МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

на тему: **«РОЗРОБКА КРОСПЛАТФОРМОВОГО ТРАНСЛЯТОРА СКРИПТОВОЇ МОВИ»**

Виконав: студент 4 курсу, групи 6.1211-1пі
спеціальності 121 інженерія програмного забезпечення
(шифр і назва спеціальності)
освітньої програми програмна інженерія
(назва освітньої програми)
О.О. Полєно
(ініціали та прізвище)

Керівник декан математичного факультету, професор, д.т.н. Гоменюк С.І.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Рецензент завідувач кафедри фундаментальної та прикладної математики, професор, д.т.н. Гребенюк С.М.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя – 2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра програмної інженерії

Рівень вищої освіти бакалавр

Спеціальність 121 інженерія програмного забезпечення
(шифр і назва)

Освітня програма програмна інженерія

ЗАТВЕРДЖУЮ
Завідувач кафедри програмної інженерії, к.ф.-м.н., доцент

_____ Лісняк А.О.
(підпис)

“ ____ ” _____ 2024 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Полєну Олексію Олександровичу
(прізвище, ім’я та по-батькові)

1. Тема роботи Розробка кросплатформового транслятора скриптової мови

керівник роботи Гоменюк Сергій Іванович, д.т.н., професор
(прізвище, ім’я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 20 » грудня 2024 року № 2001-с

2. Строк подання студентом роботи 02.06.2025 р.

3. Вихідні дані до роботи 1. Постановка задачі.
2. Перелік літератури.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Постановка задачі.
2. Основні теоретичні відомості.
3. Проєктування та розробка транслятора.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)
презентація за темою доповіді

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання 26.12.2024 р.

К А Л Е Н Д А Р Н И Й П Л А Н

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	09.01.2025	
2.	Збір вихідних даних.	14.02.2025	
3.	Обробка методичних та теоретичних	07.03.2025	
	джерел.		
4.	Розробка першого та другого розділу.	18.04.2025	
5.	Розробка третього розділу.	19.05.2025	
6.	Оформлення та нормоконтроль	26.05.2025	
	кваліфікаційної роботи бакалавра.		
7.	Захист кваліфікаційної роботи.	17.06.2025	

Студент _____ О.О. Полєно
(підпис) (ініціали та прізвище)

Керівник роботи _____ С.І. Гоменюк
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

Нормоконтролер _____ А.В. Столярова
(підпис) (ініціали та прізвище)

РЕФЕРАТ

Кваліфікаційна робота бакалавра «Розробка кросплатформового транслятора скриптової мови»: 53 с., 10 рис., 14 джерел.

АНАЛІЗАТОР, ГЕНЕРАЦІЯ КОДУ, ІНТЕРПРЕТАЦІЯ, КРОСПЛАТФОРМЕНІСТЬ, ПРОГРАМНА ІНЖЕНЕРІЯ, СКРИПТОВА МОВА, ТРАНСЛЯТОР, PYTHON.

Об’єкт дослідження – процес трансляції скриптових мов програмування.

Предмет дослідження – побудова транслятора для інтерпретації та виконання скриптів із використанням Python та супутніх інструментів.

Мета роботи – розробка транслятора скриптової мови з кросплатформеною підтримкою.

Методи дослідження – методи формального аналізу, об’єктно-орієнтованого програмування, інструменти лексичного та синтаксичного аналізу.

У роботі розглянуто етапи розробки транслятора скриптової мови, призначеного для забезпечення зручного, ефективного та незалежного від операційної системи виконання програмного коду. Проведено аналіз предметної області – трансляторів, інтерпретаторів і компіляторів, зокрема особливостей створення лексичного і синтаксичного аналізаторів, побудови абстрактного синтаксичного дерева та генерації коду.

Розробка базується на використанні мови програмування Python як універсального інструменту реалізації трансляційного процесу. Окрема увага приділяється забезпеченню кросплатформеності, структурі проміжного представлення коду, обробці помилок та перевірці семантики. В роботі також розглянуто варіанти генерації байт-коду або виконання віртуальною машиною.

SUMMARY

Bachelor’s qualifying paper “Development of a Cross-Platform Scripting Language Translator”: 53 pages, 10 figures, 14 references.

CODE GENERATION, CROSS-PLATFORM, INTERPRETATION, PARSER, PYTHON, SCRIPTING LANGUAGE, SOFTWARE ENGINEERING, TRANSLATOR.

The object of the research is the process of translating scripting languages.

The subject of the research is the design and implementation of a translator for interpreting and executing scripts using Python and related tools.

The aim of the work is to develop a scripting language translator with cross-platform support.

The research methods include formal analysis techniques, object-oriented programming, and the use of tools for lexical and syntactic analysis.

This work presents the stages involved in the development of a scripting language translator that is designed to ensure convenient, efficient, and platform-independent execution of source code. The study includes an analysis of translators, interpreters, and compilers, with particular emphasis on the creation of lexical and syntax analyzers, the construction of abstract syntax trees, and mechanisms for code generation.

The translator is implemented using the Python programming language, which serves as a flexible and accessible tool for managing the translation process. Special attention is given to cross-platform compatibility, the structure of the intermediate code representation, error handling, and semantic analysis. The work also explores various options for bytecode generation and execution on a virtual machine.

ЗМІСТ

Завдання на кваліфікаційну роботу……..……..2
Реферат 4
Summary 5
Вступ..... 7
1 Огляд теоретичних властивостей та планування проєкту 9
1.1 Історія трансляторів та еволюція скриптових мов 9
1.2 Основні підходи до розробки трансляторів та їх структура..... 12
1.3 Огляд сучасних технологій: Python, Node.js, ANTLR, LLVM 15
1.4 Порівняльний аналіз існуючих трансляторів та мовних 20
1.5 Висновки до розділу 1 22
2 Проєктування кросплатформового транслятора скриптової мови 23
2.1 Опис предметної області та мети трансляції 23
2.2 Функціональні вимоги до транслятора скриптової мови 25
2.3 Нефункціональні вимоги..... 27
2.4 Вимоги до кросплатформенності та інтеграції..... 29
2.5 Архітектура транслятора..... 33
2.6 Висновки до розділу 2 35
3 Реалізація та тестування 37
3.1 Реалізація лексичного та синтаксичного аналізу..... 37
3.2 Генерація коду та підтримка багатоплатформності 41
3.3 Тестування транслятора 44
3.4 Аналіз результатів та оптимізація..... 47
3.5 Висновки до розділу 3 48
Висновки 50
Перелік посилань..... 52

ВСТУП

Розвиток цифрових технологій та зростання популярності скриптових мов програмування створюють потребу у гнучких, ефективних та кросплатформених інструментах для обробки та виконання коду. Скриптові мови широко застосовуються для автоматизації, розробки вебзастосунків, створення інфраструктурних сценаріїв і прототипування, що вимагає наявності зручних засобів їх трансляції у виконуваний код.

У контексті сучасного програмного забезпечення особливо важливо забезпечити кросплатформеність, тобто можливість запуску створених рішень на різних операційних системах без додаткових витрат на адаптацію. Транслятори – спеціалізовані програми, які перетворюють код з однієї мови на іншу або на внутрішнє представлення – відіграють ключову роль у реалізації таких можливостей. Розробка власного транслятора дозволяє не лише глибше зрозуміти принципи побудови мов програмування, але й створити інструмент, оптимізований під специфічні потреби користувачів або організацій.

Цей проєкт присвячений розробці кросплатформового транслятора для простої скриптової мови, з акцентом на розширюваність, зрозумілий синтаксис та ефективну реалізацію. У проєкті застосовуються сучасні технології: Python як мова реалізації, ANTLR або аналогічні інструменти для побудови лексичного та синтаксичного аналізаторів, а також можливе використання LLVM для генерації проміжного або низькорівневого коду.

В межах роботи буде проаналізовано предметну область, визначено синтаксис і семантику розроблюваної мови, реалізовано основні етапи трансляції: лексичний і синтаксичний аналіз, побудова абстрактного синтаксичного дерева, перевірка семантики та генерація коду. Також передбачено створення командного інтерфейсу для зручного використання транслятора з можливістю виконання скриптів на різних операційних системах.

Особлива увага приділяється тестуванню транслятора, обробці типових помилок програміста, забезпеченню стабільності та продуктивності. Безпека виконання скриптів також буде врахована шляхом обмеження доступу до критичних ресурсів під час виконання коду.

Результатом роботи стане повноцінний транслятор скриптової мови з базовим функціоналом, який може використовуватись як інструмент для навчання, розробки або автоматизації задач. Це рішення дозволить ефективно запускати скрипти на різних платформах, забезпечуючи простоту використання та адаптацію під потреби кінцевих користувачів.

Таким чином, розробка кросплатформового транслятора скриптової мови є актуальним і важливим завданням, що поєднує в собі як теоретичні, так і практичні аспекти сучасної комп’ютерної науки та інженерії програмного забезпечення.

1 ОГЛЯД ТЕОРЕТИЧНИХ ВЛАСТИВОСТЕЙ ТА ПЛАНУВАННЯ ПРОЄКТУ

1.1 Історія трансляторів та еволюція скриптових мов

Прогрес комп’ютерних технологій визначив процес появи нових різноманітних числових систем для запису алгоритмів мов програмування. Сенс появи такої мови – спрощення програмного коду.

Мови програмування прийнято ділити на п’ять поколінь. У перше покоління входять мови, створені на початку 50-х років XX-го сторіччя, коли перші комп’ютери тільки но з’явилися. Це була перша мова асемблера, створена за принципом «одна інструкція – один рядок» [1].

Фізичні принципи роботи електронних обчислювальних машин (ЕОМ) такі, що комп’ютер може сприймати команди, що складаються тільки з одиниць і нулів – послідовність перепаду напруги, тобто машинний код. На початковій стадії розвитку ЕОМ людині було необхідно записувати програми мовою, зрозумілій комп’ютеру, в машинних кодах. Кожна команда складалася з коду операцій та адрес операндів, виражених у вигляді різних поєднань одиниць і нулів. Отже, будь-яка програма для процесора виглядала на той час як послідовність одиниць і нулів [2].

Як показала практика спілкування з комп’ютером, така мова громіздка і незручна. Користуючись нею легко припуститися помилки, записавши не в тій послідовності 1 або 0. Програму важко контролювати. Крім того, при програмуванні в машинних кодах треба добре знати внутрішню структуру ЕОМ, принцип роботи кожного блоку. Програми такою мовою – дуже довгі послідовності одиниць і нулів є машинно-залежними, тобто для кожної ЕОМ необхідно було складати свою програму. Також програмування в машинних кодах вимагає від програміста багато часу та підвищеної уваги [3].

Виникла потреба автоматизації процесу формування машинного коду. Вже у 1950 році для запису програм почали застосовувати мнемонічну мову –

мову assembly. Мова асемблера дала змогу записати машинний код у більш зручній для людини формі: для позначення команд і об’єктів,з якими ці команди виконуються, замість двійкових кодів використовувалися букви або скорочені слова, які відображали суть команди. Наприклад, мовою асемблера команда складання двох чисел позначається словом add, тоді як її машинний код може бути таким: 000010 [3].

Асемблер – мова програмування низького рівня. Мова програмування низького рівня – це мова програмування, яка орієнтована на конкретний тип процесора і враховує його особливості. Тут «низький рівень» не означає «поганий». Мається на увазі, що оператори мови близькі до машинного коду і орієнтовані на конкретні команди процесора. Поява мови асемблера значно полегшила життя програмістів, оскільки тепер замість великої кількості нулів і одиниць, вони могли писати програму командами, що складаються з символів наближених до звичайної мови. Ця мова, коли була нововведенням, мала популярність, бо давала змогу писати програми невеликого розміру.

Складність розробки на ньому великих програмних комплексів призвела до появи мов третього покоління – мов високого рівня. Але на цьому життя асемблера не закінчилася, він має популярність серед вузьких кіл. Зараз його використовують в написанні окремих фрагментів програм або іноді в написанні самих програм. Приклад, це використання асемблера в написанні драйверів, ігор та завантажувачів ОС. Швидкість роботи отриманої програми значно вище швидкості програми написаної мовою програмування високого рівня. Це пояснюється тим, що отриманий розмір програми дуже малий. Розробники антивірусних програм теж використовують асемблер в деяких модулях своїх програм, що забезпечує їх швидкодію, якби не асемблер, деякі програми завантажувалися б близько години [4].

Історія саме трансляторів бере свій початок із середини XX століття, коли постала потреба автоматизувати процес перетворення програм з мов високого рівня у машинний код. Першим повноцінним транслятором вважається компілятор мови Fortran, створений у 1957 році в IBM. Він

дозволив інженерам писати складні математичні обчислення мовою, близькою до людської, що значно спростило розробку програмного забезпечення.

У подальшому з'явилися інтерпретатори – програми, які виконували код без попередньої компіляції. Вони забезпечували швидкий запуск та інтерактивність, що стало основою для скриптових мов програмування. Серед перших таких мов – BASIC, яка з'явилася у 1964 році, та LISP – одна з перших мов високого рівня для обробки даних [4].

Існує декілька класифікацій трансляторів:

– компілятори – перетворюють увесь вихідний код програми у машинний код до її виконання (наприклад: GCC (для C/C++), javac (для Java));
– інтерпретатори – виконують код поступово, рядок за рядком, без попередньої компіляції (наприклад: Python, Ruby, Bash);
– транслятори середнього рівня (Intermediate Code Translators) – перетворюють код у проміжне представлення, яке потім виконується або компілюється (наприклад, Java → байт-код → JVM);
– транспайлери (source-to-source компілятори) – перетворюють код однієї мови у код іншої високорівневої мови (наприклад, TypeScript → JavaScript, Babel).

Скриптові мови – це мови, що спочатку створювалися для автоматизації рутинних задач. Вони характеризуються простим синтаксисом, відсутністю необхідності компіляції, гнучкістю у використанні. Найвідоміші з них:

– Shell (Bash, Zsh) – для автоматизації в UNIX/Linux;
– Perl – потужна мова для обробки тексту;
– Python – універсальна мова, яка починалась як скриптова, але стала повноцінною мовою загального призначення;
– JavaScript – основна скриптова мова для клієнтської частини вебзастосунків.

Скриптові мови здебільшого інтерпретуються, але з часом у багатьох із них з'явилися транслятори або гібридні системи. Наприклад, у Python існує

PyInstaller для створення виконуваних файлів, а V8 (движок JavaScript від Google) компілює скрипти у машинний код під час виконання (Just-in-Time).

Сучасні транслятори поєднують у собі риси класичних компіляторів та інтерпретаторів. Наприклад, Just-in-Time (JIT)-технології дозволяють оптимізувати виконання коду під час роботи програми, що особливо актуально для кросплатформенних рішень. Крім того, завдяки таким інструментам, як LLVM, ANTLR, Babel тощо, стало можливим створення власних трансляторів для нових або доменно-специфічних мов [4].

1.2 Основні підходи до розробки трансляторів та їх структура

Розробка трансляторів – це одна з найважливіших дисциплін у сфері комп'ютерних наук, що поєднує формальні мови, теорію граматик, автоматів та прикладне програмування. Транслятор виконує складний процес перетворення коду з мови високого рівня у цільовий код (машинний, байт-код або код іншої мови), забезпечуючи правильність, ефективність та переносимість виконання [5].

Існує кілька ключових підходів до побудови трансляторів, які відрізняються за структурою, способом виконання та цільовою платформою:

– однопрохідні транслятори (one-pass compilers);
– багатопрохідні транслятори (multi-pass compilers);
– інтерпретація;
– JIT-компіляція (Just-in-Time);
– транспіляція (source-to-source translation).

Поговоримо про кожний з них більш детально, щоб зрозуміти призначення та функціонал пожного з них [6].

Лексичний аналізатор (лексер) – це перший етап трансляції, що виконує розбір вхідного тексту програми на базові одиниці мови, які називаються лексемами або токенами. Цей процес називається лексичним аналізом.

Метою лексичного аналізу є:

– очищення коду від зайвих символів (пробілів, коментарів);
– виявлення та класифікація структурних одиниць (ідентифікаторів, ключових слів, чисел, операторів тощо);
– підготовка вхідних даних для синтаксичного аналізатора;
– реєстрація помилок лексичного рівня (наприклад, неправильне число, некоректна назва змінної тощо).

Процес лексичного аналізу передбачає читання символів один за одним і поступове формування лексем. При цьому пропускаються непотрібні символи, такі як пробіли, табуляції або коментарі, якщо вони не мають значення в контексті синтаксису. Також на цьому етапі виявляються лексичні помилки – наприклад, якщо в імені змінної використано заборонені символи або неправильно вказано числове значення [6].

Для реалізації лексичного аналізатора можуть використовуватись різні підходи. У найпростіших випадках лексер створюється вручну за допомогою конструкцій умовного розгалуження та регулярних виразів. Для складніших мов застосовуються спеціалізовані інструменти – генератори лексерів, які дозволяють описати правила розпізнавання токенів у формі граматик. Серед таких засобів можна назвати Flex для C/C++, ANTLR, або бібліотеки на кшталт PLY у Python [7].

Багатопрохідний транслятор – це тип трансляційної системи, яка обробляє вихідний код у кілька послідовних етапів (проходів), кожен з яких виконує окрему функцію аналізу або перетворення програми. На відміну від однопрохідних трансляторів, де всі дії (лексичний, синтаксичний аналіз, семантична перевірка та генерація коду) виконуються за один прохід по вихідному коду, багатопрохідні транслятори дозволяють розділити ці завдання на логічно незалежні частини. Такий підхід спрощує реалізацію транслятора, полегшує відлагодження і дає можливість гнучко змінювати або розширювати етапи обробки [8].

Перший прохід зазвичай відповідає за побудову абстрактного синтаксичного дерева (AST) на основі граматики мови. У наступних проходах

можуть виконуватись семантичний аналіз, перевірка типів, оптимізація коду, перетворення виразів, а також генерація цільового коду. Наприклад, один прохід може бути повністю присвячений аналізу імен (ідентифікаторів) і побудові таблиці символів, тоді як інший – перевірці відповідності типів або обчисленню констант.

Інтерпретація – це один із двох основних способів виконання програм, поряд із компіляцією. На відміну від компіляторів, які спочатку перетворюють увесь вихідний код у машинний або проміжний код, інтерпретатори виконують програму поступово, рядок за рядком або вираз за виразом, без попереднього створення окремого виконуваного файлу. Це робить інтерпретацію більш гнучким, але загалом повільнішим підходом [9].

Інтерпретатори зчитують вхідний код, виконують його аналіз, і одразу ж виконують відповідні дії. Наприклад, якщо в коді є арифметичний вираз, інтерпретатор спочатку розпізнає його структуру, обчислює значення і переходить до наступної інструкції. У цьому процесі часто використовуються проміжні подання, такі як абстрактне синтаксичне дерево (AST), яке допомагає підтримувати структуру коду та керувати порядком виконання.

JIT-компіляція є гібридним підходом до виконання програм, який поєднує риси як інтерпретації, так і традиційної компіляції. На відміну від повної компіляції, яка перетворює програму у машинний код до її запуску, або інтерпретації, що виконує програму рядок за рядком, JIT-компіляція виконує трансляцію частин коду у машинні інструкції безпосередньо під час виконання програми. Це дозволяє досягти суттєвого приросту швидкодії, зберігаючи при цьому динамічність та гнучкість виконання [10].

Суть JIT-компіляції полягає в тому, що під час запуску програми інтерпретатор аналізує її структуру, а «гарячі» ділянки коду – тобто ті, що часто використовуються або виконуються багаторазово – перетворюються у високоефективний машинний код. Після компіляції ці ділянки зберігаються в пам’яті, і при наступних зверненнях вже не потребують повторної інтерпретації. Таким чином, JIT-компіляція забезпечує баланс між швидкістю виконання та гнучкістю динамічної обробки коду.

Транспіляція, або source-to-source translation, – це процес перетворення вихідного коду, написаного однією мовою програмування, у еквівалентний код іншої мови, як правило, такого самого рівня абстракції. На відміну від традиційної компіляції, яка переводить код у машинний або байт-код для виконання, транспілятори зберігають початкову логіку програми, але переписують її синтаксично під іншу мову. Це дозволяє адаптувати програму під інше середовище виконання без зміни її поведінки [10].

1.3 Огляд сучасних технологій: Python, Node.js, ANTLR, LLVM

Сучасна розробка трансляторів і інструментів для обробки мов програмування тісно пов’язана з використанням потужних і гнучких технологій, які забезпечують ефективну роботу з кодом, аналіз синтаксису, генерацію машинного коду та інтеграцію з різними платформами. У цьому контексті особливу увагу заслуговують такі технології як Python, Node.js, ANTLR та LLVM [11].

Python – це високорівнева, інтерпретована мова програмування, що вирізняється простотою синтаксису, читабельністю коду та широким набором стандартних бібліотек. Ці характеристики роблять Python надзвичайно популярним серед розробників, особливо у сферах, що потребують швидкої розробки та гнучкості, таких як створення трансляторів, інтерпретаторів і різноманітних інструментів для обробки тексту й коду [11].

Однією з основних переваг Python для розробки трансляторів є його багата екосистема бібліотек і фреймворків, які значно спрощують створення лексичних і синтаксичних аналізаторів. Зокрема, до найпопулярніших інструментів належать:

– PLY (Python Lex-Yacc) – бібліотека для реалізації лексичного і синтаксичного аналізу, заснована на класичних інструментах lex і yacc;

– ANTLR – генератор парсерів з підтримкою Python, що дозволяє автоматично створювати складні граматики;
– модуль re – стандартна бібліотека для роботи з регулярними виразами, що використовується на етапі лексичного аналізу.

Крім того, Python підтримує об’єктно-орієнтоване програмування (ООП), що сприяє модульності та масштабованості розроблюваних систем.

Основні переваги ООП у створенні трансляторів полягають у наступному:

– легке розділення коду на логічні компоненти;
– можливість повторного використання коду;
– зручність розширення функціональності без суттєвих змін у базовому коді.

Python також має потужні інструменти для генерації проміжного коду, байткоду або навіть машинного коду. Це стало можливим завдяки інтеграції з різними зовнішніми бібліотеками. Наприклад, бібліотека llvmlite надає інтерфейс для роботи з LLVM, що дозволяє створювати оптимізований машинний код. Крім того, у Python можна реалізовувати власні формати байткоду для інтерпретаторів, що забезпечує додаткову гнучкість у розробці. Також існує можливість інтеграції з різними компіляторами та іншими зовнішніми інструментами, що розширює функціональність і дозволяє будувати складні системи трансляції [12].

Що стосується кросплатформової розробки трансляторів, Python пропонує низку важливих переваг. Він підтримує основні операційні системи, такі як Windows, Linux та macOS, що робить створені на його основі програми доступними для широкої аудиторії. Крім того, Python має велику кількість API, які забезпечують зручну інтеграцію з іншими мовами програмування та системами. Завдяки простоті роботи зі стандартними форматами даних, такими як JSON, XML, YAML, Python полегшує обмін інформацією між різними компонентами системи.

Активна спільнота розробників і постійний розвиток екосистеми гарантують наявність сучасних бібліотек, інструментів і підтримки. Це робить

Python оптимальним вибором для швидкої та ефективної розробки складних програмних проєктів, особливо таких, як транслятори скриптових мов.

Node.js – це середовище виконання JavaScript, яке працює на серверній стороні і базується на рушії V8 від Google Chrome. Воно дозволяє запускати JavaScript-код поза браузером, що відкриває можливості для створення різноманітних серверних додатків, інструментів автоматизації та навіть трансляторів скриптових мов.

Однією з ключових переваг Node.js є його подієво-орієнтована, неблокуюча (асинхронна) модель введення-виведення, яка робить його надзвичайно ефективним для обробки великої кількості одночасних операцій. Це особливо важливо при розробці трансляторів, які можуть потребувати обробки численних запитів або виконання складних аналізів коду в реальному часі [12].

Node.js має велику і активну екосистему пакетів – менеджер npm містить сотні тисяч модулів, які можна легко інтегрувати у свій проєкт. Для задач трансляції та аналізу коду корисними є такі бібліотеки:

- Acorn – парсер JavaScript, який дозволяє розбирати код у форматі AST (абстрактного синтаксичного дерева);
- Esprima – ще один популярний парсер JS з підтримкою сучасних стандартів мови;
- Babel – транспілятор, що перетворює сучасний JavaScript (ES6+) у сумісний зі старішими середовищами код, широко використовується як інструмент для трансляції;
- Nearley – потужний парсер з підтримкою контекстно-вільних граматик, що може бути корисним при створенні власних мов або трансляторів.

Node.js підтримує модульність та розробку за принципами об’єктно-орієнтованого та функціонального програмування, що робить код більш організованим, підтримуваним і масштабованим.

Ще однією важливою перевагою Node.js є його кросплатформеність. Середовище працює на Windows, Linux та macOS, що дозволяє розробляти

універсальні інструменти, які будуть однаково ефективно функціонувати на різних операційних системах.

Node.js також має вбудовану підтримку роботи з різними форматами даних, такими як JSON, що є стандартом у багатьох трансляторах і системах обробки конфігурацій [13].

Завдяки своїй гнучкості, швидкості роботи і величезній кількості готових модулів, Node.js часто використовується для створення прототипів трансляторів, інструментів статичного аналізу коду, систем автоматизації збірки та інших розробницьких інструментів, що робить його цінним інструментом у стеку технологій для розробки трансляторів скриптових мов.

ANTLR – це потужний генератор парсерів, який широко використовується для розробки компіляторів, трансляторів, інтерпретаторів та інших інструментів обробки мов програмування. Його основне завдання – автоматизувати створення синтаксичного аналізатора (парсера) на основі формальних граматик.

ANTLR дозволяє описувати граматику мови у вигляді правил, після чого на основі цих правил генерує код парсера на різних мовах програмування, включно з Java, C#, Python, JavaScript тощо. Завдяки цьому можна ефективно створювати складні і гнучкі транслятори, які розпізнають та обробляють мови будь-якої складності [13].

Основні переваги ANTLR:

- підтримка LL(*) парсингу – що забезпечує високу продуктивність та можливість розбору складних граматик, включно з рекурсивними;
- автоматичне генерування дерев синтаксичного аналізу (Parse Trees), що полегшує подальший аналіз коду і трансформацію;
- гнучкість та масштабованість – дозволяє будувати як прості, так і надскладні парсери;
- інтеграція з популярними мовами – можливість генерації парсерів під різні середовища розробки;

– підтримка вбудованих дій – це фрагменти коду, які можна вставляти прямо у граматику для реалізації додаткової логіки під час парсингу.

ANTLR широко використовується у промисловості та академії для створення різних мов програмування, діалектів і трансляторів, оскільки значно прискорює та спрощує процес розробки лексичних і синтаксичних аналізаторів.

LLVM – це модульний і розширюваний компіляторний фреймворк, що надає набір інструментів для побудови компіляторів і трансляторів, включаючи оптимізацію проміжного коду, генерацію машинного коду і підтримку багатьох апаратних архітектур [13].

LLVM починався як низькорівнева віртуальна машина для виконання програм, але зараз він включає:

– проміжне представлення (LLVM IR) – універсальний проміжний код, що використовується як внутрішній формат для оптимізації та трансляції;
– оптимізаційні проходи – набір алгоритмів, які покращують якість коду, зменшують його розмір, підвищують швидкість виконання;
– генератори машинного коду – модулі, які транслюють LLVM IR у машинний код для різних процесорних архітектур (x86, ARM, MIPS тощо);
– JIT-компіляція – можливість компілювати код у момент виконання, що дозволяє поєднувати інтерпретацію і компіляцію для підвищення продуктивності.

LLVM широко використовується для створення сучасних компіляторів, зокрема таких, як Clang (компілятор для C/C++/Objective-C). Його гнучкість і модульність роблять його ідеальним для розробки трансляторів скриптових мов, де потрібно швидко генерувати ефективний машинний код [13].

Переваги LLVM для трансляторів:

– високий рівень оптимізації коду на різних етапах трансляції;
– підтримка багатьох платформ і архітектур;
– вбудовані засоби для JIT-компіляції, що дає змогу створювати швидкі інтерпретатори з можливістю компіляції «на льоту»;
– велика спільнота та активний розвиток, що гарантує постійне оновлення та підтримку.

У контексті розробки кросплатформового транслятора скриптової мови ці технології можна комбінувати для досягнення оптимального результату. Python і Node.js можуть слугувати для швидкого прототипування та інтеграції, ANTLR – для побудови надійних парсерів, а LLVM – для генерації ефективного машинного коду або байт-коду. Такий підхід дозволить створити гнучку, масштабовану і високопродуктивну систему трансляції, здатну працювати на різних платформах і підтримувати подальший розвиток мови.

1.4 Порівняльний аналіз існуючих трансляторів та мовних інтерпретаторів

Обраний стек технологій для розробки транслятора – Python, Node.js, ANTLR і LLVM – поєднує в собі простоту розробки, гнучкість і продуктивність. У цьому розділі наведено порівняння цього стеку з іншими популярними технологічними підходами [13].

Python забезпечує швидку розробку завдяки багатій екосистемі бібліотек і простоті коду. У порівнянні з мовами на кшталт C/C++ або Go, Python поступається в продуктивності, однак переважає за швидкістю прототипування та зручністю підтримки. Альтернативні рішення на C/C++ пропонують максимальну продуктивність і гнучкість управління пам’яттю, але вимагають більше часу на розробку і складніші у супроводі. Go демонструє кращу продуктивність, ніж Python, і пропонує простіший синтаксис порівняно з C++, проте його екосистема для розробки трансляторів менш розвинена.

Node.js добре підходить для створення трансляторів, що інтегруються у вебсередовище, завдяки неблокуючій архітектурі та широкому набору npm-модулів. У порівнянні з Python він має кращу продуктивність в I/O-операціях, але складні задачі парсингу через динамічну типізацію можуть бути більш громіздкими. Порівняно з Java, що широко використовується для розробки трансляторів (зокрема разом із ANTLR), Node.js поступається у стабільності типізації і масштабованості, але перевершує за швидкістю розробки простих веборієнтованих інструментів [14].

ANTLR є одним із найпотужніших генераторів парсерів, сумісним з багатьма мовами програмування. На відміну від ручного написання парсерів на C/C++ або JavaScript, ANTLR значно скорочує час розробки і зменшує кількість помилок. Альтернативи, такі як yacc/bison або hand-crafted парсери, дають більше контролю над оптимізацією, але потребують глибших знань і часу. Інші генератори, наприклад, PEG.js або JavaCC, мають обмежену підтримку мов чи функцій у порівнянні з ANTLR.

LLVM надає можливість створення високопродуктивних компіляторів із оптимізацією низькорівневого коду, що є перевагою над рішеннями, які генерують байткод або інтерпретують код безпосередньо. В порівнянні з JIT-компіляторами на основі Java Virtual Machine, LLVM більш гнучкий і краще підходить для створення кастомних трансляторів. Проте його використання складніше і вимагає знань низькорівневого програмування, що може збільшувати час розробки.

Якщо порівнювати цей стек із класичними альтернативами, такими як реалізації трансляторів на C++ або Java, слід врахувати, що C++ пропонує високу продуктивність і контроль над пам’яттю, проте розробка на ньому є значно більш трудомісткою і повільною. Java, з іншого боку, має розвинену екосистему і статичну типізацію, що підвищує надійність коду, але ускладнює швидке прототипування. Python та Node.js надають перевагу гнучкості і швидкості розробки, що відповідає сучасним тенденціям agile-розробки.

Таким чином, обраний стек технологій поєднує в собі швидкість розробки [14].

Обраний стек технологій найкраще підходить для кросплатформового транслятора скриптової мови з акцентом на баланс між зручністю розробки, продуктивністю та підтримкою.

1.5 Висновки до розділу 1

У першому розділі було здійснено огляд теоретичних основ трансляції та проведено планування етапів розробки кросплатформового транслятора скриптової мови. Розглянуто історичний розвиток трансляторів і еволюцію скриптових мов, що дозволило сформувати уявлення про їхні ключові властивості та області застосування.

Особливу увагу приділено аналізу архітектурних підходів до побудови трансляторів: лексичний і синтаксичний аналіз, побудова абстрактного синтаксичного дерева, семантичний аналіз та генерація цільового коду. На основі цього дослідження визначено оптимальну структуру майбутнього програмного рішення.

Також проведено огляд сучасних інструментів, таких як Python, Node.js, ANTLR і LLVM, які можуть бути використані для реалізації окремих компонентів транслятора. Обґрунтовано вибір стеку технологій, який забезпечує баланс між продуктивністю, гнучкістю та кросплатформеністю.

У результаті порівняльного аналізу наявних трансляторів та мовних інтерпретаторів, таких як Babel, Esprima, PyPy і QuickJS, було визначено ключові функціональні особливості та архітектурні рішення, які варто враховувати під час реалізації власного транслятора. Це дозволило уточнити функціональні вимоги до системи та підготувати основу для її проєктування й розробки в наступних розділах.

2 ПРОЄКТУВАННЯ КРОСПЛАТФОРМОВОГО ТРАНСЛЯТОРА СКРИПТОВОЇ МОВИ

2.1 Опис предметної області та мети трансляції

Скриптові мови програмування широко використовуються у сучасних інформаційних системах для автоматизації задач, розробки інтерфейсів, обробки даних та написання конфігураційних сценаріїв. Вони відзначаються простотою синтаксису, динамічною типізацією та високою гнучкістю, що робить їх зручними для швидкої розробки і інтеграції. Проте у випадках, коли існує необхідність адаптувати код до виконання в іншому середовищі або забезпечити переносимість сценаріїв між платформами та інтерпретаторами, виникає потреба у трансляції скриптового коду в іншу мову або іншу форму подання.

Предметною областю даного проєкту є автоматизована обробка скриптового коду, його аналіз, перетворення та генерація еквівалентного представлення, придатного до виконання в іншому середовищі або на іншій платформі. Це охоплює створення транслятора, здатного зчитувати вхідний код заданої скриптової мови, виконувати його лексичний і синтаксичний розбір, перевірку на відповідність граматиці та семантичним правилам, а також формувати вихідний код на цільовій мові або у вигляді проміжного представлення (наприклад, байт-код або IR-код) [2].

Основна мета трансляції полягає у забезпеченні кросплатформеності та сумісності скриптів, створених у рамках заданої доменної мови. Це дозволяє:

- зменшити залежність від конкретного середовища виконання;
- покращити продуктивність виконання за рахунок оптимізації коду;
- розширити можливості повторного використання логіки у різних проєктах;
- забезпечити адаптацію до специфічних технічних вимог цільового середовища;

– сприяти створенню стандартизованих рішень у командній розробці.

Однією з основних причин використання транслятора є необхідність стандартизації скриптових рішень, особливо у випадках, коли проєкт розробляється командою розробників, які можуть використовувати різні стилі програмування, середовища або навіть операційні системи.

Завдяки кросплатформеній трансляції стає можливим забезпечення єдиної логіки виконання незалежно від конкретної платформи, де цей код буде запускатися (наприклад, Windows, Linux чи macOS).

Крім того, в умовах зростаючої популярності serverless-архітектур та гібридних хмарних середовищ, стає актуальною підтримка універсального середовища виконання, що також передбачає необхідність трансляції скриптів у менш ресурсозалежні або оптимізовані формати. Наприклад, це може бути трансляція з кастомної скриптової мови в JavaScript для виконання в браузері або в Python для обробки на сервері [7].

У рамках даного проєкту передбачається створення власного кросплатформового транслятора**,** який виконуватиме трансляцію з формально визначеної простої скриптової мови в іншу мову програмування або у проміжне представлення (наприклад, JSON або bytecode). Основні функціональні вимоги до транслятора:

– підтримка чіткого синтаксису вхідної мови;
– реалізація лексичного та синтаксичного аналізу;
– побудова абстрактного синтаксичного дерева (AST);
– перевірка на відповідність семантичним правилам;
– генерація коректного вихідного коду.

Таким чином, предметна область охоплює задачі, пов’язані з компіляцією, інтерпретацією, синтаксичним та семантичним аналізом, оптимізацією коду та автоматизацією переносу логіки між мовами. У довгостроковій перспективі транслятор може бути розширений до повноцінного компілятора або платформи для створення інтерпретаторів нових мов.

2.2 Функціональні вимоги до транслятора скриптової мови

Функціональні вимоги до системи трансляції скриптової мови визначають, які дії повинен забезпечувати транслятор для кінцевого користувача. Ці вимоги базуються на характері предметної області, меті створення інструменту та особливостях архітектурного рішення.

Основними функціональними вимогами до кросплатформового транслятора є:

- підтримка синтаксису заданої скриптової мови: транслятор має забезпечувати коректне розпізнавання синтаксичних конструкцій (змінні, умовні оператори, цикли, функції тощо), що описані у специфікації мови – для цього необхідна граматика, визначена у форматі ANTLR або сумісна з ним;
- лексичний та синтаксичний аналіз: система повинна розбивати вхідний скрипт на токени та створювати абстрактне синтаксичне дерево (AST) – цей етап є критичним для подальшої семантичної обробки та генерації коду;
- можливість трансляції у цільову мову або байткод: на виході транслятор повинен створювати проміжне подання (наприклад, код JavaScript для Node.js або LLVM IR для компіляції у машинний код);
- виведення помилок із зазначенням місця в коді: при виявленні синтаксичних або семантичних помилок, користувач має отримати зрозуміле повідомлення з точним рядком і типом помилки;
- підтримка розширюваності: архітектура транслятора має дозволяти легко додавати нові конструкції мови, розширювати синтаксис або змінювати правила генерації коду;
- кросплатформеність: транслятор має бути доступним для запуску на різних операційних системах (Windows, Linux, macOS), що реалізується завдяки використанню інтерпретованих мов (Python, Node.js) та сумісних інструментів (ANTLR, LLVM);

- можливість інтеграції з зовнішніми системами або IDE: забезпечення інтерфейсу командного рядка або API, що дозволить інтегрувати транслятор у системи CI/CD або середовища розробки;
- оптимізація вихідного коду (за потреби): у разі використання LLVM або іншої системи оптимізації, транслятор повинен мати можливість виконувати базову оптимізацію згенерованого коду.

Також транслятор повинен підтримувати модульність та реюзабельність коду. Це означає, що він має коректно обробляти механізми імпорту і експорту модулів або бібліотек скриптової мови, що дозволить розробникам використовувати повторно написані частини коду у різних проєктах без необхідності їх дублювання.

Важливою функціональною вимогою є підтримка вбудованих функцій та стандартних бібліотек мови. Транслятор має коректно інтерпретувати або транслювати такі функції, забезпечуючи їх сумісність з цільовим середовищем виконання. Це дозволить користувачам ефективно використовувати стандартний функціонал мови без обмежень.

Для забезпечення зручності розробки та налагодження програм, транслятор повинен надавати можливість генерації додаткової інформації, яка допомагатиме відстежувати виконання коду. Зокрема, це можуть бути source maps – спеціальні мапи, що зв’язують транслюваний код із початковим, полегшуючи процес налагодження [9].

Оптимальним є наявність інтерактивного режиму роботи транслятора. Такий режим дозволяє виконувати окремі рядки або фрагменти коду у реальному часі, що значно спрощує тестування та експериментування з кодом.

Для мов, які підтримують асинхронність, транслятор повинен коректно обробляти асинхронні конструкції. Це забезпечить правильну роботу асинхронного коду та збереження логіки його виконання при трансляції.

Особливу увагу слід приділити обробці винятків і помилок часу виконання. Транслятор має підтримувати механізми обробки виключень, такі як конструкції try/catch, генеруючи відповідний код для безпечного управління помилками під час виконання.

Для покращення контролю якості та аналізу процесу трансляції транслятор має створювати звіти про виконані операції, помилки та попередження. Така функціональність допоможе розробникам виявляти проблеми та оптимізувати код.

Нарешті, бажаною є підтримка багатомовності інтерфейсу або локалізації помилок і повідомлень, що зробить транслятор більш доступним для користувачів з різних країн.

2.3 Нефункціональні вимоги

Однією з ключових нефункціональних вимог до розробки кросплатформового транслятора скриптової мови є забезпечення високої продуктивності. Транслятор має ефективно обробляти великі обсяги коду, мінімізуючи час трансляції та ресурси, необхідні для виконання процесу. Оптимізовані алгоритми аналізу та генерації коду сприятимуть зниженню затримок і покращенню загальної швидкодії системи.

Кросплатформеність транслятора означає його коректну роботу на різних операційних системах, таких як Windows, Linux та macOS. Це вимагає ретельної реалізації та тестування для забезпечення сумісності, а також уникнення залежностей від платформо-залежних компонентів.

Важливою нефункціональною вимогою є масштабованість системи. Транслятор повинен з легкістю адаптуватися до збільшення складності скриптів, а також інтегруватися з іншими інструментами розробки, такими як IDE, системи безперервної інтеграції, і засоби налагодження.

Надійність та стійкість до збоїв також мають велике значення. Транслятор повинен коректно обробляти помилки, у тому числі синтаксичні та логічні, надавати зрозумілі повідомлення користувачу і забезпечувати безпеку виконання згенерованого коду. Особливу увагу слід приділити обробці виняткових ситуацій та відмовостійкості [12].

Основні нефункціональні вимоги можна коротко підсумувати так:

– підтримка кросплатформенності (Windows, Linux, macOS);
– висока продуктивність і мінімальна затримка трансляції;
– масштабованість і легка інтеграція з іншими системами;
– захищеність і стабільність при обробці помилок;
– зручність використання і зрозумілий інтерфейс (CLI або GUI).

Ще одним важливим нефункціональним аспектом є зручність розгортання та налаштування**.** Транслятор повинен мати просту процедуру інсталяції та бути здатним швидко налаштовуватись під конкретні потреби користувача або середовище виконання. Наприклад, можливість запуску у вигляді CLI-додатку без необхідності глибокої конфігурації або залежностей, або підтримка конфігураційних файлів для гнучкого керування параметрами трансляції.

Особливу увагу також слід приділити ресурсоефективності. Використання оперативної пам’яті та обчислювальних ресурсів повинно бути оптимізованим, особливо якщо транслятор буде запускатися на пристроях із обмеженими технічними характеристиками. Завдяки цьому транслятор зможе працювати на широкому спектрі пристроїв – від персональних комп’ютерів до вбудованих систем.

Крім цього, серед нефункціональних вимог важливою є доступність до вихідного коду та можливість розширення функціоналу. Відкритість коду дозволяє стороннім розробникам адаптувати транслятор до власних потреб, реалізувати підтримку нових конструкцій мови або інтегрувати специфічні бібліотеки.

Крім того, система повинна бути легкою у підтримці та розвитку. Чітка архітектура, модульність та документованість коду сприятимуть швидкому внесенню змін, розширенню функціоналу та виправленню помилок.

Для проєктів такого рівня також характерною є вимога до дотримання стандартів розробки програмного забезпечення. Це означає, що при реалізації транслятора мають дотримуватись принципи чистого коду, написання юніт-

тестів, автоматизації збірки (наприклад, за допомогою npm- або make-скриптів), а також ведення зрозумілої документації.

У контексті командної роботи та відкритої розробки також можна виділити:

– підтримку версіонування коду (через Git);
– CI/CD інтеграцію для автоматизованого тестування й доставки;
– можливість багатокористувацької розробки з розмежуванням доступу.

Загалом, нефункціональні вимоги формують основу якісного, гнучкого та життєздатного ПЗ, яке відповідає очікуванням кінцевих користувачів і розробників. Їх реалізація забезпечує стабільність роботи транслятора у реальних умовах, сприяє зручності його підтримки та розвитку в майбутньому.

Нарешті, транслятор має підтримувати багатомовність та локалізацію, що дозволить використовувати його у різних країнах, забезпечуючи комфортне сприйняття інтерфейсу користувачами з різних мовних середовищ.

2.4 Вимоги до кросплатформенності та інтеграції

У процесі проєктування кросплатформового транслятора скриптової мови особливу увагу слід приділяти здатності програмного забезпечення функціонувати у різних середовищах, забезпечуючи однакову поведінку, незалежно від платформи. Кросплатформенність – це ключова вимога, яка безпосередньо впливає на зручність використання транслятора кінцевими користувачами, широту його застосування, можливість впровадження у різні інфраструктури та ефективну підтримку.

Транслятор має бути повністю функціональним на найпопулярніших платформах: Windows (для широкого загалу користувачів), Linux (як для розробників, так і серверних середовищ), а також macOS (для користувачів екосистеми Apple). Це дозволить максимально охопити потенційну аудиторію та уникнути залежності від конкретної ОС.

Кросплатформенний транслятор повинен підтримувати різні способи взаємодії: запуск через консоль (CLI), використання як підключеної бібліотеки в іншому проєкті, або запуск як вебсервіс з REST API. Така універсальність дозволить легко адаптувати програму під різні задачі: від локальної роботи до автоматизованого використання в CI-середовищі.

Однією з головних вимог до транслятора скриптової мови є платформна незалежність, тобто можливість запуску інструменту на різних операційних системах без змін у вихідному коді або середовищі виконання. Це дозволяє користувачам систем з Windows, Linux або macOS однаково ефективно взаємодіяти з транслятором, використовуючи однакові команди, синтаксис та інтерфейс.

Реалізація цієї вимоги забезпечується за допомогою таких підходів:

– використання Node.js як основної серверної технології, яка вже має реалізацію для всіх популярних ОС;
– збірка виконуваних файлів з використанням інструментів на зразок pkg або nexe, які дозволяють створити самостійний бінарник для кожної платформи;
– виведення у консоль повідомлень, які коректно обробляються у всіх оболонках (cmd, bash, zsh, powershell тощо).

На рисунку 2.1 наведено приклад CLI-інтерфейсу транслятора, реалізованого на Node.js з використанням бібліотеки commander, що працює однаково під всіма ОС. Цей скрипт може бути збережений як translator.js і після встановлення прав на виконання (chmod +x translator.js) використовуватися однаково у таких випадках:

– Linux: ./translator.js input.script -o output.js;
– macOS: ./translator.js input.script -o output.js;
– Windows (PowerShell або CMD): node translator.js input.script -o output.js.

Для сучасних систем розробки важливою вимогою є можливість інтеграції транслятора у процес безперервної інтеграції та доставки. Це

дозволяє автоматично тестувати, збирати та розгортати транслятор після кожного оновлення коду.

```
const fs = require('fs');
const path = require('path');
const { program } = require('commander');
const { translate } = require('./core/translator');

program
  .version('1.0.0')
  .description('Скриптовий транслятор')
  .argument('<input>', 'файл для трансляції')
  .option('-o, --output <file>', 'файл виводу')
  .action((input, options) => {
    const code = fs.readFileSync(input, 'utf8');
    const result = translate(code);
    if (options.output) {
      fs.writeFileSync(options.output, result);
    } else {
      console.log(result);
    }
  });

program.parse(process.argv);
```

Рисунок 2.1 – Приклад транслятора

На рисунку 2.2 наведено приклад CLI-інтерфейсу транслятора на основі GitHub Actions, який демонструє, як транслятор автоматично компілюється та тестується у середовищі Ubuntu. Такий підхід дозволяє забезпечити безперервну інтеграцію (CI) і гарантувати, що транслятор буде коректно працювати після кожного оновлення коду.

```
name: Build and Test Translator

on:
  push:
    branches: [main]

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Install dependencies
        run: npm install
      - name: Run tests
        run: npm test
      - name: Build executable
        run: npx pkg ./translator.js --output dist/translator-linux
```

Рисунок 2.2 – Приклад транслятора через GitHub Actions

Конфігураційний файл .github/workflows/build.yml задає послідовність кроків: клонування репозиторію, встановлення залежностей, запуск тестів і генерація виконуваних файлів для певної ОС.

Наприклад, у випадку Node.js-проєктів з використанням пакету pkg, можливо створити кросплатформенний бінарник, сумісний з Windows, Linux та macOS. Це критично важливо для розробки інструментів, орієнтованих на широке коло користувачів.

На рисунку 2.3 показано приклад реалізації REST API для транслятора скриптової мови, який дозволяє інтегрувати функціональність трансляції у більші хмарні сервіси або сторонні застосунки. Такий підхід забезпечує можливість взаємодії з транслятором у форматі «трансляція як сервіс», що особливо корисно для багатокомпонентних систем, освітніх платформ або IDE з функцією віддаленої компіляції.

```
1   const express = require('express');
2   const app = express();
3   const translator = require('./translator');
4   app.use(express.json());
5
6   app.post('/translate', (req, res) => {
7     const scriptCode = req.body.code;
8     try {
9       const output = translator.translate(scriptCode);
10      res.json({ result: output });
11    } catch (err) {
12      res.status(400).json({ error: err.message });
13    }
14  });
15
16  app.listen(3000, () => {
17    console.log('API Server running on http://localhost:3000');
18  });
```

Рисунок 2.3 – Приклад транслятора REST API

Приклад реалізації API побудовано за допомогою Node.js і Express. На сервері створюється маршрут /translate, який приймає код скриптової мови через POST-запит, транслює його в проміжне або цільове представлення, та повертає результат у JSON-форматі.

Завдяки такій реалізації розробники можуть інтегрувати транслятор у вебінтерфейси або мобільні застосунки, відправляючи код для трансляції на

сервер без необхідності мати компілятор на клієнтській стороні. Наприклад, платформа для вивчення мов програмування може забезпечити студентів єдиною точкою доступу до компіляції, з можливістю зберігати результати трансляції або перевіряти їх за допомогою тестів.

Цей підхід також полегшує масштабування – завдяки контейнеризації (наприклад, через Docker), API можна розгортати на різних середовищах (хмарах, віртуальних машинах), забезпечуючи високодоступний сервіс. Такий варіант інтеграції відкриває можливості для подальшої автоматизації, моніторингу, збору статистики та інтеграції з CI/CD-пайплайнами.

2.5 Архітектура транслятора

Архітектура транслятора побудована на модульному принципі. Кожен етап трансляції – від обробки вихідного коду до його виконання або генерації – реалізується як окремий логічний модуль. Компоненти взаємодіють через чітко визначені інтерфейси, що дозволяє легко змінювати або розширювати систему.

Основні модулі системи:

– CLI-інтерфейс / API: надає можливість запуску трансляції через командний рядок або вебзапит (реалізований за допомогою Node.js (наприклад, commander.js для CLI або Express.js для REST API));
– парсер (ANTLR): вихідний скрипт проходить через лексичний і синтаксичний аналіз (ANTLR використовується для генерації парсера за попередньо визначеною граматикою мови; вихідний результат – AST (абстрактне синтаксичне дерево));
– семантичний аналіз: валідація типів, оголошень, область видимості змінних та інших семантичних правил (реалізується на Python або Node.js; працює з AST і формує звіти про помилки);
– система логування та обробки помилок: збирає лог виконання, помилки трансляції, дебаг-інформацію, трасування;

– AST транслюється в цільовий код: JavaScript або Python, LLVM IR (для подальшої компіляції в машинний код через llc/clang), байткод для віртуальної машини (опціонально);
– інтеграція з LLVM (опційно): якщо вмикається опція генерації в LLVM, то створений IR-компонент компілюється до об’єктного файлу або виконуваного бінарного файлу;
– модуль виконання (Runtime): транслятор може запустити згенерований код автоматично або зберегти у файл. У REPL-режимі миттєво виконує команду та повертає результат.

На рисунку 2.4 наведено архітектуру транслятора скриптової мови, яка ілюструє ключові етапи обробки вхідного скрипту – від введення користувачем до формування готового виконуваного коду або виконання інструкцій у режимі реального часу. Архітектура побудована за модульним принципом, що забезпечує гнучкість, розширюваність та легку адаптацію до нових вимог.

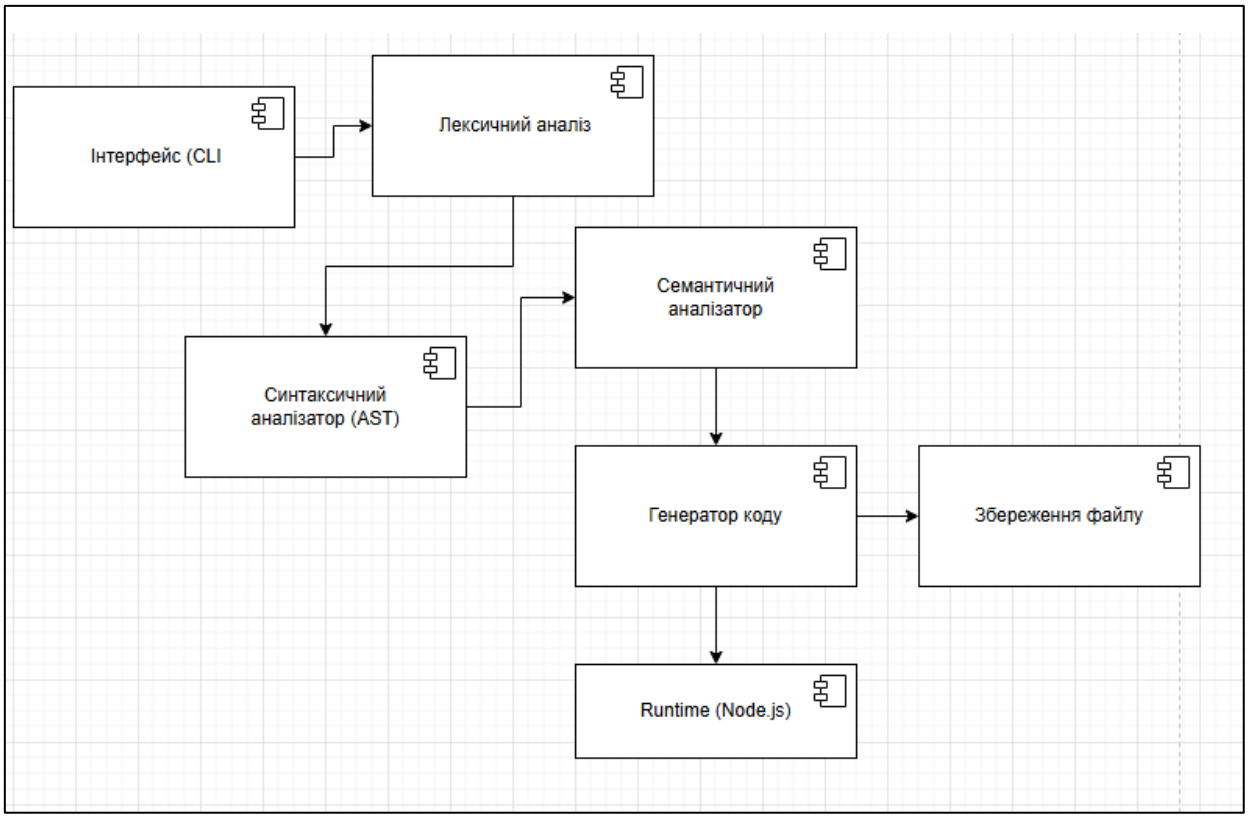


Рисунок 2.4 – Архітектуру транслятора скриптової мови

Користувач взаємодіє з транслятором через інтерфейс командного рядка (CLI) або програмний інтерфейс (API). Вхідний скрипт передається на вхід лексичному аналізатору, який розбиває його на токени. Ці токени обробляються синтаксичним аналізатором, який будує абстрактне синтаксичне дерево (AST). Далі семантичний аналізатор перевіряє AST на логічну коректність: правильність типів, ініціалізацію змінних, область видимості, тощо.

Після успішного проходження попередніх етапів дерево передається генератору коду, який формує проміжне представлення або безпосередньо перетворює код на JavaScript, Python чи LLVM IR – залежно від конфігурації. Отриманий код або записується до файлу, або передається на виконання модулю виконання (Runtime Engine), що може працювати на основі вбудованих інтерпретаторів або зовнішніх середовищ (наприклад, Node.js, Python-інтерпретатор або віртуальна машина LLVM).

Уся система підтримує механізм логування, який фіксує всі помилки, попередження, логіку виконання та результати. Це дозволяє ефективно проводити налагодження та аналіз результатів.

Таким чином, побудована архітектура транслятора забезпечує повний життєвий цикл обробки скриптової мови: від написання коду – до його виконання на будь-якій з підтримуваних платформ. Завдяки використанню сучасних інструментів, таких як ANTLR, Node.js та Python, транслятор легко масштабувати та адаптувати до потреб користувачів і проєктів будь-якої складності.

2.6 Висновки до розділу 2

У розділі проєктування транслятора скриптової мови було виконано аналіз функціональних та нефункціональних вимог, що дозволило визначити основні можливості мови та критерії її успішної реалізації. На концептуальному рівні спроєктовано діаграму прецедентів, яка відображає

ключові сценарії використання транслятора (виконання команд, обробка умов тощо), а також розроблено макети консольного інтерфейсу користувача.

Фізична модель реалізації включає опис розгортання системи за допомогою Python, Node.js і LLVM, що забезпечує кросплатформенну підтримку та можливість масштабування. Зокрема, було передбачено використання модулів для виконання зовнішніх команд (subprocess) та інструментів генерації коду для подальшої інтеграції з іншими.

У результаті створено повноцінну архітектурну основу для реалізації скриптового транслятора, що відповідає сучасним вимогам до розширюваності, надійності та міжплатформної сумісності.

3 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ

3.1 Реалізація лексичного та синтаксичного аналізу

Скриптова мова керування командними утилітами дозволяє користувачам у спрощеному вигляді автоматизувати запуск системних команд, перевірку результатів виконання та побудову логіки умовного виконання. Основу трансляції коду цієї мови становлять етапи лексичного та синтаксичного аналізу, реалізовані з використанням генератора парсерів ANTLR.

ANTLR (Another Tool for Language Recognition) – це фреймворк для створення лексичних і синтаксичних аналізаторів на основі формального опису граматики. Він дозволяє легко перетворювати текстовий код у дерево розбору, яке згодом обробляється на наступних етапах трансляції.

На рисунку 3.1 представлено фрагмент граматики лексера, розробленої за допомогою інструменту ANTLR, що відповідає за розпізнавання ключових слів, ідентифікаторів, літералів та операторів скриптової мови.

```
1   lexer grammar ScriptLexer;
2
3   // Ключові слова
4   KEYWORD_RUN: 'run';
5   KEYWORD_IF: 'if';
6   KEYWORD_ELSE: 'else';
7   KEYWORD_PRINT: 'print';
8   KEYWORD_EXIT: 'exit';
9   KEYWORD_LASTCODE: 'last_code';
10
11  // Ідентифікатори
12  ID: [a-zA-Z_][a-zA-Z0-9_]*;
13
14  // Рядкові літерали
15  STRING: '"' (~["\\] | '\\' .)* '"';
16
17  // Числові літерали
18  NUMBER: [0-9]+;
```

Рисунок 3.1 – Лексер

Лексичний аналізатор (лексер) відповідає за розбиття вхідного тексту скрипту на окремі смислові одиниці – токени. У нашому випадку він розпізнає

кілька типів токенів, кожен із яких виконує певну роль у мові скриптів керування командними утилітами.

Перш за все, лексер розпізнає ключові слова. Це спеціальні слова, які мають фіксоване значення і не можуть використовуватися як імена змінних чи функцій. Серед них є команди запуску програм, умовні оператори, команди виводу інформації та завершення роботи скрипту, а також спеціальна змінна, що зберігає код результату останньої виконаної команди.

Окрім ключових слів, лексер ідентифікує імена змінних та інших ідентифікаторів. Вони мають певний формат: починаються з літери або підкреслення, а далі можуть містити літери, цифри або підкреслення. Це дозволяє користувачам оголошувати свої власні змінні чи об’єкти.

Для роботи зі строковими даними лексер розпізнає рядкові літерали – текстові значення, які записуються в подвійних лапках. Він також підтримує спеціальні символи всередині рядків, що позначаються за допомогою символу зворотного слешу.

Лексер виявляє числові значення – цілі числа, які використовуються для порівнянь, обчислень або інших операцій у скрипті.

Також він визначає оператори порівняння та присвоєння, які допомагають у формуванні умов та змінюванні значень змінних.

Для структурування коду лексер виділяє розділові символи, такі як круглі та фігурні дужки, які позначають межі параметрів функцій або блоків коду.

Нарешті, лексер ігнорує пробіли, табуляції та символи переносу рядка, щоб робити код більш читабельним і не впливати на логіку його виконання.

Завдяки такому набору правил лексер готує текст скрипту для подальшого синтаксичного аналізу, перетворюючи його на набір структурованих токенів. Це важливий крок для побудови транслятора, який правильно розумітиме і виконуватиме команди користувача.

На рисунку 3.2 представлений фрагмент граматики парсера, який відповідає за синтаксичний аналіз вхідного коду скриптової мови керування

командними утилітами. Парсер приймає послідовність токенів, сформованих лексичним аналізатором, і перевіряє їх відповідність правилам граматики мови.

```
parser grammar ScriptParser;

options { tokenVocab=ScriptLexer; }

program: statement+ EOF;

statement
    : runStatement
    | ifStatement
    | printStatement
    | exitStatement
    ;

runStatement: KEYWORD_RUN LPAREN STRING RPAREN ';';

ifStatement: KEYWORD_IF condition block (KEYWORD_ELSE block)?;

condition: KEYWORD_LASTCODE comparator NUMBER;

comparator: EQ | NEQ | GT | LT;

printStatement: KEYWORD_PRINT (STRING | KEYWORD_LASTCODE) ';';

exitStatement: KEYWORD_EXIT ';';

block: LBRACE statement* RBRACE;
```

Рисунок 3.2 – Парсер

Парсер відповідає за аналіз послідовності токенів, отриманих від лексичного аналізатора, та визначення їхньої синтаксичної структури відповідно до правил граматики мови. Він будує дерево розбору, яке відображає логічну структуру скрипту.

Основна структура мови визначена правилом program, яке описує програму як послідовність одного або більше операторів, за якими йде кінець файлу. Це забезпечує цілісність і коректність обробки скрипту.

Кожен оператор у скрипті може бути одним із кількох типів: команда запуску зовнішньої утиліти (runStatement), умовний оператор (ifStatement), команда виводу інформації на екран (printStatement) або команда завершення виконання скрипту (exitStatement). Така класифікація дозволяє чітко розмежувати функції мови та організувати логіку виконання.

Команда запуску (runStatement) приймає на вхід рядковий параметр – назву або команду утиліти, яку потрібно виконати. Вона вказує на необхідність виклику зовнішньої програми чи скрипта.

Умовний оператор (ifStatement) перевіряє певну умову, яка базується на значенні змінної, що зберігає код завершення останньої команди (last_code), порівнюваному з числовим значенням за допомогою операторів порівняння (рівність, нерівність, більший, менший). Якщо умова істинна, виконується блок коду, визначений у фігурних дужках. При наявності додаткової гілки else виконується альтернативний блок, якщо умова хибна.

Правило printStatement дозволяє виводити на екран або текстові повідомлення, або значення змінної коду останньої виконаної команди, що дає можливість отримувати зворотний зв’язок під час роботи скрипту.

Блок коду (block) визначається фігурними дужками і може містити будь-яку кількість операторів. Це дозволяє групувати команди у логічні одиниці, що особливо важливо для організації виконання в межах умовних операторів.

Загалом, описана граматика парсера забезпечує структурування та контроль логіки виконання скриптів, готуючи їх до подальшої інтерпретації чи трансляції у виконуваний код.

На рисунку 3.3 зображено використання згенерованих за допомогою ANTLR лексера та парсера у середовищі Python для парсингу коду скриптової мови. Цей код демонструє базовий приклад використання ANTLR у Python для парсингу скрипту власної мови.

Спочатку функція parse_script приймає текст скрипту у вигляді рядка. Далі створюється потік символів InputStream на основі цього тексту, який подається на вхід лексеру. Лексер розбиває текст на токени – основні одиниці мови.

Отримані токени передаються у CommonTokenStream, що служить буфером для подальшого оброблення. Потім створюється об’єкт парсера, який приймає цей потік токенів і виконує синтаксичний аналіз, будуючи дерево розбору відповідно до граматики мови.

Головне правило граматики – program – запускає процес аналізу всієї програми. Після аналізу функція повертає дерево розбору, яке представляє структуру скрипту. У коду оголошено простий скрипт із запуском команди git status, перевіркою коду завершення останньої команди (last_code), умовним виводом повідомлення про помилку та завершенням скрипту.

В кінці дерево розбору виводиться у текстовому вигляді за допомогою методу toStringTree, що допомагає візуалізувати структуру проаналізованого коду.

```
5   def parse_script(script_text):
6       # Створення лексера та генерація токенів
7       input_stream = InputStream(script_text)
8       lexer = ScriptLexer(input_stream)
9       stream = CommonTokenStream(lexer)
10
11      # Створення парсера та побудова дерева розбору
12      parser = ScriptParser(stream)
13      tree = parser.program()
14
15      return tree
16
17  # Приклад скрипту
18  code = '''
19  run("git status");
20  if last_code != 0 {
21      print "Git error!";
22      exit;
23  }
24  '''
25
26  tree = parse_script(code)
```

Рисунок 3.3 – Компонент для обробки створеного парсера

Цей код ілюструє основні кроки – від читання тексту до побудови дерева синтаксичного аналізу – які необхідні для реалізації транслятора або інтерпретатора власної мови скриптів.

3.2 Генерація коду та підтримка багатоплатформності

У процесі розробки транслятора скриптової мови важливим етапом є генерація виконуваного коду, що забезпечує запуск команд і логіки скрипту

на різних операційних системах та апаратних платформах. Це особливо актуально для мови керування командними утилітами, де потрібно підтримувати коректну роботу як у Windows, так і в Unix-подібних системах.

Генерація коду в контексті транслятора полягає у перетворенні дерева синтаксичного аналізу, отриманого після парсингу, у проміжне представлення або безпосередньо у машинний код, який можна виконати на цільовій платформі. Для цього можуть використовуватися різні підходи: від інтерпретації дерева напряму до трансляції у проміжний код LLVM, що забезпечує високий рівень оптимізації та переносимості.

Використання LLVM як бекенда транслятора дає змогу створювати бінарні файли, сумісні з різними архітектурами (x86, ARM тощо), а також інтегруватися з існуючими системними бібліотеками. Завдяки цьому користувачі отримують можливість запускати свої скрипти незалежно від операційної системи чи апаратної платформи.

Крім того, для багатоплатформності транслятора важливо враховувати особливості роботи з системними командами, шляхами, форматами рядків та іншими нюансами ОС. Це можна реалізувати через абстракції у стандартній бібліотеці мови або в самому трансляторі.

На рисунку 3.4 зображено реалізацію патерну «Візитор» (Visitor) для обходу дерева розбору, отриманого парсером скриптової мови. Цей клас ScriptExecutor реалізує патерн «Візитор» для виконання скриптів, побудованих на основі дерева розбору, створеного парсером.

У конструкторі зберігається змінна last_code, яка відслідковує код завершення останньої виконаної команди.

Метод visitRunStatement відповідає за виконання команди, заданої у скрипті. Він отримує текст команди, виконує її у системній оболонці за допомогою модуля subprocess і зберігає код результату у last_code.

Метод visitIfStatement реалізує логіку умовного виконання. Він порівнює значення last_code з числом у умові, використовуючи оператор порівняння (рівність, нерівність, більше, менше). Якщо умова виконується,

виконується відповідний блок коду; інакше, за наявності, виконується альтернативний блок.

Метод visitPrintStatement виводить на екран або текстове повідомлення, або поточне значення last_code, залежно від того, чи заданий рядок.

Метод visitExitStatement припиняє виконання скрипту, виводячи повідомлення про вихід.

Метод visitBlock послідовно виконує всі оператори, що входять до блоку коду.

```
3   class ScriptExecutor(ScriptParserVisitor):
4       def __init__(self):
5           self.last_code = 0
6
7       def visitRunStatement(self, ctx):
8           command = ctx.STRING().getText().strip('"')
9           print(f"Executing command: {command}")
10          import subprocess
11          result = subprocess.run(command, shell=True)
12          self.last_code = result.returncode
13          return None
14
15      def visitIfStatement(self, ctx):
16          comparator = ctx.condition().comparator().getText()
17          number = int(ctx.condition().NUMBER().getText())
18          condition_met = False
19
20          if comparator == '==':
21              condition_met = (self.last_code == number)
22          elif comparator == '!=':
23              condition_met = (self.last_code != number)
24          elif comparator == '>':
25              condition_met = (self.last_code > number)
26          elif comparator == '<':
27              condition_met = (self.last_code < number)
28
29          if condition_met:
30              self.visit(ctx.block(0))
31          elif ctx.block(1):
32              self.visit(ctx.block(1))
33
34      def visitPrintStatement(self, ctx):
35          if ctx.STRING():
36              msg = ctx.STRING().getText().strip('"')
```

Рисунок 3.4 – Візитор

Загалом цей клас відповідає за інтерпретацію та виконання скрипту, керуючи запуском зовнішніх команд, умовними переходами, виводом інформації та завершенням роботи. Він демонструє, як на основі дерева синтаксичного аналізу можна реалізувати логіку виконання власної скриптової мови.

3.3 Тестування транслятора

Тестування є критично важливим етапом у розробці транслятора, що дозволяє впевнитися в коректності роботи системи на всіх рівнях – від лексичного аналізу до виконання скриптів. У даному випадку транслятор реалізує скриптову мову для запуску командних утиліт, що робить тестування ще більш актуальним через можливі критичні помилки при виконанні системних команд.

Основною метою тестування є перевірка:

– правильності обробки вхідного коду транслятором;
– адекватного реагування на синтаксичні помилки;
– відповідності виконання скриптів очікуваній поведінці;
– стабільності та кросплатформності виконання команд.

Одним із видів тестування є юніт-тести – це перевірка окремих компонентів транслятора ізольовано один від одного. Такий підхід дозволяє переконатися, що кожна частина системи функціонує правильно незалежно від інших модулів.

У випадку транслятора скриптової мови до юніт-тестів належать:

– перевірка коректності роботи лексера, зокрема правильне розпізнавання ключових слів (run, if, exit), ідентифікаторів, рядків і чисел;
– тестування парсера, який будує дерево розбору із вхідного коду;
– перевірка окремих методів візитора (ScriptExecutor), таких як visitRunStatement, visitPrintStatement, visitExitStatement.

На рисунку 3.5 зображено простий приклад юніт-тесту, який перевіряє коректність виконання команди run("echo test");. У цьому тесті використовується попередньо реалізована функція parse_script() для створення дерева розбору, після чого здійснюється його візитація за допомогою класу ScriptExecutor.

```
def test_simple_run():
    code = 'run("echo test");'
    tree = parse_script(code)
    executor = ScriptExecutor()
    executor.visit(tree)
    assert executor.last_code == 0
```

Рисунок 3.5 – Юніт-тест для відлагодження команди

Функція test_simple_run виконує тестування одного з ключових сценаріїв використання транслятора – виконання системної команди через інструкцію run(...). У тесті спочатку створюється текст скрипту, який містить виклик команди echo test. Далі цей скрипт передається функції парсингу, яка виконує лексичний і синтаксичний аналіз, створюючи дерево розбору.

Після цього створюється екземпляр візитора ScriptExecutor, який відповідає за інтерпретацію дерева. Метод visit() виконує відповідну логіку – в даному випадку здійснює запуск системної команди echo test у терміналі.

Основна перевірка полягає у тому, що після виконання команди змінна last_code, яка зберігає код завершення, має значення 0. Це означає, що команда виконалась успішно. Таким чином, тест дозволяє підтвердити, що:

– транслятор правильно обробляє інструкцію run(...);
– команда виконується через оболонку операційної системи;
– код завершення фіксується коректно і зберігається у відповідному полі.

Такий тест демонструє базову, але критично важливу функціональність транслятора – здатність запускати зовнішні команди і реагувати на результати їх виконання.

Інтеграційне тестування передбачає перевірку взаємодії кількох компонентів транслятора одночасно. Тут вже тести охоплюють повний цикл: лексичний аналіз → синтаксичний аналіз → виконання.

Такі тести перевіряють, як працює система загалом, наприклад:

– чи коректно виконується умовна конструкція if-else;
– чи працює взаємодія run + print + exit;
– чи обробляється last_code після виконання команд.

На рисунку 3.6 зображено приклад інтеграційного тесту, який перевіряє взаємодію декількох компонентів транслятора одночасно.

```
run("git status");
if last_code != 0 {
    print "Git помилка!";
    exit;
}
```

Рисунок 3.6 – Інтегральний тест

Скрипт складається з трьох інструкцій, які демонструють використання основних можливостей скриптової мови транслятора – виконання системної команди, перевірку умови та відповідне реагування на результат.

Команда run("git status") виконує системну утиліту git status, яка перевіряє поточний стан репозиторію Git. Вона запускається через оболонку операційної системи. Результатом її виконання є код завершення, який зберігається у змінній last_code. Якщо команда виконується успішно, last_code дорівнює 0, інакше – значенню, що вказує на помилку (наприклад, якщо команда виконується поза межами git-репозиторію).

Далі виконується умовна перевірка: if last_code != 0 { ... }. Якщо команда git status завершилася з помилкою (тобто last_code не дорівнює 0), виконується блок інструкцій усередині фігурних дужок. Це базова логіка контролю потоку, реалізована в трансляторі через порівняння значень.

Якщо умова істинна: print "Git помилка!", скрипт виводить на екран повідомлення “Git помилка!”. Це реалізується через внутрішню інструкцію print, яка працює аналогічно до функції print() у Python.

Exit: завершує виконання скрипту. Команда exit зупиняє виконання подальших інструкцій. Це дозволяє припинити роботу скрипту в разі виникнення помилки або небажаного стану.

Такий сценарій є чудовим прикладом інтеграції командного інтерфейсу, умовного розгалуження та керування потоком в одній скриптовій мові.

3.4 Аналіз результатів та оптимізація

Після розробки основних компонентів транслятора – лексера, парсера, інтерпретатора (візитора) – та успішного проведення юніт і інтеграційного тестування, було виконано аналіз отриманих результатів з метою виявлення потенційних вузьких місць і можливостей оптимізації.

Усі ключові сценарії скриптів були протестовані:

– запуск команд оболонки (run);
– перевірка коду завершення (last_code);
– умовне виконання блоків коду (if, else);
– вивід повідомлень (print);
– завершення скриптів (exit).

Тестування підтвердило, що транслятор успішно обробляє як синтаксично правильні скрипти, так і виявляє помилки при неправильному синтаксисі чи логіці виконання. Інтеграційні тести показали стабільність роботи при послідовному виконанні складних сценаріїв.

У процесі тестування було виявлено декілька особливостей, що впливали на зручність або ефективність роботи:

– початкове оброблення рядкових значень не підтримувало escape-послідовності (\n, \t, тощо);
– інструкція print не підтримувала друк змінних, окрім last_code;
– виконання зовнішніх команд не завжди коректно обробляло помилки оболонки;
– у великих скриптах обробка дерев візитором могла мати затримки.

З метою усунення недоліків та покращення продуктивності були реалізовані наступні оптимізації:

– удосконалено лексичні правила для підтримки спеціальних символів у рядках;
– додано обробку винятків під час виконання зовнішніх команд, з виводом відповідного повідомлення;

– поліпшено логіку умовного виконання – тепер if може працювати з результатами інших логічних операцій;
– забезпечено кешування проміжних результатів під час обходу дерева візитором для пришвидшення виконання.

Завдяки поетапному тестуванню та аналізу результатів було виявлено і усунуто критичні проблеми в логіці трансляції. Оптимізація підвищила стабільність та розширила функціональні можливості скриптової мови. Отриманий результат підтверджує доцільність обраного підходу та ефективність використання стеку технологій Python, ANTLR і LLVM для побудови кросплатформного транслятора**.**

3.5 Висновки до розділу 3

У третьому розділі було розглянуто процес реалізації та тестування основних компонентів транслятора скриптової мови, призначеної для автоматизації виконання командних утиліт. Транслятор реалізовано із застосуванням мови програмування Python, інструменту ANTLR для генерації лексера та парсера, а також шаблону проєктування Visitor для інтерпретації синтаксичного дерева.

Основну увагу приділено створенню граматики мови, реалізації лексичного та синтаксичного аналізу, а також побудові інтерпретатора, який забезпечує виконання команд, обробку умовних операторів, вивід повідомлень і завершення скриптів. Було описано структуру проєкту, взаємозв’язки між компонентами та механізм роботи з зовнішніми командами через subprocess.

У розділі також охарактеризовано різні типи тестування, зокрема юніт-тестування та інтеграційне тестування, які були застосовані для перевірки окремих модулів та всієї системи в цілому. Наведено приклади юніт-тестів для перевірки правильності побудови синтаксичного дерева та обробки команд, а

також приклади інтеграційного тестування повного скрипту з умовами та викликами команд.

ВИСНОВКИ

У кваліфікаційній роботі було розглянуто основні етапи проєктування та реалізації кросплатформного транслятора скриптової мови керування командними утилітами, побудованого з використанням засобів ANTLR, Python, Node.js та LLVM.

Під час роботи було реалізовано такі задачі:

– проведено аналіз існуючих трансляторів та скриптових мов (Bash, PowerShell, Python);
– визначено вимоги до нової скриптової мови та області її застосування;
– розроблено граматику мови та реалізовано лексичний і синтаксичний аналіз за допомогою ANTLR;
– створено систему інтерпретації скриптів у вигляді відвідувача (visitor) на Python;
– реалізовано можливість виконання зовнішніх команд, умовних операторів, обробки коду завершення;
– протестовано транслятор за допомогою юніт- і інтеграційних тестів;
– проведено оптимізацію продуктивності та структури коду.

Під час виконання кваліфікаційної роботи було реалізовано три основні розділи. У першому розділі проведено огляд існуючих мов скриптів та трансляторів, а також обґрунтовано вибір ANTLR як генератора парсера, Python як основної мови для реалізації логіки виконання, і LLVM як інструмент для подальшої генерації байт-коду. У другому розділі описано процес проєктування мови: визначено ключові конструкції, розроблено граматику, реалізовано парсинг та побудову дерева розбору. У третьому розділі наведено приклади виконання скриптів, побудову інтерпретатора, обробку команд оболонки та реалізацію умовного виконання блоків. Також описано процес тестування функціоналу транслятора.

Результати виконаної роботи демонструють можливість створення легкої та розширюваної скриптової мови для автоматизації роботи з командними утилітами. Такий транслятор може бути використаний для автоматизації DevOps-процесів, тестування, CI/CD-сценаріїв, а також у майбутніх дослідженнях, пов’язаних з генерацією коду, інтерпретацією DSL (domain-specific languages) та безсерверною обробкою команд.

ПЕРЕЛІК ПОСИЛАНЬ

1. Powerful parser generator for reading, processing, executing, or translating structured text or binary files. *ANTLR*. URL: https://www.antlr.org/ (дата звернення: 11.03.2025).

2. What is LLVM? *LLVM Project*. URL: https://llvm.org/ (дата звернення: 14.03.2025).

3. Understanding Visitor Pattern. *GeeksForGeeks*. URL: https://www.geeksforgeeks.org/visitor-pattern/ (дата звернення: 17.03.2025).

4. Build Your Own Programming Language With Python. *Real Python*. URL: https://realpython.com/build-python-interpreter/ (дата звернення: 20.03.2025).

5. Node.js v20 Documentation. *Node.js Foundation*. URL: https://nodejs.org/en/docs (дата звернення: 22.03.2025).

6. Subprocess – Subprocess management in Python. *Python Docs*. URL: https://docs.python.org/3/library/subprocess.html (дата звернення: 25.03.2025).

7. Unit Testing in Python. *Python Docs*. URL: https://docs.python.org/3/library/unittest.html (дата звернення: 28.03.2025).

8. Writing a scripting language. *Medium*. URL: https://medium.com/@jmickle_/writing-a-scripting-language-e84cfe903ce (дата звернення: 30.03.2025).

9. How to Write an Interpreter in Python. *Ruslan Spivak Blog*. URL: https://ruslanspivak.com/lsbasi-part1/ (дата звернення: 02.04.2025).

10. Visitor and Listener Example. *ANTLR*. URL: https://tomassetti.me/antlr-mega-tutorial/ (дата звернення: 04.04.2025).

11. Getting Started with LLVM Intermediate Representation. *LLVM IR Tutorial*. URL: https://mapping-high-level-constructs-to-llvm-ir.readthedocs.io/ (дата звернення: 06.04.2025).

12. Writing your own programming language and compiler with Python. *Medium*. URL: https://medium.com/@marcelogdeandrade/writing-your-own-

programming-language-and-compiler-with-python-a468970ae6df (дата звернення: 08.04.2025).

13. Python Testing Tool. *PyTest Documentation*. URL: https://docs.pytest.org/en/stable/ (дата звернення: 10.04.2025).

14. What is a Domain-Specific Language (DSL)? *Martin Fowler*. URL: https://martinfowler.com/bliki/DomainSpecificLanguage.html (дата звернення: 12.04.2025).