БАКАЛАВРСЬКА РОБОТА

БР.КІ-39.00.00.000 ПЗ

Група КІ-21-2

Проців Василь

2025

Міністерство освіти і науки України
Івано-Франківський національний технічний університет нафти і газу
факультет інформаційних технологій
Кафедра комп'ютерних систем і мереж

Проців Василь Іванович

УДК 004.45

БАКАЛАВРСЬКА РОБОТА

Розробка транслятора для формування коду на Python та C++ на основі природних мовних конструкцій

Комп'ютерна інженерія
(назва освітньої програми)

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

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

Здобувач освітнього ступеня *Проців В.І.*
(підпис, ініціали та прізвище здобувача)

Науковий керівник *Кропивницький Д.Р., доцент*
(підпис, прізвище, ім'я, по батькові, науковий ступінь, вчене звання керівника)

Допущено до захисту

Завідувач кафедри

д.т.н., професор /С. І. Мельничук/

(посада) (підпис) (дата) (ініціали та прізвище)

Івано-Франківськ – 2025 рік

Івано-Франківський національний технічний університет нафти і газу

Інститут *Інформаційних технологій*

Кафедра *Комп'ютерних систем і мереж*

Освітньо-кваліфікаційний рівень *бакалавр*

Спеціальність *123 – Комп'ютерна інженерія*

ЗАТВЕРДЖУЮ:

Зав. кафедрою КСМ

д.т.н. С.І. Мельничук

« » травня 2025 року

З А В Д А Н Н Я

НА ДИПЛОМНУ РОБОТУ СТУДЕНТУ

Процівy Василю Івановичу

(прізвище, ім'я, по батькові)

1. Тема проекту (роботи) *Розробка транслятора для формування коду на Python та C++ на основі природних мовних конструкцій.* Керівник проекту (роботи) *Кропивницький Дмитро Романович, доцент* .

затверджені наказом вищого навчального закладу від *05.05.2025 № 275\7*

2. Строк подання студентом проекту (роботи) *12 червня 2025р.* .

3. Вихідні дані до роботи *Методичні вказівки, технічна література* .

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) *1. Аналіз предметної області. 2. Проєктування архітектури програмної системи. 3. Реалізація програмного забезпечення.*

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

6. Консультанти розділів роботи

7. Дата видачі завдання *29 січня 2025 р.* .

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту (роботи)	Термін виконання етапів проекту (роботи)	Примітка
1	*Збір інформації, вивчення літератури та пошук додаткової інформації*	*Лютий 2025р*	
2	*Аналіз предметної області*	*Березень 2025р*	
3	*Проєктування архітектури програмної системи*	*Квітень 2025р*	
4	*Реалізація програмного забезпечення*	*Травень 2025р*	
5	*Оформлення додатків, дипломної роботи*	*Червень 2025р*	

Студент _____ *Процiв В.І.*

Керівник роботи _____ *Кропивницький Д.Р.*

АНОТАЦІЯ

У дипломній роботі розроблено веб-додаток, що виконує трансляцію інструкцій, поданих природною мовою українською, у програмний код мовами Python та C++. Метою дослідження є створення доступного інструменту для користувачів без глибоких знань синтаксису мов програмування, який дозволяє формулювати логіку програми у зрозумілому людині вигляді.

Розглянуто теоретичні основи трансляції природної мови, проведено аналіз існуючих аналогічних систем, визначено основні проблеми та обґрунтовано актуальність поставленої задачі. Система реалізована у вигляді клієнт-серверного веб-додатку, де клієнтська частина відповідає за зручний інтерфейс користувача, а серверна — за обробку введеного тексту, лексичний і синтаксичний аналіз, побудову дерева розбору та генерацію цільового коду.

У межах роботи реалізовано механізм збереження та відкриття файлів, перегляд таблиці лексем та синтаксичного дерева, а також можливість вибору мови генерації. Отримані результати підтверджують працездатність системи та її практичну цінність як навчального інструменту та допоміжного засобу для новачків у програмуванні.

Ключові слова: транслятор, природна мова, програмний код, лексичний аналіз, синтаксичний аналіз, генерація коду, веб-додаток, Python, C++, інтерфейс користувача.

ANNOTATION

This thesis presents the development of a web application that translates instructions written in natural Ukrainian language into program code in Python and C++. The aim of the research is to create an accessible tool for users without deep knowledge of programming syntax, enabling them to formulate program logic in a human-readable way.

The theoretical foundations of natural language translation are examined, and existing systems are analyzed to identify key challenges and justify the relevance of the proposed task. The system is implemented as a client-server web application, where the client side provides a user-friendly interface and the server side handles text processing, lexical and syntactic analysis, syntax tree construction, and target code generation.

The application includes features for saving and loading files, viewing the lexeme table and syntax tree, and selecting the output programming language. The results confirm the system’s functionality and its practical value as an educational tool and an assistant for beginners in programming.

Keywords: translator, natural language, program code, lexical analysis, syntactic analysis, code generation, web application, Python, C++, user interface.

ЗМІСТ

ПЕРЕЛІК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

ДОДАТКИ

БІБЛІОГРАФІЧНА ДОВІДКА

ВСТУП

У сучасному світі розробка програмного забезпечення відіграє ключову роль у багатьох галузях: від освіти і науки до бізнесу та виробництва. Водночас створення програм вимагає знання синтаксису і семантики мов програмування, що є суттєвою перепоною для новачків. Це спонукає до пошуку нових підходів, які спрощують взаємодію користувача з комп'ютерними системами, зокрема шляхом використання природної мови для формування програмного коду. У цьому контексті розробка транслятора, що перетворює природномовні конструкції на програмний код, є надзвичайно актуальним напрямом.

Актуальність обраної теми зумовлена зростаючим попитом на інструменти, які роблять програмування доступнішим для користувачів без глибоких технічних знань. Це особливо важливо для інтеграції програмування в освітні процеси, швидкого прототипування, автоматизації побутових задач. Перетворення природної мови в програмний код відкриває нові можливості у сфері взаємодії людини з комп'ютером і є частиною більш широкої парадигми — Natural Language Programming (NLPg).

У наукових і технічних джерелах розглядаються різні аспекти трансляції природної мови. Такі дослідники, як R. Moore, C. Manning, Y. Goldberg вивчали методи синтаксичного аналізу, семантичного розбору та обробки природної мови за допомогою штучного інтелекту та статистичних моделей. В українських дослідженнях акцент робиться на теоретичні засади лексичного і синтаксичного аналізу (зокрема, роботи, пов'язані з формальними граматиками та побудовою компіляторів). Проте питання трансляції саме природномовних конструкцій у синтаксично правильний код залишається недостатньо розробленим, що й обумовлює вибір теми дослідження.

Об'єктом дослідження є процес трансляції природномовних конструкцій у програмний код, включаючи етапи обробки тексту, побудови внутрішніх представлень і генерації програм.

Предметом дослідження є методи й алгоритми реалізації транслятора природної мови, зокрема синтаксичний і семантичний аналіз природномовних

описів програм та генерація коду мовами Python і C++.

Мета роботи полягає в створенні транслятора, здатного перетворювати природномовні конструкції у програмний код на мовах Python та C++. Транслятор повинен забезпечувати основні етапи обробки: лексичний аналіз, синтаксичний розбір, побудову дерева розбору, генерацію й вивід кінцевого коду.

Для досягнення мети було поставлено такі завдання:

- провести огляд наукових та технічних джерел за темою трансляції природної мови;
- визначити основні труднощі обробки природної мови при генерації програмного коду;
- розробити архітектуру транслятора, що включає інтерфейс користувача та ядро обробки;
- реалізувати парсинг природномовних конструкцій із побудовою синтаксичного дерева;
- здійснити генерацію програмного коду мовами Python та C++;
- протестувати транслятор з використанням позитивних і негативних прикладів.

Методи дослідження: у роботі використано метод аналізу і синтезу для вивчення наявних підходів до трансляції, метод побудови формальних граматик для створення синтаксичного аналізатора, метод структурного програмування для реалізації транслятора, а також метод експериментального тестування для перевірки його працездатності.

Практичне значення отриманих результатів полягає у створенні програмного прототипу транслятора, який може бути використаний як освітній інструмент для початківців у програмуванні, а також як основа для подальшої розробки більш складних систем взаємодії з комп’ютером через природну мову.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1.Огляд наукової, технічної та нормативної документації

У сучасних умовах стрімкого розвитку штучного інтелекту та мовних технологій особливого значення набувають дослідження, спрямовані на створення систем, здатних інтерпретувати та трансформувати природну мову у формальні структури. Одним із перспективних напрямів у цій галузі є трансляція природних мовних конструкцій у програмний код.

Наукові дослідження останніх років підтверджують актуальність цієї теми. Зокрема, у працях закордонних та вітчизняних дослідників розглядаються різні підходи до побудови систем Natural Language to Code (NL2Code).

Робота присвячена розробці навчального інструменту для школярів, який дає змогу вводити програмні команди українською мовою з подальшим автоматичним перекладом їх у синтаксис мов програмування Python або C++. Важливим технічним аспектом є вивчення структур та синтаксису мов програмування Python і C++, що мають різні підходи до типізації, оголошення змінних, роботи з пам’яттю та структур управління. Документація Python (PEP – Python Enhancement Proposals) акцентує увагу на простоті синтаксису, тоді як стандарт C++ (ISO/IEC 14882) має значно глибшу формальну структуру, орієнтовану на ефективність виконання. Це необхідно враховувати при побудові двомовного транслятора.

Складнощі, з якими можуть стикатися учні, полягають у відмінностях між динамічною типізацією Python та статичною типізацією C++, різній системі оголошення змінних, обробці помилок, синтаксисі циклів і умовних операторів, а також у складності розуміння концепцій роботи з пам’яттю в C++. Така різнорідність вимагає створення зрозумілого інтерфейсу транслятора та адаптивного підходу до навчального контенту.

З нормативної точки зору, для навчального програмного забезпечення чинними є вимоги до функціональної придатності, коректності результатів, безпечного використання, визначені у документах типу ДСТУ ISO/IEC 25010:2016

«Якість програмного забезпечення». Також враховуються методичні рекомендації щодо проєктування освітніх систем та принципи інтерфейсної доступності.

Таким чином, проведений огляд літератури, технічних стандартів та прикладних рішень створює основу для формування вимог до майбутнього навчального транслятора, що дозволяє інтерпретувати прості інструкції природною мовою у код мовами Python та C++.

1.2 Вимоги до формування програмного коду з природної мови

Формування програмного коду на основі природномовних конструкцій передбачає трансляцію вхідних текстових інструкцій, сформульованих природною мовою, у синтаксично коректний та семантично осмислений програмний код. Такий підхід спрямований на спрощення процесу програмування, зменшення порогу входу для початківців, а також демонстрацію базових принципів трансляції природної мови в штучні мови, зокрема мови програмування.

З огляду на складність повноцінного розуміння природної мови, транслятор орієнтований на роботу з нормалізованими командами, які відповідають низці структурних та лексичних обмежень. Це дозволяє зменшити неоднозначність при розборі та спростити побудову граматичної моделі.

Основні вимоги до природномовного введення:

– лаконічність та структурованість: команди мають бути короткими, без зайвих мовних зворотів;

– однозначність інструкцій: кожна команда повинна відповідати лише одному програмному елементу або конструкції;

– використання базових граматичних структур: прості речення з прямим порядком слів (підмет → присудок → додаток);

– обмежений словник: транслятор підтримує визначений набір ключових команд, таких як: створи, виведи, присвой, перевір, додай, відніми, якщо, повтори;

– уникнення синонімії та омонімії: для уникнення неоднозначностей користувачу слід використовувати лише рекомендовану лексику;

– послідовність: логічний порядок операцій повинен бути дотриманий в межах одного речення.

Наприклад, конструкція:

«Присвой змінній x значення 5»

має чітку інструкцію, однозначно інтерпретується і дозволяє побудувати дерево розбору без глибокого контекстуального аналізу.

Згенерований транслятором код має відповідати наступним критеріям:

– синтаксична коректність: код повинен відповідати граматичним правилам відповідної мови (Python або C++);

– семантична відповідність: дії, описані у природній мові, мають бути адекватно реалізовані у програмному коді;

– компільованість / виконуваність: згенерований код має успішно компілюватися (для C++) або запускатися (для Python) без помилок;

– узгодженість стилю: у випадках, де це можливо, код повинен дотримуватися базових стилістичних вимог обраної мови програмування;

– особливості формування коду для Python та C++ відрізняються через специфіку мов.

Таблиця 1.1 – Приклад трансляції природномовиних інструкцій

Природна інструкція	**Python**	**C++**
Створи змінну x зі значенням 5	x = 5	int x = 5;
Виведи x	print(x)	std::cout << x << std::endl;
Якщо x більше 0, виведи "позитивне"	if x > 0: print("позитивне")	if (x > 0) { std::cout << "позитивне"; }

Python є мовою з динамічною типізацією, простим синтаксисом та зручною структурою блоків за рахунок відступів. C++ вимагає явного оголошення типів змінних, використання фігурних дужок для блоків та включення заголовкових файлів.

Для побудови дерева розбору важливо формалізувати відповідність між частинами мови та програмними елементами:

– іменники → ідентифікатори (змінні, параметри, функції);

– дієслова → дії (оператори: присвоїти, обчислити, вивести);

– числівники → числові значення, параметри діапазонів;

– сполучники → логічні оператори (і, або, інакше → &&, ||, else).

Таким чином, речення "якщо a більше 10 і b менше 5" транслюється як логічна умова if (a > 10 && b < 5).

У межах навчального програмного продукту приймаються такі обмеження:

1. Підтримуються лише базові конструкції: присвоєння, виведення, арифметичні операції, умовні оператори, прості цикли.

2. Лексичний словник є фіксованим і не підлягає динамічному розширенню.

3. Морфологічний аналіз не проводиться — розпізнаються лише точні словоформи.

4. Типи в C++ не виводяться автоматично — за потреби користувач має дописати їх вручну.

Таким чином, успішна реалізація трансляції природної мови у програмний код передбачає встановлення чітких вимог до вхідних інструкцій і вихідного коду. Попереднє нормалізоване формулювання команд, обмеження лексики та орієнтація на просту структуру речень дають змогу забезпечити ефективну побудову дерев синтаксичного розбору і подальшу генерацію коректного коду. Отримані вимоги стали основою для розробки формальної граматики та логіки синтаксичного аналізатора, реалізованих у програмному продукті.

1.3. Аналіз сучасних методів і засобів формування коду

Формування програмного коду на основі природномовних конструкцій передбачає реалізацію систем, здатних інтерпретувати лексичні та синтаксичні одиниці природної мови й трансформувати їх у структуровані інструкції однієї або кількох мов програмування. Існує кілька основних підходів до реалізації подібних систем, які поділяються на шаблонні (rule-based), статистичні та нейромережеві (глибинного навчання). Кожен із них має власні особливості, переваги й обмеження.

1. Шаблонні (rule-based) системи. Ці методи ґрунтуються на жорстко визначених правилах відповідності між фразами природної мови та фрагментами коду. Вони реалізуються за допомогою регулярних виразів, граматик та вручну створених правил. Перевагами є простота реалізації, передбачуваність результатів та відсутність необхідності у великих навчальних вибірках. Водночас такі системи є малогнучкими: будь-яке відхилення у формулюванні призводить до неможливості коректного розпізнавання.

2. Статистичні методи. Передбачають використання імовірнісних моделей, таких як дерева рішень, наївні баєсівські класифікатори або n-грамні моделі. Вони дозволяють будувати узагальнені відповідності між вхідними мовними одиницями та відповідними елементами коду, з урахуванням статистичних закономірностей у великих корпусах даних. Недоліком є необхідність попереднього навчання та обмежена здатність працювати з контекстом.

3. Методи на основі глибинного навчання. Найсучасніші системи формування коду базуються на архітектурах типу Transformer (BERT, GPT, T5), що дозволяють здійснювати семантичний аналіз природної мови на глибокому рівні. Моделі, такі як Codex від OpenAI, CodeT5 або CodeGen, навчені на великих обсягах програмного коду, здатні генерувати повноцінні функції або програми на основі коротких запитів. Їх перевага — висока гнучкість і здатність працювати з неформальними запитами. Основними недоліками є висока потреба в обчислювальних ресурсах і складність контролю за результатом.

4. Гібридні підходи. Поєднують правила та машинне навчання: наприклад, синтаксичний аналіз здійснюється вручну, а генерація коду — за допомогою моделі. Такий підхід часто використовується в освітніх чи дослідницьких системах, де важлива керованість і передбачуваність.

На практиці також використовуються різноманітні програмні засоби для реалізації обробки природної мови:

- NLTK, spaCy – інструменти попередньої обробки тексту, токенізації, частиномовного аналізу;
- Tree-sitter – парсер для побудови синтаксичних дерев;
- ANTLR – генератор лексичних та синтаксичних аналізаторів;

– Hugging Face Transformers – бібліотека для використання нейромережевих моделей (у т.ч. CodeBERT, CodeT5).

Окрім загальних засобів, активно розвиваються готові продукти та сервіси, такі як:

– GitHub Copilot – асистент, що на основі GPT-3 генерує код з коментарів;

– OpenAI Codex – трансформує природні запити в код на десятках мов;

– Amazon CodeWhisperer**,** Replit Ghostwriter – комерційні рішення для генерації коду у хмарних середовищах.

Для цілей навчального програмного забезпечення доцільним є обмеження обсягу трансляції простими конструкціями: оголошення змінних, умовні оператори, цикли, базові функції. Найбільш придатними в такому випадку є гібридні або шаблонні підходи, які дозволяють уникнути надмірної складності при збереженні функціональності.

Таким чином, сучасний стан методів формування коду з природної мови демонструє високий рівень розвитку нейромережевих систем, однак для реалізації навчального транслятора доцільним є поєднання шаблонної обробки зі структурною генерацією коду, що забезпечує контрольований результат та доступність розуміння принципів трансляції студентами.

Опис параметрів інформаційного процесу

Інформаційний процес у системі трансляції з природної мови охоплює низку етапів — від введення мовної команди до формування програмного коду. Параметрами, які підлягають обробці, є синтаксичні одиниці тексту: ключові слова, іменники, дієслова, числівники, службові частини мови. Ці елементи піддаються лексичному аналізу, що дозволяє визначити їх функцію у реченні та відповідність програмній конструкції.

Визначальними параметрами є:

– ідентифікатори (імена змінних, функцій);

– значення (константи, літерали);

– типи операцій (арифметичні, умовні, логічні);

– контрольні структури (цикли, умовні оператори);

– порядок виконання (черговість дій у коді).

Наприклад, інструкція «додай до *x* число 3, якщо *y* більше нуля» потребує виокремлення змінних (*x*, *y*), значення (3), операції додавання та умови. На основі цього будується шаблон, який трансформується у відповідний фрагмент коду мовою Python чи C++.

Важливо, щоб транслятор коректно визначав межі команд, логічні зв’язки між ними, а також забезпечував правильну інтерпретацію вказаних дій у рамках обраної мови програмування. Саме чітке визначення параметрів інформаційного процесу дозволяє створити стійку та керовану модель трансляції.

2 ПРОЕКТУВАННЯ АРХІТЕКТУРИ ПРОГРАМНОЇ СИСТЕМИ

2.1 Загальна характеристика архітектури

Програмна система, що розробляється, призначена для трансляції інструкцій, сформульованих природною українською мовою, у програмний код на мовах Python або C++. Вона реалізує принципи навчального інструменту, який дозволяє користувачам (зокрема учням) краще засвоїти основи програмування через інтуїтивно зрозумілий інтерфейс.

Загальна архітектура побудована за клієнт-серверною моделлю. Система складається з трьох основних компонентів:

1. Інтерфейс користувача (фронтенд) — реалізований з використанням HTML та JavaScript; забезпечує введення природномовних інструкцій та вивід згенерованого програмного коду.

2. Серверна частина (бекенд) — написана на мові Python із використанням мікрофреймворку Flask; відповідає за приймання запитів від клієнта, виклик зовнішньої програми-транслятора та повернення відповіді.

3. Консольний транслятор (ядро системи) — реалізований мовою C++; безпосередньо виконує лексичний, синтаксичний аналіз вхідної природномовної конструкції, побудову дерева розбору та генерацію відповідного коду мовою Python або C++.

Архітектура системи представлена на рисунку 2.1.

Рисунок 2.1 – Загальна архітектура програмної системи трансляції природної мови у програмний код

Обмін даними між клієнтом і сервером здійснюється за допомогою HTTP-запитів у форматі JSON, що спрощує інтеграцію компонентів та уніфікує передавання текстової інформації.

Розробка архітектури системи базується на таких принципах:

1. Модульність: кожен компонент реалізує окрему частину функціональності, що забезпечує гнучкість у подальшому розширенні та тестуванні.
2. Простота та доступність: інтерфейс розроблено з урахуванням потреб початківців, тому він містить мінімум елементів і забезпечує максимально просту взаємодію з системою.
3. Зрозумілість обробки: текстова інструкція проходить послідовні етапи обробки — від лексичного аналізу до генерації готового коду, що можна використати як навчальний приклад.

Враховуючи освітнє призначення системи, особливу увагу приділено коректному відображенню результату та обробці помилок, з чіткими поясненнями у випадку некоректного вводу.

Таким чином, обрана архітектурна модель дозволяє досягти цілей розробки: забезпечити доступний засіб трансляції природної мови у програмний код з можливістю гнучкого розширення функціональності в майбутньому.

2.2 Функціональна схема системи

Функціональна схема системи демонструє загальну логіку обробки вхідних даних — від моменту введення користувачем інструкції природною мовою до виведення готового програмного коду однією з цільових мов (Python або C++). Система працює як ланцюжок функціональних компонентів, кожен з яких виконує окрему задачу, формуючи послідовність трансляції, адаптовану до потреб користувача.

Схема побудована за принципом класичної послідовної обробки запиту, в якій бере участь користувач, фронтенд, серверна логіка та консольний транслятор.

Користувач взаємодіє з веб-інтерфейсом, у якому передбачене текстове поле для введення інструкції українською мовою. Приклад:

```
Створи змінну x зі значенням 10.
```

Після натискання кнопки «Транслювати», введена команда відправляється на бекенд через HTTP-запит. У даному випадку серверна частина реалізована на Flask.

На сервері відбувається базова обробка введеного тексту:

- перевірка коректності запиту (чи не є порожнім);
- нормалізація (наприклад, приведення до нижнього регістру);
- підготовка до передачі в транслятор.

Далі відбувається передача даних до транслятора. Формується системна команда, яка викликає зовнішній консольний C++ транслятор (NL2CodeConsole.exe), передаючи йому інструкцію користувача у вигляді аргументу.

У трансляторі реалізовано:

- лексичний аналізатор — розбиває текст на токени (ключові слова, оператори, ідентифікатори, значення);
- синтаксичний аналіз — перевірка структури на відповідність формальній граматиці;
- побудова дерева розбору (AST) на основі правил трансляції.

На основі побудованої структури (AST) транслятор визначає цільову мову (Python або C++) та генерує відповідний фрагмент програмного коду згідно з правилами синтаксису цієї мови.

Наприклад, інструкція «створи змінну x зі значенням 10» буде трансформована у:

`x = 10` (для Python),

`int x = 10;` (для C++).

Згенерований код повертається у вигляді текстового рядка до Flask-бекенду.

Отриманий код відображається у відповідному полі результату на веб-сторінці. Користувач має змогу скопіювати його або переглянути для подальшого використання.

2.3 Опис основних модулів системи

Система трансляції природної мови у програмний код складається з низки функціональних модулів, кожен з яких виконує чітко визначену роль у процесі перетворення природномовного запиту у синтаксично правильний фрагмент коду однією з обраних мов програмування (Python або C++). Усі модулі взаємодіють між собою через чітко визначені інтерфейси, забезпечуючи модульність, масштабованість і можливість подальшого розширення системи.

2.3.1 Модуль інтерфейсу користувача

Модуль інтерфейсу користувача є важливою складовою програмної системи, що забезпечує взаємодію кінцевого користувача із функціональними можливостями транслятора. Основна мета модуля — надати інтуїтивно зрозумілий та доступний спосіб введення інструкцій природною мовою та отримання відповідного програмного коду на вибраній мові (Python або C++).

Користувачеві надається графічний веб-інтерфейс, через який він може:

- ввести текстову інструкцію українською мовою, що описує програмну дію (наприклад: *«виведи на екран число 10»*);
- обрати мову програмування (Python або C++);
- надіслати запит на обробку;
- переглянути згенерований результат — фрагмент коду, який відповідає введеній інструкції;
- скопіювати згенерований код для подальшого використання або редагування.

Діаграма взаємодії користувача з програмою представлена на рисунку 2.2.

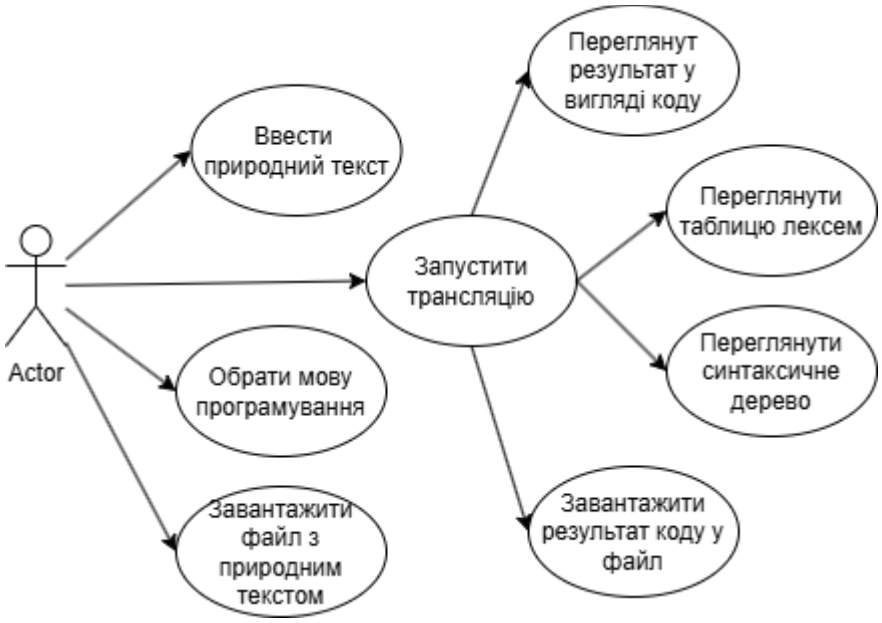


Рисунок 2.2 – Use-case діаграма

Для реалізації інтерфейсу використано стандартні веб-технології:

– HTML — для структурування елементів сторінки (текстове поле, кнопки, блок виводу результату);

– CSS — для стилізації елементів інтерфейсу, забезпечення зручного дизайну й адаптивності;

– JavaScript — для реалізації логіки клієнтської частини: обробка подій (натискання кнопки), передача даних на сервер, обробка відповіді.

– інтерфейс є односторінковим (Single Page Interface), не потребує оновлення сторінки при взаємодії та працює через асинхронні запити.

Під час проєктування інтерфейсу особливу увагу було приділено таким принципам:

1. Простота використання — інтерфейс мінімалістичний, позбавлений зайвих елементів, орієнтований на школярів і початківців.

2. Інтуїтивність — структура введення та виводу побудована лінійно: «ввів текст → натиснув кнопку → отримав код».

3. Доступність — інтерфейс не потребує встановлення програмного забезпечення, працює в будь-якому сучасному браузері.

4. Розширюваність — у разі необхідності можна додати нові функції: підказки, автодоповнення, приклади інструкцій, історію запитів тощо.

На рівні клієнта реалізовано базову перевірку введеного тексту, а саме: обмеження довжини запиту, перевірка на порожній ввід, обробка помилок під час з’єднання із сервером. Більш глибока валідація та обробка інтрукції здійснюється на серверному рівні, що мінімізує ризики некоректної обробки або зловмисного використання інтерфейсу.

Модуль інтерфейсу користувача відіграє ключову роль у доступності системи для кінцевих користувачів. Завдяки використанню стандартних веб-технологій та простого, інтуїтивного дизайну, він дозволяє легко й ефективно взаємодіяти з навчальним транслятором, що робить систему придатною для широкого кола користувачів, зокрема для учнів, які тільки починають вивчати основи програмування.

2.3.2 Модуль серверної логіки

Модуль серверної логіки є центральною частиною архітектури програмної системи, яка виконує обробку запитів користувача, взаємодію з транслятором та формування відповіді у вигляді програмного коду. Основне призначення цього модуля — приймати текстові інструкції природною мовою, передавати їх до транслятора, отримувати згенерований програмний код та надсилати його назад до клієнтської частини системи.

Серверна логіка виконує такі основні завдання:

- прийом HTTP-запитів від клієнта (frontend) через API;
- попередня валідація вхідних даних (чи не порожній запит, чи вказано мову програмування тощо);
- запуск зовнішньої консольної програми — транслятора (модуля обробки природної мови);
- обробка результату трансляції;
- формування структури відповіді (успішний результат або повідомлення про помилку);
- надсилання відповіді клієнту у форматі JSON.

Для реалізації серверної частини використано мову програмування Python разом із фреймворком Flask — легким веб-фреймворком, який забезпечує швидку розробку REST API.

Запуск транслятора здійснюється з допомогою стандартного модуля subprocess, який дозволяє отримати вихідні дані з консольної програми.

Рисунок 2.3 – Структурна схема роботи модуля серверної логіки

Принцип роботи

1. Користувач надсилає POST-запит із текстом інструкції та вибраною мовою (наприклад, "Python").

2. Flask-сервер приймає запит на певному маршруті, наприклад, /translate.

3. Дані перевіряються на наявність та коректність.

4. Якщо все коректно — сервер викликає транслятор як зовнішню програму, передаючи йому текст інструкції та параметри мови.

5. Транслятор виконує обробку запиту і повертає результат у вигляді програмного коду.

6. Отриманий код або повідомлення про помилку передаються назад клієнту у форматі JSON.

Модуль спроектовано таким чином, щоб не залежати від внутрішньої логіки транслятора. Сервер взаємодіє з ним як із "чорним ящиком", передаючи лише вхідні параметри і отримуючи результат. У разі потреби можливе розширення підтримуваних мов програмування або зміна транслятора без суттєвого втручання у код сервера. Завдяки модульній структурі та простому API, код легко розширювати, підтримувати й відлагоджувати.

Модуль серверної логіки є посередником між інтерфейсом користувача та основною обчислювальною частиною — транслятором. Завдяки використанню легкого фреймворку Flask та структурованому підходу до виклику транслятора, модуль забезпечує швидку обробку запитів, масштабованість та зручність для подальшого розширення. Саме він гарантує узгоджену взаємодію всіх компонентів системи та забезпечує правильну маршрутизацію даних.

2.3.3 Модуль транслятора

Модуль транслятора є центральною частиною всієї системи, оскільки виконує основну функцію – перетворення природномовного опису команд користувача у програмний код мовами Python або C++. Він реалізований як окрема консольна програма, яка взаємодіє з сервером через обгортку (translator_wrapper.py), приймає вхідні текстові конструкції українською мовою, обробляє їх та генерує відповідний програмний код. Архітектура модуля транслятора включає такі компоненти як: лексичний аналізатор, синтаксичний аналізатор, генератор коду.

Діаграма класів основних компонентів представлена на рисунку 2.4.

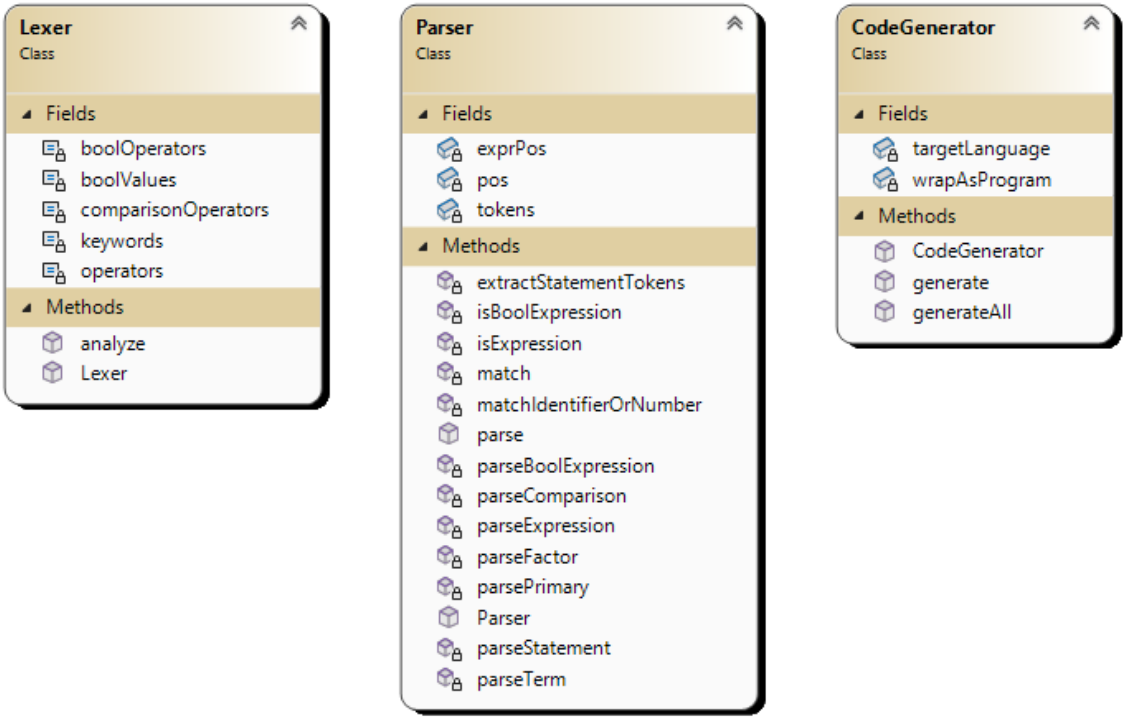


Рисунок 2.4 – Діаграма класів лексичного аналізатора, синтаксичного аналізатора та генератора коду

На етапі лексичного аналізу здійснюється попередня обробка тексту, виділяються окремі лексеми (ключові слова, ідентифікатори, оператори тощо) з природньої мови.

Алгоритм лексичного аналізу побудований на основі використання регулярних виразів — за якими здійснюється пошук і розпізнавання лексем у вхідному тексті. Кожна лексема в мові програмування має певну структурну форму: наприклад, ключові слова — це заздалегідь визначені словосполучення, ідентифікатори — це послідовності букв і цифр, які починаються з літери, числові константи — це послідовності цифр тощо.

Ці структурні шаблони можуть бути описані за допомогою регулярних виразів, що дозволяє автоматизувати процес розбиття вхідного тексту на окремі елементи. Таким чином, лексичний аналізатор не аналізує кожен символ окремо, а одразу знаходить фрагменти тексту, що відповідають тим чи іншим формальним правилам.

Рисунок 2.5 – Блок-схема алгоритму роботи лексичного аналізатора

Блок-схема алгоритму роботи лексичного аналізатора, зображена на рисунку 2.5, складається з таких етапів:

1. Початок — стандартне початкове блок-схеми, що позначає запуск процесу.

2. Отримання вхідного тексту — на цьому етапі програма отримує або зчитує текст, який буде аналізуватися. Це може бути рядок коду, введений користувачем або прочитаний із файлу.

3. Розбиття тексту на лексеми — вхідний текст розділяється на окремі лексичні одиниці (лексеми), які можуть бути ключовими словами, ідентифікаторами, операторами, літералами тощо. Це основна задача лексичного аналізу.

4. Визначення типу кожної лексеми і додавання до списку лексем — кожна виділена лексема аналізується для визначення її типу. Залежно від структури та вмісту, вона класифікується як ключове слово, ідентифікатор, оператор, числовий або рядковий літерал тощо. Після цього створюється відповідний об’єкт лексеми і додається до загального списку лексем, який згодом буде передано на етап синтаксичного аналізу.

5. Додавання токена кінця ($) — позначає дію, де до списку токенів додається спеціальний символ кінця входу ($). Цей символ часто використовується у синтаксичному аналізі для позначення завершення вхідної послідовності.

6. Повернення списку токенів — позначає дію повернення сформованого списку токенів після завершення аналізу.

7. Кінець — вказує на завершення процесу лексичного аналізу.

Блок-схема, що ілюструє процес лексичного аналізу, відображає основні етапи роботи відповідного модуля транслятора. На вхід подається програмний текст, який аналізатор поетапно обробляє з метою виділення елементарних одиниць мови — лексем. На цьому етапі здійснюється попередня фільтрація тексту: він розбивається на частини, які потенційно можуть бути ключовими словами, ідентифікаторами, операторами, рядковими або числовими літералами, логічними значеннями, дужками та іншими синтаксичними одиницями.

У процесі створення транслятора важливим етапом є синтаксичний аналіз, який відповідає за перевірку правильності побудови програми з точки зору граматики мови. На цьому етапі аналізуються послідовності лексем (отримані після лексичного аналізу) і перевіряється, чи утворюють вони коректні конструкції відповідно до синтаксичних правил.

Для реалізації синтаксичного аналізу існує кілька основних підходів, зокрема:

- висхідний аналіз (наприклад, LR, LALR, SLR);
- нисхідний аналіз (наприклад, рекурсивний спуск, LL);
- аналіз за допомогою генераторів парсерів (наприклад, ANTLR, Bison, YACC).

У рамках даного проєкту було обрано нисхідний синтаксичний аналіз методом рекурсивного спуску (Recursive Descent Parsing).

Переваги рекурсивного спуску:

1. Простота реалізації – кожне правило граматики реалізується у вигляді окремої функції, що полегшує читання та розуміння коду.

2. Природна відповідність граматиці – структура функцій повторює структуру правил, тож логіка роботи парсера добре відображає описану граматику.

3. Гнучкість – легко додавати нові конструкції до граматики або змінювати існуючі, не змінюючи кардинально архітектуру парсера.

4. Придатність для невеликих і середніх мов – для реалізації власних DSL (Domain-Specific Languages) або простих мов програмування цей підхід є оптимальним.

5. Легкість налагодження – помилки в синтаксичному аналізі можна швидко виявити через трасування викликів функцій.

Причини вибору саме цього методу:

- мова, для якої створюється транслятор, є досить простою й обмеженою за кількістю конструкцій, що добре узгоджується з можливостями рекурсивного спуску;
- пріоритетом проєкту є зрозумілість і навчальна цінність, а не максимальна продуктивність чи масштабованість;
- також важливою була можливість побудови дерева розбору (AST), яке легко реалізується саме у структурі рекурсивного синтаксичного аналізу.

Враховуючи усі зазначені аспекти, метод рекурсивного спуску був визнаний найдоцільнішим для реалізації синтаксичного аналізатора у цьому проєкті. Його реалізація дозволила ефективно обробляти ключові конструкції мови: умовні оператори, цикли, присвоєння та вирази.

Блок-схема алгоритму роботи синтаксичного аналізатора показана на рисунку 2.6.

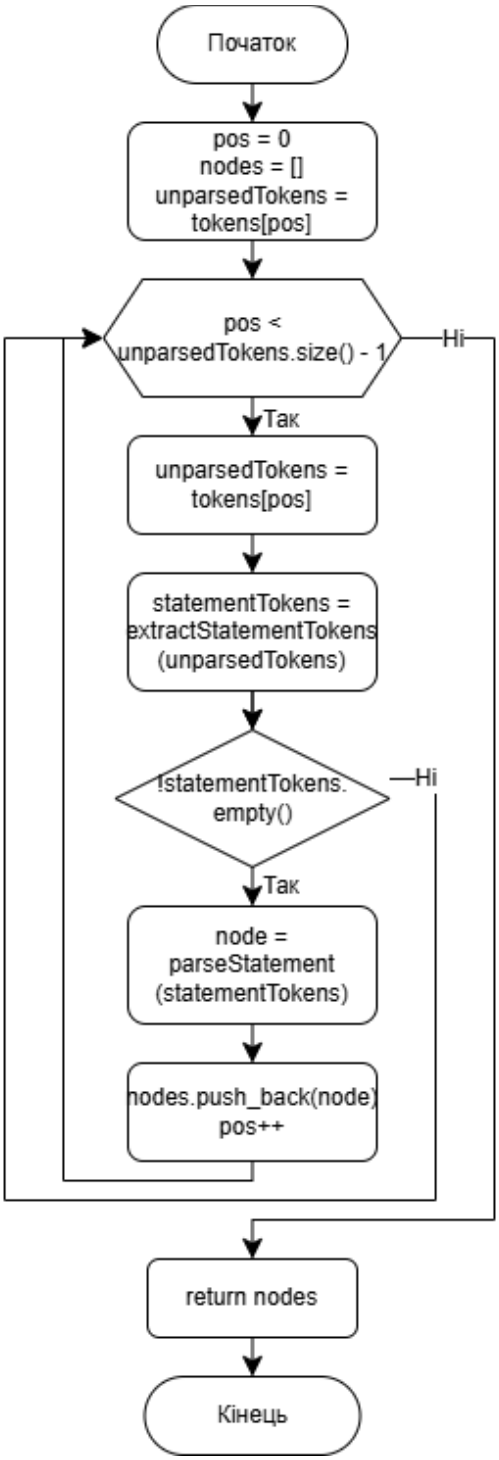


Рисунок 2.6 – Блок-схема алгоритму роботи синтаксичного аналізатора

Основні етапи алгоритму:

1. Ініціалізація

На початку методу змінна pos встановлюється у 0 — вона слугує покажчиком на поточну позицію в списку токенів. Також створюється порожній вектор nodes для зберігання результатів парсингу та копія списку токенів, що ще не були оброблені (unparsedTokens).

2. Основний цикл розбору

Метод використовує цикл while, який виконується до тих пір, поки pos менше unparsedTokens.size() - 1. У кожній ітерації відбуваються такі дії:

- оновлення списку необроблених токенів шляхом обрізання його від позиції pos до кінця;
- виділення однієї інструкції із вхідних токенів за допомогою функції extractstatementtokens();
- якщо виділена інструкція не порожня, вона передається у функцію parsestatement(), яка виконує синтаксичний аналіз і створює відповідний вузол ast;
- якщо вузол створено успішно (тобто node != nullptr), він додається до вектора nodes;
- незалежно від успішності парсингу, pos збільшується на одиницю для переходу до наступної інструкції.

3. Завершення роботи

Після завершення циклу метод повертає вектор nodes, що містить об’єкти типу ASTNode, кожен з яких відповідає одній проаналізованій інструкції.

На етапі побудови синтаксичного аналізатора одним з ключових завдань є формальне визначення граматики мови, яку транслятор має розпізнавати. У даній роботі граматика задається у формі контекстно-вільної граматики (КС-граматики), яка підходить для опису структур природної мови та програмних конструкцій.

Для вхідної мови транслятора було розроблено формальну контекстно-вільну граматику G у вигляді чотириелементної множини:

G = (T, N, P, S), де:

- T – множина термінальних символів (наприклад, оформи у вигляді програми, якщо, присвоїти, a, c, +, -, (,), тощо);
- N – множина нетермінальних символів (S, L, O, B, C, D, E, T, F);
- P – множина продукцій (правил);
- S – початковий символ (вся програма).

Основні правила граматики:

S → оформи у вигляді програми L

L → O | L.O | L.

O → якщо (B) то O інакше O. | якщо (B) то O. | { L } | роби O поки (L). | a присвоїти E.

B → B або C | C і D | не (B) | D

D → E менше E | E більше E | E дорівнює E | (B)

E → E – T | E + T | T

T → T-- | F

F → (E) | a | c

Особливості побудованої граматики:

1. Підтримка вкладених конструкцій. Граматика дозволяє використовувати вкладені умовні оператори та блоки (через O → { L }), що забезпечує виразність мови.

2. Обробка виразів з пріоритетами. Правила E, T, F задають структуру для розбору арифметичних виразів з правильною пріоритетністю операторів: множення та ділення мають вищий пріоритет, ніж додавання та віднімання.

3. Уніфікована структура логічних виразів. Завдяки розділенню логічних операцій (або, і, не) та порівнянь (менше, більше, дорівнює), граматика дозволяє комбінувати прості та складені умови.

4. Обробка заперечення. Оператор не завжди вимагає дужки після себе: не (умова), що спрощує синтаксичний аналіз.

5. Оператори завершуються крапкою (.). Це дозволяє легко визначати межу між окремими інструкціями, що особливо важливо при обробці природної мови.

Описана граматика адаптована до трансляції природних мовних конструкцій, що мають більшу варіативність, ніж формальні мови. Вона спрощена для зручності реалізації парсера методом рекурсивного спуску, а також дозволяє легко її розширювати (наприклад, додати підтримку для або функції).

Після визначення граматики та реалізації алгоритму синтаксичного аналізу важливим кроком є побудова дерева розбору – структурованого представлення синтаксичних конструкцій програми. У даному проєкті для цього реалізовано абстрактне синтаксичне дерево (Abstract Syntax Tree, AST), яке відображає ієрархію мовних конструкцій та їх логічну структуру.

Метою побудови дерева розбору є візуалізація структури програми, формалізування вхідних природномовних конструкцій у вигляді синтаксичних одиниць, створення основи для наступних етапів трансляції: семантичного аналізу та генерації коду.

Принципи побудови AST.

1. Кожна синтаксична конструкція (умова, цикл, присвоєння, вираз тощо) відповідає вузлу дерева певного типу.

2. Дочірні вузли містять складові частини конструкції. Наприклад, для умовного оператора це буде:

- умова (condition);
- тіло гілки if (then_branch);
- тіло гілки else (else_branch).

3. Усі вузли дерева реалізовані як об’єкти класу ASTNode.

Діаграма класів вузлів дерева представлена на рисунку 2.7

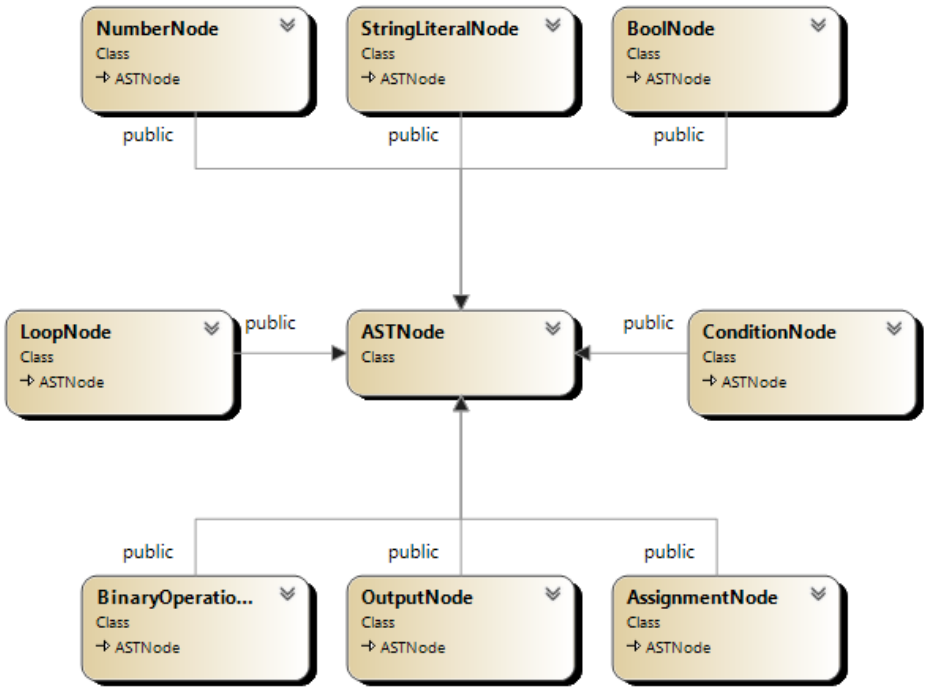


Рисунок 2.7 – Діаграма класів вузлів дерева

Етап генерації коду є завершальною фазою трансляції, яка полягає у перетворенні побудованого абстрактного синтаксичного дерева (AST) у програмний код на обраній цільовій мові програмування. Цей процес здійснюється

відповідно до набору визначених принципів, що забезпечують збереження структури, логіки та семантики початкової конструкції, заданої користувачем у природній мові.

Кожен елемент AST транслюється відповідно до синтаксису цільової мови програмування (Python або C++). Наприклад:

Присвоєння x = 5:

У Python транслюється як: x = 5

У C++: x = 5;

Вивід значення:

Python: print("Hello")

C++: std::cout << "Hello";

Вибір мови здійснюється через параметр targetLanguage, що передається до генератора коду.

Під час трансляції важливо не лише правильно оформити інструкції, але й зберегти логіку виконання, вкладеність та пріоритети операцій. Для цього AST зберігає відношення між вузлами, які відображають логічну структуру програми.

Наприклад, конструкція:

Якщо x більше 5, то вивести “Більше”, інакше – “Не більше”

перетворюється у дерево умовної конструкції, яке далі відображається як:

Python:

```
if x > 5:
    print("Більше")
else:
    print("Не більше")
```

C++:

```
if (x > 5) {
    std::cout << "Більше";
} else {
    std::cout << "Не більше";
}
```

Одна з основних цілей транслятора – здатність генерувати код на різних мовах програмування з одного й того ж AST. Це досягається завдяки абстракції

типів вузлів (AssignmentNode, ConditionNode, BinaryOperationNode тощо) та перевірці поточної мови на кожному кроці генерації.

Так, оператор виводу:

Для Python – print(...)

Для C++ – std::cout << ...;

Це дає змогу в майбутньому розширити транслятор підтримкою нових мов (наприклад, JavaScript, Java) без необхідності змінювати основну логіку синтаксичного аналізу.

У багатьох мовах, зокрема в C++, потрібна мінімальна структура повної програми (наприклад, наявність функції main). Тому генератор має функціональність обгортання згенерованих інструкцій у таку структуру – залежно від мови й вимог компілятора. Це дозволяє генерувати як повноцінні програми, готові до компіляції, так і фрагменти коду для вставки у більші проєкти.

Генерація коду виконується без побудови проміжного подання (наприклад, тріад, байткоду чи LLVM IR). Це робить трансляцію простішою і більш прозорою – кожна природномовна команда безпосередньо перетворюється у синтаксично правильну інструкцію на цільовій мові. Такий підхід називається синтаксично-керованим перекладом (syntax-directed translation).

Таким чином, принципи генерації коду у трансляторі забезпечують коректне, зручне та масштабоване перетворення природномовних конструкцій у програмний код з урахуванням особливостей обраної мови програмування.

2.4 Формати зберігання та обробки даних

У процесі функціонування системи для трансляції природномовних конструкцій у програмний код важливу роль відіграє вибір форматів зберігання та обробки даних. Ці формати повинні забезпечувати зручність передачі даних між модулями, читабельність для користувача, простоту обробки на рівні коду, а також відповідність обраній архітектурі.

Вхідними даними є текстові команди природною українською мовою, які користувач вводить у відповідне текстове поле через інтерфейс користувача. Ці

команди передаються до серверної частини у вигляді JSON-об’єкта через HTTP-запит (метод POST). Наприклад:є

```
{
  "language": "python",
  "input_text": "створи змінну результат і присвой їй суму чисел 5 і 10"
}
```

Такий формат є універсальним, компактним та легко обробляється мовами програмування як на клієнті (JavaScript), так і на сервері (Python).

У процесі трансляції система створює проміжні структури, зокрема:

- список лексем після лексичного аналізу;
- абстрактне синтаксичне дерево (AST), представлене як вкладені об'єкти або вузли;
- журнали діагностики та помилок, які фіксуються у вигляді структурованих повідомлень для подальшого відображення користувачу або для журналювання (логування).

Для внутрішнього представлення AST та діагностичних даних також використовується формат JSON, що спрощує налагодження та тестування системи.

Результатом обробки є згенерований програмний код, який повертається у вигляді текстового рядка (або багаторядкового блоку коду) залежно від вибраної мови програмування. Він передається з сервера у відповідь на запит у такому форматі:

```
{
  "generated_code": "result = 5 + 10"
}
```

Код може бути виведений на екран, скопійований користувачем або збережений у текстовий файл (наприклад, .py або .cpp) для подальшого використання.

У поточній реалізації система не вимагає довготривалого зберігання даних у базі даних. Усі обчислення й трансляції виконуються у режимі «на льоту» без

постійного збереження історії запитів. Це дозволяє зменшити складність системи та забезпечити високу швидкодію.

У цьому розділі було здійснено проєктування архітектури програмної системи трансляції природномовних інструкцій у програмний код. Визначено загальну архітектуру системи, яка складається з трьох основних модулів: інтерфейсу користувача (frontend), серверної логіки (backend) та транслятора. Така модульна структура забезпечує гнучкість, масштабованість та можливість незалежної розробки й тестування окремих компонентів.

Для кожного модуля було детально описано його функціональне призначення, взаємодію з іншими частинами системи та принципи реалізації. Визначено, що обмін даними між модулями здійснюється у форматі JSON через HTTP-запити, що дозволяє ефективно реалізувати клієнт-серверну модель.

Окрему увагу приділено форматам зберігання й обробки даних, зокрема описано вхідні, проміжні та вихідні структури, які формуються в процесі трансляції.

Функціональна та логічна схема, наведена у відповідних підпунктах, ілюструє основні етапи обробки даних у системі та підтверджує цілісність архітектурного рішення. Результати цього етапу слугують основою для реалізації прототипу транслятора та його подальшого тестування у наступних розділах.

3 РЕАЛІЗАЦІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Загальні принципи реалізації

Програмна система реалізована на основі клієнт-серверної архітектури. Клієнтська частина (frontend) написана з використанням HTML, CSS та JavaScript, і забезпечує інтерфейс користувача для введення інструкцій природною мовою, вибору мови програмування (Python або C++) та перегляду результату трансляції. Серверна частина (backend) реалізована за допомогою мови Python з використанням мікрофреймворку Flask, який забезпечує обробку HTTP-запитів та взаємодію з консольною утилітою транслятора.

Транслятор розроблений мовою C++ у вигляді окремої консольної програми. Він виконує основні етапи трансляції — лексичний, синтаксичний та семантичний аналіз, побудову дерева розбору (AST) та генерацію програмного коду.

Обмін даними між модулями здійснюється у форматі JSON, що забезпечує уніфікованість структури запитів і відповідей. Система не зберігає дані постійно (тобто база даних не використовується), що відповідає її навчальному призначенню та спрощує реалізацію.

Особливу увагу під час реалізації було приділено забезпеченню коректності трансляції інструкцій, захисту від помилок користувача, збереженню структури вхідних команд та підтримці обох мов програмування. Для обробки помилок на серверному рівні реалізовано логування, а для фронтенду — повідомлення про помилки користувачу у зрозумілій формі.

3.2 Реалізація модуля інтерфейсу користувача

Клієнтська частина системи реалізована за допомогою стандартних вебтехнологій — HTML, CSS та JavaScript. Вона забезпечує зручний інтерфейс для взаємодії користувача з транслятором, а також виконує роль передавача запитів до серверної частини та виводу результатів на екран.

Основні елементи інтерфейсу:

1. Поле введення інструкцій природною мовою. Це текстове поле дозволяє користувачу вводити програмні команди українською мовою, наприклад: створи змінну результат і присвой їй значення 10.

2. Вибір мови програмування. Два перемикачі (radio buttons) дозволяють обрати, у яку мову транслювати: Python або C++. Ця інформація передається на сервер у складі запиту.

3. Кнопка трансляції. Після натискання кнопки «Транслювати» відбувається виклик JavaScript-функції, яка формує запит до серверного API у форматі JSON. Запит містить вхідний текст та обрану мову програмування.

4. Поле виводу результату. Після обробки запиту сервер повертає згенерований код, який виводиться у спеціальне поле — textarea або стилізований блок, з підсвічуванням синтаксису (опційно).

5. Повідомлення про помилки. У разі помилки (наприклад, синтаксична неузгодженість або некоректна конструкція), сервер повертає відповідь з поясненням, яке виводиться окремим блоком у нижній частині вікна.

Приклад HTML-структури:

```
<div class="translator-container">
  <textarea id="input-text" placeholder="Введіть інструкції природною мовою..."></textarea>

  <div class="language-options">
    <label><input type="radio" name="lang" value="python" checked> Python</label>
    <label><input type="radio" name="lang" value="cpp"> C++</label>
  </div>

  <button onclick="sendRequest()">Транслювати</button>

  <textarea id="output-code" readonly placeholder="Згенерований код з’явиться тут..."></textarea>
  <div id="error-message"></div>
</div>
```

JavaScript-функція translateText() відповідає за надсилання введеного користувачем тексту природньою мовою до серверної частини для трансляції у програмний код.

```
function translateText() {
    const inputText =
document.getElementById("inputText").value;
    const language =
document.getElementById("languageSelect").value;

    fetch("/translate", {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify({ text: inputText, language:
language })
    })
        .then(res => res.json())
        .then(data => {
            document.getElementById("output").textContent =
data.code || "Помилка трансляції";
        })
        .catch(() => {
            document.getElementById("output").textContent =
"Помилка з'єднання з сервером";
        });
}
```

Призначення даної функції полягає в зчитуванні введеного тексту та обраної мови, формування запиту до /translate на сервері з тілом у форматі JSON, отримання відповіді з транслювання та виводу її в полі результат, а у випадку помилки – відображення відповідного повідомлення.

Функція loadTextFromFile() дозволяє завантажити вхідні інтрукції природною мовою з текстового файлу, що зручно для тестування або роботи з заготовками.

```
function loadTextFromFile() {
    const fileInput = document.getElementById("loadTextFile");
```

```
        const reader = new FileReader();
        reader.onload = () => {
            document.getElementById("inputText").value =
reader.result;
        };
        if (fileInput.files[0]) {
            reader.readAsText(fileInput.files[0], 'UTF-8');
        }
    }
```

Ця функція використовує API FileReader для читання вмісту локального файлу та після завантаження вставляє текст у поле введення.

Функція downloadCode() забезпечує можливість завантажити згенерований програмний код у вигляді окремого файлу з відповідним розширенням.

```
    function downloadCode() {
        const code = document.getElementById("output").textContent;
        const language =
document.getElementById("languageSelect").value;
        const ext = language === "python" ? "py" : "cpp";
        const blob = new Blob([code], { type:
"text/plain;charset=utf-8" });
        const a = document.createElement("a");
        a.href = URL.createObjectURL(blob);
        a.download = `translated_code.${ext}`;
        a.click();
    }
```

Призначенням даної функції є створення тимчасового об’єкту типу Blob з отриманого коду, формування посилання на завантаження та автоматичної активації його. Файл зберігається з розширенням .py або .cpp, залежно від вибраної мови.

Функція showSyntaxTree() активує відображення синтаксичного дерева, яке формується у серверній частині і передається у вигляді HTML-коду.

```
    function showSyntaxTree() {
        document.getElementById("syntaxTree").innerHTML = treeHTML;
```

```
        document.getElementById("syntaxTree").style.display =
"block";
        document.getElementById("lexemeTable").style.display =
"none";

        var toggler = document.getElementsByClassName("caret");
        var i;

        for (i = 0; i < toggler.length; i++) {
            toggler[i].addEventListener("click", function () {

this.parentElement.querySelector(".nested").classList.toggle("active
");
                this.classList.toggle("caret-down");
            });
        }
    }
```

Призначення:

- показує блок з синтаксичним деревом;
- приховує таблицю лексем, якщо вона була активною;
- додає можливість згортати й розгортати вузли дерева.

Ці функції у сукупності забезпечують повноцінну взаємодію користувача з вебінтерфейсом системи трансляції: від введення даних до перегляду й завантаження результату та синтаксичного дерева. Вони реалізовані з урахуванням простоти використання, інтуїтивності та підтримки сучасних браузерів.

Особливості реалізації:

- вся логіка реалізована без використання сторонніх бібліотек, що забезпечує простоту і зрозумілість коду;
- інтерфейс є адаптивним — його можна використовувати на різних пристроях (пк, планшети);
- результат зберігається лише у сесії користувача і не передається третім сторонам, що відповідає вимогам освітніх рішень.

3.3 Реалізація серверної логіки

Серверна частина програмного забезпечення реалізована з використанням вебфреймворку Flask, що є мікрофреймворком для створення вебдодатків мовою Python. Вона забезпечує обробку запитів від клієнтської частини, виклик консольного транслятора та повернення результату користувачеві у форматі JSON.

Файл app.py — основний серверний скрипт.

Цей файл містить Flask-додаток, який виконує наступні завдання:

- надає головну HTML-сторінку;
- приймає POST-запити на /translate;
- зберігає вхідні дані тимчасово у файл;
- викликає консольну програму транслятора (NL2CodeConsole.exe);
- читає результат трансляції з тимчасового файлу;
- надсилає результат клієнту.

Основні етапи роботи:

1. Ініціалізація Flask-додатку:

```
app = Flask(__name__, static_folder='../frontend', static_url_path='/')
CORS(app)
```

Забезпечується підтримка CORS для доступу з frontend-частини, яка розташована в іншій директорії.

2. Обробка запиту на /translate:

При запиті від клієнта на сервер надходить текст природною мовою. Цей текст зберігається у тимчасовий файл temp_input.txt, після чого запускається транслятор:

```
result = subprocess.run(
    [TRANSLATOR_PATH, temp_input_path, temp_output_path],
    check=True,
    stdout=subprocess.PIPE,
    stderr=subprocess.PIPE,
    text=True
)
```

Програма читає вміст згенерованого вихідного файлу та надсилає його у відповіді клієнту:

```
with open(temp_output_path, 'r', encoding='utf-8') as f:
    generated_code = f.read()
return jsonify({"code": generated_code})
```

3. Обробка помилок:

У разі виникнення помилки під час виконання трансляції сервер повертає код 500 з описом помилки, отриманим із stderr.

4. Очищення тимчасових файлів:

Після обробки запиту, незалежно від результату, тимчасові файли видаляються:

```
os.remove(temp_input_path)
os.remove(temp_output_path)
```

Файл translator_wrapper.py — допоміжний модуль. Цей файл реалізує функцію run_translator(), яка дублює логіку запуску транслятора у вигляді окремої функції. Вона також:

- створює тимчасові файли для введення й виведення;
- запускає транслятор через subprocess.run();
- повертає результат або повідомлення про помилку;
- очищає ресурси після завершення роботи.

Призначення модуля — можлива повторна інтеграція з іншими сервісами, або перенесення логіки трансляції у більшу систему чи API, з можливістю повторного використання.

Запуск транслятора здійснюється як окрема консольна програма (NL2CodeConsole.exe), з передачею шляхів до вхідного та вихідного файлу як аргументів командного рядка. Такий підхід забезпечує модульність і незалежність реалізації транслятора від вебсервісу, а також дозволяє легко замінити або адаптувати транслятор без зміни логіки вебсерверу.

Безпека та оптимізація:

- завдяки використанню tempfile.NamedTemporaryFile() гарантується унікальність і безпека тимчасових файлів;

– за допомогою механізму обробки винятків забезпечується стабільність сервісу;

– механізм CORS дає змогу взаємодіяти з сервером із клієнтської частини, яка розміщена на іншому хості або порті (наприклад, під час локальної розробки).

Ця серверна частина є зв’язуючою ланкою між користувачем та ядром системи — транслятором, забезпечуючи зручну взаємодію, обробку запитів, а також гнучкість для подальшого розширення функціональності.

3.4 Реалізація транслятора

Транслятор є основною частиною програмної системи. Він виконує обробку вхідного тексту, написаного природною мовою, з подальшою побудовою абстрактного синтаксичного дерева (AST) і генерацією коду мовами програмування Python або C++. Реалізація транслятора виконана як консольна програма мовою C++.

Лексичний аналізатор відповідає за перетворення вхідного тексту природною мовою у послідовність токенів — найменших смислових одиниць, які далі обробляються синтаксичним аналізатором. У реалізації використовується регулярний вираз для розбиття вхідного тексту на токени, а також словники ключових слів, операторів та інших конструкцій, які розпізнаються за їх синтаксичною формою.

Код лексера реалізовано на C++ з використанням модуля regex для розпізнавання токенів у рядку. Основні кроки:

– визначено множини зарезервованих слів (keywords), математичних операторів (operators), логічних операторів (boolOperators), операторів порівняння (comparisonOperators), а також булевих значень (boolValues). Це дозволяє ідентифікувати семантику кожного слова чи символу;

– за допомогою регулярного виразу відбувається розбиття вхідного рядка на потенційні токени:

```
std::wregex tokenRegex(LR"((for|do|:=|[<>]=?|[+\-*/().]|"(?:[^"\\]|\\.)*"|[a-zA-Za-яA-ЯiїєIЇЄ_][a-zA-Za-яA-ЯiїєIЇЄ0-9_]*|[IVXLCDM]+|\d+|;))");
```

Кожне слово аналізується та класифікується за типом:

Keyword — якщо слово належить до зарезервованих ключових слів;

Assignment, Operator, ComparisonOperator — для математичних чи логічних операторів;

Identifier — для імен змінних;

Number — для числових значень;

StringLiteral — для рядкових значень у лапках;

Unknown — якщо токен не розпізнано, тощо.

Для прикладу, обробка ключового слова та оператора виглядає так:

```
if (keywords.find(word) != keywords.end()) {
    tokens.push_back({ LexicalTokenType::Keyword, utf8Word });
}
else if (operators.find(word) != operators.end()) {
    tokens.push_back({ LexicalTokenType::Operator, utf8Word });
}
```

Кожен знайдений токен зберігається у вектор tokens, який потім передається на вхід синтаксичному аналізатору. Після завершення аналізу додається спеціальний токен завершення End з текстом $, який сигналізує кінець вхідного потоку.

Такий підхід дозволяє легко підтримувати розширення мови: додавання нових ключових слів або операторів достатньо лише в списках, не змінюючи основну логіку розбору.

Синтаксичний аналіз реалізовано методом рекурсивного спуску, що забезпечує просту й гнучку обробку вхідних конструкцій природної мови. Кожна конструкція (наприклад, присвоєння, виведення, умовна конструкція, цикл) розпізнається окремо, з подальшим побудуванням вузлів абстрактного синтаксичного дерева (AST).

Клас Parser приймає на вхід список токенів, які були сформовані лексичним аналізатором. Метод parse() послідовно розбиває вхідний потік токенів на окремі

інструкції за допомогою функції extractStatementTokens(), а потім кожну інструкцію обробляє через parseStatement() для побудови відповідних вузлів абстрактного синтаксичного дерева (AST).

Код методу parse():

```
std::vector<std::shared_ptr<ASTNode>> Parser::parse() {
    pos = 0;
    std::vector<std::shared_ptr<ASTNode>> nodes;

    std::vector<Token> unparsedTokens =
std::vector<Token>(tokens.begin() + pos, tokens.end());

    while (pos < unparsedTokens.size() - 1) {
        unparsedTokens =
std::vector<Token>(unparsedTokens.begin() + pos,
unparsedTokens.end());

        auto statementTokens =
extractStatementTokens(unparsedTokens);

        if (!statementTokens.empty()) {
            auto node = parseStatement(statementTokens);
            if (node != nullptr) {
                nodes.push_back(node);
            }
            pos++;
        }
    }
    return nodes;
}
```

В цьому методі ініціалізується позиція pos і порожний список вузлів, далі в циклі виділяється кожна інструкція по черзі. Кожна інструкція парситься окремо в parseStatement(), а результат додається до дерева.

Метод extractStatementTokens() виділяє окрему логічну інструкцію (statement) з потоку токенів. Зупиняється при знаходженні крапки, яка є маркером кінця інтрукції. Обробляє вкладені конструкції – умовні та цикли.

```
    std::vector<Token> Parser::extractStatementTokens(const
std::vector<Token>& statement) {
        size_t i = 0;
        std::vector<Token> statementTokens;
        int depth = 0;
        bool insideComplexBlock = false;

        while (i < statement.size()) {
            if (statement[i].type == LexicalTokenType::Keyword) {
                if (statement[i].value == "якщо" ||
statement[i].value == "повторити") {
                    depth++;
                    insideComplexBlock = true;
                }
            }

            if (statement[i].value == "." && depth == 0) {
                i++; // Пропускаємо крапку
                break;
            }

            if (depth > 0 && statement[i].value == "." &&
statement[i + 1].value == ".") {
                depth--;
            }
            statementTokens.push_back(statement[i]);
            i++;
        }
        return statementTokens;
    }
```

Конструкція extractStatementTokens() дозволяє виокремлювати окремі інструкції навіть у складних вкладених умовах і циклах. Вона враховує вкладеність

конструкцій (depth) і завершує поточну інструкцію по одному символу крапки (.), що відіграє роль роздільника.

Метод parseStatement() виконує розпізнавання синтаксичних конструкцій:

– присвоєння: конструкція виду X присвоїти вираз. Для виразів використовується рекурсивна функція parseExpression(), яка підтримує операції додавання, віднімання, множення та ділення. Підтримуються також рядкові та числові літерали;

– виведення: ключове слово вивести із наступним значенням або змінною;

– умовна конструкція (якщо ... то ... інакше ...): підтримується вкладена структура з парсингом логічних виразів (parseBoolExpression()) і розділенням на гілки trueBranch та falseBranch. Виявлення вкладених умов реалізовано за допомогою depth в extractStatementTokens(), що дозволяє враховувати структуру конструкції навіть при наявності вкладених блоків;

– цикл: підтримується базова форма циклу з обмеженнями за початковим, кінцевим значенням і змінною циклу. Тіло циклу поки що не розбирається повністю, але підготовлена структура (LoopNode) для подальшого розширення.

Для обробки виразів реалізовано три рівні пріоритету:

- parseExpression() — операції +, -
- parseTerm() — операції *, /
- parseFactor() — числа, змінні, дужки, рядки

Код методу parseStatement:

```
std::shared_ptr<ASTNode> Parser::parseStatement(const
std::vector<Token>& statement) {
        pos = 0;

        if (isAssignment(statement)) {
            return parseAssignment(statement);
        }
        else if (isOutput(statement)) {
            return parseOutput(statement);
        }
        else if (isCondition(statement)) {
```

```
            return parseCondition(statement);
        }
        else if (isLoop(statement)) {
            return parseLoop(statement);
        }

        return nullptr;
    }
```

Після лексичного та синтаксичного аналізу відбувається побудова абстрактного синтаксичного дерева (AST — Abstract Syntax Tree). Це структуроване представлення програми, яке описує її логіку у вигляді деревоподібної ієрархії об'єктів. Кожен вузол дерева відповідає певному елементу мови (вираз, оператор, літерал тощо).

Базовий клас ASTNode:

```
class ASTNode {
public:
    ASTNodeType type;
    virtual ~ASTNode() = default;
};
```

Усі інші вузли успадковують цей базовий клас. Поле type містить тип вузла, що визначається через перелік ASTNodeType.

```
enum class ASTNodeType {
    Assignment,
    Number,
    StringLiteral,
    Bool,
    BinaryOperation,
    Arithmetic,
    Condition,
    Loop,
    Output
};
```

Після побудови абстрактного синтаксичного дерева (AST) відбувається перетворення вузлів цього дерева у текстовий програмний код за допомогою

модуля генерації коду. Цей модуль реалізовано у вигляді класу CodeGenerator, який забезпечує підтримку мов Python та C++. Він дозволяє перетворити окремі конструкції (наприклад, арифметичні вирази, умовні оператори, операції присвоєння) або цілі послідовності інструкцій у відповідний синтаксис цільової мови.

Структура класу CodeGenerator:

```
class CodeGenerator {
    std::string targetLanguage;
    bool wrapAsProgram;

public:
    CodeGenerator(std::string language, bool wrap);
    std::string generate(std::shared_ptr<ASTNode> node);
    std::string generateAll(const
std::vector<std::shared_ptr<ASTNode>>& nodes);
};
```

targetLanguage — мова цільового коду ("Python" або "C++");

wrapAsProgram — прапорець, що вказує, чи слід обгортати згенерований код у повну програму з точкою входу (main).

Метод generate реалізує рекурсивну генерацію коду для окремого вузла дерева, відповідно до його типу. Вузли представлені за допомогою похідних класів від ASTNode.

Приклади генерації:

1. Арифметичні вирази

Вузол типу BinaryOperationNode генерується у вигляді інфіксного виразу з дужками (наприклад, (5 + 3)).

```
auto binOp =
std::dynamic_pointer_cast<BinaryOperationNode>(node);
return "(" + generate(binOp->left) + " " + binOp->op + " " +
generate(binOp->right) + ")";
```

2. Операції присвоєння

Вузол AssignmentNode перетворюється у присвоєння змінній:

x = (5 + 3); – для C++

x = (5 + 3) – для Python

Код вузла:

```
auto assign = std::dynamic_pointer_cast<AssignmentNode>(node);
if (targetLanguage == "C++")
    return assign->variableName + " = " + generate(assign->expression) + ";";
else if (targetLanguage == "Python")
    return assign->variableName + " = " + generate(assign->expression);
```

3. Виведення повідомлення

Вузол OutputNode перетворюється у команду виводу:

std::cout << "Привіт"; – для C++

print("Привіт") – для Python

Код вузла:

```
if (targetLanguage == "C++")
    return "std::cout << " + output->message + ";";
else if (targetLanguage == "Python")
    return "print(" + output->message + ")";
```

4. Умовні конструкції

Вузол ConditionNode генерує умовну конструкцію з двома гілками — основною (if) та додатковою (else, за наявності):

Код вузла:

```
if (targetLanguage == "C++") {
    std::string result = "if (" + condExpr + ") {\r\n        " + bodyCode + "    }";
    if (!conditionNode->falseBranch.empty()) {
        result += "\r\n    else {\r\n        " + elseBody + "    }";
    }
    return result;
}
```

Метод generateAll приймає список інструкцій (вузлів AST) і формує повноцінну програму або просто послідовність операторів, залежно від параметра wrapAsProgram.

```
    std::string CodeGenerator::generateAll(const
std::vector<std::shared_ptr<ASTNode>>& nodes) {
        std::string code;
        std::string result;

        if (wrapAsProgram) {
            if (targetLanguage == "C++") {
                for (auto& node : nodes) {
                    code += "    " + generate(node) + "\r\n";
                }
                result += R"(using namespace System;)";
                result += "\r\n\r\n";
                result += R"(void main(array<String^>^ args) {)";
                result += "\r\n";
                result += code + "}";
                return result;
            }
            else if (targetLanguage == "Python") {
                for (auto& node : nodes) {
                    code += generate(node) + "\r\n";
                }
                return code;
            }
        }
        else {
            for (auto& node : nodes) {
                code += generate(node) + "\r\n";
            }
            return code;
        }
    }
```

Модуль генерації коду є завершальним етапом трансляції. Він дозволяє інтерпретувати структури, побудовані синтаксичним аналізатором, у повноцінні інструкції мов програмування Python та C++. Завдяки підтримці кількох типів

вузлів AST і можливості налаштування мови виводу, цей компонент легко розширюється для підтримки нових конструкцій або мов.

3.5 Приклади роботи програми

У цьому розділі наведено приклади обробки природномовних конструкцій транслятором. Для кожного прикладу демонструється:

- вхідна команда користувача;
- побудоване абстрактне синтаксичне дерево (AST);
- згенерований програмний код для Python та C++.

Приклад 1: Присвоєння змінній

Вхідна конструкція: x присвоїти 5 + 4

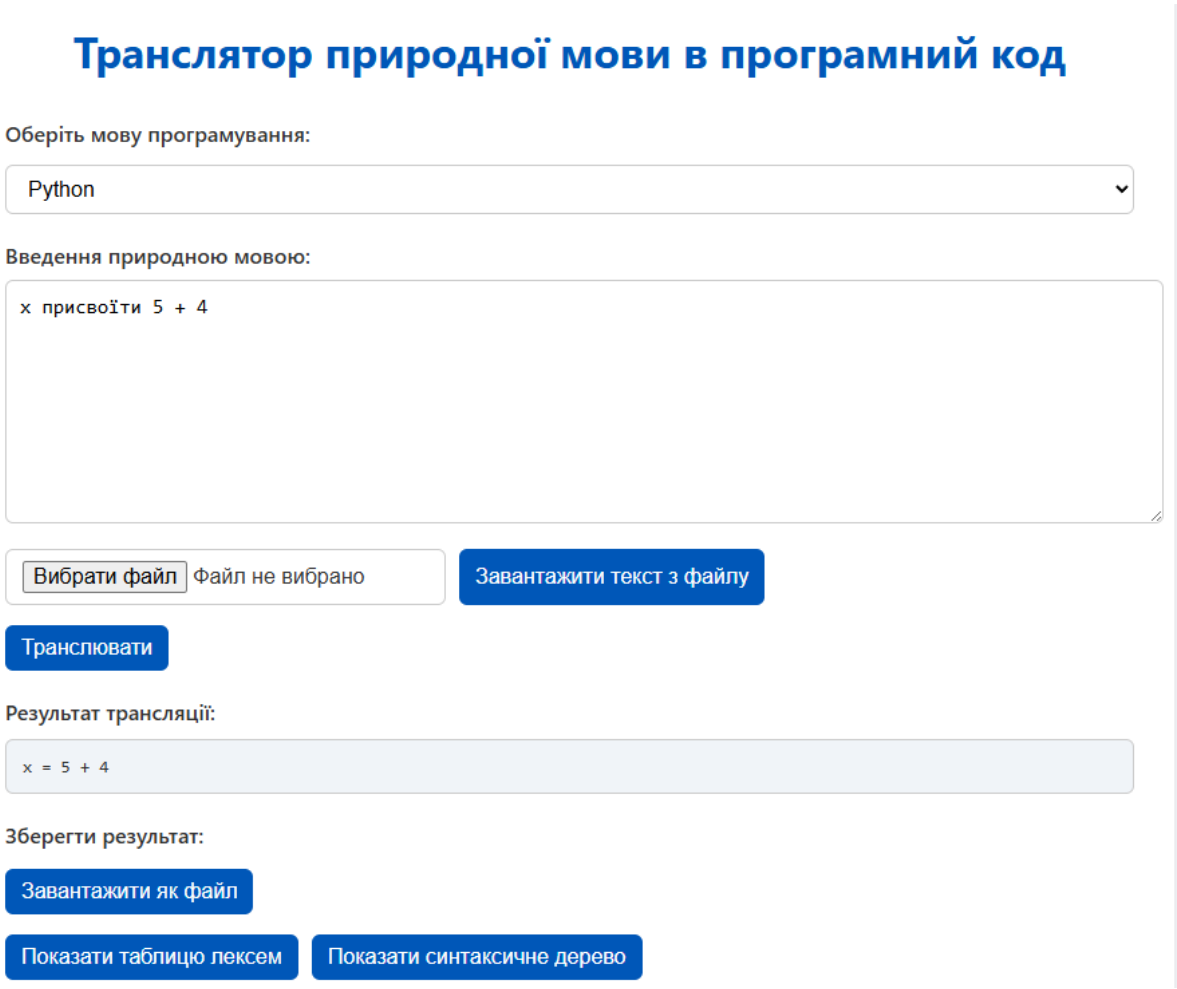


Рисунок 3.1 – Результат трансляції конструкції присвоєння в Python

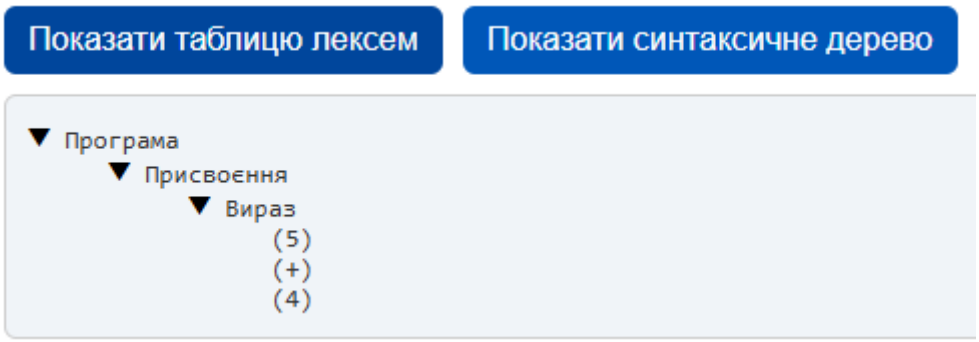


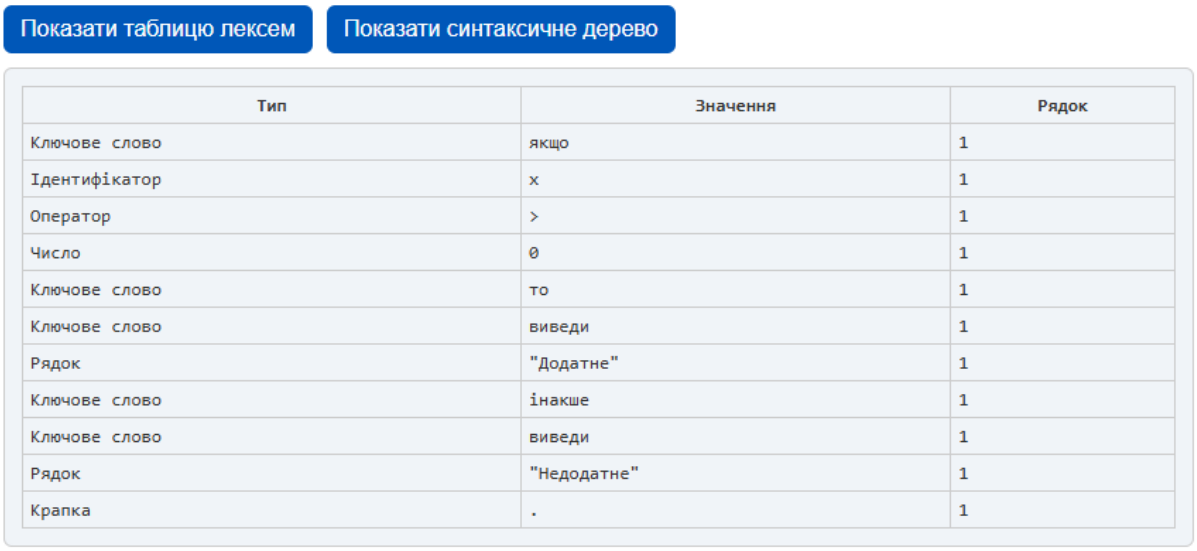
Рисунок 3.2 – Побудоване дерево для операції присвоєння

Ця інструкція має просту структуру: іменована змінна (x) та вираз (5 + 4), результат якого їй присвоюється. Транслятор визначає ключові слова ("присвоїти"), розпізнає змінну та число, після чого будує дерево AST для операції присвоєння.

Приклад 2: Умова (if-else)

Вхідна команда користувача: якщо x більше 0, то виведи "Додатне", інакше виведи "Недодатне".

Таблиця лексем показана на рисунку 3.3.



Тип	Значення	Рядок
Ключове слово	якщо	1
Ідентифікатор	x	1
Оператор	>	1
Число	0	1
Ключове слово	то	1
Ключове слово	виведи	1
Рядок	"Додатне"	1
Ключове слово	інакше	1
Ключове слово	виведи	1
Рядок	"Недодатне"	1
Крапка	.	1

Рисунок 3.3 – Таблиця лексем введеної конструкції

Ця команда містить умовну конструкцію (if-else) з логічним порівнянням і двома гілками виконання. Транслятор визначає порівняльний оператор (більше → >), будуючи логічне піддерево, а також формує гілки trueBranch і falseBranch.

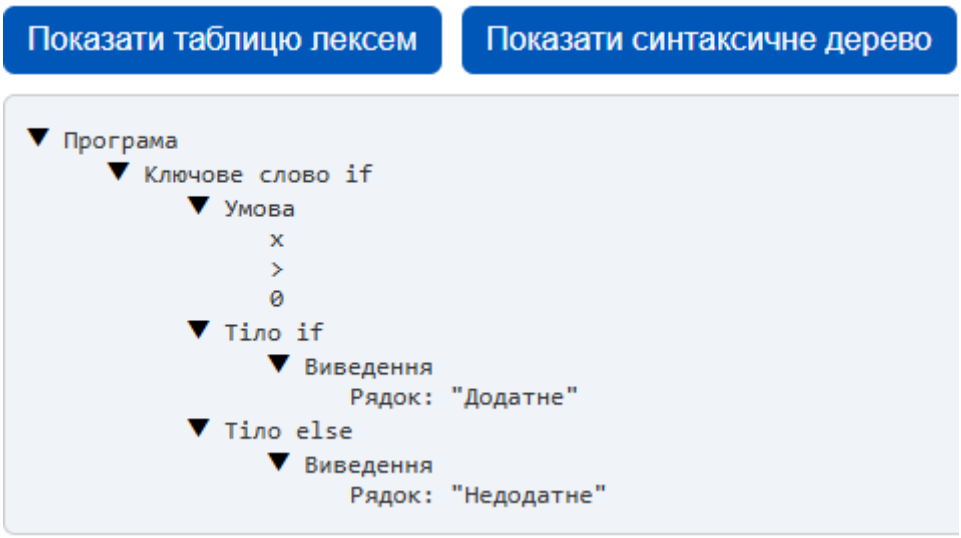


Рисунок 3.4 – Побудоване дерево для умовної конструкції

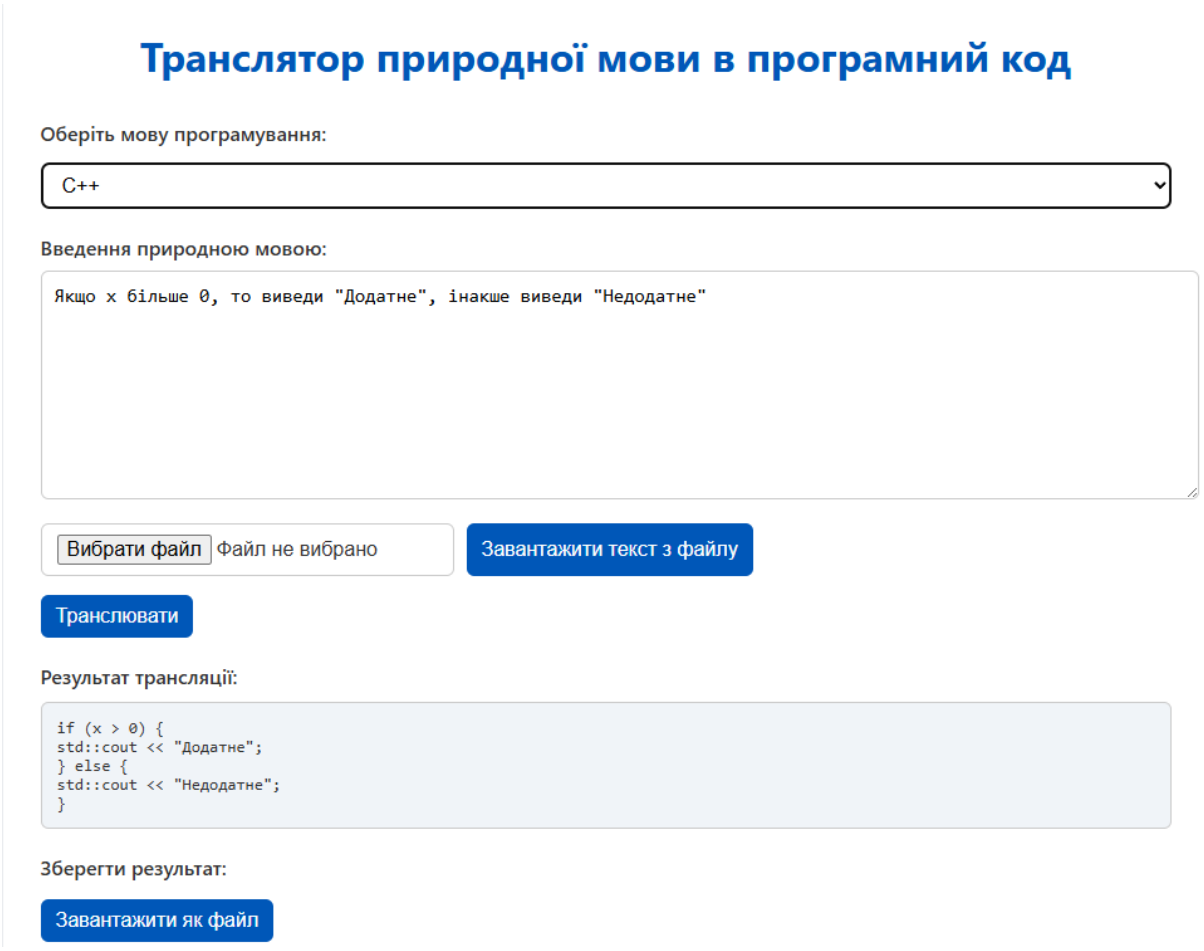


Рисунок 3.5 – Результат трансляції умовної конструкції в C++

Подібні перевірки проводилися для всіх підтримуваних конструкцій: умовних операторів, циклів, логічних виразів, операторів виводу, створення функцій тощо. Для кожного прикладу здійснювалася перевірка як для генерації коду Python, так і для генерації C++.

На наступному етапі здійснювалося тестування системи на стійкість до помилок. Зокрема, вводилися неповні або граматично некоректні конструкції, наприклад: “якщо а б то вивести "а менше b".”. У цьому випадку транслятор не генерував код, натомість виводив повідомлення про помилку: “Помилка: очікувався оператор після ідентифікатора x”. Це підтвердило коректну роботу системи обробки помилок у синтаксичному аналізаторі.

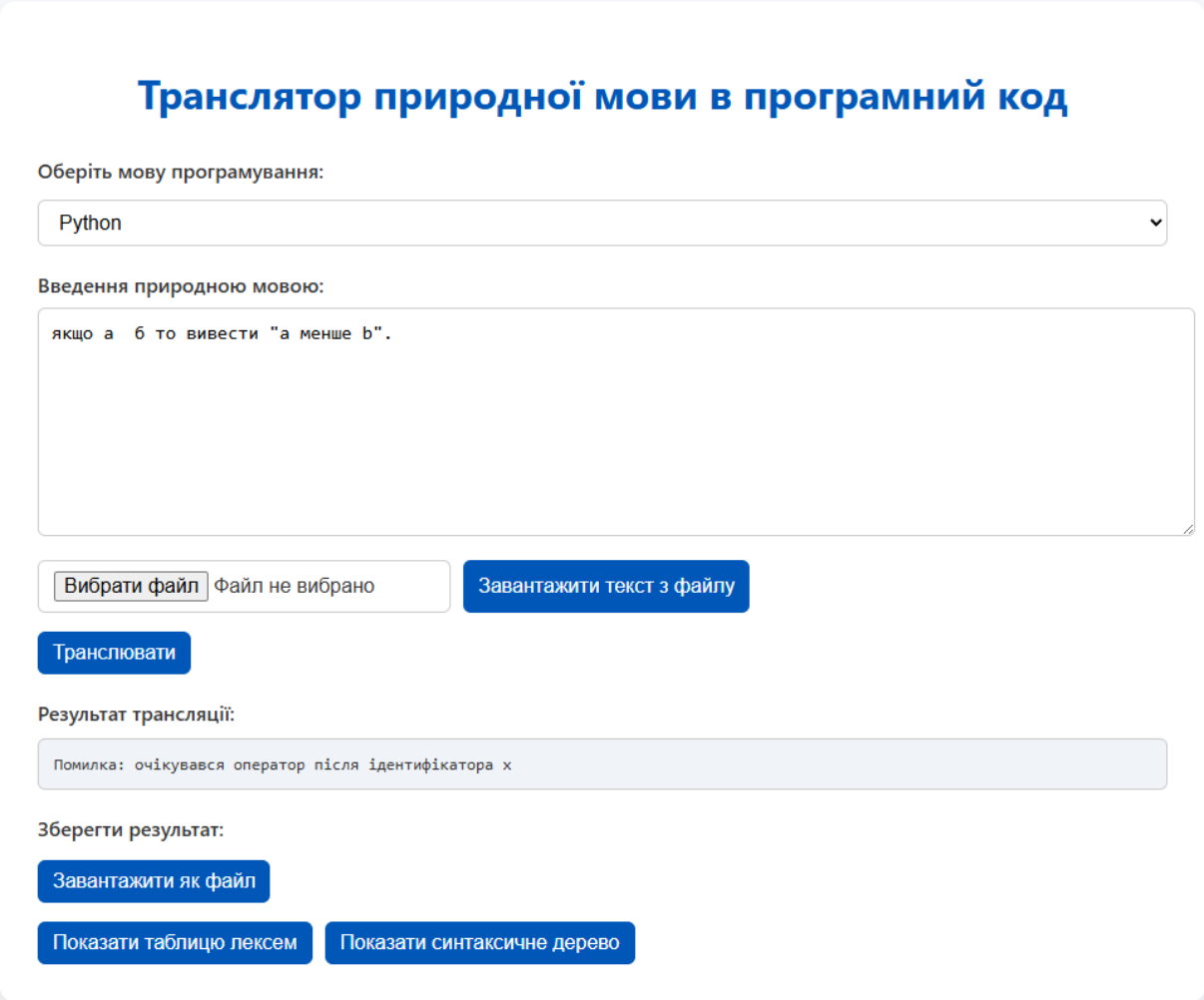


Рисунок 3.5 – Повідомлення про помилку при некоректному вводі

Наведені приклади показують, як транслятор виконує синтаксичний аналіз природномовних інструкцій, будує відповідні вузли AST та генерує програмний код у форматах, що можуть бути скомпільовані або виконані без змін. Такий підхід відкриває можливості для використання інструменту в освітніх цілях, автоматизації простих скриптів або генерації шаблонного коду на основі зрозумілої мови.

Згенерований код додатково тестувався шляхом вставлення у середовище виконання (інтерпретатор Python або компілятор C++) з метою перевірки, чи є він робочим і синтаксично правильним. У більшості випадків код виконувався без зауважень, що свідчить про правильну реалізацію генератора.

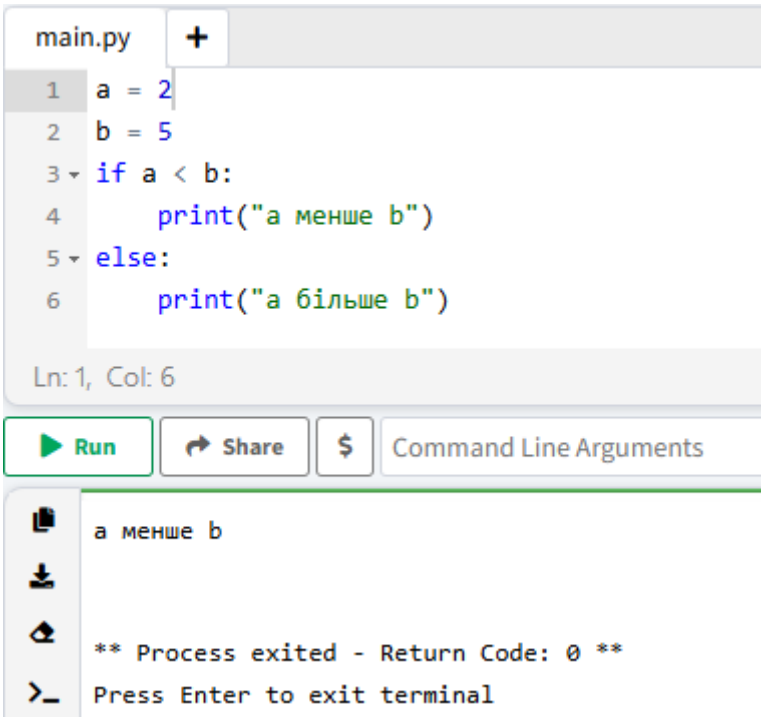


Рисунок 3.6 – Виконання згенерованого коду в інтерпретаторі Python

Проведене тестування показало, що реалізований веб-додаток успішно справляється зі своєю основною функцією — трансляцією природномовних конструкцій у програмний код. Усі ключові модулі (лексичний, синтаксичний аналізатори, генератор коду та інтерфейс користувача) працюють злагоджено та коректно обробляють як звичайні, так і виняткові ситуації.

Система виявилася стійкою до помилок, а її модульна архітектура забезпечила легке виявлення та усунення можливих проблем. Результати підтверджують працездатність та практичну цінність створеного програмного продукту.

ВИСНОВКИ

У результаті виконання дипломної роботи було розроблено та реалізовано програмний транслятор, який здійснює перетворення текстів, написаних природною мовою, у синтаксично коректний програмний код мов Python або C++.

В процесі розробки було досягнуто таких основних результатів:

1. Проведено аналіз предметної області, виявлено актуальність та перспективність підходу трансляції природної мови в код, зокрема в освітньому контексті та для новачків у програмуванні.
2. Розроблено власну формалізовану граматику, яка охоплює ключові конструкції: присвоєння, арифметичні вирази, умовні оператори, логічні вирази, оператори виводу.
3. Реалізовано лексичний та синтаксичний аналізатори, зокрема:

– лексичний аналізатор, що розбиває вхідний текст на лексеми та визначає їх тип;

– синтаксичний аналізатор на основі рекурсивного спуску, який будує дерево розбору та перевіряє коректність структури.

4. Розроблено механізм генерації коду, який формує програмний код відповідно до вибраної мови (Python або C++), з урахуванням типів змінних, синтаксису та особливостей форматування.
5. Створено зручний графічний інтерфейс користувача, що забезпечує:
   - введення природних мовних конструкцій;
   - вибір цільової мови;
   - перегляд таблиці лексем і дерева розбору;
   - виведення результату трансляції.

Розроблена система продемонструвала здатність автоматично аналізувати та інтерпретувати природномовні інструкції, спрощуючи перехід від текстового опису до програмного представлення. Такий підхід має високу освітню цінність, зокрема як демонстраційний інструмент для початківців або як основа для подальшого розвитку в напрямку Natural Language Programming.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Загальна схема роботи компіляторів: веб-сайт. URL: https://studfile.net/preview/5465784/ (дата звернення: 22.03.2025).

2. Ахо Альфред В., Мережі Раві, Ульман Джеффрі Д. Компілятори: принципи, технології і інструменти – М.: Видавничий дім «Вільямс», 2003. 768 с.

3. Робін Хантер. Основні концепції компіляторів. М.: Видавничий дім «Вільямс», 2002. 256 с.

4. Бржезовський А. В., Корсакова Н.В., Фільчаков В.В. Лексичний і синтаксичний аналіз. Формальні мови і граматики. Л.: ЛІАП, 1990. 342 с.

5. Люіс Ф. і ін. Теоретичні основи побудови компіляторів. М.: Мир, 1979. 654 с.

6. Aho A., Ullman J. Theory of syntactic analysis, translation and compilation. Prentice Hall, Inc, 1973, Vol. 1. 483 p.

7. Хопкрофт Д., Мотвані Р., Ульман Дж. Введення в теорію автоматів, мов та обчислень . К.: Видавництво "Вільямс", 2002. 480 с.

8. Grinberg M. Flask Web Development. O'Reilly Media, Incorporated, 2014.

9. Wirth N. Compiler Construction. Addison Wesley, 1996. 176 p.

10. Compilers, Principles, Technics, & Tools / Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, 2nd edition. Addison Wesley, 2006. 1040 p.

11. Maia I. Building Web Applications with Flask. Packt Publishing, 2015. 160 p.

12. Siek G. Essentials of Compilation: An Incremental Approach in Python. MIT Press, 2023.

ДОДАТКИ

Додаток А

Index.html

```
<!DOCTYPE html>
<html lang="uk">
<head>
    <meta charset="UTF-8">
    <title>Транслятор природної мови</title>
    <style>
        body {
            margin: 0;
            font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
            background: #f7f9fc;
            color: #333;
            padding: 20px;
        }

        .container {
            max-width: 900px;
            margin: auto;
            background: #ffffff;
            border-radius: 12px;
            padding: 30px;
            box-shadow: 0 8px 20px rgba(0, 0, 0, 0.1);
        }

        h1 {
            text-align: center;
            color: #0057b8;
            margin-bottom: 30px;
        }

        label {
            font-weight: 600;
            display: block;
            margin-top: 20px;
            margin-bottom: 8px;
        }

        select, input[type="file"], button {
            font-size: 16px;
            padding: 8px 12px;
            border: 1px solid #ccc;
            border-radius: 6px;
            margin-top: 5px;
        }

        select {
            width: 100%;
        }

        button {
            background-color: #0057b8;
            color: white;
            border: none;
            cursor: pointer;
            transition: background-color 0.2s ease;
        }

            button:hover {
                background-color: #003f91;
            }

        textarea {
```

```
            width: 100%;
            height: 160px;
            border-radius: 6px;
            border: 1px solid #ccc;
            padding: 10px;
            font-family: monospace;
            font-size: 15px;
            resize: vertical;
        }

        .output {
            background: #f0f4f8;
            border: 1px solid #ccc;
            padding: 12px;
            border-radius: 6px;
            margin-top: 10px;
            font-family: monospace;
        }

        .btn-group {
            display: flex;
            flex-wrap: wrap;
            gap: 10px;
            margin-top: 10px;
        }
        ul, #myUL {
            list-style-type: none;
        }

        #myUL {
            margin: 0;
            padding: 0;
        }

        .caret {
            cursor: pointer;
            -webkit-user-select: none; /* Safari 3.1+ */
            -moz-user-select: none; /* Firefox 2+ */
            -ms-user-select: none; /* IE 10+ */
            user-select: none;
        }

            .caret::before {
                content: "\25B6";
                color: black;
                display: inline-block;
                margin-right: 6px;
            }

        .caret-down::before {
            -ms-transform: rotate(90deg); /* IE 9 */
            -webkit-transform: rotate(90deg); /* Safari */
            ' transform: rotate(90deg);
        }

        .nested {
            display: none;
        }

        .active {
            display: block;
        }
    </style>
</head>
```

```
    <body>

        <div class="container">
            <h1>Транслятор природної мови в програмний код</h1>

            <label for="languageSelect">Оберіть мову програмування:</label>
            <select id="languageSelect">
                <option value="python">Python</option>
                <option value="cpp">C++</option>
            </select>

            <label for="inputText">Введення природною мовою:</label>
            <textarea id="inputText" placeholder="Наприклад: створити змінну x
присвоїти 5"></textarea>

            <div class="btn-group">
                <input type="file" id="loadTextFile" accept=".txt">
                <button onclick="loadTextFromFile()">Завантажити текст з
файлу</button>
            </div>

            <div class="btn-group">
                <button onclick="translateText()">Транслювати</button>
            </div>

            <label>Результат трансляції:</label>
            <div id="output" class="output">
                if (x > 0) {<br />
                std::cout << "Додатне";<br />
                } else {<br />
                std::cout << "Недодатне";<br />
                }
            </div>

            <label>Зберегти результат:</label>
            <div class="btn-group">
                <button onclick="downloadCode()">Завантажити як файл</button>
            </div>

            <div class="btn-group">
                <button onclick="showLexemeTable()">Показати таблицю
лексем</button>
                <button onclick="showSyntaxTree()">Показати синтаксичне
дерево</button>
            </div>

            <div id="lexemeTable" class="output" style="display: none;"></div>
            <div id="syntaxTree" class="output" style="display: none;">

            </div>

        </div>

        <script>
            function translateText() {
                const inputText = document.getElementById("inputText").value;
                const language =
document.getElementById("languageSelect").value;

                fetch("/translate", {
                    method: "POST",
                    headers: { "Content-Type": "application/json" },
                    body: JSON.stringify({ text: inputText, language: language
})
```

```
                })
                    .then(res => res.json())
                    .then(data => {
                        document.getElementById("output").textContent =
data.code || "Помилка трансляції";
                    })
                    .catch(() => {
                        document.getElementById("output").textContent = "Помилка
з'єднання з сервером";
                    });
            }

            function loadTextFromFile() {
                const fileInput = document.getElementById("loadTextFile");
                const reader = new FileReader();
                reader.onload = () => {
                    document.getElementById("inputText").value = reader.result;
                };
                if (fileInput.files[0]) {
                    reader.readAsText(fileInput.files[0], 'UTF-8');
                }
            }

            function downloadCode() {
                const code = document.getElementById("output").textContent;
                const language =
document.getElementById("languageSelect").value;
                const ext = language === "python" ? "py" : "cpp";
                const blob = new Blob([code], { type: "text/plain;charset=utf-8"
});
                const a = document.createElement("a");
                a.href = URL.createObjectURL(blob);
                a.download = `translated_code.${ext}`;
                a.click();
            }

            function loadCodeFile() {
                const fileInput = document.getElementById("uploadCodeFile");
                const reader = new FileReader();
                reader.onload = () => {
                    document.getElementById("output").textContent =
reader.result;
                };
                if (fileInput.files[0]) {
                    reader.readAsText(fileInput.files[0], 'UTF-8');
                }
            }
            function showLexemeTable() {
                const tableHTML = ``;

                document.getElementById("lexemeTable").innerHTML = tableHTML;
                document.getElementById("lexemeTable").style.display = "block";
                document.getElementById("syntaxTree").style.display = "none";
            }

            function showSyntaxTree() {

                const treeHTML = ``;
                document.getElementById("syntaxTree").innerHTML = treeHTML;
                document.getElementById("syntaxTree").style.display = "block";
                document.getElementById("lexemeTable").style.display = "none";
                var toggler = document.getElementsByClassName("caret");
                var i;
```

```
                for (i = 0; i < toggler.length; i++) {
                    toggler[i].addEventListener("click", function () {

this.parentElement.querySelector(".nested").classList.toggle("active");
                        this.classList.toggle("caret-down");
                    });
                }
            }

        </script>

    </body>
    </html>
```

Додаток Б
App.py

```
# -*- coding: utf-8 -*-

from flask import Flask, request, jsonify, send_from_directory
from flask_cors import CORS
import subprocess
import tempfile
import os

app = Flask(__name__, static_folder='../frontend', static_url_path='/')
CORS(app)

TRANSLATOR_PATH = "../translator/NL2CodeConsole.exe"

@app.route('/')
def serve_index():
    return send_from_directory(app.static_folder, 'index.html')

@app.route('/translate', methods=['POST'])
def translate():
    data = request.get_json()
    if not data or 'text' not in data:
        return jsonify({"error": "Немає тексту для трансляції"}), 400

    user_input = data['text']

    with tempfile.NamedTemporaryFile(mode='w+', delete=False, suffix='.txt',
encoding='utf-8') as temp_in:
        temp_in.write(user_input)
        temp_in.flush()
        temp_input_path = temp_in.name


    with tempfile.NamedTemporaryFile(mode='r', delete=False, suffix='.txt')
as temp_out:
        temp_output_path = temp_out.name

    try:
        result = subprocess.run(
            [TRANSLATOR_PATH, temp_input_path, temp_output_path],
            check=True,
            stdout=subprocess.PIPE,
```

```
            stderr=subprocess.PIPE,
            text=True
)
        print("[DEBUG] Translator stdout:")
        print(result.stdout)  # вивід дебагу в консоль сервера
        print("[DEBUG] Translator stderr:")
        print(result.stderr)

        with open(temp_output_path, 'r', encoding='utf-8') as f:
            generated_code = f.read()

        return jsonify({"code": generated_code})

    except subprocess.CalledProcessError as e:
        return jsonify({"error": f"Помилка транслятора: {e.stderr}"}), 500

    finally:
        os.remove(temp_input_path)
        os.remove(temp_output_path)

if __name__ == '__main__':
    app.run(debug=True)
```

Додаток В

translator_wrapper.py

```
import subprocess
import tempfile
import os

def run_translator(input_text):
    with tempfile.NamedTemporaryFile(delete=False, suffix='.txt', mode='w+',
encoding='utf-8') as input_file:
        input_file.write(input_text)
        input_file_path = input_file.name

    with tempfile.NamedTemporaryFile(delete=False, suffix='.txt', mode='r',
encoding='utf-8') as output_file:
        output_file_path = output_file.name

    translator_exe_path = "../translator/NL2CodeConsole.exe"  # або
translator.out для Linux

    try:
        result = subprocess.run(
            [translator_exe_path, input_file_path, output_file_path],
            capture_output=True,
            text=True
        )
        if result.returncode != 0:
            return f"Error: {result.stderr}"

        with open(output_file_path, 'r', encoding='utf-8') as f:
            return f.read()

    finally:
        os.remove(input_file_path)
        os.remove(output_file_path)
```

Кінець додатку В

Додаток Г
Token.h

```
// Token.h
#pragma once
#include <string>

enum class LexicalTokenType {
    Keyword,
    Identifier,
    Operator,
    Number,
    Bool,
    BoolOperator,
    ComparisonOperator,
    StringLiteral,
    RomanNumber,
    Assignment,
    Period,
    OpeningBracket,
    ClosingBracket,
    End,
    Unknown
};

// Окремо функції
inline std::string tokenTypeToString(LexicalTokenType type) {
    switch (type) {
    case LexicalTokenType::Keyword: return "Keyword";
    case LexicalTokenType::Identifier: return "Identifier";
    case LexicalTokenType::Operator: return "Operator";
    case LexicalTokenType::Number: return "Number";
    case LexicalTokenType::Bool: return "Bool";
    case LexicalTokenType::BoolOperator: return "BoolOperator";
    case LexicalTokenType::ComparisonOperator: return "ComparisonOperator";
    case LexicalTokenType::StringLiteral: return "StringLiteral";
    case LexicalTokenType::RomanNumber: return "RomanNumber";
    case LexicalTokenType::Assignment: return "Assignment";
    case LexicalTokenType::Period: return "Period";
    case LexicalTokenType::OpeningBracket: return "OpeningBracket";
    case LexicalTokenType::ClosingBracket: return "ClosingBracket";
    case LexicalTokenType::End: return "End";
    case LexicalTokenType::Unknown: return "Unknown";
    default: return "Unknown";
    }
}

inline std::string tokenToString(const LexicalTokenType& type, const
std::string& value) {
    return "{ \"type\": \"" + tokenTypeToString(type) + "\", \"value\": \""
+ value + "\" }";
}

// Структура
struct Token {
    LexicalTokenType type;
    std::string value;

    std::string toString() const {
        return tokenToString(type, value);
    }
};
```

Lexer.cpp

```
#include "Lexer.h"
```

```
#include <regex>
#include <codecvt>
#include <locale>

const std::unordered_set<std::wstring> Lexer::keywords{
    L"присвоїти", L"створити", L"вивести", L"якщо", L"то", L"інакше",
L"повторити",
    L"від", L"до", L"крок", L"робити", L"для кожного", L"функція",
L"оформи",
    L"у", L"вигляді", L"програми"
};

const std::unordered_set<std::wstring> Lexer::operators{ L"+", L"-", L"*",
L"/" };
const std::unordered_set<std::wstring> Lexer::boolOperators{ L"і", L"або",
L"не" };
const std::unordered_set<std::wstring> Lexer::comparisonOperators{
L"більше", L"менше", L"дорівнює" };
const std::unordered_set<std::wstring> Lexer::boolValues{ L"істинне",
L"хибне" };



std::vector<Token> Lexer::analyze(const std::wstring& input) {
    std::vector<Token> tokens;

    std::wregex tokenRegex(LR"((for|do|:=|[<>]=?|[+\-
*/().]|"(?:[^"\\]|\\.)*"|[a-zA-Za-яА-ЯіїєІЇЄ_][a-zA-Za-яА-ЯіїєІЇЄ0-
9_]*|[IVXLCDM]+|\d+|;))");
    auto words_begin = std::wsregex_iterator(input.begin(), input.end(),
tokenRegex);
    auto words_end = std::wsregex_iterator();

    std::wstring_convert<std::codecvt_utf8<wchar_t>> converter;

    for (auto it = words_begin; it != words_end; ++it) {
        std::wstring word = it->str();
        std::string utf8Word = converter.to_bytes(word); // ← Конвертація

        if (keywords.find(word) != keywords.end()) {
            tokens.push_back({ LexicalTokenType::Keyword, utf8Word });
        }
        else if (word == L":=") {
```

```
                tokens.push_back({ LexicalTokenType::Assignment, utf8Word });
            }
            else if (operators.find(word) != operators.end()) {
                tokens.push_back({ LexicalTokenType::Operator, utf8Word });
            }
            else if (boolOperators.find(word) != boolOperators.end()) {
                tokens.push_back({ LexicalTokenType::BoolOperator, utf8Word });
            }
            else if (comparisonOperators.find(word) !=
comparisonOperators.end()) {
                tokens.push_back({ LexicalTokenType::ComparisonOperator,
utf8Word });
            }
            else if (boolValues.find(word) != boolValues.end()) {
                tokens.push_back({ LexicalTokenType::Bool, utf8Word });
            }
            else if (std::regex_match(utf8Word, std::regex("^\\d+$"))) {
                tokens.push_back({ LexicalTokenType::Number, utf8Word });
            }
            else if (word == L".") {
                tokens.push_back({ LexicalTokenType::Period, utf8Word });
            }
            else if (word == L"(") {
                tokens.push_back({ LexicalTokenType::OpeningBracket, utf8Word
});
            }
            else if (word == L")") {
                tokens.push_back({ LexicalTokenType::ClosingBracket, utf8Word
});
            }
            else if (std::regex_match(utf8Word, std::regex("^\".*\"$"))) {
                tokens.push_back({ LexicalTokenType::StringLiteral, utf8Word });
            }
            else if (std::regex_match(utf8Word, std::regex("^[a-zA-Za-яA-
ЯiїєIЇЄ_][a-zA-Za-яA-ЯiїєIЇЄ0-9_]*$"))) {
                tokens.push_back({ LexicalTokenType::Identifier, utf8Word });
            }
            else {
                tokens.push_back({ LexicalTokenType::Unknown, utf8Word });
            }
        }
        tokens.push_back({ LexicalTokenType::End, "$" });
```

```
        return tokens;
    }

    Lexer::Lexer()
    {
    }
```

Parser.cpp

```
    #include "Parser.h"
    #include <stdexcept>
    #include <unordered_map>
    #include <memory>
    #include "CodeGenerator.h"

    Parser::Parser(const std::vector<Token>& lexems) {
        tokens = lexems;
    }

    bool Parser::match(const std::vector<Token>& statement, LexicalTokenType
type, const std::string& value) {
        if (pos < statement.size() &&
            statement[pos].type == type &&
            (value.empty() || statement[pos].value == value)) {
            pos++;
            return true;
        }
        return false;
    }

    bool Parser::matchIdentifierOrNumber(const std::vector<Token>& statement) {
        return match(statement, LexicalTokenType::Identifier) ||
match(statement, LexicalTokenType::Number);
    }

    std::shared_ptr<ASTNode> Parser::parseStatement(const std::vector<Token>&
statement) {
        pos = 0;
        if (match(statement, LexicalTokenType::Identifier) &&
            match(statement, LexicalTokenType::Keyword, "присвоїти")) {
            if (isExpression(statement) ||
                statement[pos].type == LexicalTokenType::StringLiteral) {
                exprPos = 2;
```

```
                auto expressionNode = parseExpression(statement);

                if
(std::dynamic_pointer_cast<BinaryOperationNode>(expressionNode) ||
                    std::dynamic_pointer_cast<NumberNode>(expressionNode) ||

std::dynamic_pointer_cast<StringLiteralNode>(expressionNode)) {
                    auto node =
std::make_shared<AssignmentNode>(statement[0].value, expressionNode);
                    return node;
                }
                else {
                    std::string result = "Error";
                }
            }
        }
        else if (match(statement, LexicalTokenType::Keyword, "вивести")) {
            auto node = std::make_shared<OutputNode>(statement[1].value);
            //node->message = statement[1].value;
            //node->type = ASTNodeType::Output;
            pos++;
            return node;
        }
        else if (match(statement, LexicalTokenType::Keyword, "якщо")) {
            if (isBoolExpression(statement)) {
                auto expressionNode = parseBoolExpression(statement);
                // Очікуємо ключове слово "то"
                if (!match(statement, LexicalTokenType::Keyword, "то")) {
                    throw std::runtime_error("Очікувалося 'то' після умови");
                }
                // Парсимо тіло умови (до "інакше" або кінця)
                std::vector<Token> trueBranchTokens;
                std::vector<std::shared_ptr<ASTNode>> trueBranchNodes;
                size_t startPos = pos + 1;

                while (pos < statement.size() && statement[pos].value !=
"інакше") {
                    auto innerStatement = std::vector<Token>(statement.begin() +
pos, statement.end());
                    trueBranchTokens = extractStatementTokens(innerStatement);
                    trueBranchNodes.push_back(parseStatement(trueBranchTokens));
                    pos += startPos;
```

```
                }
                std::vector<std::shared_ptr<ASTNode>> falseBranchNodes;
                if (match(statement, LexicalTokenType::Keyword, "інакше")) {
                    std::vector<Token> falseBranchTokens;
                    startPos = pos + 1;
                    while (pos < statement.size()) {
                        falseBranchTokens =
extractStatementTokens(std::vector<Token>(statement.begin() + pos,
statement.end()));

falseBranchNodes.push_back(parseStatement(falseBranchTokens));
                        pos += startPos;
                    }
                }

                if
(std::dynamic_pointer_cast<BinaryOperationNode>(expressionNode) ||
                    std::dynamic_pointer_cast<BoolNode>(expressionNode)) {
                    // Формуємо вузол умови
                    auto conditionNode = std::make_shared<ConditionNode>();
                    conditionNode->condition = expressionNode;
                    conditionNode->trueBranch = trueBranchNodes;
                    conditionNode->falseBranch = falseBranchNodes;
                    pos++;
                    return conditionNode;
                }
            }
        }
        else if (statement[0].value == "повторити") {
            std::vector<std::shared_ptr<ASTNode>> loopBody;
            auto node = std::make_shared <LoopNode>(statement[1].value,
statement[3].value, statement[5].value, loopBody);
            /*node->variable = statement[1].value;
            node->rangeStart = statement[3].value;
            node->rangeEnd = statement[5].value;*/
            // TODO: парсинг тіла циклу
            return node;
        }
        // TODO: інші конструкції
        return nullptr;
    }
```

```
bool Parser::isExpression(const std::vector<Token>& statement) {
    size_t startPos = pos;
    int bracketCount = 0;
    bool expectOperand = true;

    while (pos < statement.size()) {
        auto& token = statement[pos];

        if (match(statement, LexicalTokenType::OpeningBracket)) {
            bracketCount++;
        }
        else if (match(statement, LexicalTokenType::ClosingBracket)) {
            if (bracketCount == 0) return false;
            bracketCount--;
            expectOperand = false;
        }
        else if (expectOperand && matchIdentifierOrNumber(statement)) {
            expectOperand = false;
        }
        else if (expectOperand && match(statement,
LexicalTokenType::StringLiteral)) {
            expectOperand = false;
        }
        else if (!expectOperand && match(statement,
LexicalTokenType::Operator)) {
            expectOperand = true;
        }

        else {
            pos = startPos; // якщо не вдалося — відкат позиції
            return false;
        }
    }
    if (bracketCount == 0)
        return true;
    else
        return false; // незакриті дужки

}

bool Parser::isBoolExpression(const std::vector<Token>& statement) {
    size_t startPos = pos;
```

```
        int bracketCount = 0;
        bool expectOperand = true;
        bool lastWasBoolLiteral = false;

        while (statement[startPos].type != LexicalTokenType::Keyword && startPos
< statement.size()) {
            const Token& token = statement[startPos];

            if (token.type == LexicalTokenType::OpeningBracket) {
                bracketCount++;
                startPos++;
            }
            else if (token.type == LexicalTokenType::ClosingBracket) {
                if (bracketCount == 0) return false;
                bracketCount--;
                startPos++;
            }
            else if (token.type == LexicalTokenType::Bool) {
                if (!expectOperand) return false;
                if (lastWasBoolLiteral) return false; // два bool підряд
                lastWasBoolLiteral = true;
                expectOperand = false;
                startPos++;
            }
            else if (token.type == LexicalTokenType::Identifier || token.type ==
LexicalTokenType::Number) {
                if (!expectOperand) return false;
                lastWasBoolLiteral = false;
                expectOperand = false;
                startPos++;

                // очікуємо оператор порівняння після змінної або числа
                if (startPos >= statement.size()) return false;
                if (statement[startPos].type !=
LexicalTokenType::ComparisonOperator) return false;
                startPos++;

                // очікуємо число або змінну після оператора порівняння
                if (startPos >= statement.size()) return false;
                if (statement[startPos].type != LexicalTokenType::Identifier &&
statement[startPos].type != LexicalTokenType::Number)
                    return false;
```

```
                startPos++;

            }
            else if (token.type == LexicalTokenType::BoolOperator) {
                if (expectOperand) return false;
                expectOperand = true;
                lastWasBoolLiteral = false;
                startPos++;
            }
            else {
                // будь-який інший токен — помилка
                return false;
            }
        }

        return bracketCount == 0 && !expectOperand;
    }

    // Парсинг виразу з операціями + і -
    std::shared_ptr<ASTNode> Parser::parseExpression(const std::vector<Token>&
statement) {
        auto left = parseTerm(statement);

        while (exprPos < statement.size() &&
            (statement[exprPos].value == "+" || statement[exprPos].value == "-
")) {
            std::string op = statement[exprPos].value;
            exprPos++;
            auto right = parseTerm(statement);
            left = std::make_shared<BinaryOperationNode>(op, left, right);
        }

        return left;
    }

    // Парсинг множення і ділення
    std::shared_ptr<ASTNode> Parser::parseTerm(const std::vector<Token>&
statement) {
        auto left = parseFactor(statement);

        while (exprPos < statement.size() &&
```

```
            (statement[exprPos].value == "*" || statement[exprPos].value ==
"/")) {
                std::string op = statement[exprPos].value;
                exprPos++;
                auto right = parseFactor(statement);
                left = std::make_shared<BinaryOperationNode>(op, left, right);
            }

            return left;
        }

        // Парсинг числа, змінної або дужок
        std::shared_ptr<ASTNode> Parser::parseFactor(const std::vector<Token>&
statement) {
            if (exprPos >= statement.size()) return nullptr;

            Token token = statement[exprPos];

            if (token.type == LexicalTokenType::Number) {
                exprPos++;
                return std::make_shared<NumberNode>(token.value);
            }
            else if (token.type == LexicalTokenType::Identifier) {
                exprPos++;
                return std::make_shared<NumberNode>(token.value); // або спеціальний
VariableNode
            }
            else if (token.type == LexicalTokenType::StringLiteral) {
                exprPos++;
                return std::make_shared<StringLiteralNode>(token.value);
            }
            else if (token.type == LexicalTokenType::OpeningBracket) {
                exprPos++; // пропустити (
                auto node = parseExpression(statement);
                if (exprPos < statement.size() && statement[exprPos].type ==
LexicalTokenType::ClosingBracket) {
                    exprPos++; // пропустити )
                }
                else {
                    // Помилка: не знайдено )
                }
                return node;
```

```
        }

        return nullptr;
    }

    std::shared_ptr<ASTNode> Parser::parseBoolExpression(const
std::vector<Token>& statement) {
        auto left = parseComparison(statement); // або parsePrimary()

        while (match(statement, LexicalTokenType::BoolOperator, "і") ||
match(statement, LexicalTokenType::BoolOperator, "або")) {
            std::string op = statement[pos - 1].value;
            auto right = parseComparison(statement);

            auto node = std::make_shared<BinaryOperationNode>(op, left, right);
            left = node;
        }

        return left;
    }

    std::shared_ptr<ASTNode> Parser::parseComparison(const std::vector<Token>&
statement) {
        auto left = parsePrimary(statement);

        if (match(statement, LexicalTokenType::ComparisonOperator)) {
            std::string op = "";
            if (statement[pos - 1].value == "більше") op = ">";
            else if (statement[pos - 1].value == "менше") op = "<";
            else if (statement[pos - 1].value == "дорівнює") op = "=";
            auto right = parsePrimary(statement);

            auto node = std::make_shared<BinaryOperationNode>(op, left, right);
            return node;
        }

        return left;
    }

    std::shared_ptr<ASTNode> Parser::parsePrimary(const std::vector<Token>&
statement) {
        if (match(statement, LexicalTokenType::Bool)) {
```

```
            return std::make_shared<BoolNode>(statement[pos - 1].value);
        }
        else if (matchIdentifierOrNumber(statement)) {
            return std::make_shared<NumberNode>(statement[pos - 1].value); //// 
або спеціальний VariableNode
        }
        else if (match(statement, LexicalTokenType::OpeningBracket)) {
            auto expr = parseBoolExpression(statement);
            if (match(statement, LexicalTokenType::ClosingBracket)) {
                return expr;
            }
        }

        throw std::runtime_error("Очікувався вираз");
    }

    std::vector<Token> Parser::extractStatementTokens(const std::vector<Token>& 
statement) {
        size_t i = 0;
        std::vector<Token> statementTokens;
        int depth = 0;
        bool insideComplexBlock = false;
        while (i < statement.size()) {
            if (statement[i].type == LexicalTokenType::Keyword) {
                if (statement[i].value == "якщо" || statement[i].value == 
"повторити") {
                    depth++;
                    insideComplexBlock = true;
                }
            }
            if (statement[i].value == "." && depth == 0) {
                i++; // пропускаємо крапку
                break;
            }
            // Якщо закінчується вкладена конструкція — зменшуємо глибину
            if (depth > 0 && statement[i].value == "." && statement[i + 1].value 
== ".") {
                depth--;
            }
            statementTokens.push_back(statement[i]);
            i++;
        }
```

```
        return statementTokens;
    }

    std::vector<std::shared_ptr<ASTNode>> Parser::parse() {
        pos = 0;
        std::vector<std::shared_ptr<ASTNode>> nodes;
        std::vector<Token> unparsedTokens = std::vector<Token>(tokens.begin() +
pos, tokens.end());

        while (pos < unparsedTokens.size() - 1) {
            unparsedTokens = std::vector<Token>(unparsedTokens.begin() + pos,
unparsedTokens.end());
            auto statementTokens = extractStatementTokens(unparsedTokens);
            if (!statementTokens.empty()) {
                auto node = parseStatement(statementTokens);
                if (node != nullptr) {
                    nodes.push_back(node);
                }
                pos++;
            }
        }

        return nodes;
    }
```

AST.h

```
    #pragma once
    #include <string>
    #include <vector>

    enum class ASTNodeType {
        Assignment,
        Number,
        StringLiteral,
        Bool,
        BinaryOperation,
        Arithmetic,
        Condition,
        Loop,
        Output
    };

    class ASTNode {
```

```
    public:
        ASTNodeType type;
        virtual ~ASTNode() = default;
    };

    class AssignmentNode : public ASTNode {
    public:
        std::string variableName;
        std::shared_ptr<ASTNode> expression;

        AssignmentNode(const std::string& varName, std::shared_ptr<ASTNode>
expr) {
            type = ASTNodeType::Assignment;
            variableName = varName;
            expression = expr;
        }
    };
    class StringLiteralNode : public ASTNode {
    public:
        std::string value;

        StringLiteralNode(const std::string& val) {
            type = ASTNodeType::StringLiteral;
            value = val;
        }
    };

    class NumberNode : public ASTNode {
    public:
        std::string value;

        NumberNode(const std::string& val) {
            type = ASTNodeType::Number;
            value = val;
        }
    };

    class BinaryOperationNode : public ASTNode {
    public:
        std::string op;
        std::shared_ptr<ASTNode> left;
        std::shared_ptr<ASTNode> right;
```

```
        BinaryOperationNode(const std::string& oper, std::shared_ptr<ASTNode>
lhs, std::shared_ptr<ASTNode> rhs) {
            type = ASTNodeType::BinaryOperation;
            op = oper;
            left = lhs;
            right = rhs;
        }
    };

    class BoolNode : public ASTNode {
    public:
        std::string value;

        BoolNode(const std::string& val) {
            type = ASTNodeType::Bool;
            value = val;
        }
    };

    class ConditionNode : public ASTNode {
    public:
        std::shared_ptr<ASTNode> condition;
        std::vector<std::shared_ptr<ASTNode>> trueBranch;
        std::vector<std::shared_ptr<ASTNode>> falseBranch;

        ConditionNode() {
            type = ASTNodeType::Condition;
        }
    };

    class LoopNode : public ASTNode {
    public:
        std::string variable;
        std::string rangeStart;
        std::string rangeEnd;
        std::vector<std::shared_ptr<ASTNode>> body;

        LoopNode(const std::string& var, std::string start,
            std::string end,
            std::vector<std::shared_ptr<ASTNode>> loopBody)
        {
```

```
            type = ASTNodeType::Loop;
            variable = var;
            rangeStart = start;
            rangeEnd = end;
            body = std::move(loopBody);
        }
    };

    class OutputNode : public ASTNode {
    public:
        std::string message;

        OutputNode(const std::string& msg) {
            type = ASTNodeType::Output;
            message = msg;
        }
    };
```

CodeGenerator.cpp

```
    #include "CodeGenerator.h"
    #include <iostream>

    CodeGenerator::CodeGenerator(std::string language, bool wrap)
        : targetLanguage(language), wrapAsProgram(wrap) {
    }

    std::string CodeGenerator::generate(std::shared_ptr<ASTNode> node) {
        switch (node->type) {
        case ASTNodeType::Assignment: {
            auto assign = std::dynamic_pointer_cast<AssignmentNode>(node);
            if (targetLanguage == "C++") {
                return assign->variableName + " = " + generate(assign-
>expression) + ";";
            }
            else if (targetLanguage == "Python") {
                return assign->variableName + " = " + generate(assign-
>expression);
            }
            break;
        }
        case ASTNodeType::BinaryOperation: {
            auto binOp = std::dynamic_pointer_cast<BinaryOperationNode>(node);
```

```
            return "(" + generate(binOp->left) + " " + binOp->op + " " +
generate(binOp->right) + ")";
        }
        case ASTNodeType::Number: {
            auto num = std::dynamic_pointer_cast<NumberNode>(node);
            return num->value;
        }
        case ASTNodeType::StringLiteral: {
            auto str = std::dynamic_pointer_cast<StringLiteralNode>(node);
            return str->value;
        }
        case ASTNodeType::Output: {
            auto output = std::dynamic_pointer_cast<OutputNode>(node);
            if (targetLanguage == "C++") {
                return "std::cout << " + output->message + ";";
            }
            else if (targetLanguage == "Python") {
                return "print(" + output->message + ")";
            }
            break;
        }
        case ASTNodeType::Condition: {
            auto conditionNode = std::dynamic_pointer_cast<ConditionNode>(node);
            if (!conditionNode) return "";

            std::string condExpr = generate(conditionNode->condition);
            std::string bodyCode;

            for (const auto& stmt : conditionNode->trueBranch) {
                bodyCode += generate(stmt);
                if (targetLanguage == "C++" || targetLanguage == "Python")
                    bodyCode += "\r\n";
            }

            if (targetLanguage == "C++") {
                std::string result = "if (" + condExpr + ") {\r\n        " +
bodyCode + "    }";
                if (!conditionNode->falseBranch.empty()) {
                    std::string elseBody;
                    for (const auto& stmt : conditionNode->falseBranch) {
                        elseBody += generate(stmt) + "\r\n";
                    }
```

```
                result += "\r\n    else {\r\n        " + elseBody + "    }";
            }
            return result;
        }
        else if (targetLanguage == "Python") {
            std::string result = "if " + condExpr + ":\r\n";
            for (const auto& stmt : conditionNode->trueBranch) {
                result += "    " + generate(stmt) + "\r\n";
            }
            if (!conditionNode->falseBranch.empty()) {
                result += "else:\r\n";
                for (const auto& stmt : conditionNode->falseBranch) {
                    result += "    " + generate(stmt) + "\r\n";
                }
            }
            return result;
        }
        break;
    }
    case ASTNodeType::Loop: {
        auto loop = std::dynamic_pointer_cast<LoopNode>(node);
        // TODO: реалізуй генерацію циклів для C++ та Python
        return "";
    }
    default:
        return "";
    }
    return "";
}

std::string CodeGenerator::generateAll(const
std::vector<std::shared_ptr<ASTNode>>& nodes) {
    std::string code;
    std::string result;

    if (wrapAsProgram) {
        if (targetLanguage == "C++") {
            for (auto& node : nodes) {
                code += "    " + generate(node) + "\r\n";
            }
            result += R"(using namespace System;)";
            result += "\r\n\r\n";
```

```
                result += R"(void main(array<String^>^ args) {)";
                result += "\r\n";
                result += code + "}";
                return result;
            }
            else if (targetLanguage == "Python") {
                for (auto& node : nodes) {
                    code += generate(node) + "\r\n";
                }
                return code;
            }
        }
        else {
            for (auto& node : nodes) {
                code += generate(node) + "\r\n";
            }
            return code;
        }
    }
```

NL2CodeConsole.cpp

```
    #include <iostream>
    #include <fstream>
    #include <sstream>
    #include <string>
    #include <codecvt>
    #include <locale>
    #include "Lexer.h"
    #include "Parser.h"
    #include "CodeGenerator.h"
    #include "AST.h"
    #include <iomanip>

    void printFileHex(const std::string& filename) {
        std::ifstream f(filename, std::ios::binary);
        char c;
        while (f.get(c)) {
            std::cout << std::hex << std::setfill('0') << std::setw(2)
                << (static_cast<unsigned>(static_cast<unsigned char>(c))) << "
";
        }
        std::cout << std::dec << std::endl;
    }
```

```
int main(int argc, char* argv[]) {
    // Встановити локаль для широкого виводу в консоль
        std::wcout.imbue(std::locale(""));
    if (argc < 3) {
        std::cerr << "Usage: NL2CodeConsole.exe input.txt output.txt\n";
        return 1;
    }
    std::string inputFile = argv[1];
    std::string outputFile = argv[2];
    printFileHex(inputFile);
    // Читання вхідного файлу в UTF-8
    std::wifstream in(inputFile);
    in.imbue(std::locale(std::locale(), new std::codecvt_utf8<wchar_t>));
    std::wstringstream buffer;
    buffer << in.rdbuf();
    std::wstring inputText = buffer.str();

    // ВИВЕДЕННЯ ВХІДНОГО ТЕКСТУ
    std::wcout << L"[DEBUG] Input (wstring):\n" << inputText << std::endl;

    // Lexer працює зі std::string — конвертуємо:
    std::wstring_convert<std::codecvt_utf8<wchar_t>> converter;
    std::string inputUtf8 = converter.to_bytes(inputText);
    std::cout << "[DEBUG] Input (UTF-8):\n" << inputUtf8 << std::endl;
    // Лексичний аналіз і генерація
    Lexer lexer;
    std::vector<Token> tokens = lexer.analyze(inputText);
    Parser parser(tokens);
    std::vector<std::shared_ptr<ASTNode>> ast = parser.parse();

    CodeGenerator generator("C++", true);
    std::string resultUtf8 = generator.generateAll(ast);

    // Запис UTF-8 в output файл
    std::wofstream out(outputFile);
    out.imbue(std::locale(std::locale(), new std::codecvt_utf8<wchar_t>));
    std::wstring resultWStr = converter.from_bytes(resultUtf8);
    out << resultWStr;

    return 0;
}
```

БІБЛІОГРАФІЧНА ДОВІДКА

Тема бакалаврської роботи: **Розробка транслятора для формування коду на Python та C++ на основі природних мовних конструкцій**

Обсяг пояснювальної записки 90 аркушів:

1 таблиця;

14 рисунків;

4 додатка.

Дата завершення роботи: *10 червня 2025р*.

Підпис студента - _____*Проців В.І.*.