МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ХЕРСОНСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ

Факультет комп’ютерних наук, фізики та математики

Кафедра комп'ютерних наук та програмної інженерії

СТАТИЧНИЙ АНАЛІЗ SQL-ЗАПИТІВ БАЗОВАНИЙ НА ГРАМАТИЦІ SQL

Кваліфікаційна робота (проєкт)

на здобуття вищої освіти «бакалавр»

Виконав: здобувач 4 курсу 12-431 групи
Спеціальність 122 Комп’ютерні науки
Освітньо-професійна програма:
Комп’ютерні науки
Засухін Ростислав Русланович

Керівник: к. м.-ф. н., доц. Кравцов Г. М.

Рецензент: вчитель інформатики Херсонського фізико-технічного ліцею, магістр комп’ютерних наук, Тарасюк А.О.

Івано-Франківськ – 2025

ЗМІСТ

ВСТУП *4*
РОЗДІЛ 1. ОСНОВНІ КОМПОНЕНТИ ТА ТЕОРІЯ *6*
1.1. Статичний аналіз **6**
1.2. AutoComplete **7**
1.3. Декларативне програмування. **8**
1.4. Процедурне програмування. **9**
1.5. Lexer, Parser їх робота та взаємодія. **10**
1.6. Поняття граматики. **12**
1.7. ANTLR3, ANTLR4. Їх схожість та відмінності. **13**
1.8. Лексеми. **15**
1.9. YACC **17**
1.10. FLEX **18**
1.11 Test-Driven Development **19**
1.12 Ієрархія Чомскі **19**
РОЗДІЛ 2. SQL-ГРАМАТИКИ *22*
2.1. Огляд граматик та роботи з ними **22**
2.2. Керівничі конструкції у граматиках **23**
РОЗДІЛ 3. ПІДГОТОВКА ГРАМАТИК *27*
3.1. Огляд правил форматування граматик. **27**
3.2. Зберігання граматик на диску **27**
3.3. Python Script **28**
РОЗДІЛ 4. ПАРСУВАННЯ ТА ЗАВАНТАЖЕННЯ ГРАМАТИКИ У ДЕРЕВО *30*
4.1. Огляд процесу парсування та завантаження **30**
4.2. Опис класу SqlGrammarManager **30**
4.3. Опис класу Grammar **31**
4.4. Опис класу Lexer **32**
4.5. Опис класу Rule **34**
4.6. Опис класів сімейства Node. **35**
4.6.1. Node_Token 35

4.6.2. Node_Opt 36
4.6.3. Node_Loop_Plus та Node_Loop_Star. 37
4.6.4 Node_Alt. 38
4.6.5 Node_Call 38
4.7. Робота класу SqlLoader 39
4.8 Тестування класів Node та Loader. 40
РОЗДІЛ 5. ІТЕРАТОР 44
5.1. Огляд класу RuleIterator 44
5.2. Огляд класу Vertex 44
5.3. Огляд руху ітератора у правилах з різною конструкцією 46
5.4. Тестування класу RuleIterator 48
РОЗДІЛ 6. АЛГОРИТМ AUTOCOMPETE 51
6.1. AutoComplete 51
6.2. AutoComplete для Database Manager-ів (DBM). 52
6.3. Тестування алгоритму AutoComplete 54
6.4. Демонстрація роботи алгоритму 55
ВИСНОВКИ 58
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 59

ВСТУП

Актуальність роботи полягає в тому, що у сучасній розробці програмного забезпечення та баз даних важливу роль відіграють системи автозаповнення (AutoComplete), які допомагають користувачам швидше писати запити, зменшують кількість синтаксичних помилок і підвищують зручність роботи. Проте більшість існуючих рішень ґрунтуються на простих шаблонах або фіксованих наборах ключових слів, що не враховують повну структуру мови запитів.

Використання граматик SQL для створення алгоритмів автозаповнення дозволяє значно підвищити якість підказок, оскільки аналіз відбувається на рівні синтаксису, а не окремих слів. Це дає можливість будувати підказки залежно від контексту запиту, типу СУБД та конкретної версії SQL-синтаксису.

Таким чином, тема роботи є актуальною, оскільки поєднує класичні підходи до побудови мов програмування (лексичний і синтаксичний аналіз) з практичними задачами підвищення зручності та ефективності роботи з базами даних.

Мета роботи – розробка алгоритму автодоповнення (AutoComplete), який базується на SQL-граматиках. Особливу увагу приділено тому, щоб алгоритм ефективно працював з основними граматиками SQL, використовуваними в популярних системах управління базами даних (СУБД), таких як PostgreSQL, MySQL, SQLite та інших.

Об’єктом дослідження виступає статичний аналіз SQL-запитів та його особливості.

А **предметом дослідження** виступає створення алгоритму AutoComplete на основі SQL граматик.

Практичне значення роботи полягає в інтеграції розробленого алгоритму автодоповнення в систему управління базами даних Valentina Studio. Це дозволить значно підвищити ефективність роботи користувачів цієї платформи, а також забезпечить можливість подальшого вдосконалення та розширення функціональності алгоритму.

Задачі дослідження:

- Дослідити компоненти для роботи зі статичним аналізом та існуючі граматики SQL;
- Спроектувати методи/класи для статичного аналізу;
- Реалізувати ці методи/класи;
- Створити алгоритм AutoComplete на основі попередніх задач.

РОЗДІЛ 1. ОСНОВНІ КОМПОНЕНТИ ТА ТЕОРІЯ

1.1. Статичний аналіз

Статичний аналіз - це процес аналізу коду програмного забезпечення без його фактичного виконання. Він використовується для виявлення помилок, недоліків у коді, потенційних проблем з безпекою та результативністю. Ось основні аспекти статичного аналізу:

- Виявлення помилок і недоліків: Статичний аналіз допомагає знайти помилки та недоліки, такі як невикористовувані змінні, витік пам'яті, неправильне управління ресурсами, порушення стандартів кодування та інші типові помилки.
- Безпека коду: Він може виявляти вразливі місця, такі як недоліки в управлінні доступом, ін'єкції SQL, переповнення буферу, які можуть призвести до атак хакерів.
- Оптимізація продуктивності: Через аналіз коду можна виявити місця, де можна покращити продуктивність, наприклад, оптимізувавши цикли або усунувши непотрібні операції.
- Дотримання стандартів: Статичний аналіз перевіряє, чи дотримується код конкретних стандартів кодування або кращих практик, що є важливим для підтримки якості коду.
- Інтеграція з розробкою: Цей процес часто інтегрується в процес розробки програмного забезпечення, особливо в рамках методології безперервної інтеграції та безперервного розгортання.
- Інструменти статичного аналізу: Існує багато інструментів для статичного аналізу, таких як SonarQube, Coverity, Checkmarx, а також специфічні для мови інструменти, такі як CppCheck для C++, або ESLint для JavaScript.
- Обмеження: Важливо зазначити, що статичний аналіз не може виявити всі помилки або проблеми безпеки, і він не замінює потребу в динамічному аналізі або тестуванні.

1.2. AutoComplete

Автозаповнення (AutoComplete) в сфері програмування - це важлива функціональність, яка спрощує і прискорює написання програмного коду. Вона дозволяє розробникам отримувати автоматичні рекомендації та підказки щодо можливих функцій, методів, змінних, ключових слів і інших елементів мови програмування, що використовуються в проекті. Це дуже важливо для підвищення продуктивності розробки та зменшення кількості помилок.

Використання автозаповнення в програмуванні допомагає збільшити ефективність розробки, зменшити кількість помилок і полегшити процес навчання новим мовам програмування та бібліотекам. Крім того, ця функція дозволяє розробникам швидше знайти і використовувати необхідні ресурси для своїх проектів.

Основні аспекти автозаповнення включають такі елементи:

- Редактори та інтегровані середовища розробки (IDEs): Більшість популярних IDEs, таких як Visual Studio, IntelliJ IDEA, PyCharm, а також текстові редактори, такі як Visual Studio Code, підтримують автозаповнення. Вони аналізують код, який ви пишете, і надають вам варіанти завершення, які відповідають контексту.
- Мови програмування: Автозаповнення базується на знаннях про синтаксис та структуру конкретної мови програмування. Таким чином, редактор або IDE має доступ до інформації про ключові слова, функції, методи, класи, змінні та інші складові мови.
- Контекст коду: Система аналізує контекст вашого коду, включаючи змінні, функції, класи, модулі і бібліотеки, які ви вже використовуєте. Вона намагається зрозуміти, які саме об'єкти та функції можуть бути доступні на даному етапі розробки і пропонує варіанти, що відповідають цьому контексту.

- Документація: Деякі редактори надають докладні описи та документацію для об'єктів, що запропоновуються автозаповненням, що допомагає розробникам краще розуміти, як вони можуть використовувати ці об'єкти в своєму коді.
- Підказки та валідація: Автозаповнення також може надавати підказки щодо можливих помилок у коді, наприклад, невідомих методів або неправильних типів даних.

1.3. Декларативне програмування.

Декларативне програмування - це парадигма програмування, яка фокусується на описі того, що потрібно виконати, а не на тому, як це робити. Цей підхід відрізняється від імперативного програмування, де основна увага приділяється опису послідовності кроків для досягнення результату.

В цілому, декларативне програмування є потужною парадигмою, яка дозволяє розробникам зосередитися на "що" потрібно зробити, замість "як" це зробити, тим самим сприяючи створенню чистішого, більш модульного і легшого для підтримки коду. Ось декілька ключових аспектів декларативного програмування:

- Опис проблеми, а не процесу: У декларативному програмуванні ви описуєте, що потрібно виконати, не вказуючи детальний алгоритм досягнення цього. Наприклад, SQL для запитів до бази даних описує, які дані потрібні, а не як їх отримати.
- Чистота і простота: Декларативний код зазвичай коротший і чистіший. Він зосереджений на цілях, а не на процедурах, що робить його часто більш зрозумілим і легшим для читання.
- Функціональне програмування: Ця парадигма тісно пов'язана з декларативним програмуванням. Функціональне програмування

(наприклад, Haskell, Scala) використовує функції для досягнення результатів і зазвичай є декларативним.

- Приклади мов: Мови декларативного програмування включають SQL для баз даних, HTML для веб-дизайну, XSLT для перетворення XML, а також специфічні доменні мови (DSL).
- Переваги: Декларативне програмування може спростити написання коду, зробити його більш зрозумілим та легшим для підтримки. Також воно часто сприяє більшій повторному використанню коду.
- Обмеження: Декларативний підхід може бути менш гнучким у деяких сценаріях, де потрібен детальний контроль над процесом. Іноді він може бути менш ефективним у термінах продуктивності для певних задач.
- Застосування: Декларативне програмування часто використовується у розробці веб-додатків, управлінні базами даних, конфігурації інфраструктури (наприклад, використання Terraform для опису інфраструктури в коді) та у багатьох інших областях.

1.4. Процедурне програмування.

Процедурне програмування - це підхід до програмування, який акцентує на використанні процедур або підпрограм (функцій). Це одна з найбільш ранніх парадигм програмування і становить основу багатьох мов вищого рівня. Ось основні аспекти процедурного програмування:

- Фокус на функціях і процедурах: Процедурне програмування базується на понятті виклику процедур, які є наборами послідовних кроків або інструкцій. Програма структурована як набір процедур або функцій, кожна з яких виконує конкретне завдання.
- Локалізація стану та поведінки: В процедурному програмуванні дані та процедури, які ними маніпулюють, часто відділені. Функції працюють з даними, які передаються як параметри.

- Модульність та повторне виконання коду: Однією з основних переваг є здатність до модульності і повторного використання коду. Функції можуть бути написані один раз і викликані з різних частин програми, зменшуючи повторення коду.
- Послідовний підхід: Програми, написані в процедурному стилі, зазвичай виконуються послідовно, зверху вниз у тексті програми.
- Приклади мов, які підтримують процедурне програмування: C, Pascal, Fortran та BASIC.
- Структуроване програмування: Процедурне програмування часто пов'язане зі структурованим програмуванням, що є підходом до розробки програмного забезпечення, що зосереджений на створенні чіткої, тестової структури програми.
- Переваги:
    - Чітка Структура: Програми легко відстежувати і розуміти завдяки чіткій структурі.
    - Легкість Утримання: Модульність і перевикористання коду роблять утримання програм більш простим.
- Обмеження:
    - Обмежена Абстракція: Не завжди ефективно для дуже великих або складних програм.
    - Проблеми з Управлінням Станом: Управління спільним станом у великих програмах може стати складним.

1.5. Lexer, Parser їх робота та взаємодія.

Lexer, також відомий як лексичний аналізатор, є першою фазою процесу компіляції. Його основна задача - перетворення вхідного коду (написаного на мові програмування) у послідовність лексем.

Лексема — це мінімальна одиниця, яка має лінгвістичне значення, наприклад, ключові слова, ідентифікатори, літерали, оператори тощо.

Процес Роботи:

- Читання Вхідного Коду: Lexer читає вхідний код і розбиває його на символи.
- Визначення Лексем: Кожен символ аналізується з метою ідентифікації лексем. Наприклад, послідовність символів, що відповідає шаблону ідентифікатора, розпізнається як ідентифікатор.
- Видалення Пробілів та Коментарів: Всі непотрібні символи, такі як пробіли та коментарі, видаляються.
- Генерація Токенів: Для кожної лексеми генерується токен, який містить інформацію про тип лексеми та її значення.

Parser або синтаксичний аналізатор слідує за лексером у процесі компіляції. Його роль полягає у перетворенні послідовності токенів, отриманих від лексера, у синтаксичне дерево (Abstract Syntax Tree - AST), яке представляє синтаксичну структуру коду. Процес його роботи:

- Читання Токенів: Parser отримує токени від лексера.
- Перевірка Синтаксису: Parser використовує правила граматики мови програмування для перевірки синтаксису. Якщо він зустрічає токен, який порушує ці правила, генерується помилка синтаксису.
- Побудова AST: Під час аналізу синтаксису parser створює вузли AST, які представляють різні конструкції мови (наприклад, вирази, оператори, оголошення тощо).

Взаємодія між Lexer та Parser.

- Послідовність Роботи: Lexer спочатку перетворює вхідний код на токени, які потім передаються Parser.

- Співпраця: Lexer і Parser тісно співпрацюють. Lexer забезпечує Parser необхідними токенами для синтаксичного аналізу.
- Оптимізація Обробки: Ця взаємодія дозволяє оптимізувати обробку коду, розділяючи лексичний та синтаксичний аналізи, що спрощує конструкцію компілятора.

1.6. Поняття граматики.

Граматика в контексті комп'ютерних наук, особливо в теорії мов програмування і обробці природних мов, відіграє ключову роль. Вона описує набір правил, які визначають структуру коректних виразів у мові.

Граматика у комп'ютерних науках є фундаментальним інструментом для визначення та аналізу мов, як програмування, так і природних. Вона дозволяє розробникам та дослідникам точно описувати структуру мови, що є критично важливим для створення надійних і ефективних компіляторів, інтерпретаторів та інших інструментів обробки мови.

Ось основні аспекти граматик:

1. **Основні Компоненти:**

- Термінали: Це базові символи, з яких будуються рядки у мові. В контексті мов програмування, це можуть бути літери, цифри, оператори.
- Нетермінали: Це символи, які використовуються для представлення наборів рядків, що складаються з терміналів та/або інших нетерміналів.
- Правила Виводу (або продукції): Визначають, як нетермінальні символи можуть бути перетворені в послідовності термінальних і/або інших нетермінальних символів.
- Стартовий символ: Визначає початкову точку, з якої починається виведення рядків у мові.

2. **Типи Граматик:**

- Контекстно-незалежні граматики: Використовуються у багатьох програмних мовах. Правила виводу залежать лише від нетермінального символу, який вони перетворюють.
- Контекстно-залежні граматики: Правила виводу можуть залежати від контексту символу.

3. Застосування:

- Розробка мов програмування: Для визначення синтаксису мов програмування.
- Компілятори і інтерпретатори: Для аналізу та обробки програм, написаних на певній мові.
- Обробка природних мов: У лінгвістиці для аналізу і розуміння структури природних мов.

4. Генерування та Розпізнавання Мови:

- Генерування: Використання граматики для створення коректних рядків у мові.
- Розпізнавання: Визначення, чи належить даний рядок до мови, описаної граматикою.

5. Інструменти:

- Генератори синтаксичних аналізаторів: Інструменти, як-от Yacc або ANTLR, які генерують синтаксичні аналізатори на основі граматик.

6. Формальні Визначення:

- Формальні граматики: Використовуються в теоретичній інформатиці та математичній лінгвістиці для визначення формальних мов.

1.7. ANTLR3, ANTLR4. Їх схожість та відмінності.

ANTLR (Another Tool for Language Recognition) - це потужний генератор парсерів, який використовується для читання, обробки,

виконання або перекладу структурованих текстів або бінарних файлів. Він широко використовується для створення мов, інструментів та фреймворків. Розробка ANTLR розпочалася у 1989 році, і відтоді він зазнав кількох змін. ANTLR3 та ANTLR4 - це дві значущі версії цього інструменту, кожна зі своїми особливостями та удосконаленнями.

ANTLR3:

1. Реалізація: ANTLR3 був випущений у 2007 році. Він ввів багато нових функцій у порівнянні з попередніми версіями.
2. Алгоритм LL(*): ANTLR3 використовував алгоритм LL(*), який був значним покращенням порівняно з традиційними LL(k) алгоритмами. Це дозволило ANTLR3 обробляти більш складні граматики.
3. Потребує ручної роботи: Для деяких складних граматик користувачам доводилося вручну втручатися для розв'язання неоднозначностей і оптимізації продуктивності.

ANTLR4:

1. Реалізація: ANTLR4, випущений у 2013 році, представляє собою вдосконалення ANTLR3 з багатьма нововведеннями та покращеннями.
2. Алгоритм ALL(*): ANTLR4 використовує алгоритм ALL(*), який є ще більш потужним і гнучким. Він забезпечує більш ефективний і точний розбір складних мовних конструкцій без необхідності ручного втручання.
3. Спрощений синтаксис: Синтаксис у ANTLR4 спрощений, що робить написання та розуміння граматик більш доступним.
4. Краща обробка помилок: Покращена система обробки помилок, що важливо для створення надійних парсерів.

Порівняння ANTLR3 та ANTLR4:

1. Алгоритми розбору: Основна відмінність між ANTLR3 та ANTLR4 полягає в алгоритмах розбору: LL() для ANTLR3 та ALL() для ANTLR4. ALL(*) більш ефективний і потужний.
2. Спрощення граматики: ANTLR4 полегшує написання складних граматик завдяки спрощеному синтаксису та покращеному алгоритму розбору.
3. Обробка помилок: ANTLR4 пропонує кращу обробку помилок порівняно з ANTLR3.
4. Ручне втручання: У ANTLR3 іноді потрібне ручне втручання для оптимізації граматик, тоді як ANTLR4 мінімізує цю потребу.

Переваги ANTLR4:

- Більш потужний алгоритм розбору: ALL(*) значно підвищує ефективність і точність парсера.
- Простота використання: Спрощений синтаксис та краща обробка помилок роблять ANTLR4 більш зручним для користувачів всіх рівнів.
- Менше потреби у ручній оптимізації: Зменшується потреба у ручному налаштуванні та оптимізації граматик, що економить час та зусилля.

1.8. Лексеми.

Лексема — це основна одиниця лексичного аналізу в області програмування та компіляції. Вона відображає мінімальну послідовність символів, яка має семантичне значення у мові програмування. Лексеми є будівельними блоками мови програмування та служать для створення токенів, які використовуються під час синтаксичного аналізу.

Типи Лексем:

- Ключові Слова: Зарезервовані слова мови програмування, такі як if, else, while, function, тощо.
- Ідентифікатори: Назви змінних, функцій, класів та інших користувацьких структур.

- Літерали: Константи, які можуть бути числовими (наприклад, 123), рядковими ("Hello"), логічними (true/false).
- Оператори: Символи, що представляють операції, наприклад, +, -, *, /.
- Розділові Знаки: Символи, які використовуються для розділення виразів, наприклад, коми (,), крапки з комою (;), дужки ((), {}, []).

Процес Розпізнавання Лексем:

Лексичний аналізатор (lexer) в компіляторі читає вхідний код і розбиває його на лексеми. Кожна лексема визначається на основі заздалегідь встановлених правил та шаблонів, що відповідають структурі мови програмування. Після розпізнавання, кожна лексема перетворюється в токен, що містить інформацію про її тип та значення.

Важливість Лексем у Компіляції:

Лексеми відіграють критичну роль у процесі компіляції, оскільки вони дозволяють компілятору розбити вхідний код на мінімальні одиниці для подальшого аналізу та обробки. Вони забезпечують механізм для розуміння структури та семантики програми, дозволяючи компілятору перевірити синтаксис і генерувати відповідний машинний код або проміжне представлення.

Застосування Лексем:

- Синтаксичний Аналіз: У синтаксичному аналізі, лексеми використовуються для побудови синтаксичних структур, таких як дерева розбору.
- Оптимізація Коду: Вони допомагають у виявленні та оптимізації зайвих або невикористовуваних фрагментів коду.
- Підсвічування Синтаксису: В середовищах розробки вони використовуються для підсвітки синтаксису, допомагаючи програмістам у навігації та розумінні коду.

1.9. YACC

YACC (Yet Another Compiler Compiler) — це інструмент для автоматичного створення синтаксичних аналізаторів (парсерів) для мови програмування, який дозволяє розробникам описувати граматику мови та генерувати відповідний код для парсера. YACC широко використовується в компіляторах та інших інструментах обробки тексту.

- Основні аспекти: Граматика: YACC приймає опис граматики у формі контекстно-незалежних правил. Користувач визначає правила, які описують, як різні конструкції мови можуть бути сформовані.

- Генерація коду: YACC генерує C-код, який реалізує синтаксичний аналізатор на основі наданої граматики. Синтаксичний аналізатор перетворює потік токенів, отриманих від лексичного аналізатора, у структуровану форму.

- Обробка помилок: YACC надає механізми для обробки помилок синтаксису. Розробники можуть вказати, як парсер має реагувати на помилки та які повідомлення слід виводити.

- Інтеграція з лексичними аналізаторами: YACC зазвичай використовується в парі з лексичним аналізатором, таким як Lex, що дозволяє відокремити лексичний та синтаксичний аналіз.

- Складність: YACC може обробляти складні граматики, але користувачам може бути потрібно враховувати деякі обмеження та конфлікти, які можуть виникнути під час визначення граматики.

- Застосування YACC: YACC часто використовується в проектах, пов'язаних із компіляторами, мовами програмування, трансляторами, а також в інструментах для обробки природних мов.

1.10. FLEX

FLEX (Fast Lexical Analyzer) — це інструмент для створення лексичних аналізаторів (lexer) на основі опису регулярних виразів. FLEX є вдосконаленою версією оригінального лексичного аналізатора Lex і має багато переваг, які роблять його популярним серед розробників.

- Основні аспекти: регулярні вирази: FLEX дозволяє користувачам описувати лексеми за допомогою регулярних виразів, що забезпечує гнучкість та простоту в описанні шаблонів лексем.
- Генерація коду: FLEX генерує C-код, який реалізує лексичний аналізатор на основі регулярних виразів, заданих у вихідному файлі.
- Швидкість:

FLEX оптимізований для швидкого виконання, що робить його ефективним інструментом для обробки великих обсягів тексту.

- FLEX дозволяє одночасно визначати кілька типів токенів, що забезпечує зручність при аналізі коду.
- FLEX часто використовується в парі з YACC, де FLEX виконує лексичний аналіз, а YACC — синтаксичний.
- FLEX використовується в компіляторах, трансляторах, а також в будь-яких програмах, які потребують аналізу тексту або коду. Це можуть бути програми для обробки мов, генерації кодів, а також інструменти для аналізу даних.

1.11 Test-Driven Development

Test-Driven Development (TDD) – це методологія розробки програмного забезпечення, заснована на циклічному написанні тестів перед реалізацією функціоналу.

Основні етапи TDD:

1. **Написання тесту** – створюється тест, який перевіряє ще не реалізовану функціональність.
2. **Запуск тесту** – тест має не пройти (fail), оскільки функціонал ще відсутній.
3. **Реалізація коду** – пишеться мінімальний код, необхідний для проходження тесту.
4. **Запуск тесту** – тест повинен пройти (pass).
5. **Рефакторинг** – покращується структура коду без зміни його поведінки.

Переваги TDD:

- Зменшує кількість помилок у коді.
- Полегшує рефакторинг та подальший розвиток системи.
- Покращує архітектуру коду.
- Підвищує впевненість у коректності роботи програми.

Цей підхід широко застосовується в Agile-розробці та сприяє створенню більш якісного програмного забезпечення.

1.12 Ієрархія Чомскі

Ієрархія Чомскі — це класифікація формальних граматик за їхньою потужністю, тобто за типами мов, які вони можуть описати. Складається з 4 рівнів (від найзагальніших до найобмеженіших):

1. **Тип 0 — Рекурсивно перелічувані мови**
    - Найзагальніші.
    - Граматика: без обмежень.
    - Розпізнаються: машиною Тюрінга.
    - Можуть бути нерозв'язні (наприклад, проблема зупинки).
2. **Тип 1 — Контекстно-залежні мови**
    - Граматика: правила залежать від контексту.
    - Розпізнаються: лінійно обмеженою машиною Тюрінга.
    - Приклад: мова $\{a^n b^n c^n \mid n \geq 1\}$.
3. **Тип 2 — Контекстно-вільні мови**
    - Граматика: лівий бік правил — один нетермінал.
    - Розпізнаються: автоматом із стеком.
    - Приклад: мова дужок, арифметичні вирази.
4. **Тип 3 — Регулярні мови**
    - Граматика: дуже проста (правосторонні або лівосторонні правила).
    - Розпізнаються: скінченним автоматом.
    - Приклад: регулярні вирази (наприклад, email-адреси).

РОЗДІЛ 2. SQL-ГРАМАТИКИ

2.1. Огляд граматик та роботи з ними.

Ціллю роботи є створення власного статичного аналізатору, який буде базуватися на SQL-граматиках.

Перше питання яке потребувало вирішення -- граматики на якій мові обрати для подальшої роботи? З варіантів були мови граматик ANTLR4 та YACC. У результаті розмірковувань, за основу були обрані граматики другого типу (контекстно-вільні) на ANTLR4, оскільки ця мова більш схожа на людську, а форматування просте для розуміння.

Далі було знайдено граматики потрібних нам баз даних на платформі GitHub, та додано до нашого репозиторію. Граматика, як структура, складається з правил, які у свою чергу складаються з токенів.

Розглянемо приклад правил з граматики Valentina DB:

```
specific_routine_designator
    :   SPECIFIC routine_type specific_name
    |   routine_type member_name ( FOR schema_qualified_user_defined_type_name )?
    ;

routine_type
    :   ROUTINE
    |   FUNCTION
    |   PROCEDURE
    |   ( INSTANCE | STATIC | CONSTRUCTOR )? METHOD
    ;

member_name
    :   member_name_alternatives ( data_type_list )?
    ;

member_name_alternatives
    :   schema_qualified_routine_name
    |   method_name
    ;
```

Рис. 2.1 – Фрагмент граматики Valentina DB.

Як можемо побачити, структура будь-якого правила складається з: імені, двокрапки, тіла правила, крапка-коми.

Токени – лексична одиниця у правилі. Вони бувають декількох видів: синтаксичний, лексичний та ключові слова.

Якщо слово складається тільки з великих букв, воно являється ключовим словом (KEYWORD). Наприклад: SPECIFIC, FOR, STATIC, METHOD, DROP, USER і т.д.

До лексичних та синтаксичних токенів відносяться символи: “ ; ”, “ : ”, “ ’ ”, “ . ”, “ , ” тощо.

2.2. Керівничі конструкції у граматиках

Перейдемо до детального розгляду тіла правила, а саме до керівничих-конструкцій які присутні у ньому:

1) Оператор “|” відповідає за альтернативи, тобто при випадку ТОКЕН1 | ТОКЕН2 буде матися на увазі що користувач обов’язково введе або перший або другий токен.

```
test_rule
    : DROP
    | CREATE
    ;
```

Рис. 2.2 – Застосування альтернативи.

2) Оператор “?” відповідає за опціональність відповідного токену, тобто цей токен може бути відсутнім у запиті користувача.

```
test_rule
    : DROP LEX_IDENT IF?
    ;
```

Рис. 2.3 – Застосування опціональності.

3) Оператор “+” відповідає за те що відповідний токен може вводитися циклічно, а також використання цього оператора у граматиці запевняє нас що потрібно буде ввести хоча б один токен, тобто ітерація циклу повинна буде виконатися хоча б один раз (можна зіставити з циклом do while() в C++)

```
test_rule
    : DROP LEX_IDENT+
    ;
```

Рис. 2.4 – Застосування циклу «плюс».

4) Оператор “*” досить схожий з попереднім оператором (“+”), але його відмінність полягає в тому, що “*” не гарантує що відповідний токен буде існувати. Тобто токени можуть або вводитися декілька разів, або бути відсутніми зовсім (можна зіставити з циклом while() в C++)

```
test_rule
    : DROP LEX_IDENT*
    ;
```

Рис. 2.5 – Застосування циклу «зірочка».

5) Оператор “()”. Це конструкція групування, тобто вона відповідає за створення групи. Завдяки цьому керівничі-конструкції можна буде використовувати не тільки до одного токену, а одразу до групи.

```
test_rule
    : DROP (LEX_IDENT, LEX_IDENT)
    ;
```

Рис. 2.6 – Застосування оператору групування.

6) Цей оператор являється допоміжним, тобто його можна використовувати у парі з усіма раніше перерахованими операторами.

Наприклад його можна використовувати з альтернативою, запит буде виглядати таким чином:

```
test_rule
    :   CREATE
    |   (DROP IF EXISTS)
    ;
```

Рис. 2.7 – Використання оператору групування з альтернативами

7) Оператор групування можна використовувати з оператором опціональності, тим самим позначаючи що група токенів являється опціональними, тобто або буде присутня, або ні.

```
test_rule
    : DROP LEX_IDENT (IF EXISTS)?
    ;
```

Рис. 2.8 – Використання оператору групування з опціональністю.

8) Також групування можна використовувати з циклом +. В такому разі ми позначимо що група токенів буде повторюватися 1 або декілька разів.

```
test_rule
    : DROP (LEX_IDENT, LEX_IDENT)+
    ;
```

Рис. 2.9 – Використання оператору групування з циклом “+”.

9) Використовуючи оператор групування з оператором зірочка(“*”), ми позначаємо що група токенів може бути введена циклічно, або бути відсутньою взагалі.

```
test_rule
    : DROP (LEX_IDENT, LEX_IDENT)*
    ;
```

Рис. 2.10 – Використання оператору групування з циклом “*”.

РОЗДІЛ 3. ПІДГОТОВКА ГРАМАТИК

3.1. Огляд правил форматування граматик.

Одна з проблем використання великої кількості різних граматик полягає в тому, що кожна з них зроблена окремими людьми, через це немає певного спільного стилю у їх оформленні.

Ми, як люди, читаючи граматику будемо розуміти та бачити що саме робить те чи інше правило незалежно від стилю. Однак ми створюємо алгоритм для машини, яка робить тільки те, що запрограмовано, через це було вирішено переробити інші SQL граматики під єдиний стиль.

Підготовка граматик потрібна для можливості їх обробки з використанням універсального парсеру, який зможе обробити будь яку з них незалежно від стилю.

Усі взяті граматики, були перероблені за такими правилами:

1) усі лексичні токени повинні бути у верхньому регістрі;
2) усі ключові слова повинні бути у верхньому регістрі;
3) Ім’я правила повинно бути в нижньому регістрі, після нього повинен стояти символ “:”, після цього повинні стояти декілька токенів, правило закінчується символом “;”;
4) Правило найвищого рівню повинно починатися з префіксу “sql_”;
5) Правила вищого рівню, які також є SQL-командами, мають мати суфікс “_stmt” (приклад: alter_table_stmt, drop_trigger_stmt);

3.2. Зберігання граматик на диску

Граматики являються невід’ємною частиною VStudio. Як було сказано у попередньому пункті, усі файли граматики були взяті з платформи GitHub. Для їх збереження створено окрему директорію. Граматикам були дані імена відповідно баз даних та версій до яких вони відносяться, і після цього згруповано по папкам відповідних БД.



Рис. 3.1 – Збереження граматик

3.3. Python Script

Першим етапом роботи стала обробка імен правил у граматиках. У когось вони в форматі Camel у когось у форматі snake. Оскільки у граматиці дуже багато ключових слів, які складаються тільки з великих літер, для контрасту було обрано записувати імена правил у форматі snake. Через ці висновки було сформовано пункт №3 у правилах вище.

Наступним етапом, було замінити усі імена правил у граматиках на snake-формат. Для вирішення цього питання було створено скрипт-парсер, алгоритм роботи якого виглядає таким чином:

1. Отримати шлях до файлу;
2. Пройти цей файл та запам’ятати усі імена правил;
3. Знайти усі використання кожного з цих правил;
4. Замінити знайдені місця на snake-формат імені правила.

Після цього граматики, які були створені різними авторами набувають єдиного стилю, через що легше створити під них парсер.

Однак при такому “рефакторингу” граматик виникає таке питання: “Чи буде тривіальним підтримка єдиного стилю в файлах граматик, адже при виході нової версії потрібно буде багато чого переробляти?”. Проаналізувавши останні версії було виявлено що зазвичай змінам піддається 1-5 правил, через що підтримка буде досить простою.

РОЗДІЛ 4. ПАРСУВАННЯ ТА ЗАВАНТАЖЕННЯ ГРАМАТИКИ У ДЕРЕВО

4.1. Огляд процесу парсування та завантаження

У цьому розділі розглянемо як ми завантажуємо граматику з дискового файлу у пам’ять та структури, які використовуються для цього.

В цілому для виконання цього завдання у нас створено такі класи:

- Lexer. Його задача обробити граматику та розбити її на послідовність токенів;
- SqlGrammarManager. Його задача зберігати вже завантажені граматики та повертати їх по запиту.
- Grammar. Клас задача якого полягає у зберіганні правил граматики SQL завантажених у вигляді екземплярів класів Rule;
- Loader. Його задача полягає у взаємодії з усіма вищеназваними класами для завантаження обраної граматики з дискового файлу в новий екземпляр класу Grammar.

4.2. Опис класу SqlGrammarManager

SqlGrammarManager – це Singleton-клас, задача якого зберігати вже завантажені граматики та повертати їх по запиту. Функціонал цього класу невеликий, однак важливий для структури алгоритму загрузки граматик. Головним методом класу являється get_Grammar().

Він являє собою функцію для того що б отримати граматику за заданими назвою та версією. Сама функція отримавши вхідні параметри повертає нам вказівник на екземпляр класу SqlGrammar, в який була завантажена граматика даної бази даної версії. Після цього цей екземпляр використовуємо як «незмінний» об'єкт, що означає, що користувач не має можливості змінювати його стан, а може лише використовувати його для виконання операцій.

4.3. Опис класу Grammar

Цей клас створений для зберігання правил граматики SQL та буде використовуватися для розбору SQL-команд. Його основні функції включають:

- зберігання набору правил SQL-граматики.
- зберігання ключових слів SQL-граматик;
- зберігання правил типу «_stmt»;
- зберігання назви типів об’єктів схеми бази даних.

Для кожної бази даних та для кожної її версії існує окремий дисковий файл граматики. Під час першого підключення до конкретної комбінації БД та її версії для неї створюється новий екземпляр класу Grammar, якщо його ще не існувало в SqlGrammarManager.

Основні компоненти класу:

- Методи для ключових слів;
- Методи для роботи с правилами;
- Масиви: ключових слів, правил, правил-запитів, імен SQL-об’єктів

Стосовно роботи з ключовими словами. Для неї було створено просту однойменну структуру Keyword, яка складається з двох полів: імені та коду ключового слова. Саме з цими параметрами працюють методи для роботи з ключовими словами: addKeyword(), get_Keyword(name), get_Keyword(code), get_KeywordsCount().

Робота з правилами відбувається наступним чином. У коді існує клас Rule, взаємодія з ним відбувається завдяки методам: add(), addSubRule(), get_RuleCount(), get_Rule().

Метод add() перевіряє чи існує правило у асоціативному масиві правил, і якщо ні, додає його.

4.4. Опис класу Lexer

SqlLexer – це клас, який представляє собою клас-обробник, задача котрого обробляти вхідний потік символів та розпізнавати токени. Також у проекті існує його дочірній клас SqlLexerWithAllocator. Він був створений для розділення функціональності, а точніше для більш гнучкого виділення токенів (лексем).

Розглянемо клас SqlLexer. Він має у собі масив токенів (лексем), який заповнюється під час виклику методу GetAllTokens().

Ця функція послідовно викликає другий метод GetNextToken(), вона аналізує вхідний потік символів, оброблюючи кожен з них, виділяє пропуски (символ SPACE), якщо вхідний символ НЕ пропуск починається вибірка через оператор switch, якщо символ – пропуск, до масиву токенів додається відповідний токен.

```
    switch( charClass )
    {

// STRICTLY 1-char tokens:

        case '(':   { inTokenInfo.mType = VSQLTokenTypes::LPAREN;   tokKind = TokenKind::kSimple; } break;
        case ')':   { inTokenInfo.mType = VSQLTokenTypes::RPAREN;   tokKind = TokenKind::kSimple; } break;

        case '{':   { inTokenInfo.mType = VSQLTokenTypes::LCURLY;   tokKind = TokenKind::kSimple; } break;
        case '}':   { inTokenInfo.mType = VSQLTokenTypes::RCURLY;   tokKind = TokenKind::kSimple; } break;

        case '[':   { inTokenInfo.mType = VSQLTokenTypes::LBRACK;   tokKind = TokenKind::kSimple; } break;
        case ']':   { inTokenInfo.mType = VSQLTokenTypes::RBRACK;   tokKind = TokenKind::kSimple; } break;

        case ',':   { inTokenInfo.mType = VSQLTokenTypes::COMMA;    tokKind = TokenKind::kSimple; } break;
        case ';':   { inTokenInfo.mType = VSQLTokenTypes::SEMI;     tokKind = TokenKind::kSimple; } break;

        case '?':   { inTokenInfo.mType = VSQLTokenTypes::QUESTION; tokKind = TokenKind::kSimple; } break;

        case '%':   { inTokenInfo.mType = VSQLTokenTypes::PERCENT;  tokKind = TokenKind::kSimple; } break;
        case '+':   { inTokenInfo.mType = VSQLTokenTypes::PLUS;     tokKind = TokenKind::kSimple; } break;
        case '/':   { inTokenInfo.mType = VSQLTokenTypes::DIV;      tokKind = TokenKind::kSimple; } break;
```

Рис. 4.1 – Фрагмент коду з визначенням токену

На даному фрагменті коду представлена ідентифікація того, чи являється поточний токен 1-символьним. Після ідентифікації токену метод повертає оброблений токен у функцію GetAllTokens().

Після закінчення роботи методу GetAllTokens() у масиві будуть знаходитися тільки токени, без пропусків, ентерів, табуляцій тощо. В цілому цього достатньо для обробки та роботи з граматикою, однак можуть бути

задачі, по типу згенерувати файл граматики по запиту, для котрих потрібно пам’ятати форматування у файлі. Для подобних цілей створено клас SqlLexerWithAllocator.

Цей клас породжений від SqlLexer та має розширену функціональність та логіку віртуального методу GetAllTokens(), та більшу кількість потоків токенів. Сумарно дочірній клас має такі потоки: ключові слова, ідентифікатори, коментарі, пропуски, строкові літерали, токени (без пропусків).

```
// TOKENS LISTS:

    LSqlTokens                  mAllTokens;

// FILTER-1: spaces vs others:

    LSqlTokens                  mSpaceTokens;
    LSqlTokens                  mCommentTokens;

    LSqlTokens                  mNewLines;

    LSqlTokens                  mTokens;


// FILTER-2: not keyword tokens:

    LSqlTokens                  mIdents;

    LSqlTokens                  mStringLiterals;


// FILTER-3: keywords.

    LSqlTokens                  mKeywordTokens;

    LSqlTokens                  mBeginEnd;
```

Рис. 4.2 – Потоки токенів.

Підбиваючи підсумки можемо сказати, що клас SqlLexerWithAllocator обробляє вхідний потік символів з файлу граматики, виділяючи в ньому токени. Отримання токенів відбувається по запиту. Як результат, після роботи методу GetAllTokens() ми отримаємо відсортовані та впорядковані масиви токенів, с можливістю при потребі відновити/створити граматику по цим даним тощо.

4.5. Опис класу Rule

Задача цього класу полягає у зберіганні та оперуванні певним правилом граматики.

Він має поля для зберігання імені, текстового представлення та списку вузлів (Nodes), які входять до його складу. Також у цьому класі присутні методи для обробки та взаємодії з полями, а саме:

- get_Name(), set_Name() – для взаємодії з іменем
- get_Text(), set_Text() – для взаємодії з текстом
- get_OwnerNode(), set_OwnerNode() – для виставлення вказівника на вузол-власник

```
    // ---------------------
    // Properties:

        std::string         get_Name() const { return mName; }
        void                set_Name( std::string inName ) { mName = inName; }

        bool                get_StopHere() const { return mStopHere; }
        void                set_StopHere( bool inValue ) { mStopHere = inValue; }

        std::string         get_Text() const { return mText; }
        void                set_Text( const std::string inText ) { mText = inText; }

virtual bool                isSubRule() const { return false; }

virtual Node_Ptr            get_OwnerNode() const { return nullptr; }
virtual void                set_OwnerNode( Node_Ptr inParentNodePtr ) { std::ignore = inParentNodePtr; }
```

Рис. 4.3 – Get- та Set-методи класу Rule

Ім’я правила слугує його ідентифікатором, саме по ньому виконується пошук та виклик правила в граматиці.

Текстом правила являється все що йде у файлі граматики після імені правила, а саме: двокрапка, тіло правила, крапка з комою.

Вузол-власник – це поле класу Rule, яке зберігає в собі вказівник на вузол, який володіє даним правилом. Зроблено це для можливості реалізації ієрархії правил. Однак у звичайному правилі немає «власника», через це методи get_OwnerNode() та set_OwnerNode() не мають прописаної логіки, а просто повертають null значення, та ігнорують вхідний вказівник

відповідно. Це поле та відповідні віртуальні методи створені для класу SubRule.

Клас SubRule являється похідним від класу Rule. Відмінність цього класу від батьківського полягає у тому, що він не має власного імені, і він завжди прив'язаний до «власника». Цей клас був створений для роботи з підправилами вузлів типу Node_Opt, Node_Loop та Node_Alt.

4.6. Опис класів сімейства Node.

Наступне питання яке треба було розглянути: якими саме логічними елементами можна представити усю логіку правил в граматиці? Проаналізувавши декілька граматик було виявлено що різних синтаксичних конструкцій не так вже й багато. Класи які будуть їх реалізовувати будемо називати вузлами (Node).

Для їх реалізації сімейства цих класів було застосовано механізм спадкування, що забезпечує можливість створення узагальненого контейнера, здатного зберігати екземпляри будь-яких похідних класи від Node.

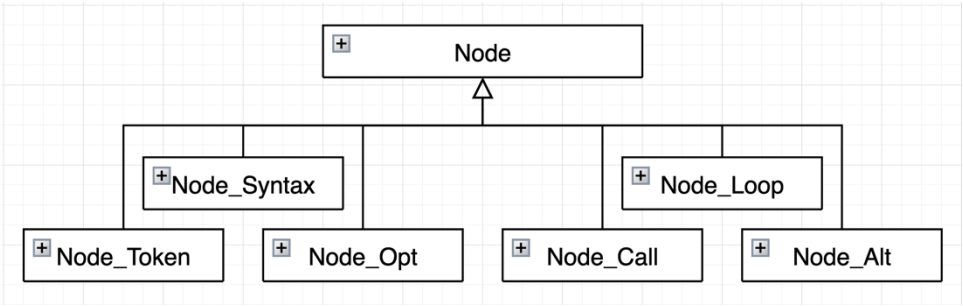


Рис 4.4 – Похідні класи від класу Вузол (Node)

4.6.1. Node_Token

Цей клас створений для зберігання токенів. Один екземпляр класу може зберігати тільки один токен.

Токени можуть бути трьох видів:

- ключові слова (DROP, SELCT, WHERE тощо);

- ідентифікатори (імена таблиць, полів тощо);
- синтаксичні токени (кома, крапка, двокрапка, скобки тощо).

На зображенні показано, як правило drop_table представляється у вигляді об’єкта класу Rule, який має під собою три об’єкти Node_Token, які зберігають ключові слова DROP, TABLE, LEX_IDENT відповідно.

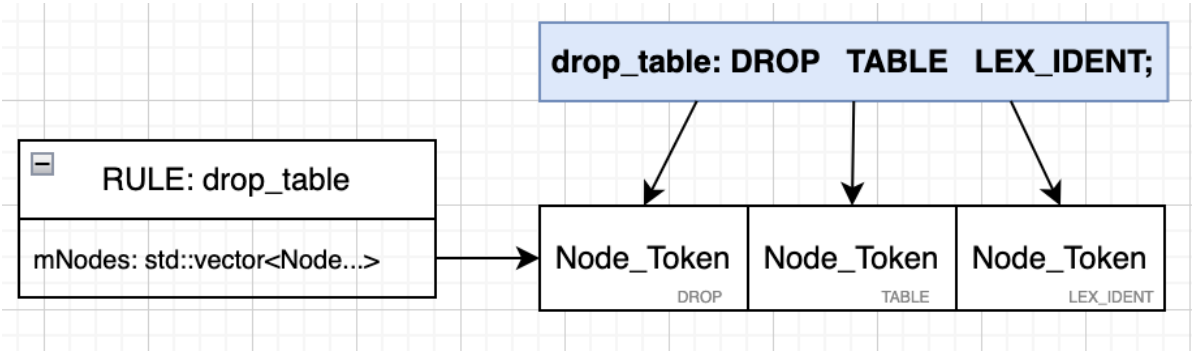


Рис. 4.5 – Лінійне правило

4.6.2. Node_Opt

Цей клас являє собою відображення розгалуження у правилі. Реалізація відбувається шляхом створення всередині об’єкту класу Node_Opt екземпляру класу SubRule, яке зберігає у собі опціональну частину правила-власника.

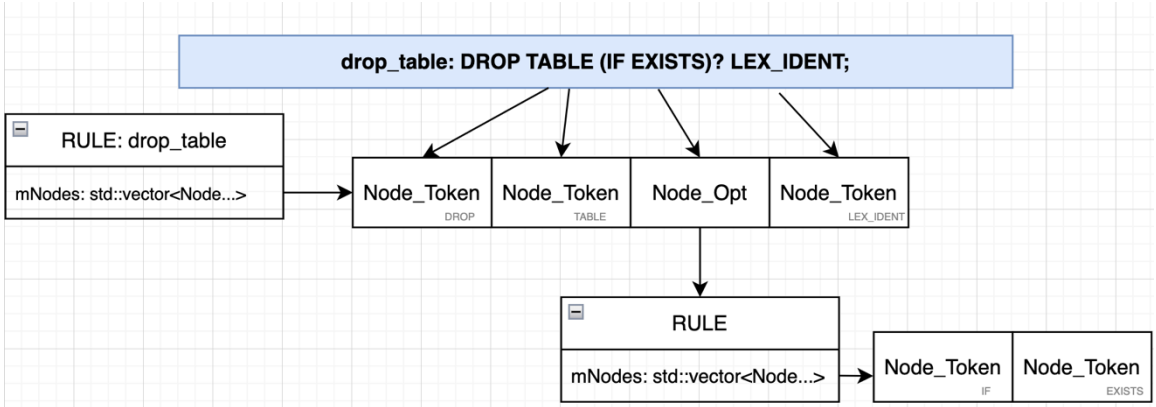


Рис 4.6 – Правило з опціональним вузлом

4.6.3. Node_Loop_Plus та Node_Loop_Star.

Задача цих двох класів полягає у циклічному повторенні SubRule, яке, так само як у класі Node_Opt, створюється всередині об’єкту класу. Використовуються вони в правилах, які передбачають введення даних, заповнення таблиці тощо.

Ці два класи можна порівняти з циклами do while та while. Перший з них (Node_Loop_Plus) являє собою цикл, який стовідсотково виконується хоча б один раз, тобто потребує від користувача введення даних хоча б раз, після чого можливе повторне введення. Другий (Node_Loop_Star) схожий з циклом while, оскільки, якщо введених даних немає, то правило відпрацює і без них, тож його можна назвати “опціональним циклом”.

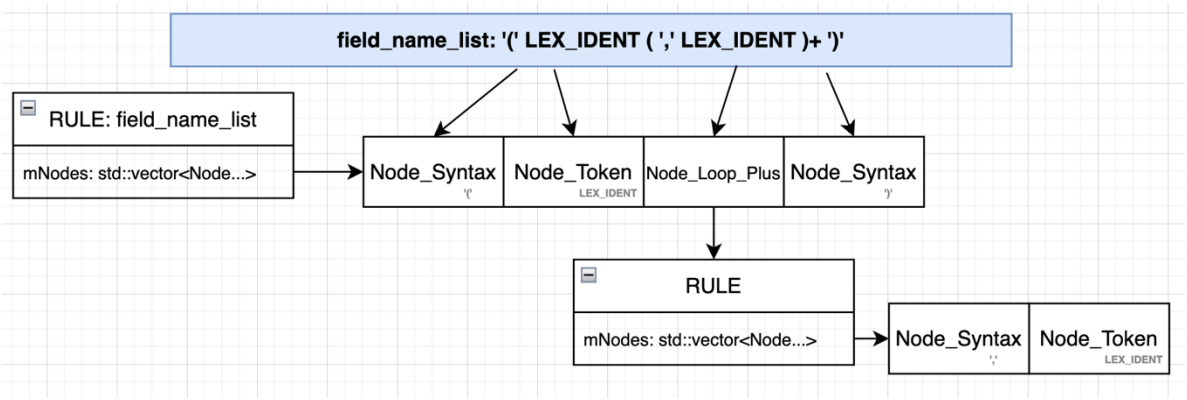


Рис 4.7 – Правило з використанням Node_Loop_Plus.

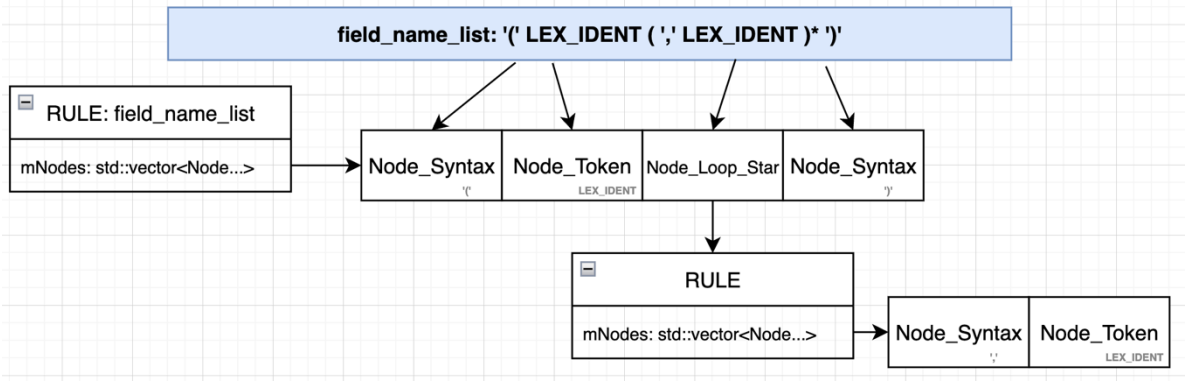


Рис 4.8 – Правило з використанням Node_Loop_Star.

4.6.4 Node_Alt.

Цей клас призначений для реалізації вибору серед декількох альтернатив у правилі. Реалізація відбувається шляхом створення всередині об’єкту Node_Alt вектору SubRules, з якого у процесі обереться один з альтернативних варіантів.

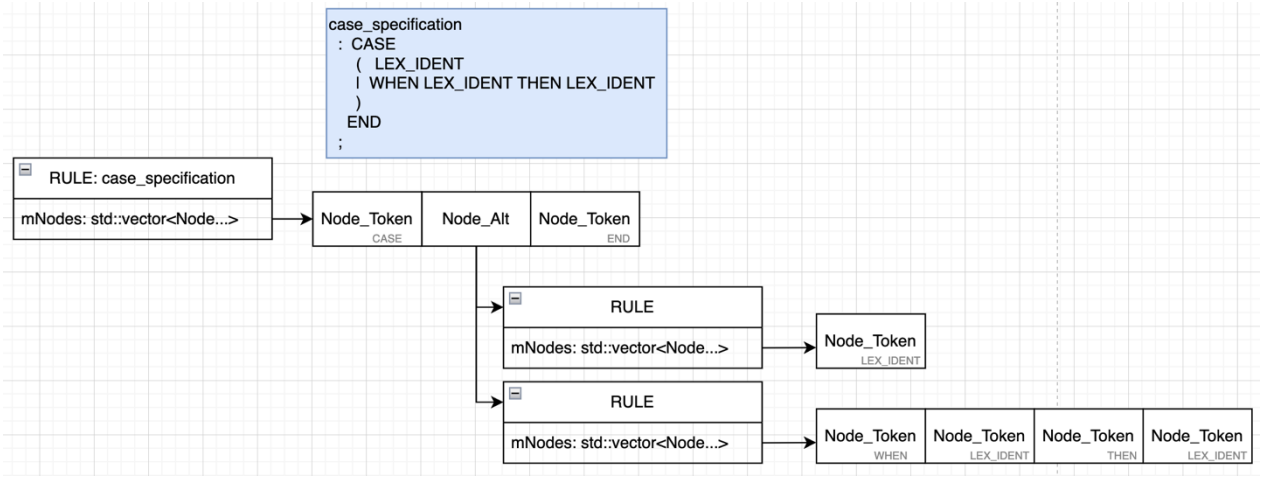


Рис 4.9 – Правило з альтернативами

4.6.5 Node_Call

Клас Node_Call містить у собі посилання на об’єкт правила, яке необхідно викликати. Це дозволяє створювати рекурсивні та взаємозалежні правила.

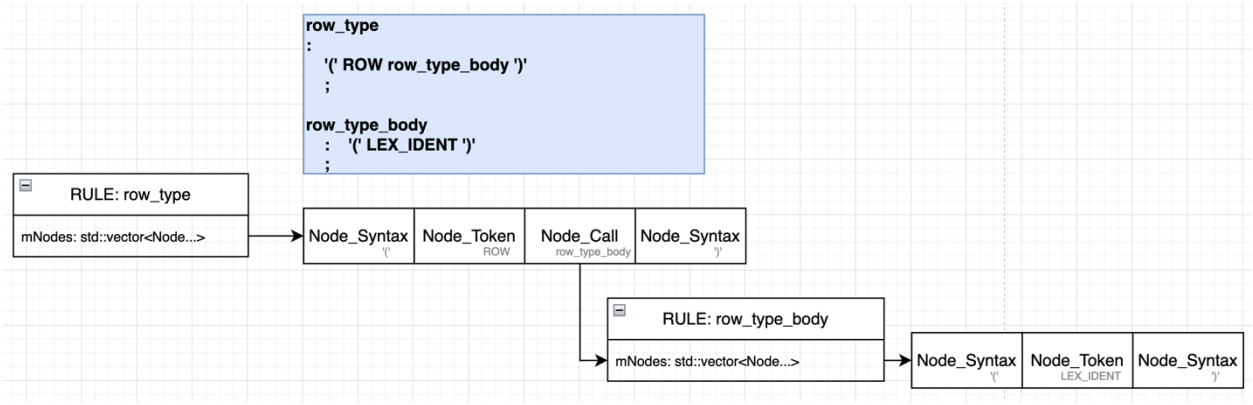


Рис 4.10 – Правило з використанням Node_Call

4.7. Робота класу SqlLoader

Клас SqlLoader відповідає за створення у оперативній пам’яті дерева-граматики.

Алгоритм його роботи починається з отримання вхідного потоку символів (з файлу граматики), після цього створюється об’єкт класу SqlLexer (розглянуто у пункті 3.4), якому ми передаємо даний потік символів. Далі створюється порожній екземпляр класу граматики після чого для нього викликається функція loadAllRules().

```
void                    loadAllRules();

void                    loadRule(
                            TOK_INDEX   l,
                            TOK_INDEX   r );

void                    loadRuleMainBody(
                            Rule_Ptr    r1,
                            TOK_INDEX   l,
                            TOK_INDEX   r );

void                    loadRuleBody(
                            Rule_Ptr    r1,
                            TOK_INDEX   l,
                            TOK_INDEX   r );
```

Рис. 4.11 – “Робочі” функції класу SqlLoader.

Задача цієї функції полягає у послідовних запитах до лексера для отримання правил. Для цього використовується циклічний виклик функції loadRule(). Вона перебирає токени з файлу граматики та визначає їх тип. Робота закінчується при знаходженні “кінцевого” токену правила “;”. При закінченні роботи сформоване під час ітерацій правило повертається як результат та додається до граматики. Завдяки цьому в кінці роботи функції loadAllRules() будемо мати створену в пам’яті граматику у вигляді дерева.

Також у класі існують ще дві важливі функції, це loadRuleMainBody() та loadRuleBody(). Задача loadRuleMainBody() завантажити «головне» тіло правила. Першим ділом вона перевіряє що у правилі немає альтернатив, якщо вони є, створюється екземпляр класу Node_Alt та додається до правила. Після перевірки запускається функція loadRuleBody().

Функція loadRuleBody() створена для обробки тіла правила. Принцип її роботи полягає у створенні структури правила граматики з потоку токенів наданого лексером. Ця функція циклічно перебирає кожен токен, виділяючи у ньому альтернативи, опціональність, цикли та викликає допоміжні функції, які потрібні для їх обробки.

4.8 Тестування класів Node та Loader.

Для початку розглянемо процес тестування загрузки граматик у дерево та перевірку коректності побудованого дерева правил та вузлів. Процес тестування виглядає таким чином:

1. створюємо строку у форматі файла граматики;
2. заповнюємо його одним або декількома правилами, які ми хочемо протестувати, тим самим створюючи «міні-граматику»;
3. після цього відпрацьовує Loader, який, як було сказано раніше, створює у пам’яті комп’ютера дерево по цій граматиці;
4. тестуємо через «геттер»-методи створене дерево на коректність, що включає у себе тестування правила та вузлів з якого воно складається;
5. тестуємо кожний екземпляр класу Rule перевіркою його розміру та перевіркою вузлів;
6. тестуємо кожний вузол отримуючи та звіряючи їх типи та дані з тими що прописані в тесті.

```
22 VLEXER_TEST_BEGIN
23 {
24     std::string data =
25         "emptyRule : ;";
26
27     loadSqlGrammar( ESqlDialect::kTest, data.c_str(), data.size() );
28
29     SqlGrammarManager* pManager = SqlGrammarManager::instance();
30
31     // Grammar tests:
32     SqlGrammar* pGrammar = (pManager->get_Grammar( ESqlDialect::kTest )).get();
33     TEST_CHECK( pGrammar != nullptr );
34     if( !pGrammar )
35         return;
36
37
38     // WORK with Grammar:
39     TEST_EQ( pGrammar->get_RuleCount(), 1 );
40
41
42     // Rule test:
43     Rule* r1 = (pGrammar->get_Rule("emptyRule")).get();
44     TEST_EQ(r1->size(), 0);
45 }
46 VLEXER_TEST_VALENTINA( "[StaticAnalyse][SqlGrammars][Load][EmptyRule]", "" )
```

Рис. 4.12 – Код для тестування пустого правила.

Давайте розберемо цей код. У рядках 24 та 25 виконується пункт 1 та 2 процесу тестування. Рядок 27 відповідає за загрузка нашої міні-граматики до класу Loader. Далі йде робота з менеджером (рядки 29-33) і власне перевірка створеного дерева. Рядок 33 – перевірка того що дерево існує, рядок 39 – перевірка що у граматиці знаходиться 1 правило, рядок 43 – перевіряємо що ім’я цього правила “emptyRule” та кількість токенів у ньому дорівнює 0 (рядок 44).

```
146 VLEXER_TEST_BEGIN
147 {
148     std::string data =
149         "optRule1 : IF (NOT)? EXISTS ;"
150         "optRule2 : IF NOT? EXISTS ;"
151         "optRule3 : IF (NOT TMP)? EXISTS ;"
152         "optRule4 : IF (',')? EXISTS ;"
153         "optRule5 : IF ','? EXISTS ;";
154     
155     loadSqlGrammar( ESqlDialect::kTest, data.c_str(), data.size() );
156 
157     // Grammar tests:
158     SqlGrammarManager* pManager = SqlGrammarManager::instance();
159 
160     SqlGrammar* pGrammar = (pManager->get_Grammar( ESqlDialect::kTest )).get();
161     TEST_CHECK( pGrammar != nullptr );
162     if( !pGrammar )
163         return;
164 
165 
166     // WORK with Grammar:
167     TEST_EQ(pGrammar->get_RuleCount(), 5);
168 
169 
170     // Rule 1 test:
171     {
172         Rule* r1 = (pGrammar->get_Rule("optRule1")).get();
173         TEST_EQ(r1->size(), 3);
174         TEST_EQ((*r1)[0]->get_NodeType(), kNode_Token);
175         TEST_EQ((*r1)[1]->get_NodeType(), kNode_Opt);
176         TEST_EQ((*r1)[2]->get_NodeType(), kNode_Token);
177     }
178 
179     // Rule 2 test:
180     {
181         Rule* r2 = (pGrammar->get_Rule("optRule2")).get();
182         TEST_EQ(r2->size(), 3);
183         TEST_EQ((*r2)[0]->get_NodeType(), kNode_Token);
184         TEST_EQ((*r2)[1]->get_NodeType(), kNode_Opt);
185         TEST_EQ((*r2)[2]->get_NodeType(), kNode_Token);
186     }
187 
188     // Rule 3 test:
189     {
190         Rule* r3 = (pGrammar->get_Rule("optRule3")).get();
191         TEST_EQ(r3->size(), 3);
192         TEST_EQ((*r3)[0]->get_NodeType(), kNode_Token);
193         TEST_EQ((*r3)[1]->get_NodeType(), kNode_Opt);
194         TEST_EQ((*r3)[2]->get_NodeType(), kNode_Token);
195 
196         Node_Ptr pNode1 = (*r3)[1];
197         Rule_Ptr pSubRule = pNode1->get_SubRule();
198         TEST_EQ( pSubRule->size(), 2 );
199         TEST_EQ((*pSubRule)[0]->get_NodeType(), kNode_Token);
200         TEST_EQ((*pSubRule)[1]->get_NodeType(), kNode_Token);
201     }
```

Рис. 4.13 – Фрагмент тестування правила з опціональними вузлами

Це тест більш складного компонента, а саме тестування опціонального-вузлу. Рядки 148-153 – створення «міні-граматики» з п’яти правил, у яких використані усі можливі варіанти використання оператору опціональності.

Рядки 155-163 так само створюють дерево по граматиці.

Далі починається тестування граматики, правил та вузлів:

- Рядок 167 – перевіряємо що у граматиці 5 правил.
- Рядки 170-177 почергово перевіряють: ім’я правила, кількість токенів, типи кожного з токенів.

- Усі наступні рядки роблять аналогічну задачу для кожного з правил міні-граматики.

РОЗДІЛ 5. ІТЕРАТОР

5.1. Огляд класу RuleIterator

Для обходу побудованого дерева правил граматики розроблено клас RuleIterator, який може переміщатися по правилам (за вертикаллю) та по вузлам правила (за горизонталлю).

Відмітимо, що може існувати декілька незалежних ітераторів одночасно, кожен з них має свою позицію. Що дає можливість мати багатопоточну реалізацію алгоритмів.

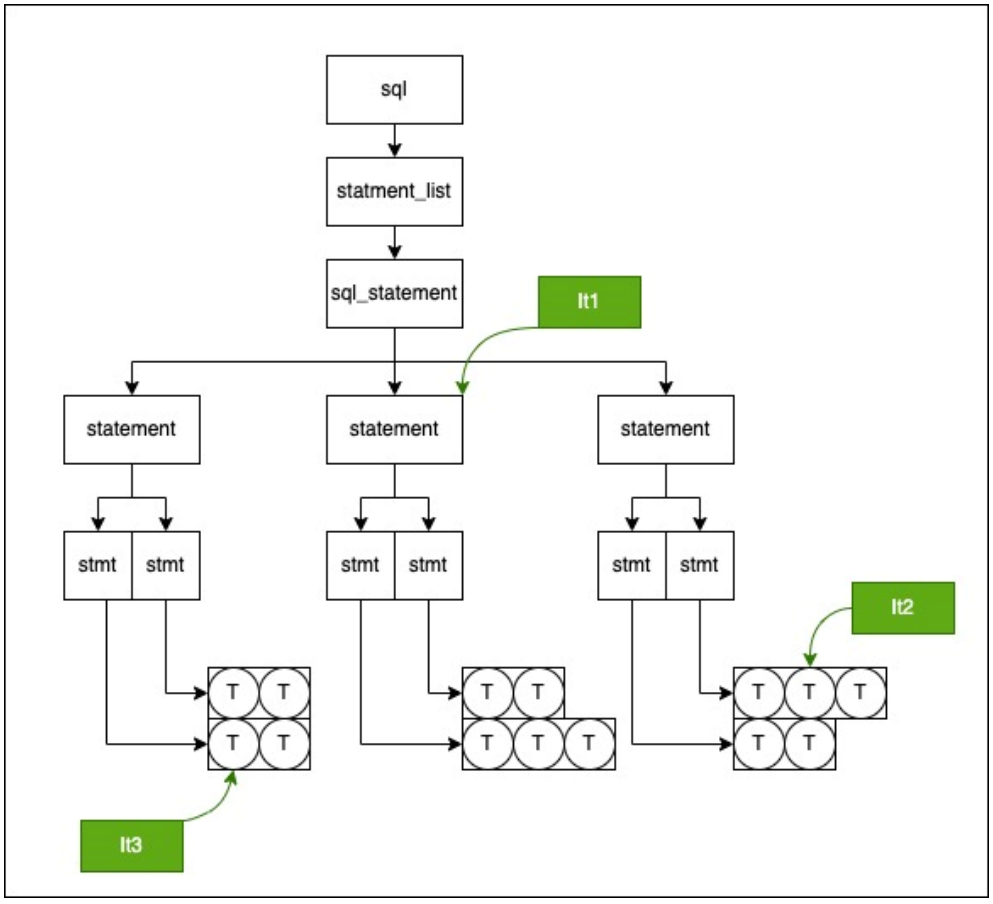


Рис. 5.1 – Статичне дерево граматики з декількома ітераторами

5.2. Огляд класу Vertex

Клас Vertex – це допоміжний клас, задача котрого полягає у реалізації стеку для переміщення по вузлам SQL-правила. Як дані класу він зберігає: вказівники на попередній вузол, батьківський вузол а також індекс даного вузлу-компоненту.

Цей клас має методи для:

- роботи з вузлами:
  - get_Rule();
  - get_Index();
  - get_PrevVertex();
  - get_ParentVertex();
  - get_RuleVertex().
- роботи зі стеком:
  - hasNextNodeInRule();
  - node();
  - getNodeAsString();
  - getStackAsString();
  - printStack().

Під час роботи екземпляри класу Vertex утворюють «ланцюг», кожен з них має вказівники на попередній а також на початок (батьківський вузол). В цілому вони створюють аналог PathBar у програмах візуалізації даних (провідник, Finder і т.д.), тобто ми можемо рухатися вглибину, а також при потребі повертатися назад.

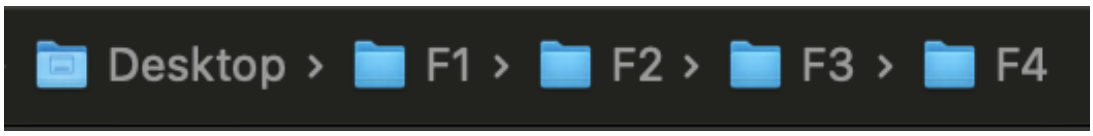


Рис. 5.2 – PathBar Mac OS “Finder”



Рис. 5.3 – PathBar Windows “Explorer”

5.3. Огляд руху ітератора у правилах з різною конструкцією

Для початку візьмемо лінійне правило, тобто всі вузли мають тип Node_Token чи Node_Syntax. Рух ітератора буде лінійним по кожному з них.

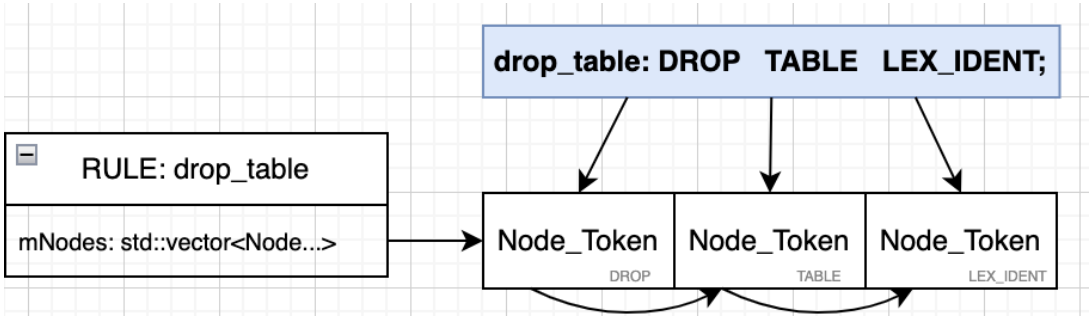


Рис. 5.4 – Рух ітератора в лінійному правилі

Рух ітератора у правилі з опціональним вузлом. Із складнощів тут, це можлива відсутність опціонального токена(-ів). Для визначення цього у коді були створені перевірки для подібних випадків, а саме чи наявні в даному SQL-запиті Node_Optional.

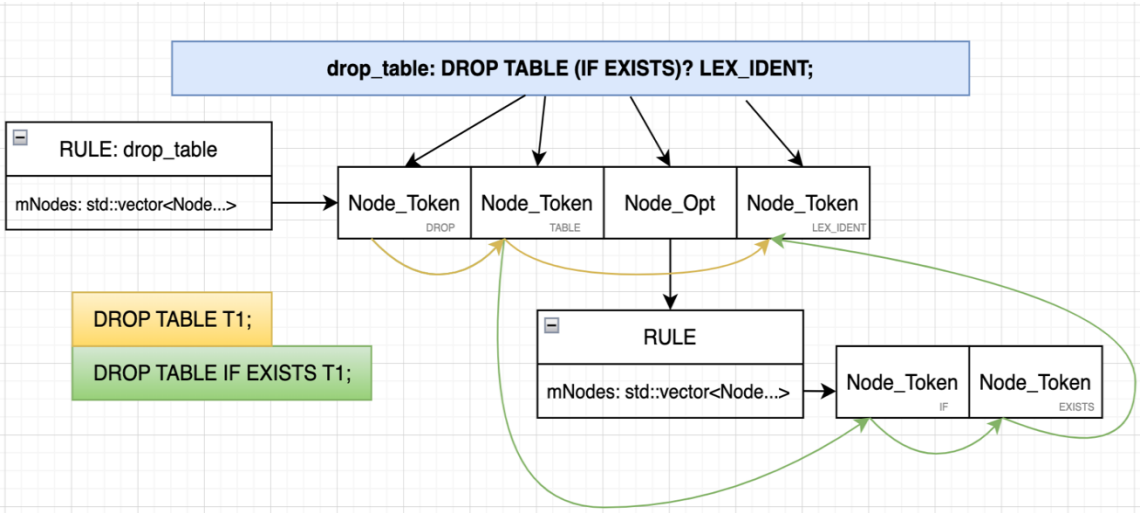


Рис. 5.5 – Рух ітератора в правилі з опціональним вузлом

Рух ітератора в правилі, яке має під собою цикл. У наведеному прикладі присутній цикл, через що, ітератор переходить у тіло циклу, в коді воно реалізовано як нове «під-правило» (subrule), та починає виконувати його роблячи ітерації при знаходженні символу “,”. При відсутності символу-тригеру ітерації, ітератор йде на наступний токен (який йде після циклу).

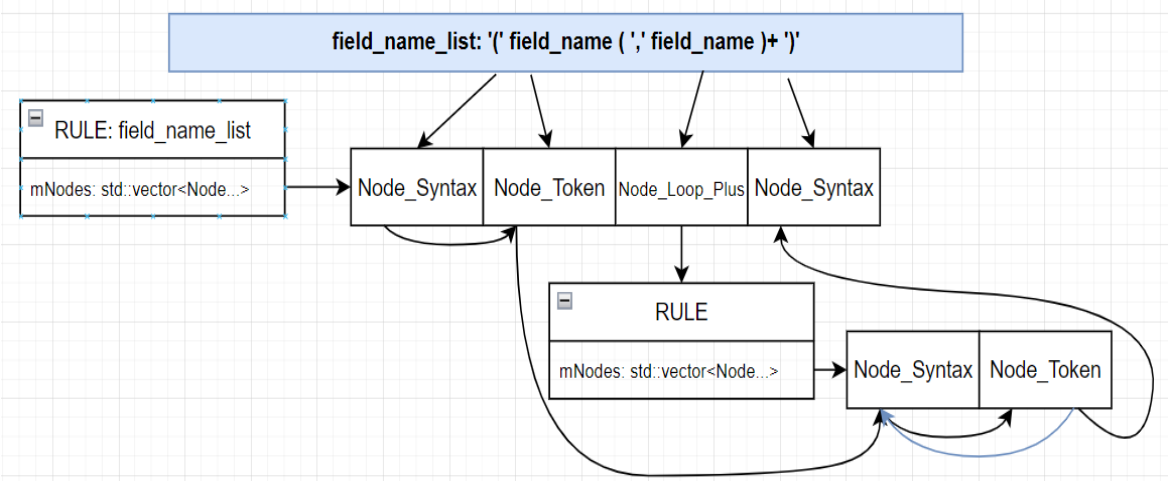


Рис. 5.6 – Рух ітератора в правилі з циклом

Рух ітератора в правилі з опціональним циклом. Його відмінність від попереднього циклу полягає в його варіативності. Мається на увазі що він так само як і Node_Opt може не існувати. У такому випадку ітератор просто перейде до наступних вузлів.

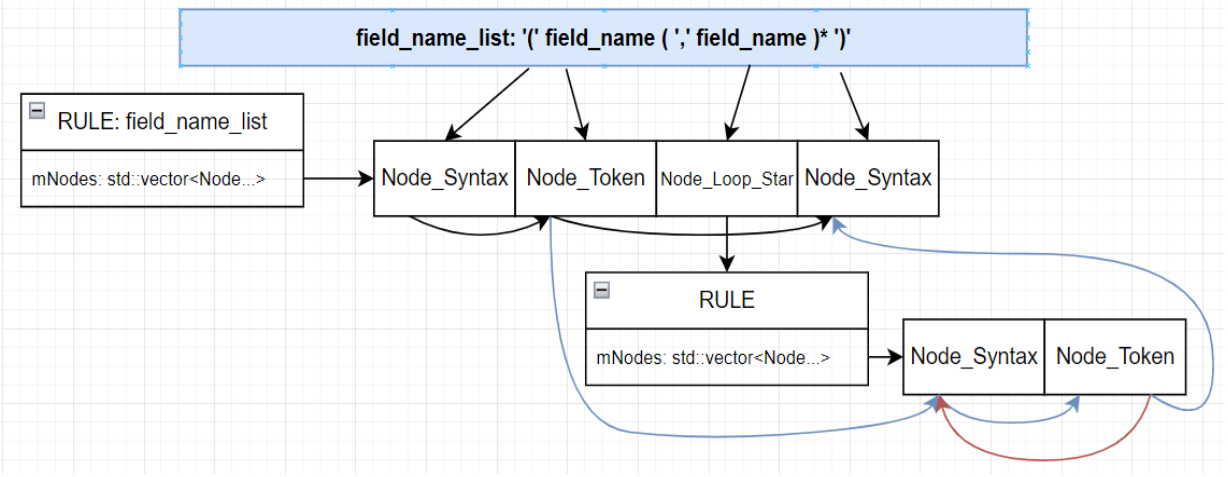


Рис. 5.7 – Рух ітератора в правилі з опціональним циклом

Рух ітератора в правилі з альтернативним вузлом. Node_Alt являє собою нод, який може мати декілька варіантів продовження правила. Рух ітератора в данному випадку буде залежити від запита який вводить. У коді клас Node_Alt зберігає під собою

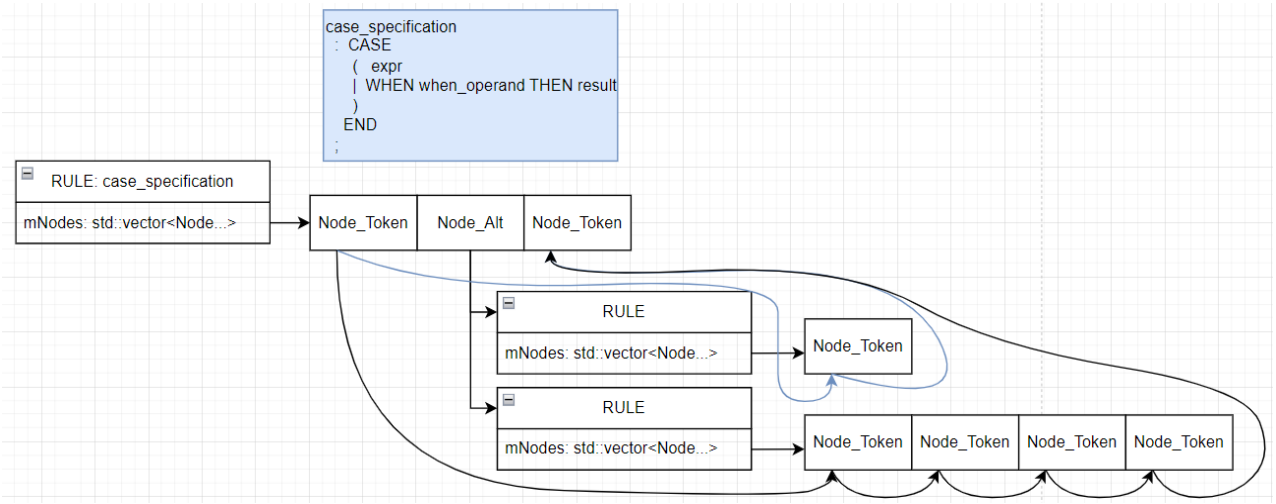


Рис. 5.8 – Рух ітератора в правилі з альтернативним вузлом

Рух ітератора в правилі з вузлом викликом. Node_Call – вузол який «викликає» інше правило. На відміну від попередніх вузлів (Alt, Loop і тд.) під-правило, яке зберігається у батьківському має власне ім’я. По ньому ми у дереві знаходимо відповідне правило, та «викликаємо» його.

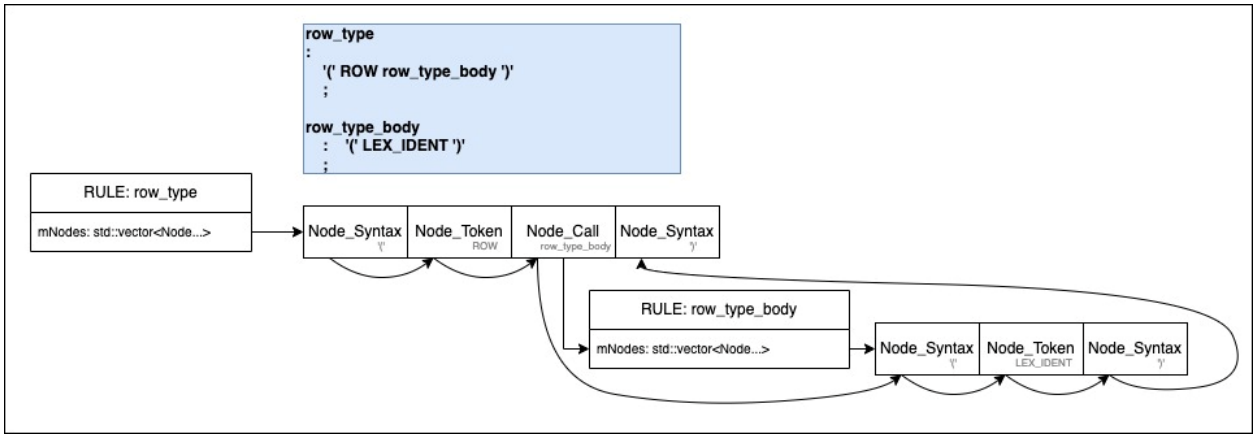


Рис. 5.9 – Рух ітератора в правилі з вузлом-викликом

5.4. Тестування класу RuleIterator

Для тестування класу RuleIterator було створено допоміжний метод GetTestSqlGrammar(). Задача цього метода полягає у створенні «тестової» міні-граматики з невеликим набором правил для тестування поведінки ітератору.

Далі розглянемо декілька тестів:

```
42  // Test simple Linear rule, i.e. no Optional, Alt, Loops.
43  //
44  VLEXER_TEST_BEGIN
45  {
46      SqlGrammar& gr = GetTestSqlGrammar();
47
48      // column_def: LEX_IDENT, VARCHAR
49      if( 1 )
50      {
51          doTest_match( gr, "column_definition",
52              { LEX_IDENT, LITERAL_varchar } );
53      }
```

Рис. 5.10 – Фрагмент тестування ітератору на лінійних правилах

Тест, рядки 49-53, шукає у граматиці gr правило з іменем “column_definition” та зіставляє його з потоком токенів { LEX_IDENT, LITERAL_varchar } на рядку 52, який формується, наприклад, з SQL-запиту «fldName VARCHAR».

Для більшого розуміння «column_definition» використовується у більш складних правилах SQL, наприклад у «create_table». Тобто приклад повного запиту виглядає як «CREATE TABLE tblPerson(fldName VARCHAR)».

```
129 // Test Optional node.  { DROP TABLE [IF EXISTS] LEX_IDENT }
130 //
131 VLEXER_TEST_BEGIN
132 {
133     SqlGrammar& gr = GetTestSqlGrammar();
134
135     // DropTable: DROP TABLE (IF EXISTS)? LEX_IDENT;
136     if( 1 )
137     {
138         doTest_match( gr, "DropTable",
139             { LITERAL_drop, LITERAL_table, LEX_IDENT } );
140     }
141     if( 1 )
142     {
143         doTest_match( gr, "DropTable",
144             { LITERAL_drop, LITERAL_table, LITERAL_if, LITERAL_exists, LEX_IDENT } );
145     }
146
147
148     // OPTIONAL with sub-rule:
149     //      OptionalWithRule: DROP TABLE column_def? LEX_IDENT;
150     //      column_def: LEX_IDENT, VARCHAR;
151     if( 1 )
152     {
153         doTest_match( gr, "OptionalWithRule",
154             { LITERAL_drop, LITERAL_table, LEX_IDENT } );
155     }
156     if( 1 )
157     {
158         doTest_match( gr, "OptionalWithRule",
159             { LITERAL_drop, LITERAL_table, LEX_IDENT, LITERAL_varchar, LEX_IDENT } );
160     }
161
162
163 }
164 VLEXER_TEST_VALENTINA( "[StaticAnalyse][SqlGrammars][Optional]", "" )
```

Рис. 5.9 – Фрагмент тестування ітератору на опціональних правилах

Перший тест складається з двох частин (рядки 136-140 та 141-145 відповідно), а саме з перевірок двох можливих (опціональних) варіантів правила DropTable. У першому випадку ми зіставляємо його з потоком токенів (рядок 139), який складається з 3 токенів, у другому (рядок 144) додається опціональна конструкція з двох токенів IF EXISTS.

Другий тест правила з під-правилом, а саме з опціональним Node_Call. Перевіряємо варіант без нього (рядки 151-155) та з ним (ряідки 156-160).

РОЗДІЛ 6. АЛГОРИТМ AUTOCOMPETE

6.1. AutoComplete

Автозаповнення (AutoComplete) є алгоритмічною технікою, яка використовується в обчислювальних системах для передбачення частково введеного тексту користувачем. Ця технологія знайшла широке застосування в різних областях, включаючи веб-пошук, редактори коду, мобільні додатки та інші інтерфейси користувача.

Принцип роботи Autocomplete (далі AC) полягає в тому, що він аналізує введені символи та на основі цих даних пропонує можливі завершення команди/речення/тощо. Ці пропозиції генеруються на основі набору даних, який може включати словники, історію користувача або інші джерела. Зазвичай, для автозаповнення використовуються алгоритми, які ґрунтуються на структурах даних, таких як trie (префіксні дерева), хеш-таблиці та інші методи пошуку, їх розглянемо пізніше.

Найчастіше AC використовується в інтегрованих середовищах розробки (IDEs) та текстових редакторах. Автозаповнення допомагає програмістам швидше писати код, автоматично заповнюючи імена змінних, функцій та інші конструкції мови програмування. Також AC часто використовують у веб-розробці, наприклад у веб-формах автозаповнення може підвищити зручність користувача, пропонуючи завершення для полів імені, адреси, пошукового запиту тощо.

Тепер розглянемо те, як можна реалізувати алгоритм AutoComplete.

Перший варіант – реалізація через статичні методи, наприклад:

- Словниковий Підхід: Використання фіксованого списку слів або фраз. Це простий, але менш гнучкий метод.
- Префіксні Дерева (Tries): Використовуються для швидкого пошуку словникових слів, що починаються з заданого префіксу.

Другий через динамічні методи, а саме:

- Засновані на Історії Користувача: Системи, які використовують інформацію про попередні запити користувача для пропонування варіантів.
- Алгоритми Пошуку Патернів: Використовують різні алгоритми, такі як фаззі-пошук, для знаходження збігів, які не обов'язково точно відповідають введеному префіксу.

Третій варіант – інтелектуальні методи:

- Машинне Навчання та Штучний Інтелект: Використовуються для створення більш інтелектуальних систем автозаповнення, які можуть адаптуватися до конкретного користувача та його уподобань.
- Контекстно-Залежні Алгоритми: Використовують контекст, наприклад, геолокацію або тип діяльності користувача, для більш точних пропозицій.

Також є варіанти реалізації через інтеграцію з базами даних, як SQL так і NoSQL. А також є варіант з Клієнт-серверною інтеграцію, з використанням асинхронних запитів, чи певних фреймворків.

6.2. AutoComplete для Database Manager-ів (DBM).

Тривіальний та найбільш розповсюджений варіант, який використовується у багатьох DBM, використовує 2 списки:

- Ключові слова SQL;
- Імена об’єктів баз даних.

Як відомо, існує кілька SQL-стандартів: SQL92, SQL99, SQL2003, SQL2008, і тд. Виробники СУБД намагаються дотримуватися цих стандартів, але також вони мають власні SQL-команди. Зазвичай СУБД має команди, які повертають список ключових слів для даної версії та список

імен об’єктів (ідентифікаторів) для обранної бази даних. Наприклад у Valentina DB це:

- SHOW VKEYWORDS;
- SHOW VIDENTS.

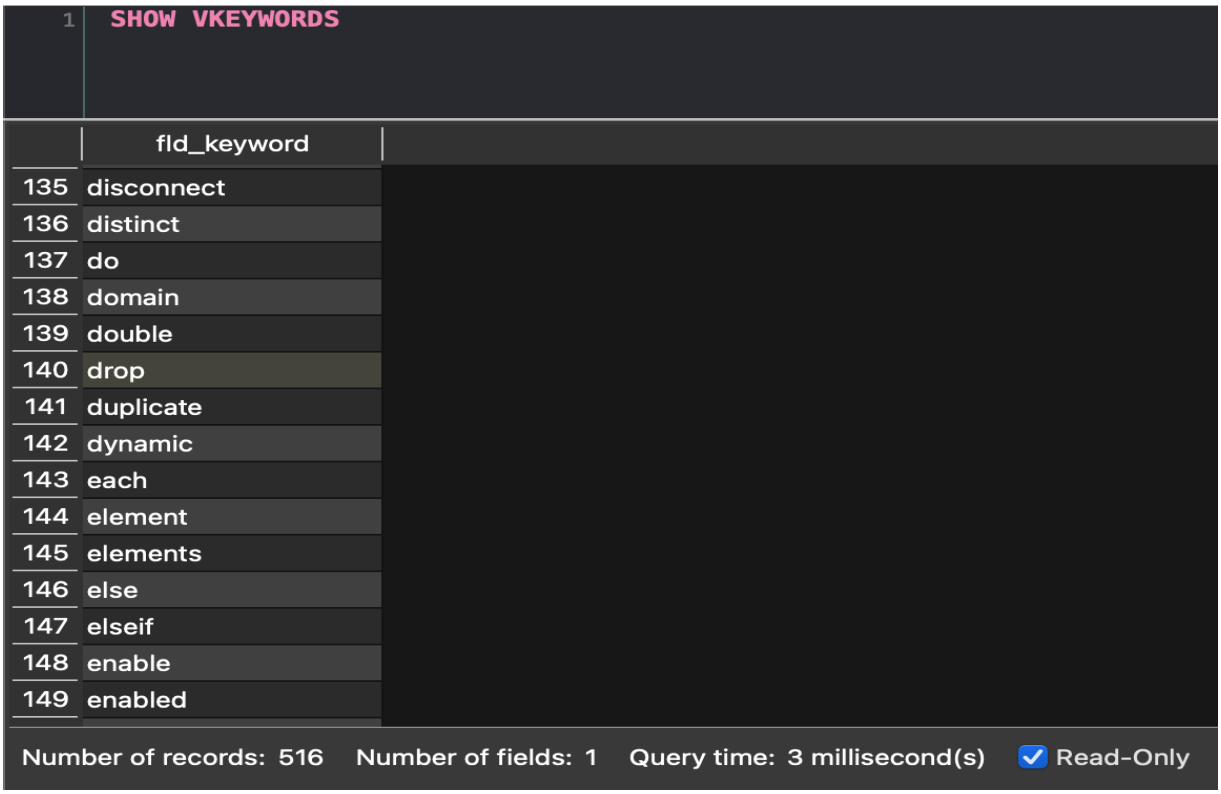


Рис. 6.1 – Результат виконання команди SHOW VKEYWORDS у Valentina DB

Помітимо, що список ключових слів статичний для конкретної версії конкретної СУБД.

```
1  SHOW VIDENTS
```

	fld_ident	fld_KindString	fld_Kind
1	actor	TABLE	1
2	actor.actor_id	FIELD	3
3	actor.first_name	FIELD	3
4	actor.last_name	FIELD	3
5	actor.last_update	FIELD	3
6	address	TABLE	1
7	address.address	FIELD	3
8	address.address_id	FIELD	3
9	address.address2	FIELD	3
10	address.city_id	FIELD	3
11	address.district	FIELD	3
12	address.last_update	FIELD	3
13	address.phone	FIELD	3
14	address.postal_code	FIELD	3
15	category	TABLE	1
16	category.category_id	FIELD	3
17	category.last_update	FIELD	3
18	category.name	FIELD	3
19	city	TABLE	1
20	city.city	FIELD	3
21	city.city_id	FIELD	3
22	city.country_id	FIELD	3

Number of records: 304 Number of fields: 3 Query time: 2 millisecond(s) Read-Only

Рис. 6.2 – Результат виконання команди SHOW VIDENTS у Valentina DB

База даних має списки: таблиць, полів, тригерів, процедур тощо. Маючи цей список ми створюємо під нього масив та вносимо дані. Далі при введенні 2-4 букв починає працювати AC, який опираючись на граматику “підказує” що буде йти далі.

Також треба зазначити, що AC використовує REGEX, тобто алгоритм буде шукати у першу чергу імена, які починаються на введені букви, а потім імена, які містять введені букви. Через це ми маємо алгоритм, який проходить весь масив, через що він є дуже витратним.

6.3. Тестування алгоритму AutoComplete

Алгоритм AutoComplete у нашій реалізації працює використовуючи усі вищезгадані компоненти.

Його робота полягає у:

1) Визначенні команди яку вводить користувач;
2) Визначенні позиції на якій знаходиться користувач;
3) Визначенні наступних можливих токенів цієї команди;
4) Створення масиву рядків з можливими командами;
5) Відображення їх у UI SQL Editor-у.

Тому задача тестування полягала у перевірці даних етапів роботи. Для тесту створювався рядок який «начебто» наразі пише користувач, по ньому запускався алгоритм АС та прогнозував які токени можуть йти далі.

Цей набір токенів ми порівнюємо з підготовленим і дивимося на збіжність. Однак іноді «передбачення» може не бути правильним, але не збігатися з нашим набором даних (наприклад токени для підказки будуть правильні, але в різному порядку), для таких випадків також підготовлений набір софт-тестів, які перевіряють усі можливі збіжності.

6.4. Демонстрація роботи алгоритму

У цьому розділі ми продемонструємо різність роботи нового та старого алгоритмів. Для цього будемо використовувати дві версії Valentina Stuido. Перша з них – реліз-версія 15.1 (далі реліз-версія), друга – зібрана у нашій feature-branch (далі дебаг-версія).

Старий алгоритм в основному використовував алгоритм перебору серед ідентифікаторів та ключових слів. Також у деяких місцях була спроба використовувати ручний статичний аналізатор, щоб покращити результат фільтрації. Наприклад, для команди “INSERT INTO tblN|” алгоритм розуміє, що в цьому місці може бути тільки ім’я таблиці, або представлення (View).

Проблема цього алгоритму полягає в тому, що на це потрібен величезний об’єм ручного коду (на кожну команду кожного діалекту).

У новому алгоритмі статичний аналіз SQL-команди відбувається автоматично на основі граматики обраної бази даних. Через це АС стає більш “розумним” помічником під час розробки.

Давайте розглянемо результати роботи старого та нового алгоритмів на прикладах деяких командах.

Для початку візьмемо команду “CREATE TABLE”. Згідно граматики Valentina SQL ми маємо для неї такий синтаксис:

```
create_table_stmt
    :   CREATE
            (OR REPLACE)?
            ( ( GLOBAL | LOCAL )? TEMPORARY )?
            (RAM | SYSTEM)?
        TABLE
            (IF NOT EXISTS)?
        table_name
        (   table_element_list ( ON COMMIT ( DELETE | PRESERVE ) ROWS )?
        |   AS direct_select_multiple_rows_stmt
        )
    ;
```

Рис. 6.3 – Представлення команди “CREATE TABLE” у граматиці

Нехай користувач написав “CREATE TABLE” та літеру “I” намагаючись написати “IF NOT EXISTS”. Старий алгоритм дає нам величезний список ключових слів, полів, об’єктів які починаються чи мають у собі цю літеру.

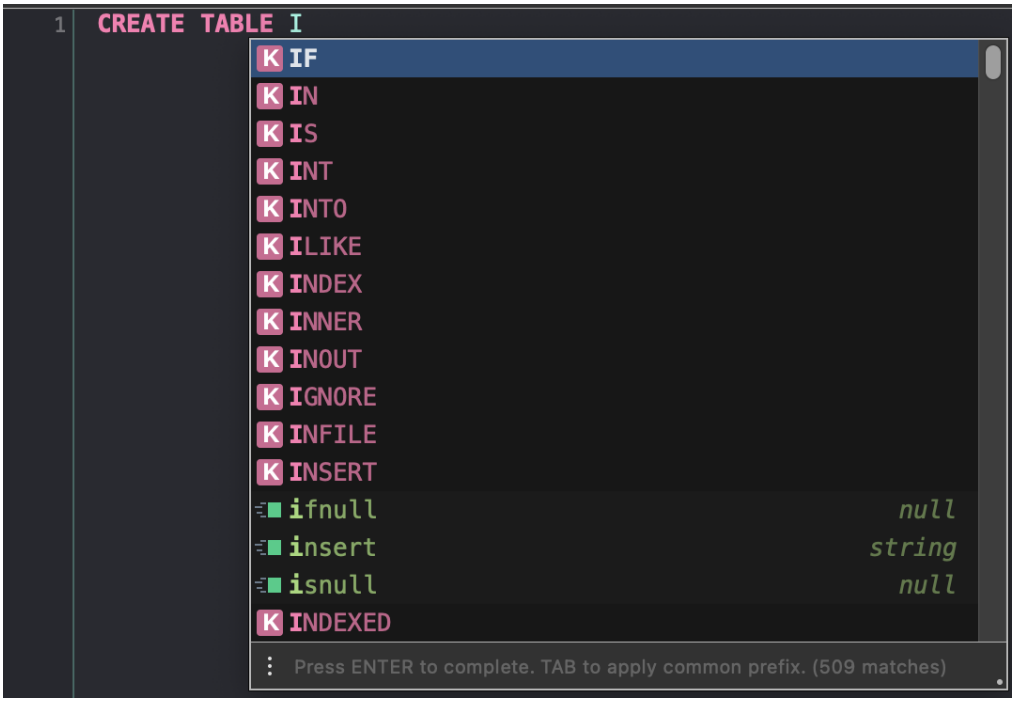


Рис. 6.4 – Результат роботи старого алгоритму

Новий алгоритм показує лише 2 можливих варіанти, які можуть бути зараз.

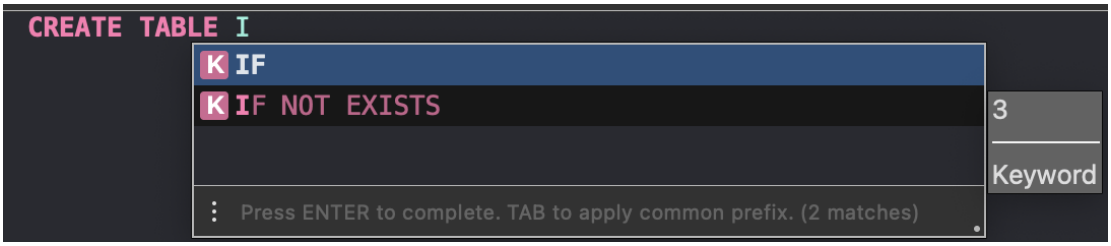


Рис. 6.5 – Результат роботи нового алгоритму

ВИСНОВКИ

Під час цієї роботи було проаналізовано компоненти для статичного аналізу SQL-запитів, а саме: лексери, граматики, AutoComplete та його аспекти. Після аналізу та дослідження кожної з цих тем було почато розробку програмного забезпечення AutoComplete для Database Manager-у Valentina Studio.

Спочатку відбувалося приведення граматик до спільного стилю, оскільки це дає можливість робити універсальне програмне забезпечення для парсингу та подальшої роботи з даними. Наступним кроком був аналіз логіки SQL команд, та виявлення в них спільних рис. На цьому етапі було виявлено, що «різних» логічних команд існує не більше 10 (звідси й взялися Node_ класи).

Після цього почалося детальне продумування логіки кожного з них, а також реалізація цих класів. На даний момент робота над цим проектом продовжується, наступним кроком буде створення класу ітератора, який буде аналізувати та «проходити» по кожному елементу правила.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Статичний аналіз коду: https://uk.wikipedia.org/wiki/Статичний_аналіз_коду
2. Генератор ANTLR: https://uk.wikipedia.org/wiki/ANTLR
3. Формальні граматики: https://ru.wikipedia.org/wiki/Формальная_грамматика
4. Декларативне програмування: https://uk.wikipedia.org/wiki/Декларативне_програмування
5. Дейт, К. Дж. (2009). SQL і реляційна теорія: Як писати точний SQL-код. O'Reilly Media.
6. Челко, Дж. (2015). SQL для розумників від Джо Челко: Розширене програмування на SQL. Morgan Kaufmann.
7. Крейбіх, Дж. (2010). Використання SQLite. O'Reilly Media.
8. Молінаро, А. (2006). SQL Куховарська книга. O'Reilly Media.
9. Оппел, А. Дж. (2008). Бази даних для чайників. McGraw-Hill Education.
10. Ернандес, М. (2013). Проектування баз даних для простих смертних: Практичний посібник з реляційного проектування баз даних. Addison-Wesley Professional.
11. Болю, А. (2009). Вивчення SQL. O'Reilly Media.
12. Тейлор, А. (2015). SQL Все-в-одному для чайників. For Dummies.
13. C++
14. Страуструп, Б. (2013). Мова програмування C++. Addison-Wesley.
15. Маєрс, С. (2005). Ефективне C++: 55 конкретних способів покращити ваші програми та проекти. Addison-Wesley.
16. Саттер, Г., та Александреску, А. (2005). Стандарти кодування на C++: 101 правило, рекомендація та найкращі практики. Addison-Wesley.
17. Ліппман, С. Б., Лажуа, Дж., та Моо, Б. Е. (2012). C++ Практичний посібник. Addison-Wesley.
18. Йозуттіс, Н. М. (2012). Стандартна бібліотека C++: Посібник і довідник. Addison-Wesley.

19. Страуструп, Б. (2014). Програмування: Принципи та практика використання C++. Addison-Wesley.
20. Александреску, А. (2001). Сучасний дизайн на C++: Генеративне програмування та патерни проектування. Addison-Wesley.
21. Маєрс, С. (1997). Більше ефективного C++: 35 нових способів покращити ваші програми та проекти. Addison-Wesley.