ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

НАВЧАЛЬНО–НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра інженерії програмного забезпечення

Пояснювальна записка

до магістерської роботи
на ступінь вищої освіти магістр

на тему: **«РОЗРОБКА УДОСКОНАЛЕНОГО ГЕНЕРАТОРА ПАРСЕРІВ КОНТЕКСТНО-ВІЛЬНОЇ ГРАМАТИКИ ЗА ДОПОМОГОЮ АЛГОРИТМІВ ПОШУКУ ТА МАШИННОГО НАВЧАННЯ»**

Виконав: студент 7 курсу, групи ППЗМ–71
спеціальності
121 Інженерія програмного забезпечення
(шифр і назва спеціальності)
Іваницький Денис Сергійович
(прізвище та ініціали)
Керівний Негоденко О.В.
(прізвище та ініціали)
Рецензент
(прізвище та ініціали)

Київ – 2021

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра - Інженерії програмного забезпечення
Ступінь вищої освіти - «Магістр»
Спеціальність - 121 «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення
_____О.В. Негоденко
“____”_____20___року

З А В Д А Н Н Я
НА МАГІСТЕРСЬКУ РОБОТУ СТУДЕНТУ

Іваницькому Денис Сергійовичу
(прізвище, ім’я, по батькові)

1. Тема роботи: «Розробка удосконаленого генератора контекстно-вільної граматки за допомогою алгоритмів пошуку та машинного навчання»
Керівник роботи к.т.н., доцент Негоденко Олена Василівна ,
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом вищого навчального закладу від “12” березня 2021 року №65.
2. Строк подання студентом роботи 01.06.2021
3. Вихідні дані до роботи:
 3.1 Вимоги до кваліфікаційної роботи магістра з актуальних завдань спеціальності;
 3.2 Нормативні матеріали (стандарти);
 3.3 Технічні вимоги;
 3.4 Науково-технічна література з питань, пов‘язаних з темою роботи.
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити):
 4.1 Порівняльний аналіз результатів, отриманих іншими авторами;
 4.2 Методика дослідження;
 4.3 Результати дослідження;
 4.4 Висновки

5. Перелік графічного матеріалу.
 5.1 Титулка
 5.2 Актуальність
 5.3 Мета та об’єкт
 5.4 Методи роботи генератора
 5.5 Недоліки попереднього методу
 5.6 Удосконалений генератор
 5.7 Алгоритм роботи створеного ГК
 5.8 Алгоритм роботи оптимізатора

5. Перелік графічного матеріалу.
5.9 Створення моделі передбачень
5.10 Порівняння роботи аглоритмів
5.11 Висновки

6. Дата видачі завдання 19.04.2021

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів магістерської роботи	Строк виконання етапів роботи	Примітка
1	Підбір науково-технічної літератури	19.04.2021	Виконано
2	Огляд існуючих рішень	23.04.2021	Виконано
3	Огляд існуючих алгоритмів пошуку	27.04.2021	Виконано
4	Дослідження систем машинного навчання	05.05.2021	Виконано
5	Визначення ефективності алгоритмів	10.05.2021	Виконано
6	Вступ, реферат, розділи, висновки	25.05.2021	Виконано
7	Розробка обов‘язкових демонстраційних матеріалів	30.05.2021	Виконано
8	Попередній захист роботи	31.05.2021	Виконано

Студент _____ _____
(підпис) (прізвище та ініціали)

Керівник роботи _____ _____
(підпис) (прізвище та ініціали)

РЕФЕРАТ

Текстова частина магістерської роботи 50 с, 10 рис, 4 табл, 10 джерел.

ПАРСИНГ, АНАЛІЗАТОР, ВИРАЗ, КОТНЕКСТНО-ВІЛЬНА ГРАМАТИКА, ДЕРЕВО, АГЛОРИТМ, ПОШУК, ПРАВИЛА РОЗБОРУ.

Об'єктом дослідження – процес збору великого об'єму інформації.

Мета роботи – підвищення швидкої функціонування генератора контекстно-вільної граматики, а також удосконалення генератора за допомогою машинного навчання та оптимізації алгоритмів пошуку.

Предмет дослідження - генератор парсерів контекстно-вільної граматики.

Методи дослідження. Теорія графів, математична статистика, теорія ймовірності, математичне імітаційне моделювання, машинного навчання та інтелектуального аналізу (обробки) даних.

Проведено аналіз існуючих рішень генераторів, які в реальному часі можуть генерувати парсера до однієї з мов програмування.

Проведено аналіз швидкодії генератора на прикладі генерування парсера з синтаксисом базових математичних виразів.

Розглянуто особливості кожного з генераторів їх методи та алгоритми генерування синтаксичного дерева.

На основі результатів виконаних досліджень запропоновано універсальний алгоритм швидкого пошуку даних та генерація правила на основі тексту за допомогою машинного навчання.

Отриманні результати дозволяють швидше знаходити з великого об'єму певну інформація. Також дозволяють моделювати та розробляти правила для пошуку інформації, а також на основі певної кількості інформація (тексту) генерувати правила за допомогою машиного навчання.

ЗМІСТ

ВСТУП 9
1 ДОСЛІДЖЕННЯ РЕАЛІЗАЦІЙ І КЛІСИФІКАЦІЙ ГЕНЕРАТОРА-ПАРСЕРІВ 11
1.1 Значення генераторів парсерів контекстно-вільної граматики 11
1.2 Значення генераторів парсерів контекстно-вільної граматики 13
1.3 Види та реалізації парсерів 13
1.4 Методи і алгоритми формувань парсерів 15
2 АНАЛІЗ ЗАДАЧ, ЩО ВИНИКАЮТЬ ПРИ РОЗРОБЦІ ГЕНЕРАТОРІВ, ТА ОГЛЯД ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 20
2.1 Генератор парсерів «GNU Bison» 20
2.2 Генератор парсерів «Yacc» 21
2.3 Генератор парсерів «Lex» 22
2.4 Машинне навчання 23
2.4 Проблеми які виникають при розробці ти використанні генераторів та огляд програмного забезпечення 25
3 ПРАКТИЧНЕ ВДОСКОНАЛЕННЯ ГЕНЕРАТОРА ПАРСЕРІВ НА ОСНОВІ ЗАСТОСУВАННЯ АЛГОРИТМІВ ПОШУКУ ТА МАШИННОГО НАВЧАННЯ 30
3.1 Архітектура генератора парсерів контекстно-вільної граматики 30
3.2 Практичне моделювання генератора конекстно-вільної граматики 31
3.2.1 Моделювання граматики 31
3.2.2 Моделювання парсера вхідної граматики 36
3.3 Оцінка стану вихідного генератора 43
ВИСНОВКИ 45

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

БНФ – Нотація Бекуса-Наура

КВ граматика – контекстно-вільна граматика

ІКП – інтерпретатор комп’ютерних програм

LL – аналізатор – синтаксичний аналізатор методом рекурсивного спуску

LR – аналізатор – лівосторонній аналізатор з правосторонньою граматикою

SEO – оптимізація пошукових систем

ВСТУП

Актуальність теми. Починаючи з 20 століття, люди почали все частіше використовувати інформацію яка подається у цифровому вигляді. Зазвичай інформація подається в текстовому вигляді та об’єми її дуже великі. І часто виникає необхідність проаналізувати вхідну інформацію та виконати її структурування. Людина не може сама за короткий термін проаналізувати такий великий об’єм інформації. Для цієї роботи використовується синтаксичні аналізатори або як ще їх називають – парсери.

Синтаксичний аналіз, або парсинг, у лінгвістиці – це процес порівняння лінійної послідовності лексем (слів) природньої або формальної мови з її формальної граматикою. Результатом парсингу є класифікування даних або представлення цих даних у вигляді синтаксичного дерева.

Аналізатор – це програма, яка аналізує вхідний текст по заданим правилам, записує результат у вигляді дерева або просто класифікує дані залежності від їх значень.

Синтаксичний аналіз є дуже важливим компонентом для опрацювання тексту і спрямований на розпізнавання та в подальшому класифікування і групування даних. Наприклад SEO (оптимізація пошукових систем) інструменти створені для аналізу: посилань, знаходження ключових слів в запитах. На основі аналізу користувачу буде видано тільки актуальні дані до його запиту.

Використовуючи парсинг, можна обробити дуже велику кількість даних за малий термін. В цілому парсинг є ефективним рішення для автоматизації збору інформації та її можливої подальшої зміни.

Але однією з подальших проблем є те, що як правило конкретний парсер підходить тільки для однієї задачі. Це унеможливлює застосування одного і того ж парсера для різних процесів. Створення парсера відбувається за допомогою коду, що робить цей процес важким для звичайної не підготовленій людини яка не має певних знань в даній галузі.

Для вирішення таких проблем і були створені генератори парсерів. Як правило вони створювалися лише для створення мов програмування. До тож не могли класифікувати дані та не мали машинного навчання.

Метою магістерської роботи є підвищення швидкої функціонування генератора контекстно-вільної граматики, а також удосконалення генератора за допомогою машинного навчання та оптимізації алгоритмів пошуку.

Завдання магістерської роботи: пришвидшити алгоритм парсингу, оптимізувати пошуку в дереві, створити прототипу генератора.

Об'єктом дослідження – процес збору великого об'єму інформації.

*Предмет дослідженн*я - генератор парсерів контекстно-вільної граматики.

Для досягнення поставленої мети необхідно вирішити такі завдання:

- провести аналіз і зробити загальну характеристику предметної області;
- розробити вимоги до генератора КВ граматики;
- провести аналіз існуючих генераторів КВ граматики;
- провести обґрунтування вибір засобів розробки і проектування
- спроектувати структуру бази даних;
- реалізувати інформаційну систему.

Практична значущість результатів: Даний продукт може бути використаний різними розробниками, а також компаніями. Додаток або бібліотека може використовуватися як звичайний генератор парсерів для витягу потрібної інформації та її класифікування, а також в більш специфічних сферах таких як: створення мов програмування.

1 ДОСЛІДЖЕННЯ РЕАЛІЗАЦІЙ І КЛІСИФІКАЦІЙ ГЕНЕРАТОРА-ПАРСЕРІВ

1.1 Значення генераторів парсерів контекстно-вільної граматики

На сьогоднішній день, суспільство сприймає дуже багато інформації. Та все більш дана інформація подається у вигляді цифровому вигляді. Цифрова інформація представляється у текстовому вигляді, щоб людині було зручніше її приймати, але обсяги її можуть бути дуже великі. Здебільшого, отриману інформацію треба аналізувати та виконати її структурування. Людина не може виконати таку роботу за короткий термін, а інформація може приходити безперервно. В такому випадку використовують синтаксичні аналізатори або як їх ще називаються - парсера.

Синтаксичний аналізатор – це програма або модуль, який на вхід приймає дані, структурує їх, та подає на вихід [2]. Сам парсер будується на раніше створених правилах. Тому, як правило, один синтаксичний аналізатор для однієї задачі.

Однак самі парсери дуже важко писати, а змінювати їх у використанні ще важче. Тому для побудови парсерів використовують генератори парсерів.
Генератор парсерів – це програма або модуль, які можуть генерувати парсера по заданих правилах. Правила задаються у вигляді формальної граматики, записаною в нотації, подібній до БНФ [3].

Як правило, граматика генераторів, опису їх вхідних правил, є формальна граматика. Формальна граматика або граматика формальних мова – це спосіб опису формальної мови, тобто виділення деякої підмножини з множини всіх слів деякого скінченого алфавіту. Сама формальна граматика як правило поділяється на породжувальні та на аналітичні граматики. Перші створюються правила, за допомогою яких можна побудувати будь-яке слово, а другі визначають по вхідному слову чи належить воно до цієї мови [4].

Поняття алфавіту в формальній мові – це скінченна множина будь-яких символі. Сформулюємо це у вигляді підмножини. ε – порожній список, будь-яке

слово або послідовність. Алфавіт є об'єднання алфавітів, їх перетин або різниця. Позначимо A – алфавіт, тоді:

A^+ - це є множина усіх можливих послідовностей, що складені з елементів даного алфавіту та до яких не входить пустий список ε.

A^* - це є множина усіх можливих послідовностей, які були створені з цього алфавіту, будь-якої довжини. Їм можна записати таким чином: $A^* = A^+ \cup \{\varepsilon\}$

A^i – це є множина усіх послідовностей, які були створені з цього алфавіту і довжина не повинна бути більшою як і.

Мова в алфавіті A це множина ланцюжків скінченної довжини в цьому алфавіті.

Формальна граматика – являє собою $G = \{N, T, P, S\}$, де:

- T – алфавіт термінальних символів
- N – алфавіт нетермінальних символів
- S – аксіома, символ який був спеціально виділений, з якого і починається опис граматики.
- P – правила виводу, скінченна підмножина множин $(T \cup N)^+ \times (T \cup N)^*$

В такому випадку ліва сторона не можу бути порожньою [4].

Контекстно-вільна граматика одна з підмножин формальної граматики.

Ця граматика G є уособленням (N, T, P,S), де:

- $S \in N$
- N та T є скінченними множинами, що не перетинаються
- P є скінченною множиною, де $N \times (N \cup T)^*$

До цих літер застосовуються такі назви: N – множина не термінальних символів, T – множина термінальних символів, P – множина правил виводу, S – перший вхідний символ (початковий). В лівій частині правила виводу має міститися одна змінна як є нетермінальним символом. Тобто, повинна виконуватися умова, де $\alpha \in N, \beta \in (N \cup T)^*$, де $|\beta| \geq 1$. Продовженням КВ-граматики є стохастичні КВ-граматики, тобто там де є невпевненість в правильності вводу даних.

1.2 Значення генераторів парсерів контекстно-вільної граматики

Зазвичай парсери дуже важків в написанні. Особливо це помітно в тих завданням в яких вхідних даних дуже багато. Тому в такому випадку застосовують генератор парсерів. Генератор парсерів – це програма як потрібна для генерування парсерів, за допомогою заданої тільки вільної граматики. Сам генератор складається з декількох речей. Це граматичний аналізатор в якому задається сама граматаки для вихідного парсера, генератор синтаксичного дерева та генератор вихідного компілятор. В граматичному аналізаторі задається список правил. Зазвичай вони задаються на спрощеній мові, котра будується та використовується як ієрархічна. Генератор повинен продивлюватися правила та знаходити помилки. Помилки можуть бути заховані в самій граматиці тобто генератору треба прогодитися по кожному правилу та зпівставляти всі правила які можуть бути реалізовані в заданій граматиці.

1.3 Види та реалізації парсерів

Види парсерів є різні. Вони можуть розрізнятися по кількості вхідних даних це як однопрохідні та і багатопрохідні. Однопрохідні парсери можуть тільки 1 раз пройтися по граматиці в той час коли багатопрохідні можуть декілька разів прикладати правила для граматики. Як правило для багатопрохідних парсерів треба більший час для створення визначеної структури. Також парсери можна поділити на такі види:

- Черга класифікованих лексем
- Абстрактне дерево
- Ієрархічні структури
- Таблиці даних

Перший вид це те коли лексеми класифікуються та створюють собою чергу.

Другий види, це абстрактне дерево. Воно представлено скінченою множиною, всі вершини якого співставленні зі спеціальними операторами котрі були задані в правилах раніше як зображено на рис 1.0

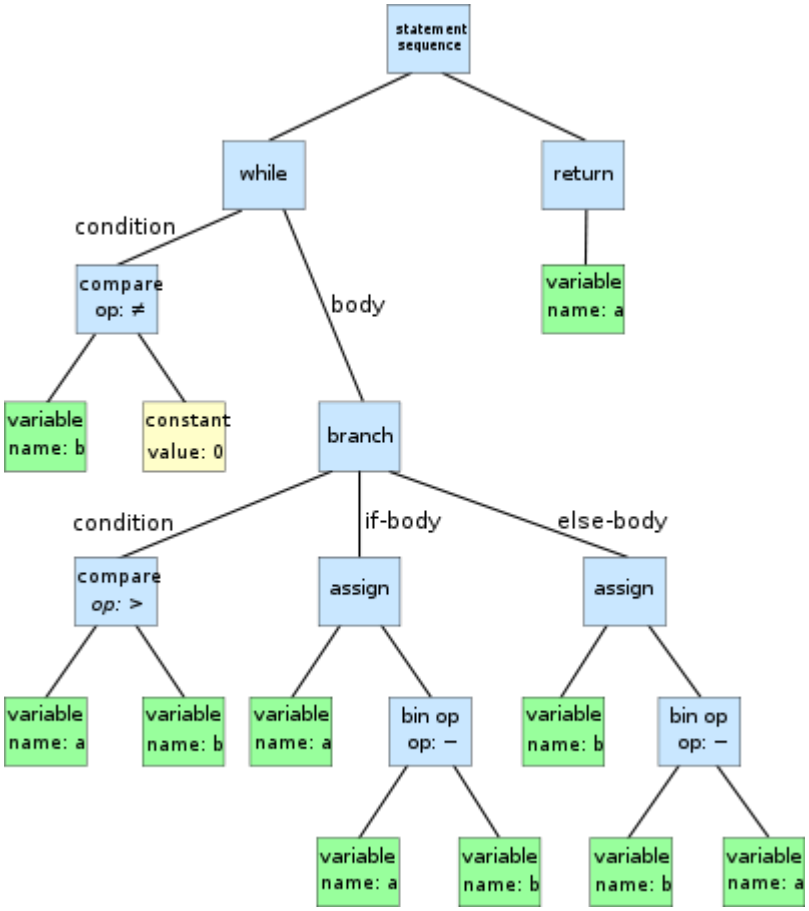


Рисунок. 1.0 - Приклад абстрактного дерева

Синтаксичні дерева використовують в парсерах для проміжного представленнях програм між деревом розбору і структурою даних, яка потім використовується як внутрішнє представлення компілятора або ІКП для оптимізації і генерації коду [5].

Третій тип – це ієрархічна структура. Це множина будь-чого, частково упорядкована так, що існує тільки один елемент цієї множини, який не має попереднього, а всі інші елементи мають тільки один попередній [6]. Також це й тип можу бути представлений як об’єкт котрий має більш узагальнені ознаки свого головного предка.

Четвертий тип – це таблиці даних. Їх суть полягає в тому що всі проміжні дані будуть записувати в таблицю. За рахунок цього можна швидко знайти правило, яке і буде відповідати даній структурі.

Останній тип поділу – це написання вихідного коду. Він може бути написаний статично, тобто програмістом, а може і бути згенерований за допомогою таких програм як GNU Bison та Yacc.

Генератор може бути реалізований як локально так і мати повноціний генератор. Локальний парсер має на вхід завантажити правила і вони не будуть створені локальні. Тобто парсер повинен отримати вхідними даними гарматику та сам текст для компіляції. Далі генератор починає парсити код який вже був завантажений, відразу без створення віртуальних правил.

Якщо ж прасер має окремий генератор, то правила будуться створені у вигляді дерева. І вже тільки потім можна буде відсилати їх на обробку генераторм вихідних парсерів. Також локьний генератор не буде такий швидкий так як він відразу посинає парсити текст, в той час коли генератор має окерему реалізацію правил, то вони будуть парситися в певну структуру, зазвичай це синтаксичне дерево, а вже потім буде проводитися генерація вихідного парсеру. В такому випадку ця реалізація буде швидше працювати для обробки синтаксичного тексту (правил).

1.4 Методи і алгоритми формувань парсерів

Існує дуже багато різноманітним варіантів реалізації методів синтаксичного парсеру.

Низхідний синтаксичний аналіз – це аналіз який починає свою роботу згори і спускається донизу. Таким чином цей аналіз намагається знайти найбільший початок z слова x. Цей спосіб працює за допомогою синтаксичного дерева за допомогою який буде пройдений згори донизу за допомогою раніше заданих формальних правил граматики. Токени зчитаються зліва на правою. Ліва сторона

не повинна бути порожньою. Він застосовує формальну мову яка описується в LL-граматиці. Умови застосування цієї граматики:

Нехай в даній формальній граматиці N – скінченна множина нетермінальних символів. Тоді $\sum$ - це скінченна множина термінальних символів, тоді метод рекурсивного спуску можна записувати тільки, якщо кожне правило цієї граматики такий вигляд: $A \to a$, де $a \in (\sum \cup N)^*$, це і єдине правило виводу для цього нетермінала [7].

Один з популярних методів створення синтаксичного дерева це – запис польської анотації або як його ще називаються польський зворотній запис. Це одна із форм запису математичних операцій, де знак цих операцій записується в кінці після символів. Якщо знаки записуються до символів, то такий запис називається польська анотація. Розглянемо декілька прикладів математичних рівнянь.

$$a + b * c/q$$

де a,b,c,q – можуть бути довільними виразами, числами, словами.

В зворотній польській анотації це рівняння буде записано так:

$$abcq/* +$$

Тобто для початку парсер повинен розпізнати першість математичних знаків. Всі вони повинні бути записані в кінці рядка в правильному порядку. Всі вирази записуються з початку. Тому коли цей вираз буде парситися, то це буде відбуватися так: спочатку він (парсер) зчитає літеру а потім впише її в стек. Всі наступні літери також будуть зчитані в стек. Коли парсер дійде до виконання математичних операцій, то він буде брати всі дані з кінця стека. Тобто всі дані будуть братися з кінця стека.

Наступний приклад буде такий:

$$(x + y) * (x - x)$$

Це рівняння не є простим тому в польській анотації будуть змінюватися запис так в ньому є дужки. В такому випадку це рівняння в зворотній польській анотації буде виглядати так:

$$(xy)(xx) + - *$$

В такому випадку всі дані будуть записані так само як і в попередньому варіанті, але додаються дужки щоб знати в якій послідовності йдуть вирази.

Для цього виразу розберемо алгоритм парсингу більш детальніше.

- Якщо вхідний знак x_1, то записуємо стек
- Якщо вхідний знак y_1, то записуємо його в стек
- Якщо вхідний знак x_1, то записуємо його в стек
- Якщо вхідний знак x_1, то записуємов його в стек
- Якщо вхідний знак буде термінал, то записуємо в кінець

Так і відбувається парсинг та записування зворотної польської анотації.

Висхідний синтаксичний парсер – його завдання полягає в почати аналіз з низу та йти доверху заміняючи на більші структури на його місці.

Це загальні методи формувань парсерів. В свою чергу методи використовують певні алгоритми. Серед них є LL-аналізатор LR-аналізатор.

LL-аналізатор – це алгоритм синтаксичного аналізу методом рекурсивного спуску для підмножини конекстно-вільних граматик. Він обробляє вхід зліва на право (підказка в першій букві L – left, як означає ліво) та будує ліворекурсивне виведення рядка. Клас граматики, що розпізнаються цим аналізатором, називається LL-граматики [8].

LR-граматика – це найбільш розповсюджений підклас контекстно-вільних граматик. Цей тип забезпечує розпізнавання ланцюжка при перегляді зліва направо, і дозволяє виконувати правостороннє згортання. Існують декілька варіантів такого парсера це SLR-парсери, LALR-парсери, Canonical- LR- парсери. Вони відрізняються своєю структурою і самим алгоритмами парсингу тексту. SLR-парсер, більший чим LR клас граматик. Він може використовуватися тоді коли LR не в змозі зчитувати (парсити) текст через конфлікти які виникли. Зазвичай такі проблема мають характер. Наприклад проблема через здвиг-приведення або приведення-приведення. LALR – теж більш обширний парсер який має висхідний алгоритм синтаксичного розбору. Він може використовуватися також в тих умовах

де звичайного LR аналізатор не буде вистачати або в звичайного аналізатор відбулася помилка. Граматика класу LR є більша.

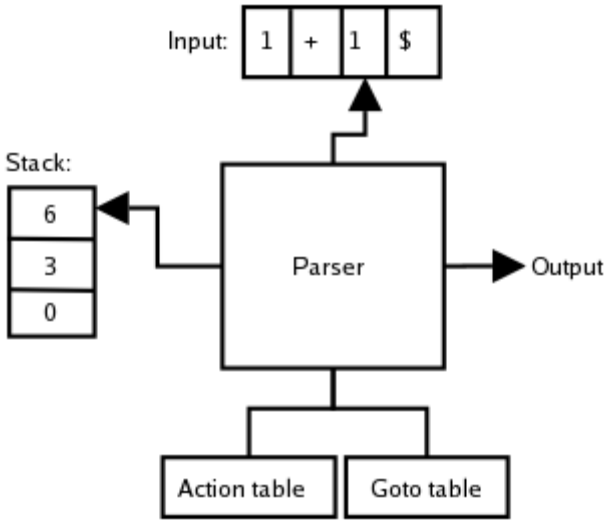


Рисунок. 1.1 – Вигляд парсера

На рис. 1.1 можна побачити зображення аналізатора. Input – це вхідні дані. В даному прикладі використовується математичний рядок. Stack – використовується для зберігання проміжних даних. Action table – список дій котрі виконуються в даний момент. Goto table – є переходом до наступної дії.

SLR – також являється висхідним алгоритмом синтаксичного розбору. Він може працювати в тому випадку, коли побудова LR таблиці розбору для даної граматики неможливе через конфлікт зрушення-приведення або приведення-приведення. Таким чином, клас граматик, що розбираються по SLR ширше, ніж клас LR-граматик.

Для кожного нетермінала A в граматиці генерується безліч терміналів перших (A), створене наступним чином:

- якщо в граматиці є правило з A в лівій частині і правою частиною, що починається з терміналу, то даний термінал входить в першого (A)

- якщо в граматиці є правило з A в лівій частині і правою частиною, що починається з нетермінала (позначимо B), то перших (B) строго входить в першого (A)
- ніякі інші термінали не входять до першого (A)

Ці ж безлічі використовуються для побудови аналізатора LL (1). Далі на основі множин першого (A) генеруються безлічі наступних (A), певні наступним чином:

- якщо в граматиці є правило, де зустрічається A нема на останній позиції, і за ним слід термінал, то термінал входить в наступних (A).
- якщо в граматиці є правило, де зустрічається A нема на останній позиції, і за ним слід нетермінал (позначимо B), то першого (B) строго входить в наступного (A)
- якщо в граматиці є правило, де зустрічається A на останній позиції, то наступного (C) строго входить в наступного (A), де C - ліва частина правила.
- ніякі інші термінали не входять до наступного (A)

Далі проводиться генерація таблиці розбору, як для LR (0), з відзнакою при занесенні дій приведення. Приведення для стану S і вхідного символу C заноситься в таблицю тільки тоді, коли в S є ланцюжок, відповідна цілої правій частині правила, і нетермінал N з лівої частини правила задовольняє умові «C входить в послідовного (N)». Це призводить до того, що для SLR (1) відбувається менше спроб поставити «приведення» в клітинку таблиці розбору, що зменшує ризик виникнення конфліктів з привидами, іноді зовсім позбавляє від них і робить граматику, не розбирається по LR (0), розглядуваної.

2 АНАЛІЗ ЗАДАЧ, ЩО ВИНИКАЮТЬ ПРИ РОЗРОБЦІ ГЕНЕРАТОРІВ, ТА ОГЛЯД ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Генератор парсерів «GNU Bison»

GNU Bison (бізон) – це генератор парсерів загального призначення, який перетворює анотовану контекстно-вільну граматику у детермінований LR парсер або узагальнений синтаксичний аналізатор LR із використанням таблиць синтаксичного аналіз LALR. Bison також може генерувати IELR або канонічні LR таблиці синтаксичного аналізу LR. Як тільки ви володієте Bison, ви можете використовувати його для розробки широкого кола мовних парсерів, від тих, що використовуються в простих настільних калькуляторах до складних мов програмування [9]. Bison є сумісним з Yacc. Всі правильно написані граматики Yacc будуть працювати з Bison буз змін. Кожен, хто знайомий з Yacc, повинен мати можливість використовувати Bison`s з невеликими труднощами. Для користування бізоном, потрібно досить добре володіти мовами програмування такими як C, C++ або Java що в повній мірі користуватися бізоном.

У режимі POSIX Bison не тільки сумісний з Yacc, але також має кілька розширень щодо цієї програми, зокрема:

- генерування анотацій для конфліктів (відстежує місцезнаходження помилки в файлі, рядку та стовпцю)
- розширення та інтернаціоналізація повідомлень про помилки синтаксису в створених парсерах
- генерація налаштовувальних синтаксичних помилок
- синтаксичні аналізатори
- push-повідомлення з автозавершенням
- підтримка іменованих посилань
- кілька типів звітів (графічні, XML) про згенерований парсер
- підтримка декількох мов програмування

Одним делікатним питанням генератора парсерів LR ж вирішення конфліктів (зміщення, зменшення конфліктів). Вирішення конфліктів зазвичай вимагає аналізу автоматичного аналізатора, як описано у звітах, та певного досвіду від користувача. Анотації допомагають швидко зрозуміти деякі конфлікти і навіть можуть фактично довести, що проблема пролягає в тому, що граматика насправді неоднозначна [10]. Наприклад, щодо граматики, яка страждає від сумнозвісної проблеми, що зависає бізон повідомляє. В термінал виводить значення помилки і рекурсивно показується її місце як зображено на рис. 1.2

```
Example: "if" expr "then" "if" expr "then" stmt • "else" stmt
Shift derivation
  if_stmt
  ↳ "if" expr "then" stmt
                     ↳ if_stmt
                        ↳ "if" expr "then" stmt • "else" stmt
Example: "if" expr "then" "if" expr "then" stmt • "else" stmt
Reduce derivation
  if_stmt
  ↳ "if" expr "then" stmt                      "else" stmt
                     ↳ if_stmt
                        ↳ "if" expr "then" stmt •
```

Рисунок 1.2. Повідомлення про циклічну помилку

Так, наприклад і повідомляється про рекурсивну помилку, яка відображається рекурсивно.

2.2 Генератор парсерів «Yacc»

Yacc – це ще один генератор компіляторів. Дослівно назва перекладається як «Ще один компілятор-компіляторів», розроблена Стівеном Джонсоном. Це генератор синтаксичного аналізу LARA, який генерує синтаксичний аналізатор LALR (частина компілятора, яка намагається внести синтаксичний сенс у вихідний код) на основі формальної граматики, написаної в позначенні, подібному до BNF. Цей генератор поставляється як стандартна утиліта для різних Unix систем [11].

Синтаксичний аналізатор, створений Yacc, є синтаксичним аналізатором LALR з декількома прагматичними розширеннями для вирішення неграмотних граматик та інших проблем, пов’язаних з тим, що граматики іноді помиляються, і Yacc повинен сигналізувати про ці проблеми в таким чином, щоб творець граматики міг її вдосконалити. Прикладами цих розширень є обробка помилок (наприклад, у разі конфліктів). Теоретично, за наявності конфліктів, можна сказати, що парсер не буде скомпільований, але Yacc вирішує проблему, кодуючи поведінку за замовчуванням, коли це потрібно. Можливо, така поведінка потрібна, але, загалом, помилка у використанні граматик, що породжує конфлікти в парсері. Після успішного запуску Yacc створить принаймні функцію $yyparse()$, яка запускає парсер і виробляє синтаксичну оцінку семантичних дій у правилах. Ця функція, коли кодуються відповідні дії, створює синтаксичне дерево як структуру даних (дерево, яке може не настільки здивувати).

2.3 Генератор парсерів «Lex»

Lex – це додаток, який генерує лексичні аналізатори. Lex зазвичай використовується з генератором парсера Yacc. Lex зчитує вхідний потік із зазначенням лексичного аналізатора та виводить вихідний код, що реалізує лексер на мові програмування C [12]. Структура файлу Lex навмисно схожа на структуру файлу Yacc. Файли розділені на три розділи, розділені рядками, які містять лише два відсотки знаків, як показано нижче:

- Розділ визначення визначає макроси та імпортує файли заголовків, написані мовою C. Тут також можна написати будь-який код C, який буде дослівно скопійований у згенерований вихідний файл.
- Розділ правил пов'язує шаблони регулярних виразів із операторами C. Коли лексер побачить текст у вхідних даних, що відповідає заданому шаблону, він виконає відповідний код C.

- Розділ коду С містить оператори С та функції, які дослівно копіюються у згенерований вихідний файл. Ці висловлювання, ймовірно, містять код, що викликається правилами в розділі правил. У великих програмах зручніше розміщувати цей код в окремому файлі, підключеному під час компіляції.

Lex - це інструмент, який дозволяє вирішувати широкий клас проблем, що виникають при обробці текстів, шифруванні, реалізації компіляторів і в інших областях. При текстової обробці Ви можете перевірити правильність написання слів, при шифруванні - переводити певні шаблони символів в інші, при реалізації компіляторів - визначити, які лексеми (мінімальні значущі послідовності символів) зустрічаються в компілюємій програмою. Загальною для всіх цих завдань є необхідність розпізнавати ланцюжка символів, що задовольняють певним критеріям. Якщо Ви розробляєте компілятор, Вам доведеться реалізованих лексичний аналізатор. Звідси і назва - Lex. В принципі, щоб впоратися з перерахованими проблемами, які необов'язково використовувати Lex. Можна просто писати програми на стандартній мові, подібному С. Справді, Lex також породжує С-програми. Справа, однак, в тому, що Lex дозволяє досягти мети про-
ще й швидше. Його слабкість в тому, що він часто породжує С програми, які довше, ніж необхідно для конкретної задачі, і які працюють повільніше, ніж могли б. Але в багатьох додатках це несуттєво, і в цілому переваги використання значно перевершують недоліки. Lex можна також використовувати для збору статистичної інформації про властивості вхідного тексту, таких, як число символів, довжина слів, число входжень слів.

2.4 Машинне навчання

Машинне навчання – це штучний інтелект який має певні класи методів, для якого характерною рисою яких є не пряме рішення задач, а навчання в процесі якого здійснюється прийняття рішень в багатьох задач. Для побудови таких методів використовуються засоби математичної статистики, чисельних методів,

математичного аналізу, методів оптимізації, теорії ймовірностей, теорії графів, різні техніки роботи з даними в цифровій формі. Розрізняють два типи навчання:

- Навчання по прецедентах, або індуктивне навчання, засноване на виявленні емпіричних закономірностей в даних.
- Дедуктивне навчання передбачає формалізацію знань експертів і їх перенесення в комп'ютер у вигляді бази знань.

Дедуктивне навчання прийнято відносити до області експертних систем, тому терміни машинне навчання і навчання по прецедентах можна вважати синонімами. Багато методів індуктивного навчання розроблялися як альтернатива класичним статистичним підходам. Багато методи тісно пов'язані з витяганням інформації інтелектуальним аналізом даних.

Є безліч об'єктів (ситуацій) і безліч можливих відповідей (відгуків, реакцій). Існує деяка залежність між відповідями і об'єктами, але вона невідома. Відома тільки кінцева сукупність прецедентів - пар «об'єкт, відповідь», звана навчальної вибіркою. На основі цих даних потрібно відновити неявну залежність, тобто побудувати алгоритм, здатний для будь-якого можливого вхідного об'єкта видати досить точний класифікують відповідь. Ця залежність не обов'язково виражається аналітично, і тут нейромережі реалізують принцип емпірично формованого рішення. Важливою особливістю при цьому є здатність навченою системи до узагальнення, тобто до адекватного відгуку на дані, що виходять за межі наявної навчальної вибірки. Для вимірювання точності відповідей вводиться оціночний функціонал якості. Дана постановка є узагальненням класичних задач апроксимації функцій. У класичних задачах апроксимації об'єктами є дійсні числа або вектори. У реальних прикладних задачах вхідні дані про об'єкти можуть бути неповними, неточними, нечисловими, різнорідними. Ці особливості призводять до великої різноманітності методів машинного навчання.

Розділ машинного навчання, з одного боку, утворився в результаті поділу науки про нейромережах на методи навчання мереж і види топології їх архітектури, з іншого боку - увібрав в себе методи математичної статистики. Зазначені нижче

способи машинного навчання виходять з нагоди використання нейромереж, хоча існують і інші методи, які використовують поняття навчальної вибірки - наприклад, дискримінантний аналіз, який оперує узагальненої дисперсією і коваріації спостерігається статистики. Базові види нейромереж, такі як перцептрон і багатошаровий перцептрон (а також їх модифікації), можуть навчатися як з учителем, так і без вчителя, з підкріпленням і самоорганізацією. Але деякі нейромережі і більшість статистичних методів можна віднести тільки до одного зі способів навчання. Тому, якщо потрібно класифікувати методи машинного навчання залежно від способу навчання, буде некоректним відносити нейромережі до певного виду, правильніше було б типізувати алгоритми навчання нейронних мереж.

2.4 Проблеми які виникають при розробці ти використанні генераторів та огляд програмного забезпечення

Синтаксичні парсери являють собою інструмент котрим можна користуватися для парсингу даних, які зазвичай представлені у вигляді тексту. Ці дані не є структурованими, тобто частина споріднених сполук може бути розділена. Це і породжує першу проблему синтаксичного аналізу. Якщо текст буде мати в собі різні види сполук, то синтаксичний аналіз не зможе їх розпізнати так як він будується виключно за правилами.

Наступний вид проблеми це зациклювання без виводу помилки. Це дуже часта проблема в такому виду додатків. Правила описуються на контекстно-вільній граматиці де ліва частина не повинна буди порожньою. В правій можу бути все що завгодно. Правила будуються ієрархічним способом, коли перше правило є базове, а останнє правило побудоване на базових правилах. В таких структурах може бути рекурсивне зациклювання. Прикладом може бути звичайне число. Мінімальний символ позначено як D, тоді повне чило можна буде записати так $N = D \mid (N^*)$.
Цей запис пояснюється як N є як базовий символ D або циклічний N котрий може повторюватися, а тоже і ні. В такому випадку можливе закиклювання.

Синтаксичний генератор буде підставляти новий символ і якщо він буде не число, то синтаксичний парсер знову звернеться до цього правила і це буде тривати нескінченну кількість разів поки парсер не зустріне на вході число. Таке правило є некоректним, і генератори не можуть відловити таку помилку, як результат циклічне зациклювання.

Ще одна проблема парсера це погляд на один символ вперед. Так сам парсер може двивтися тількт на $i + 1$, тобто генератор обмежений в своїх дія. До того ж сам парсинг забирає дуже багато часу так як парсеру потрбен час на те щоб дібратися до певного i-того стовбця. Ще більш склаанощів додає рекурсивний спуск.

Генератор парсерів теж має проблеми. Мала база прімітивних методів корі можна викликати. Це в свою чергу робить граматику дуже великою. Зазвичай ще одна проблема знаходить в самому генераторі. Його мова на якій описується правила для вихідного парсеру є вузько направленим. Це не дає змогу використовувати один і той сами генератор на різних напрамках.

Прикладами генераторів є JavaCC. JavaCC - засіб створення класів на мові Java для перевірки та розробки структурованого тексту. У більших системах часто потрібно виконувати збір та аналіз даних, які представляють собою текст, оформлений відповідно до особистих правил (наприклад, у СУБД - це текст SQL-запитів, у компіляторах - вихідні файли). Правила, за якими складається текст для обробки, зазвичай називають граматикою. Програма, яка передає вихідний текст у вид, який простір для програмної обробки, називається парсером. JavaCC забезпечує створення програмного забезпечення, яке розбирає та аналізує вхідний текст відповідно до заданої граматики, тому JavaCC обробляє написане парсером.
Coco/R - програма генерації компіляторів або інтерпретаторів мови. Програма читає файл з атрибутивное граматикою реалізується мови, Описане в формі EBNF і генерує ряд файлів:

- вихідні коди лексичного аналізатора (сканера); працює як детермінований кінцевий автомат.

- вихідні коди синтаксичного аналізатора (парсеру); використовує метод спадного рекурсивного спуску.
- інформаційні файли (лог, таблиця лексем мови).

Використання Coco / R є дуже простим. Створюваний програмою код є швидким і легким для розуміння. Сама програма вже реалізована на багатьох мовах, і генерує вихідні компілятора так само на багатьох мовах. Існують версії Coco / R, що створюють вихідні на Java, C ++, C #, Delphi, Pascal, Ada, Modula-2, Modula-3, Oberon, Component Pascal та іншими мовами. Розробка своєї мови полягає в розробці файлу граматики мови. В граматику мови додається спеціального виду коментарі (..), В яких укладено код для виконання додаткових дій. Як правило, це код для занесення даних в таблиці ідентифікаторів, генерації коду або його інтерпретації.

Ragel - компілятор кінцевих автоматів. Вихідним текстом кінцевого автомата для Ragel служить розширена мова регулярних виразів або діаграма станів кінцевого автомата. Ragel добре підходить для побудови лексичних аналізаторів і специфікації протоколів передачі даних. Ragel дозволяє впроваджувати в будь-якій точці виконання автомата визначені користувачем дії. З метою вирішення недетермінізма передбачена система пріоритетів для операторів регулярного мови. Ragel підтримує візуалізацію генерується автомата за допомогою graphviz.

ANTLR – розшифровується як – ще одиин додаток для розпізнавання мов. Це генератор низхідних аналізаторів для формальних мов. ANTLR перетворює контекстно-вільну граматику у вигляді РБНФ в програму. Конкретнл в цій розмітці записуються код який буде виконуватися. Зазвичай цей код написаний на C++ та мають спеціальні блоки вставки. Якщо код буде дуже великий його розділюють на спеціальні бібліотеки.

Використовується для розробки компіляторів, інтерпретаторів і трансляторів.

Його переваги:

- Безкоштовне програмне забезпечення.
- Використання одного позначення для опису лексичних та синтаксичних аналізаторів.
- Аналіз зверху вниз, а не знизу вгору.
- Зручність роботи з абстрактним деревом синтаксису.
- Надання повідомлень про помилки та відновлення після них.
- Візуальні середовища розробки (ANTLR Works, ANTLR Studio, плагіни для Eclipse та IntellliJ IDEA), що дозволяють створювати та налагоджувати граматики, підтримувати підсвічування синтаксису, автозавершення, візуальне відображення граматик, будувати в реальному часі під час набору тексту, налагоджувач , рефакторинг.

Його синтаксис можна побачити на рис 1.3

```
grammar T;

msg : 'name' ID ';'
    {
        System.out.println("Hello, "+$ID.text+"!");
    } ;

ID: 'a'..'z' + ;
WS: (' ' |'\n' |'\r' )+ {$channel=HIDDEN;} ;
```

Рисунок 1.3 – Граматика ANTLR

JFlex - це генератор лексичних аналізаторів (також відомий як генератор сканерів) для Java, написаний на Java. Генератор лексичного аналізатора бере за вхідні дані специфікацію з набором регулярних виразів та відповідних дій. Він генерує програму (лексер), яка читає вхідні дані, узгоджує вхідні дані із регулярними виразами у файлі специфікації та виконує відповідну дію, якщо регулярний вираз

відповідає. Лексери, як правило, є першим інтерфейсним кроком у компіляторах, відповідності ключових слів, коментарів, операторів тощо та генеруванням потоку вхідних токенів для парсерів. Лексери також можна використовувати для багатьох інших цілей. Лексери JFlex базуються на детермінованих кінцевих автоматах (DFA). Вони швидкі, без дорогого відступу. JFlex призначений для роботи разом з генератором парсерів LALR CUP від Скотта Хадсона та модифікацією Java Berkeley Yacc BYacc Боб Джеймісон. Він також може використовуватися разом з іншими генераторами синтаксичного аналізатора, такими як ANTLR, або як самостійний інструмент.

Іноді помилки бувають в коді, і деякі з них можна виявити за допомогою ANTLR, а саме лексичні і синтаксичні. Помилки смислові, коли код написаний відповідно до граматикою, але не робить потрібного, виявити не можна: фреймворк ж не знає, чого ви хочете.

Лексер і парсер, які ми згенерували, викидають повідомлення про помилку. Нам треба всього лише прикріпити до них слухач, в найпростішому випадку він просто виводить повідомлення в консоль.

Лістенер реалізує інтерфейс ANTLRErrorListener. У ньому чотири методи, але в даний момент нас цікавить лише один - syntaxError.

Не завжди в процесі обробки дерева синтаксичного розбору потрібно робити обчислення. Наприклад, щоб підфарбувати ключові слова, досить лише отримати їх координати. Для цього скористаємося лістенером. Робота цього класу дещо нагадує обхід XML за допомогою SAX: відповідний метод класу спрацьовує при вході і при виході з правила.

3. ПРАКТИЧНЕ ВДОСКОНАЛЕННЯ ГЕНЕРАТОРА ПАРСЕРІВ НА ОСНОВІ ЗАСТОСУВАННЯ АЛГОРИТМІВ ПОШУКУ ТА МАШИННОГО НАВЧАННЯ

3.1 Архітектура генератора парсерів контекстно-вільної граматики

Генератор парсерів контекстно-вільної граматики –це додаток котрий за допомогою описаних раніше файлів може побудувати парсер. Тобто результатом його роботи буде генератор. Правила задаються у спецаільній граматиці. Ця граматика задається самим генератором. Вона складається х простих функцій. Наприклад наступний символ буде літера або нижнє підкреслення. Якщо це буде записано на спеціальній грамаиці, то буде виглядати так $Ide = L \mid "_"$, тобто $Ide =$ це буде фінальний індинтифікатор якому буде присвоїна функція. Ця функція буде дивитися на вхідний символ, якщо це буде літера, то умова буде виконана, і рузльтат цієї операції буде позитивний. Якщо ж ні то на розгляд буде подано наступний символ “_”. Цей символ буде одначати що хочемо перевірити конкретно той символ котрий вписал в лапки. Якщо умова буде виконана, то рузультат цієї функції буде позитивним. Генератор візьмен на перевірку наступне правило. Якщо ж результат буде негативний, то генератор відхилить це правило.

Архітекту генератора парсерів контекстно-вільної граматики можна зобразити як на рис. 1.4

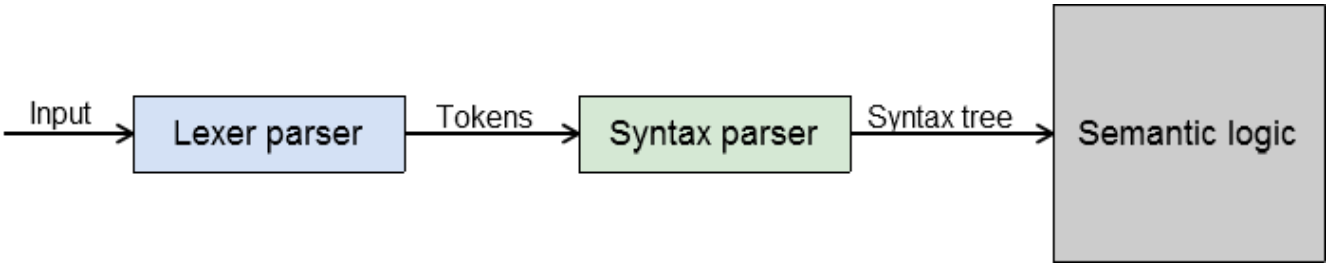


Рисунок 1.4. Архітектура генератора парсерів котекстно-вільної граматики

Архітектура генератора не досить складна. Першим етапом буде зчитування установлених правил, їх розбір. Всі правила будуть складатися в динамічний стек.

Якщо всі правила виявилися коректними, то генератор зчитує вхідний текст. Зчитуваний вхідний текст переводиться в токени. Тобто для подальшої роботи потрібні токени. В даному випадку токен – це базовий символ. Далі в дію йдуть правила. Цей модуль зображено на на рис. 1.4 світло зеленим кольором. Він називається синтаксичний парсер. Він проходиться по кожній літері та зчитує дані. Далі за рекурсивно спускається за правилами. Якщо правило вірне він створює певну структуру. Якщо ж правило не виконалося, то структура не буде виконана. Коли вся структура буде згенерована, то на виході повернеться дерево. В цьому дереві можна буде знайти всі дані аналізу. Потім це дерево передається до парсера. Парсер методом рекурсивного спуску починає спускатися донизу та будувати новий генератор котрий і був описаний в правилах. Якщо генератор досить розписаний він може спускатися по дереву до декількох разів. Він це може зробити для того щоб передбачити ті конструкцій котрі можна спростити.

3.2 Практичне моделювання генератора конекстно-вільної граматики

3.2.1 Моделювання граматики

Моделювання генератора контекстно-вільної граматики відбувається в декілька етапів. Перший етап це - створення граматки. Граматика повинна бути створена для того щоб кінцевий користувач міг розробляти на ній правила.

Таблиця 1.1. Позачення спеціальних символів у граматиці

Позначення	Опис
=	Символ присвоєння
|	Знак або
?	Знак, що означає, можливе повторення попереднього виразу 0 або 1 раз
&	Знак, що означає, що повинен виконатися попередній вираз і наступний обов’язково

Таблиця 1.1 Позначення спеціальних символів у граматиці

Позначення	Опис
!	Знак який зміню значення функції на протилежний
[]	Знак котрий показує, що вираз можу повторюватися 0 або більше разів
{}	Знак котрий показує, що вираз може виконатися 1 або більше разів
""	Між цими знаками можна писати будь-який символ, він не буде прийнятий як функція
()	Дужки, в них дозволяється об'єднувати вирази

Ці всі значення котрі позначені в Таблиці 1.1 будуть використовуватися для створення правил. При їх комбінуванні можна досягти великих результатів. Базові правила будуть створені як методи. Наприклад $letter$ – одна з базових функцій котру можна викликати. Вона буде дивитися чи є наступний символ літерую. Всі базові методи розміщені в таблиці 1.2

Таблиця 1.2 Базові метод граматики для вхідного генератора

Назва функції	Опис
letter	Перевіряє чи є наступний знак літерою, якщо так, то повертає позитивне значення
digit	Перевіряє чи є наступний знак числом, якщо так, то повертає позитивне значення
empty	Перевіряє чи є наступний знак пробілом, переходом на новий рядок, якщо так, то повертає позитивне значення

Таблиця 1.2 Базові методи граматики для вхідного генератора

token	Перевіряє наступний знак на те щоб він не належав до попередньої функцій (таких як letter, digit, empty), тобто таким чином фактично він перевіряє чи є знак токеном, якщо так, то повертає позитивне значення, а якщо ні то негативне
ident	Це комплексна функція яка перевіряє на те, що перший вхідний символ є нижнє підкреслення або літера. Наступними можуть бути числа, літери, нижнє підкреслення, але не елементи які входять до таких функцій як empty, token.

Всі ці методи є базовими. За допомогою їх потрібно оперувати коли складаються базові правила або правила нижнього рівня.

Для початку треба скласти прості правила для генератора. Перше правило буде починатися так:

$$Primary = ident\ "["\{\ Expression\ \}\ "]"\ |\ letter|\ "("\ Expression\ ")"\ |\ \text{Type}"("Expression")"$$

Базова функція котра має назву Primary, це тому що вона буде мінімально. В ній описується, що перша вираз можу бути слово далі, а може бути і масивом або це може бути простий знак або це може бути рекурсивний вираз або привід то якогось іншого типу.

Після цього типу буде наступна операція. Ця операція називається одинарна операція. Правило для неї записується так:

$$UnaryOp = \ -|"!"$$

Ця операцію означає, що лексичний аналізатор перевірить чи наступний знак буде мінус чи знак оклику.

Наступне правило буде називатися фактор. Його реалізація така:

$$Factor = [UnaryOp]\ Primary$$

Його реалізація дуже проста. Перший вхідний вираз буде одинарна операція. Її може і не бути. Другий вираз є базовий приклад котрий був представлений вище в цьому документі.

Третє правило – це математична операція яка в себе включає під операції таких як множення, ділення, та останок від ділення. Ці всі операція прораховуються на одному рівні та будуть розпарсені в тому порядку в якому вони йдуть. Записується вона так:

$$MulOp = * \mid / \mid \%$$

Четверте правило записується так $Term = Factor\ \{MulOp\ Factor\}$. Воно є контекстним. В ньому використовується повтор.

П’яте правило відображає просте математичне відношення. Записується воно так:

$$AddOp = \ +|-$$

В цьому правилі є цікавий момент так як в ньому використовується такий самий знак як і в правилі $UnaryOp$. Але воно відрізняється своєю дією так як виконує математичну функцію віднімання.

Шосте правило також є контекстним. Воно має назву $Addition$ та записується так:

$$Addition = Term\ \{\ AddOp\ Term\}$$

Сьоме правило застосовується для визначення для порівняння. Воно записується так:

$$RelOp =< | \leq | > | \geq$$

Його реалізація дуже проста. Вона визначає, що на одному рівні можуть знаки порівняння і всі вони є одного рівня і будуть визначатися по порядку.

Восьме правило визначається як контекстне так записується так:

$$Relation = Addition\ [\ RelOp\ Addition]$$

Воно показує тільки те, що правило Addition може бути рекурсивно викликаним або ні.

Наступне правило визначає прирівнювання. Значення його так, що знак дорівняю та не дорівнює можуть використовуватися на одному рівні. Записується воно так:

$$EquOp = "=="|"! = "$$

Десяте та одинадцяте правила також є контекстними так записується так:

$$Equality\ = Relation\ [EquOp\ Relation]$$

$$Conjunction = Equality\ \{\ "\&\&"\ Equality\}$$

Одинадцяте правило виконує функцію порівняння, фактично називається «та». Наступне правило, дванадцяте, є схожим для попереднього. Воно виконує функцію «або». Записується воно так:

$$Expression = Conjunction\ \{\ ||\ Conjunction\ \}$$

Цей вираз є фінальним, тому і називається як Expression. Він може використовуватися як код котрий визивається.

Всі ці виразу утворюють загальну систему правил. Вони будуть задані в конфігуратор. Далі буде проведений аналіз цієї граматики. Чи нема в неї циклічного коду, котрий породжує зациклення. Якщо всі перевірки будуть пройдені, то ці правила будуть передані далі по конвеєру.

3.2.2 Моделювання парсера вхідної граматики

Моделювання граматики відбувається на мові програмування C#. Для цього використовується окрема бібліотека під назвою «GCFCG.Core». Це зроблено для того щоб в майбутньому цю бібліотеку можна було використовувати в різних проектах. Наприклад для візуалізації цього процесу або для інших розробників. Головний файл де буде тривати розробка називається «RuleGrammaParser». Для кожного виразу було створено клас під назвою Expression. Базово він вказує на свого батька, це було зроблено щоб швидко пробігатися по структурі даних та відшукувати певні дані. Наступний базовий параметр це з якої відстані починається вираз. Він наступні клас спадкуються (повинні спадкуватися) від цього базового класу. Для парсингу правил створюється клас «RuleGrammaParser». В нього є метод Parse. На вхід він приймає текст з правилами. Далі вони починають проходити процес парсингу. Структура парсингу правил складається з різних методів.

Метод GetLetter – перевіряє чи є вхідний символ літерою, якщо так то позиційний лічильник збільшується на один, а сам літера повертається як LetterExpression.

Схожий метод є і для цифр. На вхід він бере літеру та перевіряє чи є вона числом. Якщо так, то позиційний лічильник збільшується, а метод повертає DigitExpression.

Метод котрий перевіряє на всі значення котрі не є числами або літерами, мають значення токен. Записуються вони в клас TokenExpression. Всі інші знаки (знаки пропуску) будуть записуватися в клас EmptyExpression. Але зазвичай вони будуть ігноруватися та пропускатися аналізатором так як не несуть в собі ніякої важливої інформації. Система позиційного лічильника без цієї інформації буде працювати коректно.

Наступний метод є більш збірним. В ньому застосовуються декілька правил. GetIdentifier – метод котрий має два рівні перевірки символів. Перший рівень перевіряє чи вхідний символ нижнім підкресленням або чи є він літерою. Якщо так, то далі йде наступний, другий рівень перевірки. В ньому перевіряється чи є вхідний

символи нижнім підкресленням, літерою чи цифрою. Якщо всі перевірки були пройдені вірно вихідним об’єктом буде IdentifierExpression клас.

Наступний клас є більш зібраним. Він викликає в собі різні методи. Він має назву GetPrimary. Перша його перевірка буде на індифікатор. Якщо він є та перевірка пройдено правильно, то буде повернуто саме цей клас, але якщо ні, то наступна перевірка буде дивитися чи вхідна літера є відкриття дужок. Якщо так, далі буде здійснена перевірка на пошук всіх виразів. Цей вираз виконується циклічно. Цей вираз повинен бути закритий дужкою. Якщо цього не буде виконано аналізатор поверне помилку про те що попередній вираз повинен закриватися дужками.

Наступний вираз відноситься до правил повторень. Перше правило називається CanBeExpression. Воно значить те, що всі вирази які були записані в середині цього виразу можуть бути, але також можуть бути відсутніми. Тобто вираз який знаходить в середині не є постійним.

Схожим для перевірки є метод GetRepeatExpression. З його назви стає зрозумілим, що цей метод перевіряє на те щоб всі його внутрішні вирази могли повторюватися 1 або більше разів. Заключний метод називається GetBodyExpression. Він також розділений на декілька етапів. 1й етап перевірки це повернення виразу який може бути 0 або 1 раз. Якщо його нема, то наступною перевіркою буде повернення виразу який може повторюватися 1 або більше разів. Якщо і ця перевірка буде провалена, то далі буде відшукуватися простий вираз. Якщо простий вираз не буде знайдений, то це значить, що лексичний аналізатор зчитав всі правила та завершив свою роботу.

Після того як правила будуть згенеровані, вони будуть передані далі по конвеєру. Якщо всі правила згенерувалися правильно, то вони будуть розміщатися ієрархічно. Тобто правило буде мати батька, а також вирази які знаходяться в середині цього правила.

Наступним пунктом буде побудова синтаксичного дерева. Воно будується за правилам які були подані на вхід. Після цього йде метод який будує синтаксичне дерево. Воно має певні властивості. Кожен блок цього деревам має певну

структуру. Обов'язковим властивостями є посилання на батька цього блока, а також посилання на його дітей. Приклад синтаксичного дерева зображено на рисунку 1.5

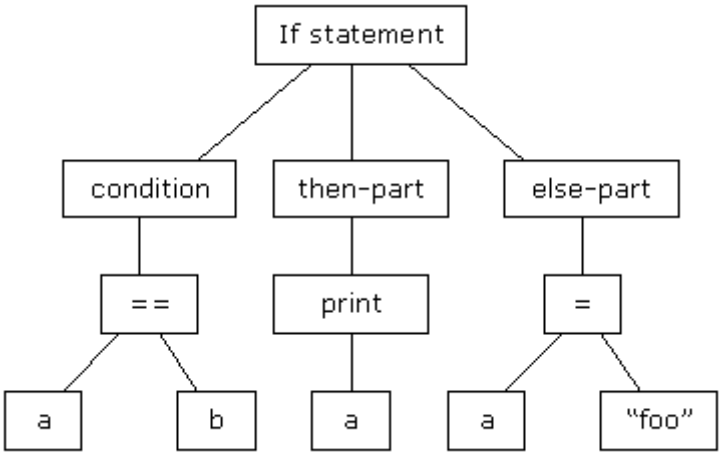


Рисунок 1.5. Приклад синтаксичного дерева

На рисунку 1.5 зображений приклад синтаксичного дерева. Цей приклад був зіставлений з коду: `if (a == b) print(a); else a = "foo";`

Його значення це умова якщо a буде рівнятися b, то треба вивести a. А якщо ж ні, то присвоїти змінній a дорівнює рядкове значення foo. Це простий приклад для прикладу створення дерева. Якщо код буде більший, то і синтаксичне дерево буде більше.

Пояснення дерева дуже просте. Всі базові блоки або вирази знаходяться знизу. Вони називаються базовими так як містять в собі базові типи такі як цифри, літери або токени (ті що являються ні цифрами, а ні літерами та не пропуски). В даному випадки це є літери та константне значення "foo". Вони всі є спадкоємцями своїх предків. Зазвичай ці вирази мають назви але для простоти показу вони показаними тими значення які і були зчитані для цього. Всі ці значення заключають вираз «Якщо». Цей вираз є самий головним. Якщо код буде більший, то цей вираз буде заключений в самий верхній вираз. Зазвичай головний виразом називається «EntryPoint», що перекладається як головна точка або головний вхід. Він в собі збирає всі під типи. Кожен під тип має посилання на свого батька.

3.2.3 Практичне моделювання генератор вихідних парсерів

Генератор вихідних парсерів працює з синтаксичним деревом, який може бути зроблений за допомогою синтаксичного аналізатора. Він має бути підлаштований під певну структуру.

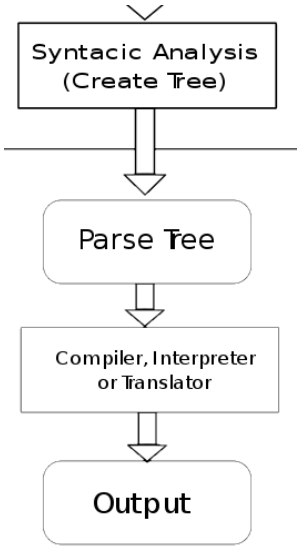


Рисунок 1.6. Алгоритм роботи генератора вихідного парсера

На рис. 1.6 зображений алгоритм роботи генератора вихідного парсера. Він починається з того, на вхід подається дерево, котре перед цим було прочитане та сформоване в певну структуру. Для це дерево відправляється на парсер дерев. Його головною задачею пройти всі елементи. Генератор дерева будує вихідний компілятор по заданим правилам. Розглянемо побудову лексичного аналізатора на практиці. Створимо класс з назвою GrammaExpresionBuilder. Його параметром буде простий об’єкт, він буде потрібний для використання вбудованих функцій. Також класс має проперті під назвою «Index» тип якого є int (цілочисельний тип). Сутність цього індексу, є те що він буде лічильником для вирзів.

Сам синтаксичний аналізато має 2 рівні як це зображено на рисунку 1.7. Перший рівне є базовим. Саме на ньому аналізатор працює з базовими символами. До нього входять такі методи як: GetDigitExpression, GetLetterExpression, GetTokenExpression, GetSpaceExpression. Всі ці методи працють напряму з текстом.

Другий рівень, або абстрактний рівень, є груповий, тобто він побудований на першому рівні і вже не працює на пряму з текстом, а тільки групує перший рівень. До нього входять так методи: GetIdentifireExpression, GetNumberExpression, GetPrimaryExpression, GetExpression, GetInitExpression, GetStepExpression, GetStringExpression, GetRepeatExpression, GetOrConditionExpression, GetSubExpression, GetPossibleExpression. Всі ці методи не працюють на пряму з текстом, а тільки маніпулюють методами з 1 рівня.

Рисунок 1.7. Візуальне зображення рівнів синатаксичного аналізатора

З рисунка 1.7, бачемо що 2 рівень завжи звертається до 1 рівня. Базовий метод які використовується всі методи, маж назву GetNextChar. Він повертає символ позиція якого буде залежати від проперті Index. Він не збільшує цю властивість. Так як може повернути не точно потрібний символ. Наступний метод який змінює властивість Index, а також створює структуру має назву GetLetterExpression. Цей

метод на початку перевірє чи є вхідний символ літерою. Якщо це так, то змінює лічильник на 1, потім циклічно починає зчитувати наступший символ, до ти пір поки не буде отриманий символ який буде відмінний від літери, наприклад це буде цифра або знак (позначення). В результаті метод поверне об’єкт або нічого, що в свою чергу повідомить аналізатору про не можливість розпарсити вхідний текст за таким правилом. На рівні з цим методом стоїть метод GetDigitExpression. Він повертає число. Працює він по аналогічному алгоритму зхожим з правилом для пошуку літери. В таком самому ключі працює і GetTokenExpression. Але вім перед самим токеном та після нього може зчитувати пробіли що не буде помилкою. Якщо ж пробіли будуть в середині це буде вже означати, що це вже інший вираз. GetSpaceEpxression зчитує всі знаки-пропуски крім перенаправлення на нова лінію. Так як цей знак входить до іншого ряду груп. Це був 1 рівень, котрий безпосереднь працював з текстом. На 2 рівні робота з текстом зустрічається рідко. Один із групових методів GetIdentifierExpression. Він маніпулює методм GetLetterExpression , а також GetDigitExpression. Перший вхідний символ має бути літера. Якщо це так, то правило буде працювати до ти пір поки не буде прорушену одну із умов, тобто вхідний символ буде відмінний від літери, цифри або знаку «_». Поряд з цим методом працює метод GetNumberExpression, воно є зхожим до правила GetIndetifierExpression. Але вхідний символ повинен бути цифра або знак «.», до того ж знак крапка може бути використаний тільки 1 раз. GetPrimaryExpression є одним із головних правил ві уміщує в собі відразу декілька правил таких як: GetIdentifireExpression, GetNumberExpression, GetStringExpression, GetPossibleExpression, GetRepeatExpression, GetSubExpression. Він проходить ці правила по порядку. Якщо одне з цих правил було виконане, то відповідно повертає його. Якщо ні, то повертає пустий об’єкт. Правило GetSubExpression є досить простив і воно виконує роль дужок як і в математиці. Тобто вираз котрий буде в дужках буде виконаний першим. Наступне правило GetRepeatExpression. Це правило на початку перевіряє чи є вхидний символ «{» математичною дужкою і тільки в таком випадку правило буде виконано. Наступне правило має назву GetPossibleExpression. Воно має аналогію з попереднім правилом і на вхід очікую

симовл «[» відкриття. Якщо це правило не виконається, то буде повернутий пустий об’єкт. Наступним правилом буде GetStepExpression. Це правило робить пропуски між виразами. Тобто якийсь вираз і пропуск, і так необмежена кількість разів. Далі буде йти правило умови. Воно має назву GetOrConditionExpression. Перший його крок буде знайти праву частину виразу якщо вона є, то 2 умовою буде щоб наступний символ був «|». Якщо ця умова виконається, то буде пошук виразу ще раз з самого його початку, тобто буде пошук лівої частини. Якщо відповідний знак не буде знайдений, то буде повернута перша частина пошуку. Також є пусте правило з ім’я GetExpression, воно створенно тільки для майбутнього розширення. Головним метод має назву Build. На вхід він приймає текст, який і буде в подальшому парситися на вирази. Сутність його циклічна. Він викликає один метод, який має назву «GetInitExpression», та після нього очікує на знак перенесення рядку. Це головний мотод який заключає в себе сутність інших, більш менших виразів. Після цього методу йде вираз який називається GetInitExpression.

3.3 Генерація синтаксних правил на основі машинного начання

Одним із ключових моментів є генерація правил на основі входного тексту. Такий алгоритм повинен працювати в один кінець. Для почасту створюється база даних, тобто проводиться навчання. Далі на вхід подається довільний текст, на основі якого будуть генеруватися правила. Побудова правил здіснюється по вхідних символах. Тобто беруться вхідні слова і до них шукається відповідність. Якщо відповідность не буде, то буде підібране відповідне правило. Ім’я для створюваних правил буде вибиратися випадковим чином. Правила будуються в 2 етапи. Перший етап це побудова самої схеми синтаксичних правил, другий етап це перевірка правил, а також створення відповідної бази даних. Генерація вхідниз символів буде повторюватися циклічно.

3.4 Оцінка стану вихідного генератора

Після того як вихідний генератор скомпілював новий парсер його можна використати в роботі. Цей парсер має туж саму граматику котру і було представлено в описі граматики. Він повинен мати ідентичну структуру до опису його правилами. Згенерований код може бути представлений у вигляді як структурного дерева та у вигляді коду для будь-якої мови програмування. На практиці розглянемо його роботу. Напишем вхідний текст. **Primary = Identifier ["["Expression"]"] |Literal**. Він буде опрацьований синтаксичним аналізатором.

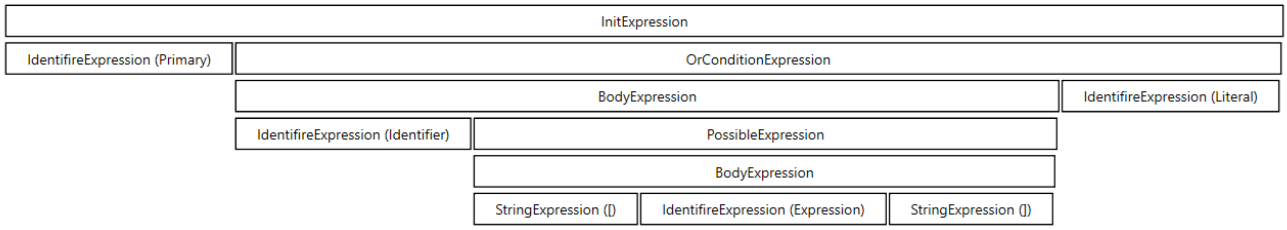


Рисунок 1.8. Дерево згенероване синтаксичним аналізатором

На цьому базовому дереві, рисунок 1.8, видно, що всі правила розмістилися вірно, до того ж нема повторень, а тому час на аналіз менший.

Верхній вираз це ініціалізація виразу, тобто один рядок який і був написай в розмітці далі йде ідентафікатор виразу, який називається Primary. Далі його голивний вираз називається умовний вираз, так як його може і не бути.

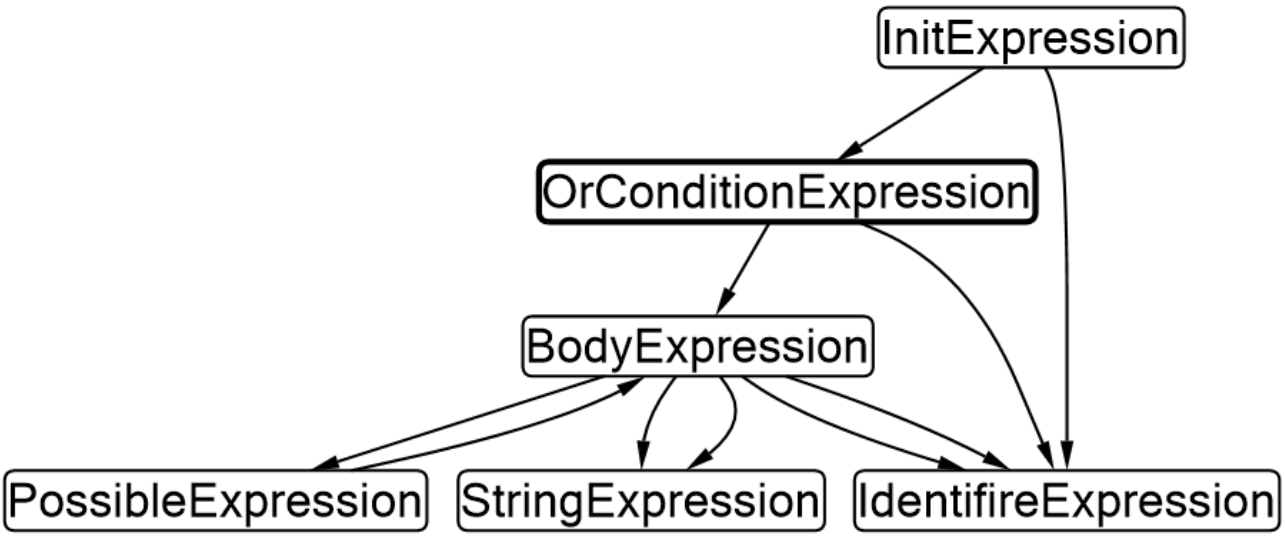


Рисунок 1.9. Граф, згенероване синтаксичним аналізатором

Далі його вирази спускається до низу та залучає у роботу базові вирази, тобто вирази 1 рівня.

На прикладі, рисунок 1.9, цього графі можна побачити що майже всі правила взаємодію між собою, тільки головний InitExpression не підлягає цьому правилу. IdentifireExpression не намагається знову, рекурсивно, зайти на початок аналізу, це додає аналізатору швидкості.

ВИСНОВКИ

В даній дипломній роботі було створено новий принцип парсингу який в свою чергу дозволить оптимізувати пошук у великому обсязі тексту, а також додатко який можна використовувати як допоміжний інструмент для розробки власних правил пошуку.

Також було перегляну то та проведено порівняння інформацію про генератори та різнів реалізації методів парсингу. Перевірено на практиці деякі види парсерів, оцінено їх швидкодію.

Проаналізовано та застосовані відомі способи оптимізації даних алгоритмів.

Розроблено метод застовування машинного навчання в системі парсингу.

Проведено загальний аналіз та зроблена характеристика предметної області.

Розроблені вимоги до генератора контекстно-вільної граматики

Проведено аналіз та порівняння існуючих генераторів контекстно-вільної граматики.

Зроблений обґрунтований вибір засобів розробки і проектвання проекта.

Була спроектована база даних.

Реалізована інформаційна система.

Була підвищенна швидкодія генератора КВ граматики, а також удосконалено генератор за допомогою механізма передбачень наступних правил.

Зроблена оптимізація алгортмів пошуку в контекстно-вільній граматиці.

ПЕРЕЛІК ПОСИЛАНЬ

1. І. Б. Швороб, Порівняльний аналіз метод синтаксичного розбору текстів. /- / - Львів: Львівська політехніка, 2015. – 197–202 с.

2. Aho, A.V., Sethi, R. and Ullman, J.D. Compilers: principles, techniques, and tools (Компілятори: принципи, прийоми, засоби). / - USA. Boston. 1986.

3. Frost, R. Hafiz, R and Callaghan, P. Modular and Efficient Top-Down Parsing for Ambiguous Left-Recursive Grammars (Модульний та ефективний синтаксичний аналіз зверху вниз для неоднозначної ліво-рекурсивної граматики) / - Prague. International Workshop on Parsing Technologies, 2007. – 109 – 120 с.

4. Frost, R. Hafiz, R. and Callaghan, P. Parser Combinators for Ambiguous Left-Recursive Grammars (Комбінатори синтаксичного аналізу для неоднозначних ліворекурсивних граматик). San Francisco: PADL, 2008. – 167 – 181с.

5. http://dinosaur.compilertools.net/yacc/ [Електронний ресурс] : [Інтернет-портал]. – Електронні дані. – Режим доступа: http://dinosaur.compilertools.net/yacc/ (дата звернення 25.03.2021). - Yacc: Yet Another Compiler-Compiler.

6. Nicola Howarth. Abstract Syntax Tree Design (Дизайн абстракнтого синтаксичного дерева). /- / United Kingdom, Cambridge, 1995. 3 – 17с.

7. Dick Grune, Kees van Reeuwijk. Modern Compiler Design Second Edition (Структура сучасних компіляторів 2 збірка). Dutch, 2012.

8. Peter Sestoft, Ken Friis Larsen. Grammars and parsing with C# 2.0 (Граматика і парсинг з C# 2.0). Denmark: IT University of Copenhagen, 2006. 24 – 26 с.

9. Levine, John R. Flex & Bison (Флекс та Бізон). USA, 2009. 304 с.

10. Hierarchical data in RDBMSs [Електроний ресурс] : [Інтернет-портал]. – Електронні дані. – Режим доступу: http://troels.arvin.dk/db/rdbms/links.

ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ/НАВЧАЛЬНО НАУКОВИЙ ІНСТИТУТ ЗАОЧНОЇ ФОРМИ НАВЧАННЯ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Розробка удосконаленого генератора парсерів контекстно-вільної граматики за допомогою алгоритмів пошуку та машинного навчання

виконав студент Групи ППЗМ-71

Іваницький Денис Сергійович

Науковий керівник: к.т.н., доцент, Негоденко Олена Василівна

Київ 2021

Актуальність

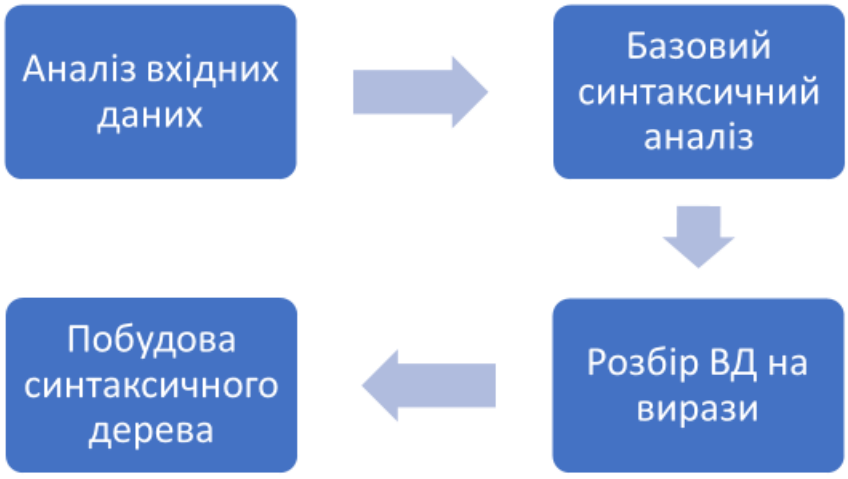
Пошук, збір та обробка інформації мають широкий спектр застосування в різних областях, починаючи від збору даних на різних сайтів, оптимізаціях пошукови систем, розпізнавання сутності мовлення, закінчуючи побудовою інтерпритаторів та компіляторів в мовах програмування.

Мета підвищення швидкодії генератора КВ граматики, удосконалення генератора за допомогою машинного навчання та оптимізації алгоритмів пошуку.

Об'єктом дослідження є процес збору великого об'єму інформації.

Предмет дослідження є алгоритм генерації парсерів контекстно-вільної граматики.

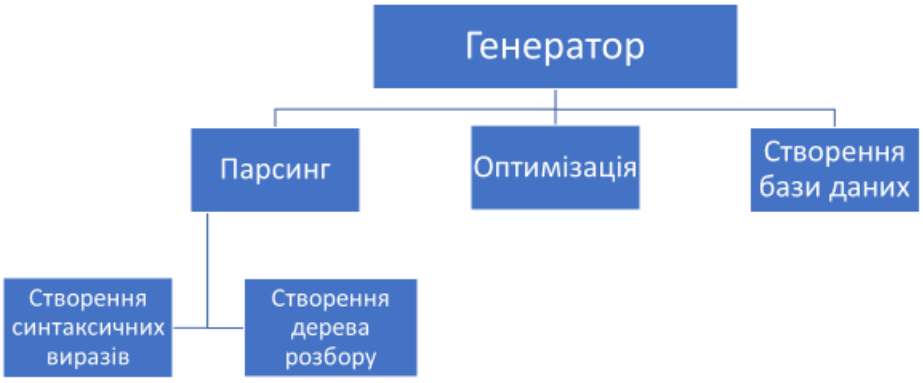
Метод роботи генератора

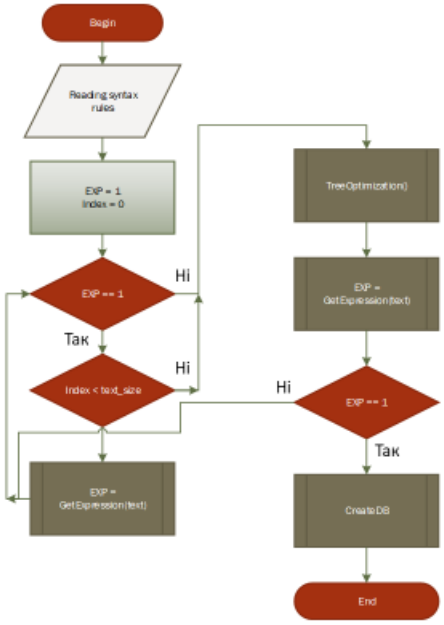


Недоліки попереднього метод

1. Відімінність вхідних даних
2. Не оптимізований процес синтаксичного аналізатора
3. Перебудова синтаксичного дерева

Удосконелений генератор





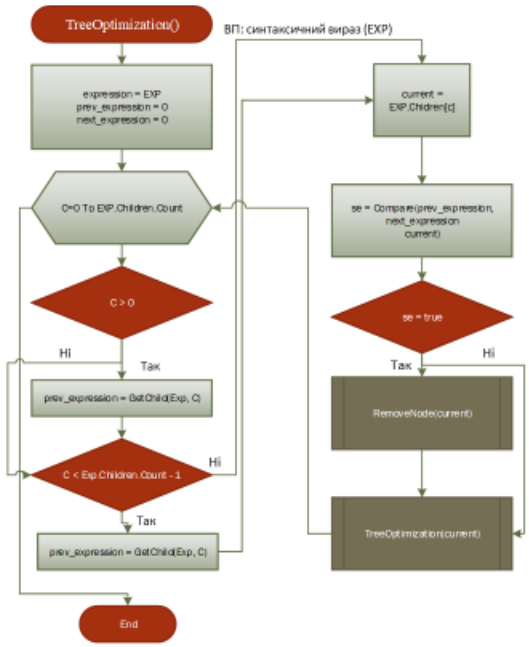
Алгоритм роботи створеного ГК

Де:

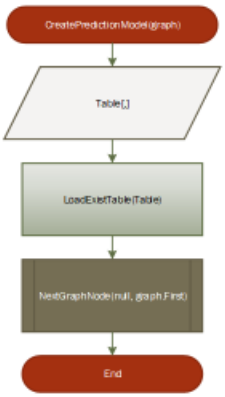
Syntax Rules – вхідні граматичні правила

EXP – базови синтаксичний вираз

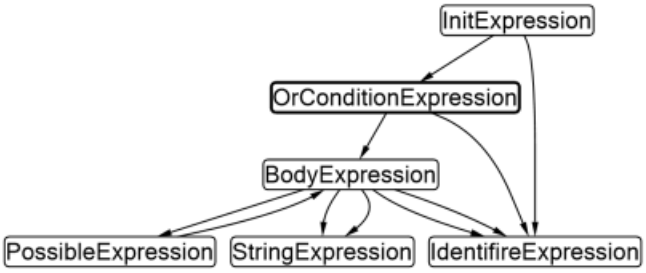
Index – початок зчитуємого значення



Алгоритм роботи оптимізатора



Створення моделі передбачень



Граф, згенероване синтаксичним аналізатором

Порівняння роботи алгоритмів

	Програма генератор Bison	Програма генератор Yacc	Удосконалений генератор	Удосконалений генератор (оптимізатор, модель передбачення)
Час виконання (базовий правило)	10 мс	11 мс	50 мс	5 мс
Переваги	Швидка генерація без додаткових затрат	Більш легка граматика правил	Генерує дерево розбору, створює модель передбачення	Генерує дерево розбору, створює модель передбачення, малий час генерації даних
Недоліки	Не генерує дерево розбору, немає передбачення	Не генерує дерево розбору, немає передбачення	Довгий час на генерацію вихідних даних	Час роботи залежить від моделі передбачення

Висновок

Для подальшого розвитку необхідно удосконалити алгорим роботи програми, та змінити наступне:

- Змінити в моделі передбачення таблицю на ланцюговий механізм
- Розширити алгоритм за допомогою віртуальних правил
- Поєднати механізм передбачення зі створенням дерева

Дякую за увагу!