлов

Для оптимизации производительности операций над текстом реализовано ограничение на максимальный размер текста в листовых узлах. Типичное ограничение составляет 64-128 символов, что обеспечивает баланс между количеством узлов и стоимостью операций внутри узла.

При превышении лимита размера листовой узел автоматически разделяется. Алгоритм разделения учитывает границы лексем, определяемые функцией `calculateFragments`. Предпочтение отдается разделению на границах между лексемами, что минимизирует необходимость пересчета стилей.

Если разделение на границе лексемы невозможно (например, при очень

длинной строке-литерале), выполняется принудительное разделение. В этом случае создаются два и более новых узла, первый из которых наследует стартовое состояние исходного узла, а второй получает стартовое состояние, вычисленное на основе функции действия предыдущего узла.

3. 3. Интеграция с редактором

Класс `Editor` в модуле `editor.js` обеспечивает полнофункциональный текстовый редактор с поддержкой раскраски синтаксиса. Редактор интегрирует структуру данных Rope с пользовательским интерфейсом браузера.

Редактор построен на основе contenteditable-элемента, что обеспечивает нативную поддержку текстового ввода браузером. Однако вместо прямого редактирования DOM, все изменения проходят через структуру Rope, что гарантирует согласованность данных и корректность раскраски.

Основными компонентами редактора являются DOM-элемент редактора, содержащий визуальное представление текста, скрытое поле ввода для перехвата событий клавиатуры в случаях, когда contenteditable-недостаточно, корневой узел дерева Rope, содержащий структурированное представление текста, двунаправленные карты для связи листовых узлов и DOM-элементов, карты координат для эффективного преобразования между позициями в тексте и визуальными координатами.

3. 3. 1. Управление DOM-элементами

Каждому листовому узлу дерева соответствует элемент `<span>` в DOM. Связь поддерживается через отображения: `leafSpanMap` отображает листовые узлы на соответствующие span-элементы, `spanLeafMap` выполняет обратное отображение.

Метод `createSpanForLeaf` создает span-элемент для листового узла. Элементу присваивается текстовое содержимое узла, CSS-класс, соответствующий стилю токена, data-атрибут для хранения информации о стиле. Созданный элемент добавляется в обе карты для поддержания

двусторонней связи.

При изменении стиля листа через механизм сигналов метод `updateLeafStyle` обновляет соответствующий span-элемент. Старый CSS-класс удаляется, новый добавляется, обновляется data-атрибут. Это обеспечивает мгновенное визуальное обновление без необходимости перестроения DOM- структуры.

3. 3. 2. Обработка пользовательского ввода

Редактор перехватывает и обрабатывает различные типы пользовательского ввода. Метод `handleInput` обрабатывает ввод обычных символов. При вводе символа определяется текущая позиция курсора, находится листовой узел в этой позиции, вычисляется смещение внутри узла, обновляется текст узла с вставленным символом, вызывается метод `onTextChange` узла. При удалении последнего символа в узле сам узел удаляется из дерева.

Система координат редактора поддерживает преобразование между различными представлениями позиции: абсолютная позиция в тексте (количество символов от начала), позиция в строке и столбце, визуальные координаты в пикселях.

Метод `rebuildCoordinateMaps` пересчитывает карты координат после изменения структуры текста. Для каждого span-элемента вычисляются начальная и конечная позиции в тексте, номер строки, визуальные координаты. Информация сохраняется в картах для быстрого доступа.

Метод `updateCursorDisplay` обновляет визуальное представление курсора. На основе текущей позиции курсора вычисляются строка и столбец, определяются визуальные координаты с учетом размера символов, позиционируется элемент курсора.

Для обеспечения эффективной работы редактора реализован принцип инкрементальных обновлений. При изменении текста обновляются только непосредственно затронутые span-элементы и их стили. Структура Rope

позволяет локализовать изменения и минимизировать количество операций с DOM. При вводе символа изменяется только один листовой узел и соответствующий ему span-элемент. Механизм сигналов обеспечивает обновление стилей только тех узлов, которые действительно изменились.

3. 3. 3. Процесс построения анализатора

На основе правил раскраски строится лексический анализатор. Процесс включает следующие этапы. Для каждого правила создается НКА. Все НКА объединяются в единый автомат с сохранением информации о стилях. Объединенный НКА преобразуется в ДКА для эффективного распознавания. ДКА преобразуется в автомат Мили для генерации сигналов раскраски. Построенный автомат кэшируется для повторного использования. При изменении правил раскраски автомат пересобирается.

3. 3. 4. Инициализация и жизненный цикл системы

Модуль `main.js` координирует инициализацию и работу всей системы раскраски синтаксиса. При загрузке страницы выполняется следующая последовательность действий. Загружаются правила раскраски для выбранного языка из модуля `rules.js`. На основе правил строится лексический анализатор: создаются НКА для каждого правила, выполняется объединение и детерминизация, создается автомат Мили.

Инициализируется структура данных: создается корневое дерево Rope с псевдо-узлами, подготавливаются структуры для хранения узлов. Создается и настраивается редактор: инициализируется класс Editor с привязкой к DOM-элементам, устанавливаются обработчики событий, выполняется начальная отрисовка.

Если предоставлен начальный текст, он загружается в редактор: текст анализируется автоматом Мили, создается дерево с соответствующими узлами, выполняется начальная раскраска.

3. 3. 5. Обработка изменения языка

При изменении языка программирования система выполняет следующие действия. Загружаются новые правила раскраски для выбранного языка. Перестраивается лексический анализатор с новыми правилами. Сохраняется текущий текст из редактора. Очищается существующее дерево и связанные структуры. Создается новое дерево с текстом, проанализированным новым анализатором. Обновляется визуальное представление в редакторе.

4. Тестирование и демонстрация работы системы

4.1. Сравнение производительности

В рамках работы было проведено сравнительное исследование двух подходов к раскраске синтаксиса: простого подхода на основе применения набора независимых регулярных выражений, используемого в данный момент в редакторе кода сервера тестирования ИУ9 [1] и подхода на основе компиляции регулярных выражений в автомат Мили с последующим его применением к тексту. Тестирование проводилось на синтетически сгенерированном коде языка Go и заточенным под него набором регулярных выражений различного размера (от 100 до 100000 строк). Для каждого подхода измерялось следующие характеристики среднее время обновления при редактировании.

Тестирование проводилось в окружении NodeJS с моделированием реальных условий редактирования текста. Для каждого размера входных данных генерировалась серия случайных изменений, включающих вставку, удаление и замену символов в различных позициях текста. На рис. 1 представлен график зависимости времени обновления раскраски от размера текста. К примеру, на файле размером 100000 строк среднее время раскраски составило:

- регулярные выражения: 366 мс;
- автомат Мили: 39 мс.

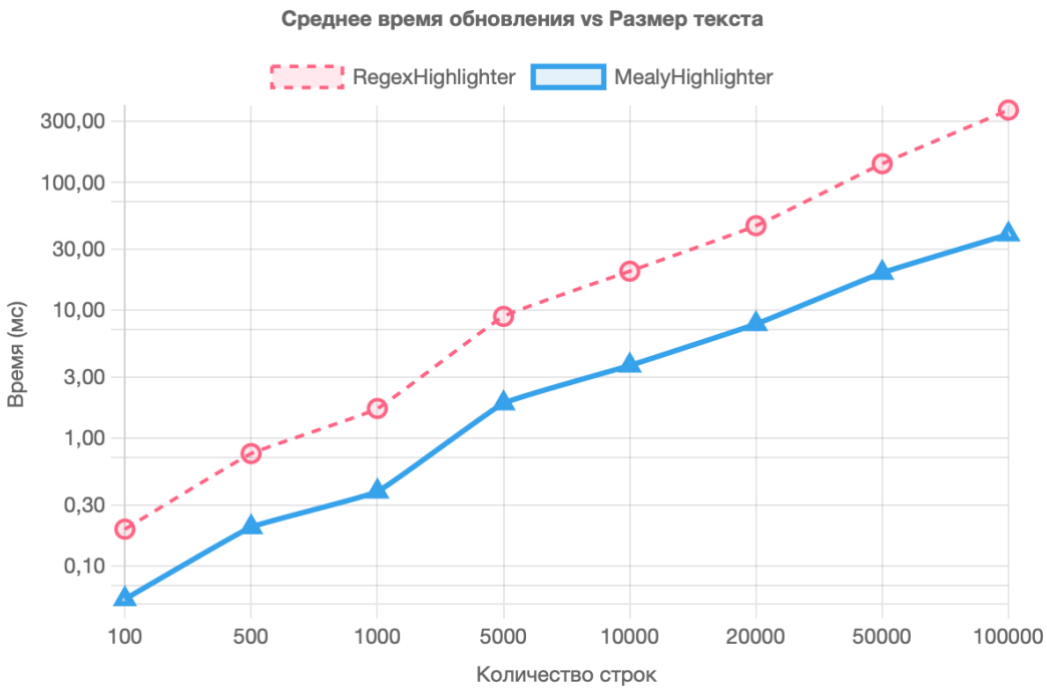

Рисунок 1 – зависимость времени обновления раскраски от размера текста для автомата Мили (MealyHighlighter) и набора регулярных выражений (RegexHighlighter)

Такая разница объясняется тем, что автомат Мили имеет линейную зависимость времени раскраски от размера входных данных O(n), в то время как применение регулярных выражений напрямую показывает квадратичную зависимость $O(n * m)$, где n - размер текста, m - количество регулярных выражений.

Данное экспериментальное исследование подтверждает преимущества предложенного подхода на основе автомата Мили: автомат обеспечивает значительно более высокую производительность как при инициализации, так и при интерактивном редактировании текста. Время обработки изменений у автомата Мили в 4-8 раз меньше по сравнению с применением регулярных выражений как есть, что критически важно для обеспечения отзывчивости пользовательского интерфейса. Таким образом применение автомата Мили вместо простого набора регулярных выражений уже дает кратное преимущество в производительности, особенно на текстах больших размеров

4.2. Веб-редактор кода

Разработанная система представляет собой веб-приложение с интерфейсом

текстового редактора. Логика работы раскраски синтаксиса основана на двух подсистемах: автомат Мили для определения стилей, которые необходимо применить к лексемам, и древовидной структуры данных, основанной на структуре Rope, позволяющей эффективно определять область, которую необходимо перекрасить после внесения изменений в текст. Основные элементы интерфейса редактора включают:

- поле для непосредственного ввода текста;
- переключатель для выбора языка программирования;
- сворачиваемое окно с отладочной информацией, такой как список узлов Rope с сигналами на раскраску и примененными стилями.

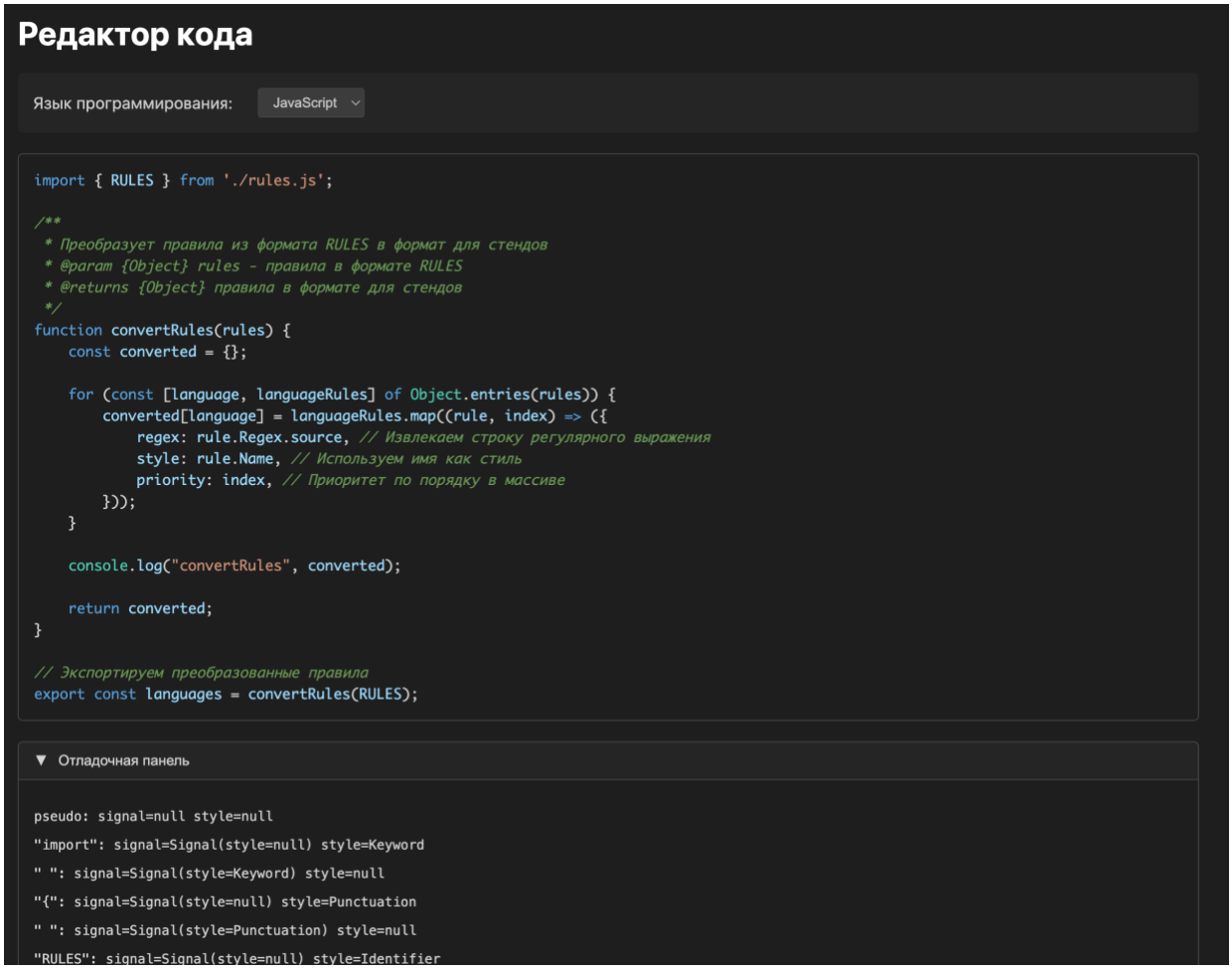

Рисунок 2 – интерфейс редактора кода с раскрытой отладочной панелью

Центральная область интерфейса занята полем для редактирования кода. Редактор поддерживает стандартные операции работы с текстом: ввод символов, удаление, выделение, копирование и вставку. Подсветка синтаксиса применяется автоматически в процессе набора текста.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Сервер поддержки учебного процесса ИУ9 [Электронный ресурс]. — URL: https://hw.iu9.bmstu.ru/ (дата обращения: 20.04.2025).

2. Monaco Editor [Электронный ресурс]. — URL: https://microsoft.github.io/monaco-editor/ (дата обращения: 20.04.2025).

3. CodeMirror [Электронный ресурс]. — URL: https://codemirror.net/ (дата обращения: 20.04.2025).

4. TextMate Grammar [Электронный ресурс]. — URL: https://macromates.com/manual/en/language_grammars (дата обращения: 01.05.2025).

5. H.-J. Boehm, R. Atkinson, M. Plass. Ropes: an Alternative to Strings // Software: Practice and Experience. — 1995. — Vol. 25, No. 12. — P. 1315-1330.