МАГІСТЕРСЬКА РОБОТА

МР.КІ – 42.00.000 ПЗ

Група КІм-24-2

Шушваль Богдан

2025

Міністерство освіти і науки України

Івано-Франківський національний технічний університет нафти і газу
Факультет інформаційних технологій

Кафедра комп'ютерних систем і мереж

Шушваль Богдан Романович
(прізвище, ім'я, по батькові)

УДК 004.7
(індекс)

МАГІСТЕРСЬКА РОБОТА

Вдосконалення сервісу візуального маркування синтаксису HAML для редактора коду Neovim на основі Tree-sitter

Комп'ютерна інженерія
(назва освітньої програми)

123 – Комп'ютерна інженерія
(шифр і назва спеціальності)

/ Б. Р. Шушваль /
(підпис, ініціали та прізвище здобувача освітнього ступеня)

Науковий керівник – Бабчук Сергій Миронович, к.т.н., доцент
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Допущено до захисту

Завідувач кафедри _____ */ С.І. Мельничук /*
(посада) (підпис) (дата) (ініціали та прізвище)

Рецензент _____ */ А.М. Топалов /*
(посада) (підпис) (дата) (ініціали та прізвище)

Робота містить результати власних досліджень, використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

Івано-Франківськ – 2025 рік

Івано-Франківський національний технічний університет нафти і газу

(повне найменування вищого навчального закладу)

Факультет *інформаційних технологій*

Кафедра *комп'ютерних систем і мереж*

Освітній ступінь *магістр*

Спеціальність *123 – Комп'ютерна інженерія*

(шифр і назва)

ЗАТВЕРДЖУЮ

Завідувач кафедри КСМ

(С.І. Мельничук)

« » 2025 року

З А В Д А Н Н Я

НА МАГІСТЕРСЬКУ РОБОТУ СТУДЕНТОВІ

Шушвалю Богдану Романовичу

(прізвище, ім'я, по батькові)

1. Тема проекту (роботи*) Вдосконалення сервісу візуального маркування синтаксису HAML для редактора коду Neovim на основі Tree-sitter*

керівник проекту (роботи) *Бабчук С. М., к.т.н., доцент*

(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом вищого навчального закладу від *05.12.2025 № 755/7*

2. Строк подання студентом роботи *10 грудня 2025*

3. Вихідні дані до роботи *Матеріали і результати отримані під час проходження переддипломної практики, методичні вказівки, технічна література.*

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) *1. Аналіз предметної області. 2. Вибір генератора парсерів для вдосконалення сервісу візуального маркування синтаксису HAML для редактора коду Neovim. 3. Вдосконалення сервісу візуального маркування синтаксису HAML для редактора коду Neovim на основі Tree-sitter.*

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень):

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	*Мойсеєнко О.В.*		

7. Дата видачі завдання *12 березня 2025 р.*

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту (роботи)	Строк виконання етапів проекту (роботи)	Примітка
1	*Постановка задачі та збір інформації*	*12.03.2025-29.09.2025*	виконано
2	*Аналіз мов шаблонів та редакторів коду*	*30.09.2025-20.10.2025*	виконано
3	*Вибір генератора парсерів для вдосконалення сервісу візуального маркування синтаксису HAML для редактора коду Neovim*	*21.10.2025-01.11.2025*	виконано
4	*Вдосконалення сервісу візуального маркування синтаксису HAML для редактора коду Neovim на основі Tree-sitter*	*02.11.2025-29.11.2025*	виконано
5	*Оформлення пояснювальної записки*	*30.11.2025-10.12.2025*	виконано

Студент _____ *Шушваль Б. Р.*
(підпис) (прізвище та ініціали)

Керівник роботи _____ *Бабчук С.М.*
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Сучасний процес розробки програмного забезпечення вимагає від інструментів високої ефективності, точності та швидкодії. Якісне візуальне маркування синтаксису є одним із ключових елементів, що безпосередньо впливає на продуктивність розробника, полегшуючи читання коду, навігацію та виявлення синтаксичних помилок.

HAML забезпечує зручну роботу з великими шаблонами в професійних проєктах. Також встановлено, що редактор коду Neovim є редактором коду перевага якого полягає у високій продуктивності та мінімальному споживанні системних ресурсів. Однак, маркування синтаксису HAML у Neovim базується на підході, який використовує регулярні вирази, що не забезпечує коректного маркування синтаксису.

За результатами аналізу генераторів парсерів, придатних для інкрементального парсингу встановлено, що для вдосконалення сервісу візуального маркування синтаксису HAML для редактора коду Neovim потрібно використати Tree-sitter.

Під час виконання магістерської роботи розроблено та інтегровано парсер мови HAML на основі Tree-sitter, що дозволило вдосконалити сервіс візуального маркування синтаксису HAML для редактора коду Neovim та забезпечити його коректну й стабільну роботу.

Ключові слова: Neovim, HAML, Tree-sitter, маркування синтаксису, парсер, інкрементальний парсинг

ABSTRACT

The modern software development process requires tools to be highly efficient, accurate, and fast. High-quality visual syntax highlighting is one of the key elements that directly affects developer productivity by facilitating code readability, navigation, and the detection of syntax errors.

HAML provides convenient support for working with large templates in professional projects. It has also been established that the Neovim code editor offers advantages in terms of high performance and minimal system resource consumption. However, HAML syntax highlighting in Neovim is based on an approach that relies on regular expressions, which does not ensure correct syntax highlighting.

Based on an analysis of parser generators suitable for incremental parsing, it was determined that Tree-sitter should be used to improve the HAML syntax highlighting service for the Neovim code editor.

During the completion of the master's thesis, a HAML language parser based on Tree-sitter was developed and integrated, which made it possible to enhance the HAML syntax highlighting service for the Neovim code editor and ensure its correct and stable operation.

Keywords: Neovim, HAML, Tree-sitter, syntax highlighting, parser, incremental parsing.

ЗМІСТ

с.

ПЕРЕЛІК ОСНОВНИХ ПОЗНАЧЕНЬ І СКОРОЧЕНЬ

ERB – Embedded Ruby;

HTML – HyperText Markup Language;

HAML – HTML Abstraction Markup Language;

DRY – Don’t repeat yourself;

API – Application Programming Interface;

ANTLR – Another Tool for Language Recognition;

AST – Abstract Syntax Tree;

CST – Concrete Syntax Tree;

LR – Left-to-right, Rightmost derivation in reverse;

EBNF – Extended Backus-Naur Form;

LL – Left-to-right, Leftmost derivation;

JS – Java Script;

JSON – Java Script Object Notation;

DSL – Domain-Specific Language;

JSON – JavaScript Object Notation;

MIT – Massachusetts Institute of Technology;

LSP – Language Server Protocol;

IDE – Integrated Development Environment;

VS Code – Visual Studio Code;

UI – User Interface;

CSS – Cascading Style Sheets;

PHP – Hypertext Preprocessor.

ВСТУП

Сучасний процес розробки програмного забезпечення вимагає від інструментів високої ефективності, точності та швидкодії. Одним із ключових елементів, що безпосередньо впливає на продуктивність розробника, полегшуючи читання коду та виявлення синтаксичних помилок, є візуальне маркування синтаксису. Його робота базується на точному розпізнаванні структурних елементів мови та їх взаємозв'язків, що дозволяє редактору коду відображати логічну структуру програми та забезпечувати стабільну роботу інструментів аналізу й редагування [3,4].

Актуальність теми даної роботи. Вдосконалення сервісу візуального маркування синтаксису HAML для редактора коду Neovim на основі Tree-sitter є актуальним завданням через постійне зростання вимог до швидкості та точності синтаксичного аналізу в сучасних середовищах розробки. Існуючі методи візуального маркування синтаксису HAML, які традиційно базуються на регулярних виразах, часто демонструють обмежений функціонал та низьку точність при роботі зі складними вкладеними структурами та вбудованим кодом [4-6].

Об'єктом дослідження є процеси синтаксичного аналізу та візуального маркування в текстових редакторах та інтегрованих середовищах розробки.

Предметом дослідження є візуальне маркування синтаксису HAML.

Метою магістерської роботи є вдосконалення сервісу візуального маркування синтаксису HAML для редактора коду Neovim на основі Tree-sitter.

Методи дослідження. Метод формалізації (теорії формальних мов) використаний для опису синтаксису HAML у вигляді формальної граматики Tree-sitter.

Практичне значення одержаних результатів полягає в вдосконаленні сервісу візуального маркування синтаксису HAML для редактора коду Neovim на основі Tree-sitter.

Апробація результатів магістерської роботи. Положення магістерської роботи обговорювались на VI Міжнародній науковій конференції “Теорія модернізації в контексті сучасної світової науки”.

Публікації. Опубліковані тези доповіді в збірнику тез доповідей VI Міжнародної наукової конференції “Теорія модернізації в контексті сучасної світової науки”.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Аналіз мов шаблонів

Мови шаблонів (templating languages) – це спеціалізовані мови, призначені для поєднання статичних шаблонів з динамічними даними з метою генерації документів, найчастіше HTML-розмітки. Одними з найпоширенішими є [7, 8]:

- ERB (Embedded Ruby);
- Liquid;
- HAML (HTML Abstraction Markup Language).

ERB – це вбудована система шаблонів Ruby, яка дозволяє вставляти код Ruby безпосередньо в HTML-подібний текст. Вона використовується в Ruby on Rails за замовчуванням і базується на прямому змішуванні HTML з Ruby [9].

Основна ідея ERB полягає в тому, щоб поєднати статичний вміст, зазвичає HTML, з динамічним кодом Ruby, використовуючи спеціальні теги для вставки логіки та результатів виконання. Це робить ERB простим і гнучким інструментом для створення динамічних веб-сторінок, де розробник може легко змішувати розмітку з програмною логікою [10, 11].

Синтаксис ERB (рис 1.1) базується на трьох основних типах тегів: <% %> – для виконання Ruby-коду без виводу результату (наприклад, для умов чи циклів), <%= %> – для виконання коду з виводом результату в шаблон (з автоматичним екрануванням HTML для безпеки), та <%# %> – для коментарів, які ігноруються під час обробки [10, 11].

Процес генерації HTML з ERB простий: код Ruby в тегах виконується, а результат заміняє тег у фінальному документі.

ERB вирізняється своєю гнучкістю та близькістю до чистого HTML, що робить її легкою для вивчення розробниками. Вона не накладає обмежень на код Ruby – можна використовувати будь-які конструкції, включаючи цикли,

умови, методи та навіть включення інших шаблонів. Це робить ERB ідеальною для простих і середніх проектів, де потрібна швидка інтеграція динамічного контенту без додаткових використання додаткових бібліотек [9, 11].

```
<!DOCTYPE html>
<html>
<head>
  <title>Random ERB Example</title>
</head>
<body>
  <h1>Welcome to the Random ERB Page!</h1>

  <% names = ["Alice", "Bob", "Charlie", "Dana"] %>
  <ul>
    <% names.each do |name| %>
      <li>
        <% if name.start_with?("A") %>
          <strong><%= name %> (special)</strong>
        <% else %>
          <%= name %>
        <% end %>
      </li>
    <% end %>
  </ul>

  <% current_time = Time.now %>
  <p>Current server time: <%= current_time.strftime("%H:%M:%S") %></p>

  <% random_number = rand(1..10) %>
  <% if random_number > 5 %>
    <p>Lucky number: <%= random_number %> 🎉</p>
  <% else %>
    <p>Better luck next time: <%= random_number %> 😅</p>
  <% end %>
</body>
</html>
```

Рисунок 1.1 – Шаблон написаний на ERB

Liquid – це відкрита шаблонна мова, розроблена компанією Shopify у 2006 році для створення тем у їхній платформі електронної комерції. Вона набула широкого поширення завдяки високій безпеці (sandboxed – обмежений доступ до коду) та простоті, і зараз використовується в статичних генераторах сайтів таких як Jekyll (за замовчуванням для GitHub Pages), а також у проєктах, де шаблони редагують люди з незначним досвідом програмування [12, 13].

Liquid (рис 1.2) базується на підході без логічних конструкцій: складна логіка виноситься в зовнішній код, а шаблон лише відображає дані з простими

операціями (цикли, умови). Синтаксис включає output ({{ }} для виводу з фільтрами), tags ({% %} для логіки) та фільтри (наприклад, upcase, date). Liquid не дозволяє довільний код, що забезпечує безпеку для шаблонів [14].

```
<h2>{{ product.title | camelize }}</h2>
{{ product.description }}

<!-- Output product images -->
{% if product.images.size > 0 %}
{% for image in product.images %}
    {% if forloop.first %}
        <img src="{{ image | product_img_url: '2048x2048' }}" />
    {% else %}
     <img src="{{ image | product_img_url: 'small' }}" />
    {% endif %}
{% endfor %}
{% else %}
No product image to show!
{% endif %}

<!-- Add to cart -->
<form action="/cart/add" method="post" enctype="multipart/form-data">
<select name="id">
    {% for variant in product.variants %}
    {% if variant.available == true %}
    <option value="{{variant.id}}"> {{ variant.title }}
      for {{ variant.price | money }}</option>
    {% endif %}
    {% endfor %}
</select>
<input type="submit" name="add" id="add" value="Add to Cart">
</form>
```

Рисунок 1.2 – Шаблон написаний на Liquid

HAML – це мова розмітки, яка використовується для чіткого та простого опису HTML будь-якого веб-документа без використання вбудованого коду. HAML (рис 1.3) замінює вбудовані системи шаблонів сторінок, такі як PHP, ERB та ASP. Однак HAML дозволяє уникнути необхідності явного написання HTML у шаблоні, оскільки фактично є абстрактним описом HTML із деяким кодом для генерації динамічного вмісту [5, 15].

Рядки, що не починаються зі спеціальних символів, трактуються як звичайний текст і вставляються в HTML без змін. Для екранування символів %, = та інших службових позначок можна використовувати зворотний слеш на початку рядка. Це дозволяє вставляти в документ текст, який інакше був би інтерпретований як код або тег [15, 16].

Теги в HAML задаються через символ %, після якого йде ім’я елемента, можливі класи, ідентифікатори та атрибути. Якщо ім’я елемента не вказати, HAML автоматично використовує HTML елемент div. Класи та ідентифікатори (id) можна задавати у вигляді CSS-подібної нотації [16].

Атрибути дозволено описувати у фігурних дужках або у форматі, схожому на HTML, з круглими дужками. Масиви у атрибутах класу та id автоматично перетворюються на рядки з відповідним об’єднанням елементів. Порожні або хибні значення атрибутів ігноруються. Теги також можуть бути самозакривними, якщо поставити символ /, і точний формат залежить від встановленого типу вихідного HTML [15].

```
%html
  %head
    %title Sample HAML Page

  %body.container
    %header#main-header
      %h1 Welcome to the Demo
      %nav
        %ul.nav
          - ["Home", "About", "Contact"].each do |item|
            %li.nav__item= item

    %section.content
      - if @user
        %h2 Hello, #{@user.name}!
        %p You've got #{@notifications.count} new notifications.
      - else
        %h2 Hey there!
        %p Please log in to continue.
```

Рисунок 1.3 – Шаблон написаний на HAML

Контроль пробілів виконується через символи > або <, розташовані наприкінці тегу. Перший прибирає пробіли навколо тегу, другий – пробіли всередині. Це допомагає отримувати акуратний HTML без зайвих переносів і відступів [5].

Коментарі в HAML можуть бути двох типів. Коментар, який починається

зі знаку /, потрапляє у кінцевий HTML як стандартний HTML-коментар. Коментар, що починається з -#, повністю ігнорується під час генерації HTML і не залишає жодного сліду. Обидва коментарі можуть містити вкладені рядки [16, 17].

Символ ~ працює подібно до =, але перед вставкою результат обробляється методом preserve, який зберігає перенос рядків. У тексті або атрибутах можна використовувати Ruby-інтерполяцію #{}, що дозволяє вставляти значення Ruby-виразів прямо у рядок [17].

HAML підтримує фільтри – спеціальні механізми для обробки блоків тексту зовнішніми процесорами. Фільтр (рис 1.4) вказується після двокрапки з назвою (наприклад, :markdown, :javascript, :sass), а за ним іде вкладений блок тексту написаний іншою мовою програмування. Вміст блоку передається відповідному обробнику (наприклад, Markdown у HTML, Sass у CSS), і результат вставляється у фінальний HTML-документ. Серед вбудованих фільтрів: :css, :javascript, :markdown, :sass, :scss, :erb, :plain та інші [17].

```
:javascript
  let str = 'Hello';
  console.log(str);

:ruby
  users = %w[Alice Bob Charlie]
  users.each { |u| puts u }

:markdown
  ## Welcome

  This block is rendered using **Markdown** inside HAML.

  - Clean
  - Simple
  - Based
```

Рисунок 1.4 – HAML фільтри

Код Ruby можна вставляти двома способами. Якщо рядок починається з =, код виконується і результат потрапляє в HTML, а спеціальні символи можуть

бути екрановані, якщо ввімкнені відповідні налаштування. Якщо рядок починається з -, код виконується, але результат не вставляється, що підходить для умов і циклів. Ruby-блоки визначаються через відступи, де збільшення відступу відкриває блок, а зменшення – закриває [17].

Результати аналізу мов шаблонів ERB, Liquid та HAML наведені в таблиці 1.1.

Таблиця 1.1 – Основні характеристики мов шаблонів

Характеристика	**ERB (Embedded Ruby)**	**Liquid**	**HAML (HTML Abstraction Markup Language)**
Основний синтаксис	Теги <% %>, <%= %>, <%# %> у HTML-подібному тексті	{{ }} для виводу, {% %} для тегів, фільтри	Використання відступів та спеціальних символів, %, ., #, =, -
Стисліть та читабельність	Середня	Висока	Висока
Безпека	Середня (можливе виконання довільного коду)	Найвища (без довільного коду)	Висока (автоматичне екранування)
Логіка в шаблоні	Повна (цикли, умови, методи)	Обмежена (простий підхід)	Повна (цикли, умови, методи)
Фільтри та обробка блоків	Відсутні	Вбудовані фільтри (upcase, date тощо)	Вбудовані (:markdown, :javascript тощо)
Швидкість рендерингу	Висока	Висока (швидкий для статичних сайтів)	Середня
Складність вивчення	Низька (близький до HTML)	Низька (простий для дизайнерів)	Середня (незвичний синтаксис)
Основне застосування	Динамічні Rails-додатки, прості проекти	Статичні сайти	Складні Rails-шаблони

З таблиці 1.1 видно, що ERB попри гнучкість і простоту, страждає від засміченості тегами та порушень DRY-принципу, що ускладнює читабельність і підтримку коду. Liquid, своєю чергою, вирізняється підвищеною безпекою та доступністю для не-програмістів, однак її спрощена логіка зменшує придатність для реалізації складних шаблонів. У результаті саме HAML постає

як найбільш ефективна шаблонна мова для професійних проєктів із великими шаблонами, забезпечуючи кращу структурованість, лаконічність та зручність супроводу коду.

1.2 Аналіз текстових редакторів

Sublime Text, Visual Studio Code та Neovim є сучасними текстовими редакторами для роботи з програмним кодом, які широко застосовуються як у навчальному, так і в професійному середовищі. Незважаючи на спільну базову функцію редагування тексту, кожен із цих інструментів реалізує власну концепцію взаємодії з користувачем, архітектурні підходи та механізми розширення, що безпосередньо впливає на ефективність роботи з вихідним кодом [18, 19].

Sublime Text (рис 1.5) є редактором коду, який зосереджений на високій продуктивності та мінімалізмі. Редактор відомий швидким запуском, плавною роботою з великими файлами та лаконічним інтерфейсом [20].

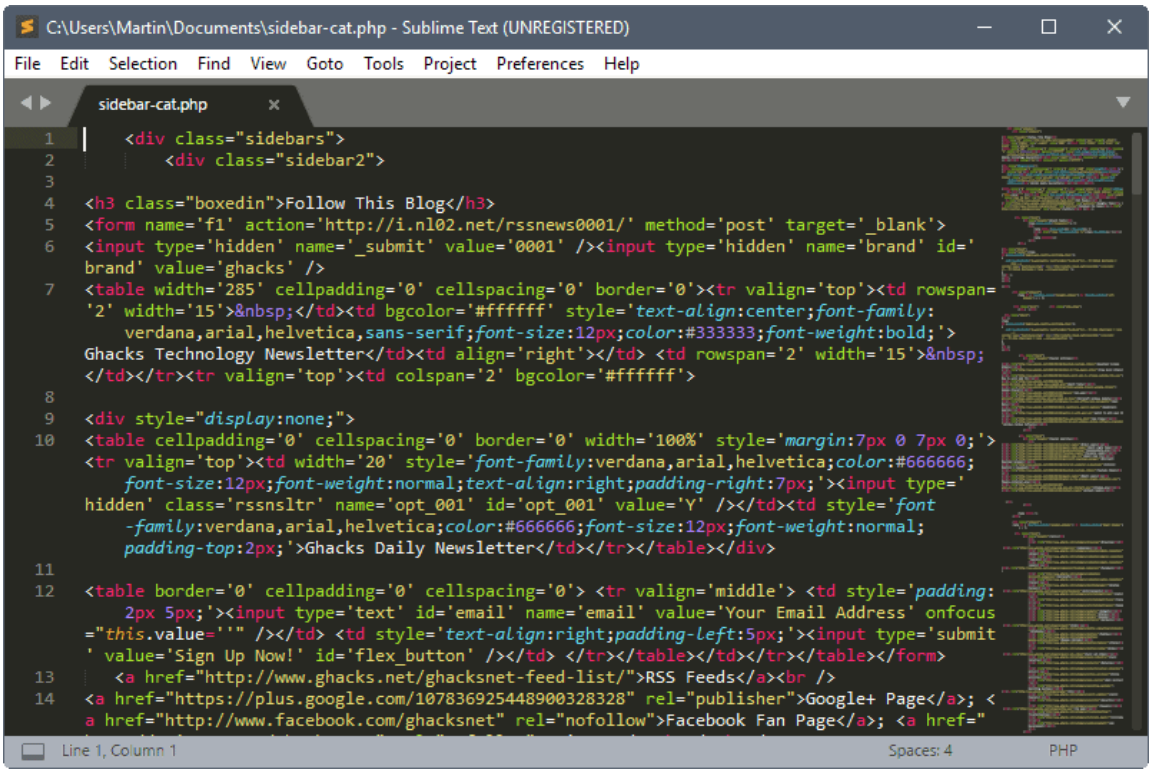


Рисунок 1.5 – Інтерфейс Sublime Text

Архітектура Sublime Text оптимізована для швидкого редагування тексту, а механізми підсвічування та навігації забезпечують комфортну роботу навіть без значної кількості додаткових компонентів. Розширення можливостей здійснюється за допомогою власного API. Підтримка сучасних інструментів аналізу коду зазвичай потребує інтеграції зовнішніх рішень, що зменшує рівень автоматизації порівняно з іншими редакторами [20, 21].

Важливим аспектом є модель розповсюдження Sublime Text: редактор є платним продуктом з одноразовою ліцензією (станом на 2025 рік вартість становить близько 99 доларів США за користувача) що відрізняється від повністю безкоштовних альтернатив таких як Neovim чи VS Code [20].

Visual Studio Code (рис 1.6) є редактором, що поєднує простоту початкового використання з широкими можливостями додаткових розширень. Більшість функціональних можливостей надається через систему плагінів, що дозволяє адаптувати середовище під різні мови програмування та задачі. VS Code має вбудовану підтримку Language Server Protocol, засобів налагодження, систем контролю версій та інструментів командної взаємодії [22].

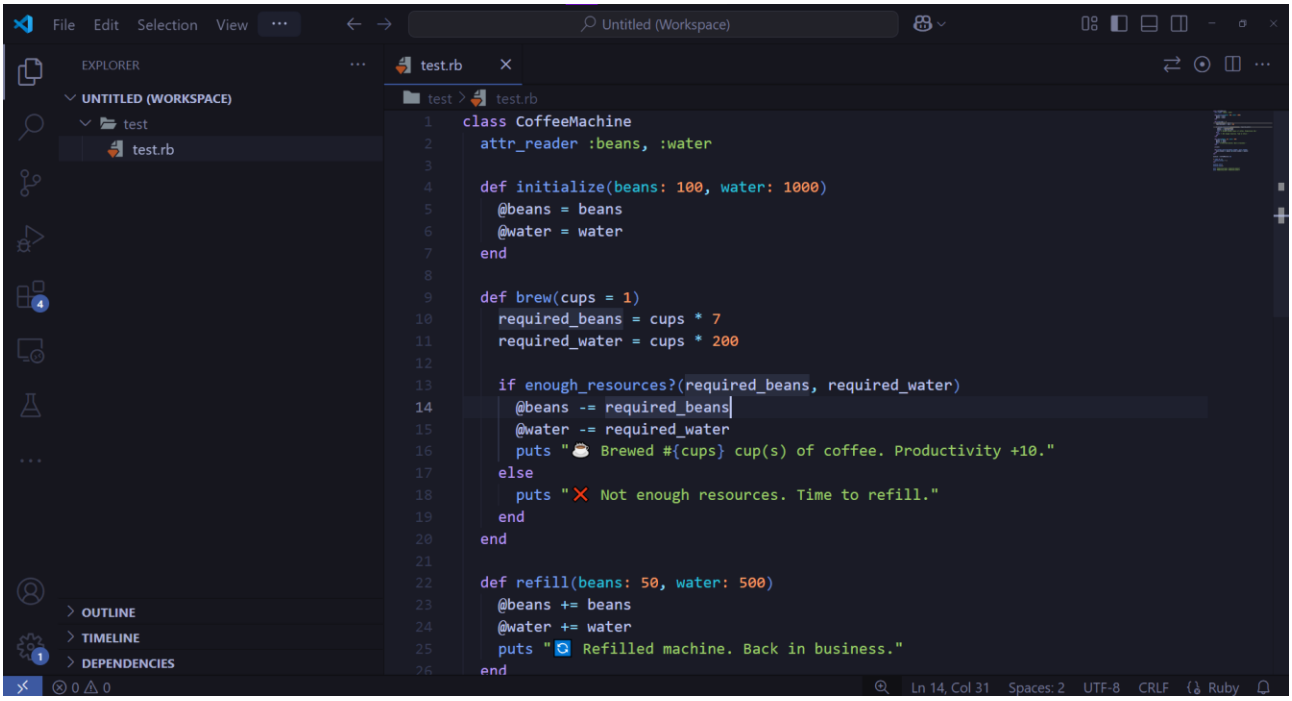


Рисунок 1.6 – Інтерфейс Visual Studio Code

Його інтерфейс орієнтований на візуальну наочність і мінімізацію необхідності ручної конфігурації. Водночас, залежність від великої кількості розширень може призводити до зростання споживання системних ресурсів, а також до ускладнення підтримки стабільного середовища при оновленнях [23].

Рисунок 1.7 – Інтерфейс Neovim

Neovim (рис 1.7) є розвитком класичного редактора Vim і орієнтований на модальну модель редагування, де введення тексту, навігація та виконання команд чітко розділені. Такий підхід дозволяє редагування тексту без мишки з використанням лише клавіатури, що позитивно впливає на швидкість роботи досвідчених користувачів [5, 24].

Архітектура Neovim побудована навколо асинхронної подієвої моделі та розширюваності за допомогою Lua, що робить його повноцінною платформою для побудови індивідуальних робочих середовищ. Важливою перевагою є інтеграція з протоколом Language Server Protocol. Разом з тим, високий рівень гнучкості супроводжується складнішим процесом початкового налаштування, що може бути бар'єром для недосвідчених користувачів [24, 25].

Аналізу редакторів коду Sublime Text Visual Studio Code та Neovim наведені в таблиці 1.2.

Таблиця 1.2 – Основні характеристики текстових редакторів

Характеристика	**Sublime Text**	**Visual Studio Code**	**Neovim**
Базова технологія	C++ з Python API	Electron (Chromium + Node.js)	C/Lua,
Продуктивність	Швидкий, легкий, ефективний з великими файлами	Ресурсоємний через Electron	мінімальне споживання ресурсів, швидкий запуск і редагування великих файлів
Підсвічування синтаксису	TextMate граматики	TextMate граматики	На основі бібліотеки інкрементального парсингу Tree-sitter
Підтримка HAML	Базове маркування синтаксису на основі пакету language-haml	Базове маркування синтаксису на основі розширення HAML/Ruby	Базове маркування синтаксису на основі регулярних виразів
Розширюваність	Висока (Package Control, Python-плагіни)	Найвища (маркетплейс з тисячами розширень)	Висока (Lua-плагіни, вбудований LSP)
Інтеграція LSP/Tree-sitter	Через плагіни (LSP частково)	Вбудована LSP	Вбудована (Tree-sitter + LSP)
Складність вивчення	Низька (простий інтерфейс)	Низька (інтуїтивний UI)	Висока (модальний режим Vim)
Модель розповсюдження	Платний	Безкоштовний (open-source з телеметрією)	Безкоштовний (open-source, MIT)

З таблиці 1.2, можна зробити висновок, що Neovim є найбільш перспективним та технічно досконалим редактором коду серед розглянутих рішень. Його перевага полягає у високій продуктивності та мінімальному споживанні системних ресурсів. Також він орієнтований на точну та формально коректну роботу з програмним текстом, проте, на даний момент підсвічування

синтаксису HAML у Neovim реалізується за допомогою підходу, що базується на регулярних виразах. Такий підхід не забезпечує коректного структурного аналізу мови, є чутливим до вкладеності та контексту і обмежує можливості точного візуального маркування складних синтаксичних конструкцій. В зв’язку з вищевказаним, актуальною задачею є вдосконалення сервісу візуального маркування синтаксису HAML для редактора коду Neovim.

2 ВИБІР ГЕНЕРАТОРА ПАРСЕРІВ ДЛЯ ВДОСКОНАЛЕННЯ СЕРВІСУ ВІЗУАЛЬНОГО МАРКУВАННЯ СИНТАКСИСУ HAML ДЛЯ РЕДАКТОРА КОДУ NEOVIM

Генератори парсерів є інструментами, призначеними для автоматизованого створення синтаксичних аналізаторів на основі формального опису граматики мови. Традиційно такі інструменти використовуються в задачах компіляції, трансляції мов програмування, побудови інтерпретаторів та аналізаторів коду. Однак із розвитком інтегрованих середовищ розробки та редакторів коду роль парсерів значно розширилась і почала охоплювати сервіси реального часу, зокрема маркування синтаксису, навігацію по коду та статичний аналіз [26, 27].

На відміну від класичних компіляторних сценаріїв, підсвітка синтаксису вимагає від парсера роботи з частково некоректним або незавершеним кодом, а також швидкого оновлення результатів аналізу після кожної зміни тексту. Це накладає суттєві обмеження на вибір генераторів парсерів, оскільки не всі з них здатні ефективно працювати в умовах інтерактивного редагування [26].

Для використання в редакторах коду генератор парсерів повинен відповідати ряду специфічних вимог, які не є критичними для класичних компіляторів. Насамперед це здатність обробляти вхідні дані, що постійно змінюються, та підтримка побудови синтаксичного дерева навіть у разі наявності помилок у коді [28, 29].

Важливим критерієм є можливість інкрементального парсингу, коли повторний аналіз обмежується лише зміненими фрагментами документа. Такий підхід суттєво знижує обчислювальні витрати та забезпечує високу швидкодію сервісів підсвітки синтаксису [29].

ANTLR (ANother Tool for Language Recognition) є одним із найбільш відомих і поширених генераторів парсерів, який активно використовується для створення компіляторів, інтерпретаторів, трансляторів та мов предметної

області. Його архітектура базується на підході top-down парсингу та використовує адаптивний алгоритм LL(*), що дозволяє аналізувати широкий клас контекстно-вільних граматик без жорстких обмежень на кількість токенів попереднього перегляду. Завдяки цьому ANTLR надає розробнику значну гнучкість у визначенні синтаксису мов програмування та дозволяє описувати складні граматичні конструкції з великою кількістю альтернатив [30, 31].

Ключовою особливістю ANTLR є його здатність автоматично генерувати абстрактне синтаксичне дерево (Abstract Syntax Tree, AST) на основі заданої граматики. Це дерево використовується як проміжне представлення програми для подальшого аналізу, трансформації або генерації коду. У контексті підсвітки синтаксису наявність AST є важливою перевагою, оскільки воно зберігає ієрархічну структуру мови та дозволяє ідентифікувати синтаксичні конструкції не лише на лексичному, але й на структурному рівні [32, 33].

ANTLR використовує розширену нотацію EBNF для опису граматик, що підвищує їх читабельність і спрощує підтримку. Розробник може поєднувати опис лексичних правил і синтаксичних конструкцій в одному файлі граматики, що зменшує кількість допоміжних інструментів та спрощує процес розробки парсера. Крім того, ANTLR підтримує семантичні дії та семантичні предикати, які дозволяють вбудовувати додаткову логіку безпосередньо в граматику. Це робить ANTLR потужним інструментом для вирішення складних синтаксичних неоднозначностей, які важко або неможливо усунути лише засобами контекстно-вільної граматики [34].

Разом із тим архітектура ANTLR орієнтована передусім на аналіз повного та синтаксично коректного вхідного коду. У випадку наявності синтаксичних помилок ANTLR, як правило, фіксує помилку та намагається виконати відновлення, проте його механізми error recovery мають обмежену ефективність у сценаріях інтерактивного редагування. Під час введення коду користувачем, коли текст часто перебуває в тимчасово некоректному стані, це може призводити до втрати або спотворення синтаксичного дерева, що негативно впливає на стабільність підсвітки синтаксису [35].

Суттєвим обмеженням ANTLR у контексті редакторів коду є відсутність підтримки інкрементального парсингу. Кожна зміна у вхідному тексті вимагає повторного запуску парсера для всього документа або його значної частини. Такий підхід є прийнятним для компіляторів або офлайнових інструментів аналізу, але не відповідає вимогам сервісів реального часу, де обробка змін повинна відбуватися за мілісекунди. У великих файлах це може призводити до затримок, що робить використання ANTLR для безперервної підсвітки синтаксису менш ефективним [34].

З точки зору теорії, ANTLR може бути використаний як основа для реалізації підсвітки синтаксису, оскільки він формує структуроване представлення коду та дозволяє точно ідентифікувати мовні конструкції. Проте на практиці це потребує створення додаткового шару оптимізацій, таких як кешування результатів парсингу, ручна обробка помилок та часткове повторне використання синтаксичного дерева. Навіть за наявності таких механізмів досягти рівня швидкодії та стабільності, характерного для спеціалізованих редакторних парсерів, є складним завданням [35].

Таким чином, ANTLR слід розглядати як потужний і універсальний генератор парсерів, який добре підходить для побудови фронтендів компіляторів і статичних аналізаторів, але лише умовно придатний для задач підсвітки синтаксису в інтерактивних редакторах коду. Його сильні сторони полягають у виразності граматик, підтримці семантичних механізмів та розвиненій екосистемі інструментів, тоді як основними обмеженнями в контексті візуального маркування синтаксису є відсутність інкрементального парсингу та недостатня стійкість до частково некоректного коду.

JavaCC (Java Compiler Compiler) є одним із класичних генераторів парсерів, орієнтованих на мову програмування Java, який широко використовувався у навчальних та промислових проєктах для побудови компіляторів і трансляторів. Його архітектура базується на підході рекурсивного спуску з попереднім переглядом токенів, що відповідає класу LL(k)-парсерів. Такий підхід забезпечує відносну простоту реалізації та

зрозумілість граматик, проте водночас накладає певні обмеження на структуру мов, які можуть бути описані без додаткових трансформацій [36, 37].

Граматики в JavaCC описуються у формі, близькій до EBNF, з чітким поділом між лексичними та синтаксичними правилами. Однією з характерних рис JavaCC є можливість вбудовування семантичних дій безпосередньо у правила граматики у вигляді Java-коду. Це дозволяє виконувати додаткову обробку під час парсингу, проте тісно пов'язує граматику з конкретною мовою реалізації, що ускладнює повторне використання та аналіз граматики як самостійної формальної специфікації [38, 39].

У контексті підсвітки синтаксису JavaCC теоретично може бути використаний для побудови структурного представлення вихідного коду, оскільки результатом його роботи є синтаксичне дерево або набір викликів семантичних дій, які відображають структуру програми. Однак, як і у випадку з ANTLR, JavaCC не підтримує інкрементальний парсинг. Будь-яка зміна у тексті вимагає повторного запуску парсера, що робить його малопридатним для сценаріїв реального часу, характерних для сучасних редакторів коду [39].

Крім того, механізми обробки синтаксичних помилок у JavaCC орієнтовані на компіляторні сценарії, де важливо коректно повідомити про помилку та завершити аналіз. У процесі інтерактивного редагування, коли код часто перебуває в некоректному стані, це може призводити до втрати цілісності синтаксичного дерева або повної неможливості його побудови. Таким чином, хоча JavaCC може бути використаний для побудови підсвітки синтаксису з теоретичної точки зору, на практиці це потребуватиме значних доопрацювань та оптимізацій, які нівелюють його початкову простоту [40].

SableCC є генератором парсерів, що розроблявся з акцентом на чітке розділення етапів лексичного та синтаксичного аналізу, а також на формальну строгість опису граматики. На відміну від JavaCC та ANTLR, SableCC використовує підхід bottom-up парсингу та будує парсери на основі LR-алгоритмів, що дозволяє обробляти ширший клас граматик без необхідності ручного усунення лівої рекурсії. Граматика в SableCC описується у

декларативному вигляді, без вбудованих семантичних дій, що сприяє кращій читабельності та формальній чистоті специфікації мови [41, 42].

Результатом роботи SableCC є побудова абстрактного синтаксичного дерева, яке має чітко визначену структуру та типізацію вузлів. Такий підхід є зручним для статичного аналізу та трансформацій програм, проте в контексті підсвітки синтаксису він має як переваги, так і обмеження. З одного боку, наявність AST дозволяє точно ідентифікувати мовні конструкції на структурному рівні, з іншого – дерево не зберігає всіх деталей вихідного тексту, таких як пробіли, форматування або коментарі, які часто є важливими для візуального маркування коду [43, 44].

Як і більшість класичних генераторів парсерів, SableCC не підтримує інкрементальний парсинг. Повторний аналіз всього документа при кожній зміні робить його непридатним для безперервної роботи в редакторі коду. Додатковим обмеженням є обмежена стійкість до синтаксичних помилок, оскільки парсер орієнтований на аналіз коректного вхідного тексту. У сценаріях інтерактивного редагування це призводить до втрати структурного представлення коду або необхідності реалізації складних механізмів відновлення [45, 46].

Tree-sitter є сучасним генератором парсерів, спеціально розробленим для використання в редакторах коду та інтегрованих середовищах розробки, де критично важливими є швидкодія, стабільність та здатність працювати з частково некоректним або незавершеним вихідним кодом. На відміну від традиційних парсерів, орієнтованих на компіляцію, архітектура Tree-sitter з самого початку проєктувалась з урахуванням вимог інтерактивного редагування тексту. Основою його роботи є інкрементальний парсинг на базі LR(1)-алгоритму, який дозволяє оновлювати синтаксичне дерево лише для змінених фрагментів документа без необхідності повного повторного аналізу всього файлу [47, 48].

Важливою особливістю Tree-sitter є здатність стабільно працювати в умовах синтаксичних помилок. У процесі введення коду користувач часто

створює тимчасово некоректні конструкції, наприклад незакриті дужки, теги або обірвані вирази. У таких випадках класичні парсери зазвичай припиняють роботу або повертають некоректний результат. Tree-sitter, навпаки, продовжує аналіз і формує синтаксичне дерево (рис 2.1), яке містить спеціальні вузли помилок, зберігаючи загальну ієрархічну структуру документа. Така поведінка є ключовою для реалізації стабільної підсвітки синтаксису, оскільки дозволяє редактору коректно відображати структуру коду навіть у процесі його активного редагування [49].

```
(document ; [0, 0] - [20, 0]
  (running_ruby ; [0, 0] - [0, 47]
    (ruby_code ; [0, 1] - [0, 47]
      (program ; [0, 2] - [0, 47]
        (assignment ; [0, 2] - [0, 47]
          left: (identifier) ; [0, 2] - [0, 14]
          right: (call ; [0, 17] - [0, 47]
            receiver: (element_reference ; [0, 17] - [0, 38]
              object: (identifier) ; [0, 17] - [0, 23]
              (simple_symbol)) ; [0, 24] - [0, 37]
            method: (identifier)))))) ; [0, 39] - [0, 47]
  (running_ruby ; [1, 0] - [1, 29]
    (ruby_code ; [1, 1] - [1, 29]
      (program ; [1, 2] - [1, 29]
        (assignment ; [1, 2] - [1, 29]
          left: (identifier) ; [1, 2] - [1, 5]
          right: (call ; [1, 8] - [1, 29]
            receiver: (element_reference ; [1, 8] - [1, 20]
              object: (identifier) ; [1, 8] - [1, 14]
              (simple_symbol)) ; [1, 15] - [1, 19]
            method: (identifier)))))) ; [1, 21] - [1, 29]
```

Рисунок 2.1 – Синтаксичне дерево Tree-sitter

Проєкт Tree-sitter був розроблений у 2017 році як відкритий програмний продукт з ліцензією MIT. Початково він створювався для потреб платформи GitHub, зокрема для редактора Atom, однак з часом набув широкого поширення та став стандартним інструментом інкрементального парсингу для багатьох

сучасних редакторів коду, таких як Neovim, Helix, Zed, а також частково Visual Studio Code. [49].

Інкрементальний підхід, реалізований у Tree-sitter, полягає в тому, що після кожної зміни тексту оновлюються лише ті вузли синтаксичного дерева, які відповідають зміненим ділянкам коду. Це дозволяє досягати дуже високої продуктивності, зокрема забезпечувати час обробки змін на рівні кількох мілісекунд навіть для великих файлів. Завдяки цьому Tree-sitter є придатним для реалізації сервісів реального часу, таких як підсвітка синтаксису, автодоповнення, навігація по коду та структурний аналіз [50].

Алгоритмічно Tree-sitter використовує узагальнений LR-підхід (Generalized LR), що базується на LR(1)-парсингу та має лінійну обчислювальну складність O(n). Це дозволяє йому коректно працювати з неоднозначними граматиками та складними синтаксичними конструкціями, що часто зустрічаються в сучасних мовах програмування та шаблонних мовах. Водночас Tree-sitter не орієнтується на глибокий семантичний аналіз, зосереджуючись саме на побудові структурного представлення коду, достатнього для потреб редактора [47].

Граматики в Tree-sitter описуються у вигляді декларативного набору правил, які задаються за допомогою спеціалізованого DSL (domain-specific language, предметно-орієнтована мова) на основі мови JavaScript або JSON-подібної нотації. Хоча такий підхід відрізняється від класичних формальних нотацій BNF або EBNF, він забезпечує достатню гнучкість для опису синтаксису сучасних мов, зокрема мов із вкладеними або змішаними структурами, таких як HAML. На основі описаної граматики Tree-sitter генерує парсер у вигляді C-коду, причому runtime-бібліотека реалізована відповідно до стандарту C11 і не має зовнішніх залежностей. Це спрощує інтеграцію Tree-sitter в різні програмні середовища та дозволяє використовувати його через біндінги для багатьох мов програмування, включаючи Rust, Go, Python та JavaScript [51].

Особливістю Tree-sitter є те, що він формує саме конкретне синтаксичне дерево (рис 2.1). Це означає, що дерево містить повну інформацію про структуру вихідного тексту, що є критично важливим для задач підсвітки синтаксису. Для безпосереднього маркування елементів коду використовуються спеціальні запити, описані у вигляді S-виразів, які дозволяють зіставляти вузли синтаксичного дерева з відповідними групами підсвітки. Такий підхід забезпечує чітке розділення між синтаксичним аналізом і візуальним представленням коду, а також дозволяє реалізовувати ін'єкції парсерів для вбудованих мов, наприклад JavaScript у HTML або Ruby у HAML.

В таблиці 2.1 наведені основні характеристики генераторів гарсерів ANTLR, JavaCC, SableCC та Tree-sitter.

Таблиця 2.1 – Основні характеристики генераторів парсерів

Характеристика	**ANTLR**	**JavaCC**	**SableCC**	**Tree-sitter**
Тип парсингу	Top-down, адаптивний LL(*)	Top-down, рекурсивний спуск LL(k)	Bottom-up, LR	Bottom-up, узагальнений LR(1) (GLR)
Інкрементальний парсинг	Ні	Ні	Ні	Так (оновлення лише змінених фрагментів)
Обробка синтаксичних помилок (fault-tolerance)	Обмежена (error recovery, але часто втрата дерева)	Обмежена (орієнтація на компілятори)	Обмежена (вимагає коректного коду)	Висока (формує дерево з вузлами помилок, зберігає структуру)
Продуктивність в реальному часі	Низька при частих змінах (повний перепарсинг)	Низька (повний перепарсинг)	Низька (повний перепарсинг)	Висока (мілісекунди навіть для великих файлів)
Нотація граматики	EBNF	Близька до EBNF, чіткий поділ	Декларативна, строга	DSL на базі JS/JSON (спеціалізована)
Придатність для підсвітки синтаксису в редакторах	Умовна (потрібні додаткові оптимізації)	Низька (значні доопрацювання)	Низька (втрата деталей тексту)	Висока (спеціально розроблений для цього)
Основна сфера застосування	Компілятори, інтерпретатори, DSL	Компілятори Java-проєктів	Статичний аналіз, формальні мови	Редактори коду (Neovim, Helix, Zed, VS Code тощо)

З таблиці 2.1 видно, що більшість класичних інструментів проєктувалися з урахуванням вимог компіляторів. Такі генератори, як ANTLR, JavaCC та SableCC, демонструють високу виразність граматик, формальну строгість та розвинені механізми синтаксичного аналізу, однак їх архітектурні рішення не відповідають ключовим вимогам середовищ реального часу, де підсвітка синтаксису повинна бути швидкою, стабільною та стійкою до частково некоректного коду.

ANTLR, будучи потужним інструментом для побудови компіляторів, орієнтується на повний аналіз вхідного тексту та генерацію абстрактного синтаксичного дерева. Відсутність інкрементального парсингу та обмежена обробка синтаксичних помилок ускладнюють його використання в редакторах коду, оскільки кожна зміна вимагає повторного аналізу документа. Аналогічні обмеження характерні для JavaCC, який використовує підхід рекурсивного спуску та тісно пов’язує граматику з семантичними діями, що знижує гнучкість та масштабованість такого рішення. SableCC, попри формальну чистоту та використання LR-парсингу, також не враховує потреби інкрементальної обробки та орієнтується на побудову абстрактних синтаксичних дерев, недостатньо придатних для точного візуального маркування вихідного тексту.

Tree-sitter, завдяки поєднанню інкрементального парсингу, високої продуктивності та чіткому поділу між синтаксичним аналізом і механізмами підсвітки, є найбільш придатним генератором парсерів для реалізації візуального маркування синтаксису в сучасних редакторах коду. На відміну від розглянутих класичних інструментів (ANTLR, JavaCC, SableCC), Tree-sitter з самого початку проєктувався саме для інтерактивного редагування тексту. Його інкрементальний підхід дозволяє оновлювати синтаксичне дерево лише в змінених фрагментах документа, забезпечуючи стабільну швидкодію навіть у великих файлах і мінімізуючи затримки до рівня мілісекунд – що критично важливо для комфортного користувацького досвіду в режимі реального часу.

Значною перевагою Tree-sitter є його стійкість до синтаксичних помилок (fault-tolerant). Під час написання коду текст майже завжди містить тимчасові

некоректні конструкції, і класичні парсери в таких випадках часто втрачають здатність формувати коректне дерево. Tree-sitter же продовжує аналіз, додаючи спеціальні вузли помилок і зберігаючи загальну структуру, що гарантує стабільну підсвітку та навігацію навіть у незавершеному коді.

Важливим аспектом є також використання конкретного синтаксичного дерева (Concrete Syntax Tree, CST) замість абстрактного. CST зберігає всі деталі вихідного тексту: коментарі, пробіли, форматування. Що дозволяє виконувати точне й контекстно-залежне маркування синтаксису. Окремий механізм підсвітки, реалізований через систему запитів (queries) на основі S-виразів, забезпечує гнучкість, розширюваність та легку підтримку вбудованих мов таких як, Ruby в HAML.

Отже, Tree-sitter є оптимальним вибором вдосконалення сервісу візуального маркування синтаксису HAML для редактора коду Neovim.

3 ВДОСКОНАЛЕННЯ СЕРВІСУ ВІЗУАЛЬНОГО МАРКУВАННЯ СИНТАКСИСУ HAML ДЛЯ РЕДАКТОРА КОДУ NEOVIM НА ОСНОВІ TREE-SITTER

3.1 Розробка структури проєкту Tree-sitter

Для Вдосконалення сервісу візуального маркування синтаксису HAML для редактора коду Neovim на основі Tree-sitter було розроблено проєкт з структурою, яка зображена на рисунку 3.1.

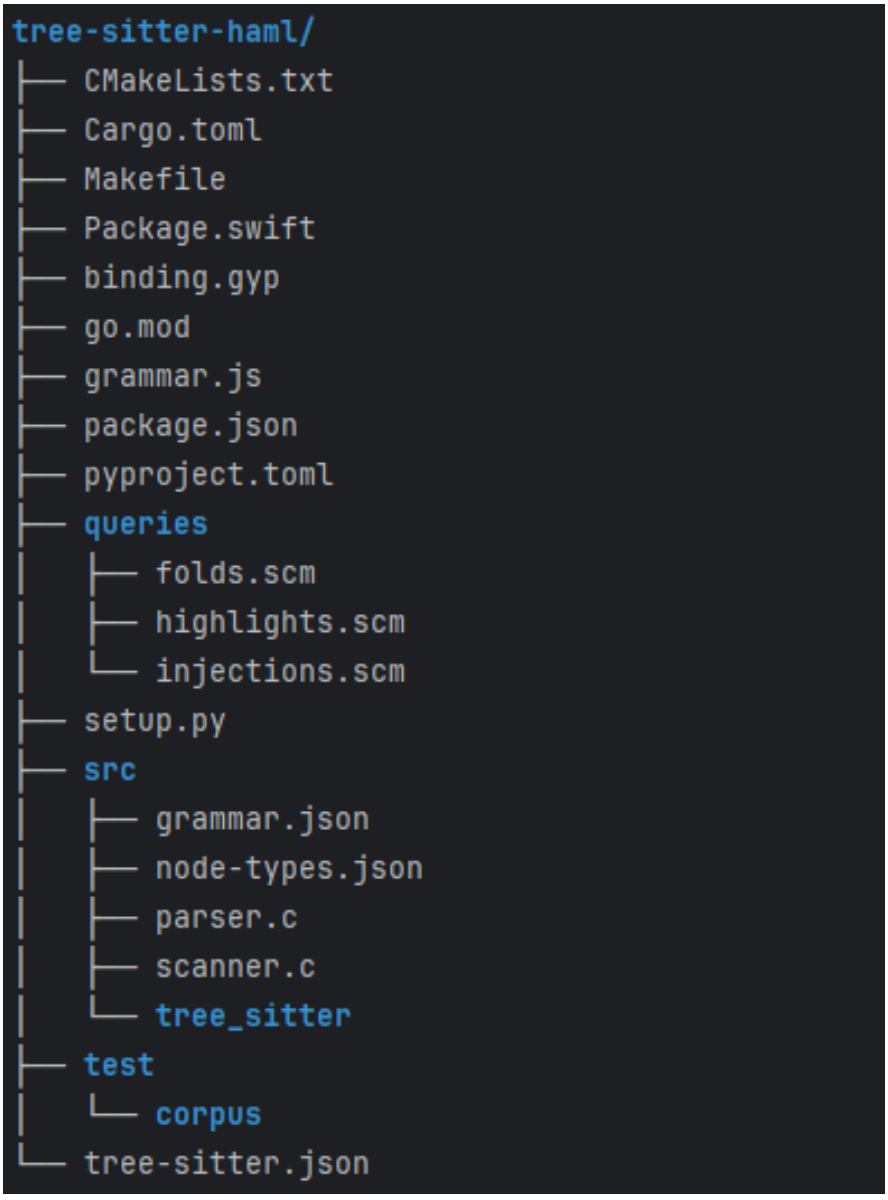


Рисунок 3.1 – Структура проєкту

Проєкт складається з конфігураційних файлів, опису граматики, згенерованого синтаксичного аналізатора, а також допоміжних компонентів для тестування та інтеграції.

У кореневій директорії розміщено файли збірки та конфігурації для різних екосистем. Наявність CMakeLists.txt, Makefile, binding.gyp, Cargo.toml, Package.swift, go.mod та pyproject.toml забезпечує можливість використання граматики в проєктах на C/C++, Node.js, Rust, Swift, Go та Python відповідно.

Файл grammar.js (Додаток А) є центральним елементом проєкту та містить формальний опис граматики мови HAML. У ньому визначаються лексичні та синтаксичні правила, ієрархія вузлів синтаксичного дерева та способи розбору конструкцій мови. Саме на основі цього файлу Tree-sitter генерує низькорівневий парсер.

Каталог src містить згенеровані артефакти синтаксичного аналізу. Файл parser.c реалізує основний алгоритм розбору, згенерований Tree-sitter на основі граматики. Файл scanner.c (Додаток Б) використовується для реалізації складніших лексичних правил, які неможливо або недоцільно описувати декларативно в grammar.js. Файли grammar.json та node-types.json є проміжними представленнями граматики та типів вузлів синтаксичного дерева, які використовуються інструментами Tree-sitter та редакторами коду. Підкаталог tree_sitter зазвичай містить допоміжні заголовки або структури для інтеграції з API Tree-sitter.

Каталог queries призначений для зберігання запитів Tree-sitter, які використовуються редакторами коду для реалізації семантичних можливостей поверх синтаксичного дерева. Файл highlights.scm визначає правила підсвічування синтаксису на основі типів вузлів дерева. Файл folds.scm описує правила згортання коду, дозволяючи редактору визначати логічні блоки. Файл injections.scm використовується для ін’єкції інших мов усередину HAML-документів, наприклад Ruby або HTML, що є критично важливим для коректної роботи підсвічування у вкладених мовних конструкціях.

Каталог test містить тестові дані для перевірки коректності граматики. Підкаталог corpus включає приклади HAML-коду разом з очікуваною структурою синтаксичного дерева. Такі тести дозволяють автоматично перевіряти правильність роботи парсера після внесення змін до граматики та забезпечують стабільність результатів синтаксичного аналізу.

Файл tree-sitter.json є метаданим описом граматики, який використовується сучасними інструментами Tree-sitter для конфігурації та автоматизації збірки. Він містить інформацію про мову, її назву, версію та інші параметри, необхідні для інтеграції.

Таким чином, структура проєкту tree-sitter-haml чітко розділяє декларативний опис граматики, згенерований парсер, редактор-орієнтовані запити та інфраструктуру тестування. Такий підхід забезпечує масштабованість, зручність супроводу та можливість використання граматики HAML у різних середовищах розробки, зокрема в редакторі коду Neovim.

3.2 Створення зовнішнього сканера

Однією з ключових особливостей мови HAML є використання відступів як основного механізму структурування документа. На відміну від класичних тегових мов розмітки, таких як HTML, рівень вкладеності елементів у HAML визначається кількістю пробілів або табуляцій на початку рядка.

На рівні граматики неможливо коректно обробляти такі конструкції, оскільки визначення відступів потребує збереження історії попередніх станів. Тому Tree-sitter підтримує використання зовнішніх сканерів, які мають власний змінний стан і можуть передавати цю інформацію парсеру.

Зовнішній сканер виконує аналіз початку кожного нового рядка та визначає:

- чи відбувається перехід до нового рядка (NEWLINE);
- чи збільшується рівень відступу відносно попереднього (INDENT);
- чи зменшується рівень вкладеності (DEDENT).

Для створення зовнішнього сканера необхідно створити файл src/scanner.c. У цьому новому файлі необхідно визначити тип enum, який міститиме назви всіх зовнішніх токенів.

```
enum TokenType {
  NEWLINE,
  INDENT,
  DEDENT,
};
```

Також необхідно визначити п’ять функцій із фіксованими назвами, що відповідають імені мови та п’яти діям: створення (create), знищення (destroy), серіалізація (serialize), десеріалізація (deserialize) та сканування (scan).

Функція tree_sitter_haml_external_scanner_create створює та ініціалізує об’єкт сканера, виділяючи пам’ять, ініціалізуючи масив відступів і додаючи початковий рівень відступу 0.

```
typedef struct {
  Array(uint16_t) indents;
} Scanner;
void *tree_sitter_haml_external_scanner_create() {
  Scanner *scanner = ts_calloc(1, sizeof(Scanner));
  array_init(&scanner->indents);
  array_push(&scanner->indents, 0);
  return scanner;
}
```

Функція tree_sitter_haml_external_scanner_destroy відповідає за звільнення ресурсів зовнішнього сканера. Вона приймає вказівник на сканер (payload), очищує масив відступів (indents) та звільняє пам’ять, виділену для структури

Scanner.

```
void tree_sitter_haml_external_scanner_destroy(void *payload) {
  Scanner *scanner = (Scanner *)payload;
  array_delete(&scanner->indents);
  ts_free(scanner);
}
```

Функція tree_sitter_haml_external_scanner_serialize() відповідає за серіалізацію стану зовнішнього сканера. Вона копіює поточний стан об'єкта Scanner у заданий буфер байтів (buffer) та повертає кількість записаних байтів.

Функція виконується щоразу, коли зовнішній сканер успішно розпізнає токен. Вона отримує вказівник на сканер та на буфер, у який потрібно зберегти стан. Максимальна кількість байтів, яку можна записати, визначається константою TREE_SITTER_SERIALIZATION_BUFFER_SIZE, оголошеною в файлі tree_sitter/parser.h.

```
unsigned tree_sitter_haml_external_scanner_serialize(void *payload, char *buffer) {
  Scanner *scanner = (Scanner *)payload;

  unsigned size = 0;
  for (unsigned i = 0; i < scanner->indents.size && size < TREE_SITTER_SERIALIZATION_BUFFER_SIZE; i++) {
    buffer[size++] = (char)*array_get(&scanner->indents, i);
  }

  return size;
}
```

Функція tree_sitter_haml_external_scanner_deserialize() відновлює стан зовнішнього сканера з буфера байтів, отриманого під час серіалізації. Вона приймає вказівник на сканер (payload), вказівник на буфер (buffer) та довжину буфера (length).

У реалізації спочатку очищується існуючий масив відступів (indents) і ініціалізується заново, з додаванням початкового рівня 0. Потім по черзі у стек додаються всі значення з буфера, відновлюючи стан сканера на момент останньої серіалізації.

Ця функція дозволяє парсеру Tree-sitter коректно обробляти редагування та неоднозначності, гарантуючи, що зовнішній сканер завжди відновлюється у правильному стані.

```
void tree_sitter_haml_external_scanner_deserialize(void *payload, const char *buffer, unsigned length) {
  Scanner *scanner = (Scanner *)payload;

  array_delete(&scanner->indents);
  array_init(&scanner->indents);
  array_push(&scanner->indents, 0);

  for (unsigned i = 0; i < length; i++) {
    array_push(&scanner->indents, (unsigned char)buffer[i]);
  }
}
```

Функція tree_sitter_haml_external_scanner_scan виконує основну роботу зовнішнього сканера, визначаючи токени NEWLINE, INDENT та DEDENT на основі відступів у документі HAML.

Спочатку сканер фіксує поточну позицію (mark_end) та ініціалізує змінні для відстеження нового рядка і довжини відступу. Потім він послідовно

пропускає початкові пробіли, табуляції та символи кінця рядка (\r, \f). Табуляція інтерпретується як 8 пробілів. Якщо досягається кінець файлу, це також вважається новим рядком. Якщо після пропуску символів новий рядок не був знайдений, функція повертає false, оскільки токен не розпізнано.

```
// Consume leading whitespace & detect newline
for (;;) {
  if (lexer->lookahead == '\n') {
    found_newline = true;
    indent_length = 0;
    skip(lexer);
  } else if (lexer->lookahead == ' ') {
    indent_length++;
    skip(lexer);
  } else if (lexer->lookahead == '\t') {
    indent_length += 8; // assume tab = 8 spaces
    skip(lexer);
  } else if (lexer->lookahead == '\r' || lexer->lookahead == '\f') {
    skip(lexer);
  } else if (lexer->eof(lexer)) {
    found_newline = true;
    indent_length = 0;
    break;
  } else {
    break;
  }
}
```

Після цього визначається поточний рівень відступу зі стеку.

```
uint16_t current_indent = *array_back(&scanner->indents);
```

Якщо довжина відступу нового рядка більша за поточний рівень і відповідний токен дозволений, значення додається у стек і повертається токен INDENT.

```
// INDENT
if (valid_symbols[INDENT] && indent_length > current_indent) {
  array_push(&scanner->indents, indent_length);
  lexer->result_symbol = INDENT;
  return true;
}
```

Якщо довжина відступу менша за поточний рівень, зі стеку видаляється верхній елемент, і повертається токен DEDENT.

```
// DEDENT
if (valid_symbols[DEDENT] && indent_length < current_indent) {
  array_pop(&scanner->indents);
  lexer->result_symbol = DEDENT;
  return true;
}
```

Якщо відступ не змінився, але знайдено новий рядок, повертається токен NEWLINE. У випадку, коли жодна з умов не виконується, функція повертає false.

```
// NEWLINE
if (valid_symbols[NEWLINE]) {
  lexer->result_symbol = NEWLINE;
  return true;
```

```
}
return false;
```

Таким чином, ця функція забезпечує коректне формування токенів відступів у HAML і підтримує історію вкладеності рядків, необхідну для побудови точного синтаксичного дерева.

3.3 Розробка граматики HAML

Масив externals відповідає за токени які розпізнаються зовнішнім сканером.

```
externals: $ => [
  $._newline,
  $._indent,
  $._dedent
],
```

document визначає основну структуру HAML-документа як послідовність елементів. До таких елементів належать Doctype, теги, Ruby-вставки, блоки Ruby-коду, фільтри, коментарі та простий текст.

```
document: $ => repeat(
  choice(
    $.doctype,
    $.tag,
    $.ruby_insert,
    $.running_ruby,
    $.filter,
    $.plain_text
```

```
  )
),
```

doctype описує декларацію типу документа. Вона починається зі спеціального маркера !!! і може містити опційне ім'я типу (наприклад, HTML або XML). Завершення рядка визначає кінець декларації. Ця конструкція дозволяє парсеру ідентифікувати стандарт документа, який буде згенерований у HTML.

```
doctype: $ => seq(
  "!!!",
  optional(alias($._text, $.doctype_name)),
  $._newline,
),
```

ruby_insert обробляє вставки Ruby-коду, які починаються з символів =, ~, &= або !=. Ці вставки дозволяють виконати код та вставити його результат у HTML з різними способами обробки пробілів або екрануванням. Опціонально можуть включати вкладені блоки контенту під відступом.

```
ruby_insert: $ => seq(
  choice(
    '=',
    '~',  // Whitespace Preservation
    '&=', // Escaping HTML
    '!=', // Unescaping HTML
  ),
  $.ruby_code,
  $._newline,
  optional($._block_content)
```

```
),
```

running_ruby відповідає за виконання Ruby-коду без вставки результату в HTML. Використовується для умов, циклів та інших логічних конструкцій. Також підтримує блоки контенту, вкладені через відступи.

```
running_ruby: $ => seq(
  '-',
  $.ruby_code,
  $._newline,
  optional($._block_content)
),
```

ruby_code описує синтаксис рядка Ruby-коду, включаючи можливість перенесення виразів на кілька рядків за допомогою ком та пробілів. Це термінальне правило, яке визначає зміст виконуваного коду.

```
ruby_code: _ => token(
  prec(1,
    seq(
      /[^\n]+/,
      repeat(
        seq(
          /,[ \t]*\n[ \t]*/,
          /[^\n]+/,
        ),
      ),
    ),
  ),
),
```

_text і plain_text відповідають за звичайний текст HAML, який не містить Ruby-коду або спеціальних символів. Він вставляється у HTML без змін, при цьому plain_text має нижчий пріоритет для розрізнення від інших конструкцій.

```
_text: $ => /[^\n]+/,
plain_text: _ => token(prec(-1, /[^\n]+/)),
```

comment визначає вузол коментаря, який може бути або HAML-коментарем, або HTML-коментарем, і просто делегує парсинг відповідному підправилу залежно від першого символа.

```
comment: $ => choice(
  $._haml_comment,
  $._html_comment
),
```

_comment_condition описує спеціальну умову в дужках, яка використовується лише в HTML-коментарях типу /[if IE]. Вона парситься як один токен з невеликим пріоритетом і дозволяє наявність опційного знака !, а також тексту всередині квадратних дужок.

```
_comment_condition: $ => token(
  prec(
    1,
    seq(
      optional('!'),
      '[',
      optional(/[^\n\]]+/),
      ']'
```

```
    )
  )
),
```

_html_comment описує HTML-коментар у HAML, який починається зі слеша, опційно може містити умову, а потім має власне контент коментаря. Це відповідає синтаксису HAML, де / використовується для коментарів, що перетворюються на HTML.

```
_html_comment: $ => seq(
  '/',
  optional($._comment_condition),
  $._comment_content
),
```

_haml_comment визначає HAML-стиль коментаря, який починається з -# і не потрапляє в HTML-вивід. Після префікса йде контент, який може бути однорядковим або блочним.

```
_haml_comment: $ => seq(
  '-#',
  $._comment_content
),
```

_comment_content описує зміст коментаря, який може бути або простим текстом із переходом на новий рядок, або блочним форматом із відступом, кількома рядками тексту та наступним поверненням рівня відступу. Це дозволяє підтримувати вкладені багаторядкові коментарі на основі індентації.

```
_comment_content: $ => choice(
```

```
    seq(
      $._text,
      $._newline
    ),
    seq(
      $._newline,
      $._indent,
      repeat1($._text),
      $._dedent
    )
  ),
```

filter_name визначає ім’я фільтру, яке використовується для передачі блоку тексту зовнішньому обробнику, наприклад Markdown, Sass, JavaScript або Ruby.

```
filter_name: $ => /[a-zA-Z0-9_-]+/,
```

filter_body описує вміст блоку фільтру як послідовність рядків тексту. Це правило визначає контент, який обробляється відповідним фільтром.

```
filter_body: $ => seq(
  repeat1($._text),
),
```

filter поєднує ім’я фільтру та його тіло. Воно дозволяє описати блок контенту, який обробляється зовнішнім механізмом, та враховує відступи для визначення меж блоку.

```
filter: $ => seq(
```

```
    ':',
    $.filter_name,
    choice(
      $._newline,
      seq(
        $._newline,
        $._indent,
        $.filter_body,
        $._dedent
      )
    ),
  ),
```

Правило tag описує синтаксичну конструкцію HAML-тегу як базового елемента мови розмітки. Воно визначає допустиму послідовність складових тегу, починаючи з його імені та закінчуючи вмістом або закриттям.

Початковий елемент правила відповідає ідентифікації тегу. Конструкція choice($.tag_name, $.tag_class, $.tag_id) дозволяє починати опис тегу або з явного імені тегу, або безпосередньо з класу чи ідентифікатора що відображає специфіку HAML, де ім’я тегу може бути неявним і виводитися за замовчуванням, наприклад, при використанні лише класів або ідентифікаторів.

```
  tag: $ => seq(
    // Tag name
    choice(
      $.tag_name,
      $.tag_class,
      $.tag_id,
    ),
```

Наступна частина правила описує список CSS-класів та ідентифікаторів. Вона дозволяє довільну кількість послідовних класів та ідентифікаторів, які додаються до тегу після його імені або замість нього.

```
// Class/id list
repeat(
  choice(
    $.tag_class,
    $.tag_id
  )
),
```

Далі правило містить групу альтернатив, яка забезпечує коректну обробку атрибутів тегу. Комбінації з hash_attributes, list_attributes та object_reference описують різні способи задання атрибутів у HAML, зокрема у вигляді Ruby-хешів, списків або посилань на Ruby-об'єкти. Використання декількох варіантів seq гарантує, що кожен тип атрибутів може з'явитися не більше одного разу, але порядок їх слідування залишається довільним.

```
choice(
  seq(optional($.hash_attributes), optional($.list_attributes), optional($.object_reference)),
  seq(optional($.hash_attributes), optional($.object_reference), optional($.list_attributes)),
  seq(optional($.list_attributes), optional($.hash_attributes), optional($.object_reference)),
  seq(optional($.list_attributes), optional($.object_reference), optional($.hash_attributes)),
  seq(optional($.object_reference), optional($.hash_attributes), optional($.list_attributes)),
```

```
    seq(optional($.object_reference), optional($.list_attributes),
optional($.hash_attributes)),
  ),
```

Опційна частина optional(choice('>', '<', '<>', '><')) відповідає за символи керування пробілами. Ці маркери використовуються в HAML для явного контролю видалення або збереження пробілів перед і після тегу при генерації HTML.

Завершальна частина правила визначає тип вмісту тегу або спосіб його закриття. Перша альтернатива описує теги без вкладеного блоку. Вона дозволяє або self-closing теги, позначені символом /, або inline-контент, представлений правилом _inline_content. У будь-якому випадку конструкція завершується символом нового рядка, що відповідає однорядковому запису тегу. Друга альтернатива описує теги з блоковим контентом: після завершення рядка виконується розбір вкладеного блоку за допомогою правила _block_content.

```
// Either a closing tag or inline content or block content.
choice(
  // Either a closing tag or inline content.
  seq(
    optional(
     choice(
       // Self-closing (void tags)
       '/',

       // Inline content.
       $._inline_content,
     ),
    ),
```

```
    // End of tag
    $._newline
  ),
  seq(
    // End of tag
    $._newline,
    $._block_content,
  )
)
```

_inline_content визначає вміст тегу, який розташовується в одному рядку, включаючи Ruby-вставки та простий текст.

```
_inline_content: $ => choice(
  $.ruby_insert,
  $.plain_text
),
```

_block_content описує контент, вкладений під тег або Ruby-блоком. Використовує відступи та деденти для визначення меж блоку і дозволяє вкладення тегів, Ruby-вставок, фільтрів та тексту.

```
_block_content: $ => seq(
  $._indent,
  repeat(
   choice(
     $.tag,
      $.ruby_insert,
      $.running_ruby,
      $.plain_text,
```

```
      $.filter
    )
  ),
  $._dedent
),
```

tag_name, tag_class і tag_id визначають основне ім’я елемента, класи та ідентифікатори тегу у CSS-подібній нотації. Це дозволяє точно розпізнати тег та його властивості.

```
_identifier: _ => /[-:\w]+/,

tag_name: $ => seq(
  '%',
  $._identifier,
),

tag_class: $ => seq(
  '.',
  $._identifier,
),

tag_id: $ => seq(
  '#',
  $._identifier,
),
```

object_reference дозволяє посилатися на зовнішні об’єкти у тегах через квадратні дужки, підтримуючи передачу складних даних.

```
object_reference: _ => seq(
  '[',
  repeat(/[^\[\]]/),  // anything except brackets
  ']'
),
```

hash_attributes та _hash_attribute_content описують атрибути тегу у форматі Ruby-хешу, включаючи підтримку вкладених хешів для складних структур. Це дозволяє описати багато атрибутів у одному тегу.

```
hash_attributes: $ => seq(
  '{',
  optional($._hash_attribute_content),
  '}'
),

_hash_attribute_content: $ => repeat1(
  choice(
    /[^{}]/,
    $._nested_hash_attributes
  )
),
```

_nested_hash_attributes – приховане правило для підтримки вкладених Ruby-хешів у атрибутах, що забезпечує коректне парсування складних структур.

```
// Hidden rule for nested hashes
_nested_hash_attributes: $ => seq(
  '{',
```

```
  optional($._hash_attribute_content),
  '}'
),
```

list_attributes описує атрибути у списковому форматі, обмеженому круглими дужками, що дозволяє вказувати кілька значень для тегу.

```
list_attributes: _ => seq(
  '(',
  repeat(/[^()]/),     // anything except parentheses
  ')'
)
```

Створена граматика описує ключові синтаксичні конструкції HAML та забезпечує їх коректне розпізнавання та дозволяє згенерувати HAML-парсер, здатний точно відтворювати ієрархічну структуру документа, коректно обробляти вкладеність на основі відступів і формувати узгоджене синтаксичне дерево для подальшого маркування синтаксису на основі запитів підсвічування (highlighting queries).

3.4 Створення засобів для маркування синтаксису

Після написання граматики важливим етапом є опис запитів (queries), які визначають спосіб використання синтаксичного дерева редактором коду. Запити дозволяють використовувати структуру абстрактного синтаксичного дерева з метою реалізації таких функцій, як підсвітка синтаксису та визначення блоків згортання коду. У контексті Neovim запити Tree-sitter зберігаються у вигляді файлів з розширенням .scm і застосовуються до конкретної мови.

Файл injections.scm використовується для визначення фрагментів коду, всередині яких необхідно активувати інший синтаксичний аналізатор. Це є

особливо актуальним для HAML, оскільки мова тісно інтегрується з Ruby та дозволяє вставляти Ruby-код у різних синтаксичних конструкціях.

```
((ruby_code) @injection.content
  (#set! injection.language ruby))

((hash_attributes) @injection.content
  (#set! injection.language ruby))

((list_attributes) @injection.content
  (#set! injection.language ruby))

((object_reference) @injection.content
  (#set! injection.language ruby))
```

У даному випадку всі вузли синтаксичного дерева, що відповідають Ruby-коду (ruby_code), атрибутам у вигляді Ruby-хешів (hash_attributes), списків атрибутів (list_attributes) та посиланням на об’єкти (object_reference), позначаються як вміст для ін’єкції з мовою ruby. Це дозволяє Neovim застосовувати Ruby-парсер до відповідних фрагментів HAML-файлу, забезпечуючи коректну підсвітку.

Окремо обробляються HAML-фільтри, які можуть містити код іншої мови, наприклад JavaScript або Markdown. Для цього ім’я фільтра використовується як ідентифікатор мови ін’єкції, тоді як тіло фільтра визначається як вміст, до якого застосовується відповідний синтаксичний аналізатор. Такий підхід дозволяє динамічно визначати мову ін’єкції на основі синтаксису самого HAML-файлу.

```
(filter
  (filter_name) @injection.language
```

```
  (filter_body) @injection.content)
```

Файл highlights.scm відповідає за зіставлення вузлів синтаксичного дерева з семантичними групами підсвітки. На відміну від класичних лексичних підсвіток, Tree-sitter дозволяє виконувати це зіставлення на основі структури коду, що підвищує точність візуального маркування.

У межах реалізованих запитів виділяються розділові символи HAML, такі як %, ., #, які використовуються для опису тегів, класів та ідентифікаторів.

```
[
  "%"
  "."
  "#"
] @punctuation.delimiter
```

Окремо обробляється директива doctype, де ключове слово та його значення маркуються різними семантичними групами.

```
;; --- Doctype ---
(doctype
  "!!!" @keyword
  (doctype_name)? @string.special)
```

Імена тегів, класів та ідентифікаторів отримують відповідні типи підсвітки, що дозволяє візуально розрізняти структуру HAML-документа.

```
;; --- Tags (%p, .cls, #id) ---
(tag_name) @tag
(tag_class) @property
(tag_id) @constant
```

Атрибути у вигляді Ruby-структур виділяються разом із дужками, підкреслюючи межі вкладеного коду.

```
(object_reference) @punctuation.bracket

(hash_attributes
  "{" @punctuation.bracket
  "}" @punctuation.bracket)

(list_attributes
  "(" @punctuation.bracket
  ")" @punctuation.bracket)
```

Особливу увагу приділено вставкам Ruby-коду. Різні оператори вставки (=, ~, &=, !=, -) маркуються як оператори, тоді як сам Ruby-код позначається як вбудований (embedded). Це дозволяє чітко відокремити шаблонну частину HAML від виконуваного коду.

```
;; --- Ruby Code ---
(ruby_insert
  "=" @operator             ;; normal insert
  (ruby_code) @embedded)    ;; code inside

(ruby_insert
  "~" @operator)            ;; whitespace preservation
(ruby_insert
  "&=" @operator)           ;; escape
(ruby_insert
  "!=" @operator)           ;; unescape
```

```
(running_ruby
  "-" @operator
  (ruby_code) @embedded)

(ruby_code) @embedded
```

Також реалізована підсвітка звичайного тексту, коментарів та спеціальних символів керування пробілами, які впливають на форматування згенерованого HTML, але не мають прямого відображення у вихідному коді.

```
;; --- Plain text ---
(plain_text) @string

;; --- Whitespace control flags '>', '<', '<>', '><' ---
(tag
  ">" @punctuation.special)
(tag
  "<" @punctuation.special)
(tag
  "<>" @punctuation.special)
(tag
  "><" @punctuation.special)

(comment) @comment
```

Файл folds.scm визначає синтаксичні конструкції, які можуть бути згорнуті в редакторі. Для HAML це є особливо важливим, оскільки структура документа значною мірою визначається вкладеністю та відступами.

```
[
  (filter)
  (running_ruby)
  (ruby_insert)
  (filter)
  (tag)
  (comment)
] @fold
```

У якості вузлів для згортання обрані теги, фільтри, коментарі, а також конструкції вставки та виконання Ruby-коду. Це дозволяє користувачу ефективно навігувати великими HAML-файлами, приховуючи деталізовані блоки коду та фокусуючись на загальній структурі документа.

Таким чином, використання Tree-sitter queries забезпечує глибоку інтеграцію HAML у Neovim, поєднуючи структурну обізнаність синтаксичного дерева з розширеними можливостями сучасного редактора коду.

3.5 Інтеграція парсера HAML в Neovim

Для інтеграції розробленого парсера HAML на основі Tree-sitter у редактор Neovim було додано відповідну конфігурацію до файлу ~/.config/nvim/init.lua.

```
local parser_config = require "nvim-treesitter.parsers".get_parser_configs()
parser_config.haml = {
  install_info = {
    url = "https://github.com/BohdaR/tree-sitter-haml",
    files = {"src/parser.c", "src/scanner.c"},

    branch = "main",
```

```
    generate_requires_npm = false,
    requires_generate_from_grammar = true,
  },
  filetype = "haml",
}
```

Зазначений фрагмент коду розширює стандартну конфігурацію плагіна nvim-treesitter, додаючи підтримку нового парсера tree-sitter-haml.

На початковому етапі з модуля nvim-treesitter.parsers отримується таблиця вже визначених парсерів, після чого для мови HAML створюється новий запис. У межах цього запису задається адреса репозиторію з вихідним кодом парсера, перелік файлів, необхідних для його компіляції (parser.c та scanner.c), а також додаткові параметри конфігурації, зокрема назва гілки репозиторію та вимога генерації парсера на основі граматичної специфікації.

Завершальним кроком є прив'язка створеного парсера до типу файлів haml, що дозволяє Neovim автоматично завантажувати та використовувати відповідний синтаксичний аналізатор.

Також було додано необхідні Tree-sitter запити для HAML. Файли folds.scm, highlights.scm та injections.scm були скопійовані з директорії парсера до локальної папки ~/.config/nvim/queries/haml. Це дозволило Neovim застосовувати правила підсвітки та визначення блоків згортання коду визначені в цих файлах, під час роботи з HAML-файлами.

3.6 Перевірка працездатності розробленої системи візуального маркування синтаксису HAML для редактора коду Neovim на основі Tree-sitter

На рисунку 3.2 зображено повідомлення про успішне встановлення парсера HAML після виконання команди TSUpdate, яка відповідає за встановлення та оновлення парсерів в Neovim.

Рисунок 3.2 – Повідомлення про успішне встановлення парсера HAML в Neovim

На рисунку 3.3 зображено файл HAML, праворуч код HAML з тегами doctype, зліва згенероване синтаксичне дерево. З рисунка 3.3 видно що синтаксичене дерево, виглядає абсолютно коректним: парсер правильно розпізнав усі директиви doctype, виділив їх у окремі вузли та точно вказав позиції у тексті. Підсвічування синтаксису на основі цього дерева також працює коректно. Всі елементи doctype мають відповідне кольорове маркування, без пропусків чи помилкових виділення.

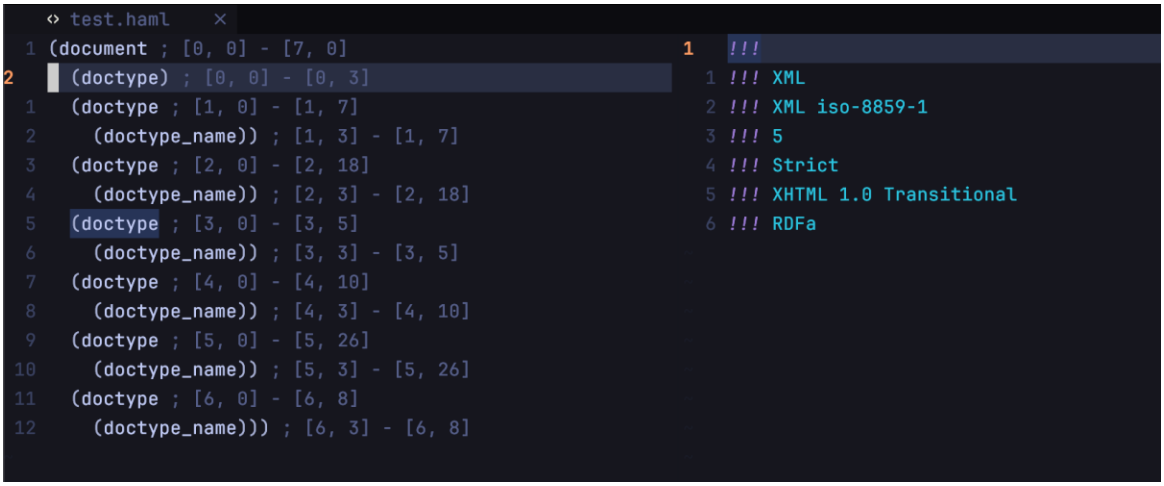


Рисунок 3.3 – Синтаксичне дерево та маркування синтаксису елементів doctype

HAML-фільтри дозволяють виконувати код інших мов програмування та використовувати результат його виконання безпосередньо в шаблоні. З рисунка 3.4 видно, що відповідні фрагменти синтаксичного дерева для мови Ruby були згенеровані коректно із застосуванням механізму ін’єкцій. Це дозволяє коректно інтегрувати Ruby-код у загальну структуру HAML-документа. Підсвітка синтаксису для вбудованого Ruby-коду відповідає його внутрішній структурі та працює стабільно.

Рисунок 3.4 – Синтаксичне дерево та маркування синтаксису HAML-фільтрів

HAML-фільтри дозволяють виконувати код інших мов програмування та використовувати результат його виконання безпосередньо в шаблоні. З рисунка 3.4 видно, що відповідні фрагменти синтаксичного дерева для мови Ruby були згенеровані коректно із застосуванням механізму ін’єкцій. Це дозволяє коректно інтегрувати Ruby-код у загальну структуру HAML-документа. Підсвітка синтаксису для вбудованого Ruby-коду відповідає його внутрішній структурі та працює стабільно.

HAML-теги є базовими структурними елементами шаблону та визначають ієрархію HTML-документа. Вони можуть містити ім’я тега, класи, ідентифікатори, атрибути, а також вкладений контент. Аналіз синтаксичного дерева, наведеного на рисунку 3.5, демонструє, що теги коректно розпізнаються парсером як окремі вузли, а їхні складові частини представлені у вигляді відповідних підвузлів, зокрема tag_name, tag_class та tag_id. Це забезпечує коректну інтерпретацію структури документа та правильне маркування синтаксису на основі реальної ієрархії HAML-шаблону.

HAML-теги є базовими структурними елементами шаблону та визначають ієрархію HTML-документа. Вони можуть містити ім’я тега, класи,

ідентифікатори, атрибути, а також вкладений контент. Аналіз синтаксичного дерева, наведеного на рисунку 3.5, демонструє, що теги коректно розпізнаються парсером як окремі вузли, а їхні складові частини представлені у вигляді відповідних підвузлів, зокрема tag_name, tag_class та tag_id. Це забезпечує коректну інтерпретацію структури документа та правильне маркування синтаксису на основі реальної ієрархії HAML-шаблону.

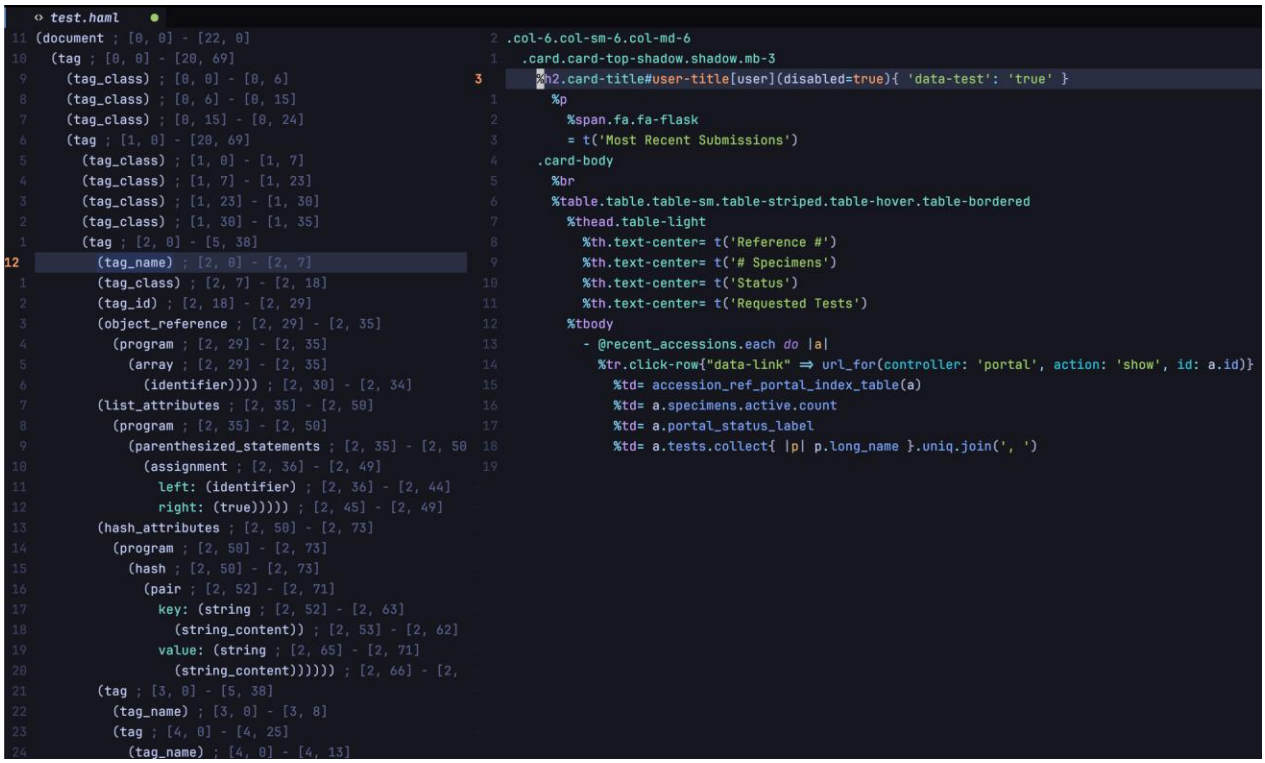


Рисунок 3.5 – Синтаксичне дерево та маркування синтаксису HAML-тегів

На рисунку 3.6 наведено приклад HAML-коду, що містить різноманітний текстовий контент, включно з Unicode-символами та спеціальними знаками. З аналізу синтаксичного дерева видно, що воно побудоване коректно: парсер правильно розпізнав усі рядки, які не починаються з керуючих символів, та виділив їх в окремі вузли типу plain_text. Це підтверджує коректну роботу правила обробки звичайного тексту та запобігає помилковому трактуванню текстових рядків як синтаксичних конструкцій HAML. Підсвітка синтаксису, побудована на основі цього дерева, працює коректно та забезпечує стабільне маркування всіх елементів без пропусків чи помилкових виділень.

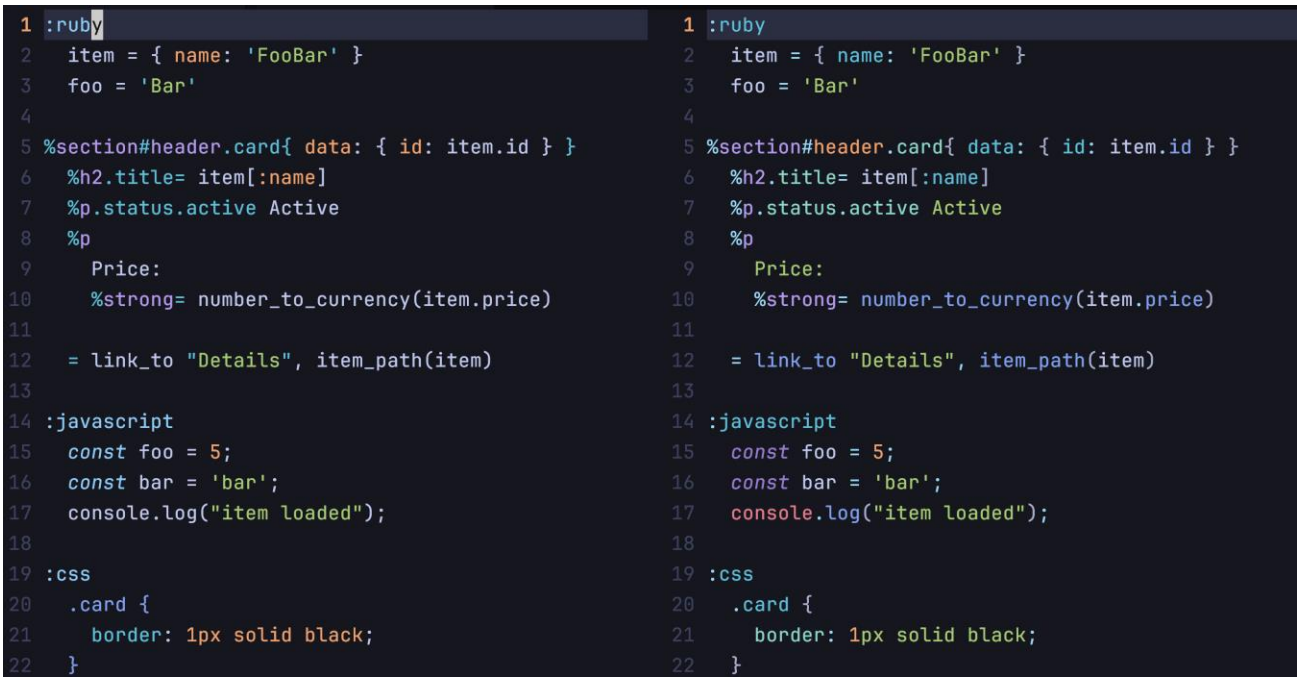
Рисунок 3.6 – Синтаксичне дерево та маркування синтаксису звичайного тексту в HAML

На лівій частині рисунка 3.7 представлено попередній варіант синтаксичного підсвічування HAML, який базується на регулярних виразах. Видно, що більшість елементів підсвічуються одноманітно: HAML-теги, Ruby-вирази та вкладений код JavaScript і CSS не мають чіткого семантичного розділення. Наприклад, ключові слова JavaScript (const) та виклики функцій (console.log) не відрізняються візуально від звичайних ідентифікаторів, а рядки та числові літерали не завжди коректно маркуються. Аналогічно, Ruby-код у HAML-атрибутах та inline-виразах не має стабільного та контекстно залежного підсвічування.

Праворуч на рисунку 3.7 показано результат використання Tree-sitter-орієнтованого підходу. Синтаксичне підсвічування є значно точнішим і семантично насиченим: рядкові літерали та методи мають різні стилі, що покращує читабельність коду. Вкладені HAML-фільтри :javascript та :css коректно розпізнаються, а їхній вміст підсвічується відповідно до граматики цільової мови. Зокрема, ключові слова JavaScript, числові значення та виклики функцій виділяються окремо, що неможливо досягти за допомогою простих регулярних виразів.

Таким чином, новий підхід забезпечує контекстно-залежне підсвічування, коректну обробку вкладених мов та значно покращує маркування синтаксичних конструкцій. Це підвищує зручність читання коду, зменшує когнітивне

навантаження на розробника та підтверджує доцільність використання Tree-sitter для візуального маркування синтаксису HAML у редакторі Neovim.



```
:ruby
  item = { name: 'FooBar' }
  foo = 'Bar'

%section#header.card{ data: { id: item.id } }
  %h2.title= item[:name]
  %p.status.active Active
  %p
    Price:
    %strong= number_to_currency(item.price)

  = link_to "Details", item_path(item)

:javascript
  const foo = 5;
  const bar = 'bar';
  console.log("item loaded");

:css
  .card {
    border: 1px solid black;
  }
```

```
:ruby
  item = { name: 'FooBar' }
  foo = 'Bar'

%section#header.card{ data: { id: item.id } }
  %h2.title= item[:name]
  %p.status.active Active
  %p
    Price:
    %strong= number_to_currency(item.price)

  = link_to "Details", item_path(item)

:javascript
  const foo = 5;
  const bar = 'bar';
  console.log("item loaded");

:css
  .card {
    border: 1px solid black;
  }
```

Рисунок 3.7 –Маркування синтаксису на основі регулярних виразів та на основі Tree-sitter

Враховуючи вищевказане, можна зробити висновок, що розроблена система візуального маркування синтаксису HAML для редактора коду Neovim на основі Tree-sitter працює належним чином.

ВИСНОВКИ

У першому розділі було показано, що HAML забезпечує зручну роботу з великими шаблонами в професійних проєктах. Також встановлено, що редактор коду Neovim є кращим рішенням серед розглянутих. Його перевага полягає у високій продуктивності та мінімальному споживанні системних ресурсів Однак, маркування синтаксису HAML у Neovim базується на підході, який використовує регулярні вирази. Такий підхід не забезпечує коректного структурного аналізу та є чутливим до вкладеності та контексту, що обмежує можливості точного візуального маркування складних конструкцій.

У другому розділі було проведено вибір генератора парсерів для вдосконалення сервісу візуального маркування синтаксису HAML для редактора коду Neovim, що дозволило вибрати Tree-sitter, як засіб вдосконалення маркування синтаксису HAML у Neovim. Його ключовими перевагами є підхід, що забезпечує стабільну продуктивність незалежно від розміру файлу, а також стійкість до синтаксичних помилок, яка гарантує коректне формування синтаксичного дерева навіть за наявності некоректних або незавершених конструкцій. Це створює передумови для точного візуального маркування синтаксису.

Під час виконання третього розділу було розроблено структуру проєкту для парсера HAML на основі Tree-sitter. Також було розроблено граматику HAML, засоби маркування синтаксису та інтегровано розроблений парсер в Neovim.

Проведена перевірка працездатності вдосконаленого сервісу візуального маркування синтаксису HAML для редактора коду Neovim на основі Tree-sitter показала, що він працює належним чином.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Мельничук С.І. Магістерська робота: методичні вказівки до змісту та оформлення для студентів спеціальності 123 – Комп'ютерна інженерія. Івано-Франківськ, ІФНТУНГ. – 2024. 28 с.
2. Бабчук С.М., Шушваль Б.Р. Сервіс візуального маркування синтаксису HAML для редактора коду Neovim на основі Tree-sitter. Збірник тез доповідей VI Міжнародної наукової конференції «Теорія модернізації в контексті сучасної світової науки». Івано-Франківськ. ІФНТУНГ. – 2025. – С. 281-282.
3. Підсвічування синтаксису. URL: https://uk.wikipedia.org/wiki/%D0%9F%D1%96%D0%B4%D1%81%D0%B2%D1%96%D1%87%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F_%D1%81%D0%B8%D0%BD%D1%82%D0%B0%D0%BA%D1%81%D0%B8%D1%81%D1%83 (дата звернення: 16.09.2025)
4. Syntax highlighting. URL: https://en.wikipedia.org/wiki/Syntax_highlighting (дата звернення: 16.09.2025)
5. HAML documentation. URL: https://haml.info/docs/yardoc/file.REFERENCE.html (дата звернення: 16.09.2025)
6. Neovim documentation. URL: https://neovim.io/ (дата звернення: 16.12.2025)
7. What is a templating language. URL: https://stackoverflow.com/questions/4026597/what-is-a-templating-language (дата звернення: 17.09.2025)
8. Web template system. URL: https://en.wikipedia.org/wiki/Web_template_system (дата звернення: 18.09.2025)
9. An easy to use but powerful templating system for Ruby. URL: https://github.com/ruby/erb (дата звернення: 19.09.2025)

10. ERB documentation. URL: https://docs.ruby-lang.org/en/master/ERB.html (дата звернення: 19.09.2025)
11. Clean Code in ERB Templates: Ruby on Rails Basics. URL: https://medium.com/nyc-ruby-on-rails/clean-code-in-erb-templates-ruby-on-rails-basics-374ad48dd95e https://medium.com/nyc-ruby-on-rails/clean-code-in-erb-templates-ruby-on-rails-basics-374ad48dd95e (дата звернення: 19.09.2025)
12. Liquid documentation. URL: https://shopify.github.io/liquid/ (дата звернення: 19.09.2025)
13. Liquid markup language. Safe, customer facing template language for flexible web apps. URL: https://github.com/Shopify/liquid (дата звернення: 19.09.2025)
14. Liquid reference. URL: https://shopify.dev/docs/api/liquid (дата звернення: 19.09.2025)
15. HAML. URL: https://uk.wikipedia.org/wiki/Haml (дата звернення: 20.09.2025)
16. HTML Abstraction Markup Language – A Markup Haiku. URL: https://github.com/haml/haml (дата звернення: 21.09.2025)
17. GitLab HAML documentation. URL: https://docs.gitlab.com/development/fe_guide/haml/ (дата звернення: 22.09.2025)
18. Best Text Editors to Speed up Your Workflow. URL: https://kinsta.com/blog/best-text-editors/ (дата звернення: 23.09.2025).
19. Code Editors: My Top 7 Picks. URL: https://maxwellj.vivaldi.net/2025/04/03/code-editors-my-top-7-picks/ (дата звернення: 24.09.2025)
20. Text Editing, Done Right. URL: https://www.sublimetext.com/ (дата звернення: 25.09.2025)
21. Sublime Text. URL: https://en.wikipedia.org/wiki/Sublime_Text (дата звернення: 25.09.2025)

22. Visual Studio Code. URL: https://en.wikipedia.org/wiki/Visual_Studio_Code (дата звернення: 26.09.2025)
23. The open source AI code editor. URL: https://code.visualstudio.com/ (дата звернення: 27.09.2025)
24. Vim-fork focused on extensibility and usability. URL: https://github.com/neovim/neovim (дата звернення: 28.09.2025)
25. A Vim hater's guide to Neovim. URL: https://medium.com/@dinithwalpitagama/a-vim-haters-guide-to-neovim-6f235551689e (дата звернення: 29.09.2025)
26. Comparison of parser generators. URL: https://en.wikipedia.org/wiki/Comparison_of_parser_generators (дата звернення: 30.09.2025)
27. Parser generators. URL: https://web.mit.edu/6.005/www/fa15/classes/18-parser-generators/ (дата звернення: 30.09.2025)
28. Practical Algorithms for Incremental Software Development Environments. URL: https://www2.eecs.berkeley.edu/Pubs/TechRpts/1997/CSD-97-946.pdf (дата звернення: 01.10.2025)
29. Incremental Analysis of Real Programming Languages. URL: https://harmonia.cs.berkeley.edu/papers/twagner-glr.pdf (дата звернення: 01.10.2025)
30. Official ANTLR Website & Docs. URL: https://www.antlr.org/ (дата звернення: 02.10.2025)
31. ANTLR GitHub Repository. URL https://github.com/antlr/antlr4 (дата звернення: 02.10.2025)
32. The ANTLR Mega Tutorial. URL: https://tomassetti.me/antlr-mega-tutorial/ (дата звернення: 03.10.2025)
33. ANTLR Runtime API Documentation. URL: https://www.antlr.org/api/ (дата звернення: 04.10.2025)
34. ANTLR Development Tools Page – Documentation on plugins and tools. URL: https://www.antlr.org/tools.html (дата звернення: 05.10.2025)

35. ANTLR. URL: https://en.wikipedia.org/wiki/ANTLR (дата звернення: 06.20.2025)
36.JavaCC official site. URL: https://javacc.github.io/javacc/ (дата звернення: 07.10.2025)
37. JavaCC GitHub repo. URL: https://github.com/javacc/javacc (дата звернення: 07.10.2025)
38. Baeldung JavaCC guide. URL: https://www.baeldung.com/javacc (дата звернення: 08.10.2025)
39. Wikipedia – JavaCC. URL: https://en.wikipedia.org/wiki/JavaCC (дата звернення: 08.10.2025)
40. Princeton JavaCC manual (PDF). URL: https://www.cs.princeton.edu/~appel/modern/java/Javacc.pdf (дата звернення: 10.10.2025)
41.tree-sitter explained. URL: https://www.youtube.com/watch?v=09-9LltqWLY (дата звернення: 11.10.2025)
42.SableCC official site. URL: https://sablecc.org/ (дата звернення: 29.09.2025)
43. SableCC GitHub repository. URL: https://github.com/SableCC/sablecc (дата звернення: 12.10.2025)
44. SableCC documentation. URL: https://sablecc.org/documentation/ (дата звернення: 17.10.2025)
45. Wikipedia – SableCC. URL: https://en.wikipedia.org/wiki/SableCC (дата звернення: 18.10.2025)
46. SableCC tutorial (University of Alberta). URL: https://www.cs.ualberta.ca/~laks/courses/cmput415/notes/sablecc.pdf (дата звернення: 18.10.2025)
47. Tree-sitter documentation. URL: https://tree-sitter.github.io/tree-sitter/ (дата звернення: 18.10.2025)
48. Tree-sitter: a new parsing system for programming tools – GitHub Universe 2017. URL: https://www.youtube.com/watch?v=a1rC79DHpmY (дата звернення: 18.10.2025)

49. Tree-sitter – a new parsing system for programming tools by Max Brunsfeld URL: https://www.youtube.com/watch?v=Jes3bD6P0To (дата звернення: 19.10.2025)
50. Tree-sitter – a new parsing system for programming tools. URL: https://www.thestrangeloop.com/2018/tree-sitter---a-new-parsing-system-for-programming-tools.html (дата звернення: 19.10.2025)
51. Tree-sitter A new parsing system for programming tools. URL: https://www.youtube.com/watch?v=0CGzC_iss-8 (дата звернення: 19.10.2025)

Додатки

Додаток А

Лістинг файлу граматики grammar.js

```
/**
 * @file HAML Parser
 * @author Bohdan Shushval <bohdanshushval@gmail.com>
 * @license MIT
 */

/// <reference types="tree-sitter-cli/dsl" />
// @ts-check

module.exports = grammar({
  name: 'haml',

  externals: $ => [
    $._newline,
    $._indent,
    $._dedent
  ],

  rules: {
    document: $ => repeat(
      choice(
        $.doctype,
        $.tag,
        $.ruby_insert,
        $.running_ruby,
        $.filter,
```

```
    $.plain_text,
    $.comment
  )
),

doctype: $ => seq(
  "!!!",
  optional(alias($._text, $.doctype_name)),
  $._newline,
),

comment: $ => choice(
  $._haml_comment,
  $._html_comment
),

_comment_condition: $ => token(
  prec(
    1,
    seq(
      optional('!'),
      '[',
      optional(/[^\n\]]+/),
      ']'
    )
  )
),
```

```
_html_comment: $ => seq(
  '/',
  optional($._comment_condition),
  $._comment_content
),

_haml_comment: $ => seq(
  '-#',
  $._comment_content
),

_comment_content: $ => choice(
  seq(
    $._text,
    $._newline
  ),
  seq(
    $._newline,
    $._indent,
    repeat1($._text),
    $._dedent
  )
),
_text: $ => /[^\n]+/,

ruby_insert: $ => seq(
  choice(
    '=',
```

```
    '~',  // Whitespace Preservation
    '&=', // Escaping HTML
    '!=', // Unescaping HTML
  ),
  $.ruby_code,
  $._newline,
  optional($._block_content)
),

running_ruby: $ => seq(
  '-',
  $.ruby_code,
  $._newline,
  optional($._block_content)
),

ruby_code: _ => token(
  prec(1,
    seq(
      /[^\n]+/,
      repeat(
        seq(
          /,[ \t]*\n[ \t]*/,
          /[^\n]+/,
        ),
      ),
    ),
  ),
```

```
),

_text: $ => /[^\n]+/,
plain_text: _ => token(prec(-1, /[^\n]+/)),

filter_name: $ => /[a-zA-Z0-9_-]+/,

filter_body: $ => seq(
  repeat1($._text),
),

filter: $ => seq(
  ':',
  $.filter_name,
  choice(
    $._newline,
    seq(
      $._newline,
      $._indent,
      $.filter_body,
      $._dedent
    )
  ),
),

tag: $ => seq(
  // Tag name
  choice(
```

```
      $.tag_name,
      $.tag_class,
      $.tag_id,
    ),
    // Class/id list
    repeat(
      choice(
        $.tag_class,
        $.tag_id
      )
    ),

    // Make sure each attribute appears at most one time.
    choice(
      seq(optional($.hash_attributes), optional($.list_attributes), optional($.object_reference)),
      seq(optional($.hash_attributes), optional($.object_reference), optional($.list_attributes)),
      seq(optional($.list_attributes), optional($.hash_attributes), optional($.object_reference)),
      seq(optional($.list_attributes), optional($.object_reference), optional($.hash_attributes)),
      seq(optional($.object_reference), optional($.hash_attributes), optional($.list_attributes)),
      seq(optional($.object_reference), optional($.list_attributes), optional($.hash_attributes)),
    ),
```

```
    // Whitespace Removal
    optional(choice('>', '<', '<>', '><')),

    // Either a closing tag or inline content or block content.
    choice(
      // Either a closing tag or inline content.
      seq(
        optional(
          choice(
            // Self-closing (void tags)
            '/',

            // Inline content.
            $._inline_content,
          ),
        ),

        // End of tag
        $._newline
      ),
      seq(
        // End of tag
        $._newline,
        $._block_content,
      )
    )
  ),
```

```
_inline_content: $ => choice(
  $.ruby_insert,
  $.plain_text
),

_block_content: $ => seq(
  $._indent,
  repeat(
    choice(
      $.tag,
      $.ruby_insert,
      $.running_ruby,
      $.plain_text,
      $.filter,
      $.comment
    )
  ),
  $._dedent
),

tag_name: $ => seq(
  '%',
  $._identifier,
),

tag_class: $ => seq(
  '.',
  $._identifier,
```

```
),

tag_id: $ => seq(
  '#',
  $._identifier,
),

_identifier: _ => /[-:\w]+/,

object_reference: _ => seq(
  '[',
  repeat(/[^\[\]]/),  // anything except brackets
  ']'
),

hash_attributes: $ => seq(
  '{',
  optional($._hash_attribute_content),
  '}'
),

_hash_attribute_content: $ => repeat1(
  choice(
    /[^{}]/,                    // normal chars
    $._nested_hash_attributes
  )

),
```

```
    // Hidden rule for nested hashes
    _nested_hash_attributes: $ => seq(
      '{',
      optional($._hash_attribute_content),
      '}'
    ),

    list_attributes: _ => seq(
      '(',
      repeat(/[^()]/),     // anything except parentheses
      ')'
    ),
  }
})
```

Додаток Б

Лістинг файлу зовнішнього сканера src/scanner.c

```
#include "tree_sitter/alloc.h"
#include "tree_sitter/array.h"
#include "tree_sitter/parser.h"

#include <stdint.h>
#include <stdbool.h>

enum TokenType {
  NEWLINE,
  INDENT,
  DEDENT,
};

typedef struct {
  Array(uint16_t) indents;
} Scanner;

static inline void skip(TSLexer *lexer) { lexer->advance(lexer, true); }

bool tree_sitter_haml_external_scanner_scan(void *payload, TSLexer *lexer,
const bool *valid_symbols) {
  Scanner *scanner = (Scanner *)payload;

  lexer->mark_end(lexer);

  bool found_newline = false;
```

Продовження додатка Б

```
uint32_t indent_length = 0;

// Consume leading whitespace & detect newline
for (;;) {
  if (lexer->lookahead == '\n') {
    found_newline = true;
    indent_length = 0;
    skip(lexer);
  } else if (lexer->lookahead == ' ') {
    indent_length++;
    skip(lexer);
  } else if (lexer->lookahead == '\t') {
    indent_length += 8; // assume tab = 8 spaces
    skip(lexer);
  } else if (lexer->lookahead == '\r' || lexer->lookahead == '\f') {
    skip(lexer);
  } else if (lexer->eof(lexer)) {
    found_newline = true;
    indent_length = 0;
    break;
  } else {
    break;
  }
}

if (!found_newline) return false;

uint16_t current_indent = *array_back(&scanner->indents);
```

Продовження додатка Б

```
  // INDENT
  if (valid_symbols[INDENT] && indent_length > current_indent) {
    array_push(&scanner->indents, indent_length);
    lexer->result_symbol = INDENT;
    return true;
  }

  // DEDENT
  if (valid_symbols[DEDENT] && indent_length < current_indent) {
    array_pop(&scanner->indents);
    lexer->result_symbol = DEDENT;
    return true;
  }

  // NEWLINE
  if (valid_symbols[NEWLINE]) {
    lexer->result_symbol = NEWLINE;
    return true;
  }

  return false;
}

unsigned tree_sitter_haml_external_scanner_serialize(void *payload, char
*buffer) {
  Scanner *scanner = (Scanner *)payload;
```

```
  unsigned size = 0;
```

Продовження додатка Б

```
  for (unsigned i = 0; i < scanner->indents.size && size <
TREE_SITTER_SERIALIZATION_BUFFER_SIZE; i++) {
    buffer[size++] = (char)*array_get(&scanner->indents, i);
  }

  return size;
}

void tree_sitter_haml_external_scanner_deserialize(void *payload, const char
*buffer, unsigned length) {
  Scanner *scanner = (Scanner *)payload;

  array_delete(&scanner->indents);
  array_init(&scanner->indents);
  array_push(&scanner->indents, 0);

  for (unsigned i = 0; i < length; i++) {
    array_push(&scanner->indents, (unsigned char)buffer[i]);
  }
}

void *tree_sitter_haml_external_scanner_create() {
  Scanner *scanner = ts_calloc(1, sizeof(Scanner));
  array_init(&scanner->indents);
  array_push(&scanner->indents, 0);
  return scanner;
}
```

Продовження додатка Б

```
void tree_sitter_haml_external_scanner_destroy(void *payload) {
  Scanner *scanner = (Scanner *)payload;
  array_delete(&scanner->indents);
  ts_free(scanner);
}
```

БІБЛІОГРАФІЧНА ДОВІДКА

Тема магістерської роботи: Вдосконалення сервісу візуального маркування синтаксису HAML для редактора коду Neovim на основі Tree-sitter

Обсяг пояснювальної записки 65 аркушів.

3 таблиці;

15 рисунків;

2 додатки.

Дата завершення роботи: *10 грудня 2025р.*

Підпис студента-дипломника _____/ Б.Р. Шушваль /