Міністерство освіти і науки України
Криворізький національний університет
Факультет інформаційних технологій
Кафедра комп’ютерних систем та мереж

Пояснювальна записка
до кваліфікаційної роботи бакалавра
за спеціальністю 123 «Комп’ютерна інженерія»

на тему: ОПТИМІЗУЮЧИЙ ІНТЕРПРЕТАТОР ДЛЯ МОВИ ПРОГРАМУВАННЯ

Проектував	_____	В. Р. Галімулін
Керівник роботи	_____	І. О. Музика
Нормоконтроль	_____	Д. І. Кузнєцов
Завідувач кафедри	_____	А. І. Купін

Кривий Ріг
2026

Криворізький національний університет
Факультет інформаційних технологій
Кафедра комп'ютерних систем та мереж

Ступінь вищої освіти бакалавр
Спеціальність 123 «Комп'ютерна інженерія»

ЗАТВЕРДЖУЮ
Завідувач кафедри, голова циклової комісії

_____ А. І. Купін

"____" _____ 20__ року

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

Галімулін Владислав Рауфович
(прізвище, ім'я, по батькові)

1. Тема роботи Оптимізуючий інтерпретатор мови програмування

керівник роботи Музика Іван Олегович, к.т.н., доц.,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом вищого навчального закладу від "27" січня 2026 року №64с

2. Строк подання студентом роботи 01.06.2026 р.

3. Вихідні дані до роботи моделі компіляторів та інтерпретаторів, моделі виконання інтерпретаторів, описи проміжних представлень програм, SSA, опис байткодної архітектури CPython

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Огляд оптимізацій
2. Проєктування оптимізуючого інтерпретатора
3. Перевірка оптимізацій

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень) схеми структур компілятора, інтерпретатора, моделей виконання, оптимізацій

6. Консультанти розділів роботи

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

7. Дата видачі завдання 01.02.2026 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Строк виконання етапів роботи	Примітка
1	Огляд стат. та дин. оптимізацій	15.04.2026	
2	Проєктування SSA	7.05.2026	
3	Проєктування статичних оптимізацій	18.05.2026	
4	Проєктування динамічних оптимізацій	25.05.2026	
5	Перевірка стат. та дин. оптимізацій	29.05.2026	

Студент _____ Галімулін В Р.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Музика І. О.
(підпис) (прізвище та ініціали)

РЕФЕРАТ

Пояснювальна записка: 66 сторінок, 25 рисунків, 22 використаних джерел.

Об’єкт дослідження – процеси оптимізації виконання програм у віртуальних машинах та байт-кодових інтерпретаторах.

Проєкт складається з трьох розділів.

У першому розділі розглянуто загальну структуру компіляторів та інтерпретаторів, принципи їхньої організації та відмінності між моделями виконання. Описано роль проміжного представлення (IR) у процесі аналізу та оптимізації програм, а також наведено основні його форми: деревоподібні (AST), інструкційні (TAC) та стекові представлення. Окремо розглянуто класи статичних і динамічних оптимізацій, що застосовуються для підвищення ефективності виконання програм.

У другому розділі виконано проєктування архітектури оптимізуючого інтерпретатора. Запропоновано використання SSA-представлення як основи для статичних оптимізацій та визначено послідовність оптимізаційних проходів, що включає поширення та згортання констант, strength reduction, peephole-оптимізації, відновлення узгодженості SSA та видалення мертвого коду. Також розглянуто динамічні оптимізації на прикладі інтерпретатора CPython, зокрема інлайнове кешування, спеціалізацію байт-коду та суперінструкції.

Третій розділ присвячено практичній перевірці запропонованих підходів. Проведено тестування статичних оптимізацій, яке підтвердило коректність роботи оптимізаційних проходів та їх здатність до послідовного спрощення проміжного представлення. Окремо проаналізовано поведінку динамічних оптимізацій у CPython, де підтверджено ефективність адаптивної спеціалізації байт-коду та зменшення накладних витрат інтерпретації після періоду розігріву (warm-up).

ОПТИМІЗУЮЧИЙ ІНТЕРПРЕТАТОР, SSA, ПРОМІЖНЕ ПРЕДСТАВЛЕННЯ, БАЙТ-КОД, СТАТИЧНІ ОПТИМІЗАЦІЇ, ДИНАМІЧНІ ОПТИМІЗАЦІЇ.

Explanatory note: 66 pages, 25 figures, 22 references.

The object of research is optimization processes in program execution within virtual machines and bytecode interpreters.

The project consists of three chapters.

The first chapter examines the general structure of compilers and interpreters, the principles of their organization, and the differences between execution models. The role of intermediate representation (IR) in program analysis and optimization is described, along with its main forms: tree-based (AST), instruction-based (TAC), and stack-based representations. Static and dynamic optimization classes used to improve program execution efficiency are also discussed.

The second chapter focuses on the design of an optimizing interpreter architecture. SSA-based representation is proposed as a foundation for static optimizations, and a sequence of optimization passes is defined, including constant propagation and folding, strength reduction, peephole optimizations, SSA consistency restoration, and dead code elimination. Dynamic optimizations are also considered using the CPython interpreter as an example, in particular inline caching, bytecode specialization, and superinstructions.

The third chapter presents a practical evaluation of the proposed approaches. Static optimizations are tested, confirming the correctness of optimization passes and their ability to progressively simplify intermediate representation. The behavior of dynamic optimizations in CPython is also analyzed, demonstrating the effectiveness of adaptive bytecode specialization and the reduction of interpretation overhead after a warm-up phase.

OPTIMIZING INTERPRETER, SSA, INTERMEDIATE REPRESENTATION, BYTECODE, STATIC OPTIMIZATIONS, DYNAMIC OPTIMIZATIONS.

ЗМІСТ

Реферат 4
Зміст 6
Перелік скорочень 7
Вступ 8
1. Аналіз методів оптимізації 9
1.1 Огляд оптимізації коду 9
1.2 Методи статичної оптимізації 12
1.3 Методи динамічної оптимізації 19
1.3 Види IR 22
1.4 Висновки 24
2. Проєктування інтерпретатора 25
2.1 Структура оптимізуючого інтерпретатора 25
2.2 Підготовка до статичних оптимізацій: мова програмування та її IR 27
2.3 Підготовка до статичних оптимізацій: створення SSA-форми та Def-Use ланцюгів 28
2.3 Статичні оптимізації: поширення/згортання констант 31
2.4 Статичні оптимізації: strength reduction 33
2.5 Статичні оптимізації: peephole 34
2.6 Статичні оптимізації: DCE 35
2.7 Статичні оптимізації: Стабілізація SSA та загальна структура оптимізатора 37
2.8 Динамічні оптимізації 39
2.9 Інлайнове кешування 41
2.10 Спеціалізація байт-коду 43
2.11 Суперінструкції 44
2.12 Висновки 46
3. Перевірка оптимізацій 47
3.1 Перевірка статичних оптимізацій 47
3.2 Перевірка динамічних оптимізацій: спеціалізація байт-коду 57
3.3 Перевірка динамічних оптимізацій: інлайнове кешування 61
3.5 Перевірка динамічних оптимізацій: суперінструкції 61
3.6 Висновки 63
Висновки 64
Список використаних джерел 65

ПЕРЕЛІК СКОРОЧЕНЬ

AST – Abstract Syntax Tree (абстрактне синтаксичне дерево);
CPU – Central Processing Unit (центральний процесор);
CPython – CPython (еталонна реалізація Python);
DCE – Dead Code Elimination (усунення мертвого коду);
GCC – GNU Compiler Collection (компіляторна система GNU);
IC – Inline Caching (інлайнове кешування);
IR – Intermediate Representation (проміжне представлення);
ISA – Instruction Set Architecture (архітектура набору команд);
JIT – Just-In-Time compilation (компіляція «на льоту»);
SSA – Static Single Assignment (форма статичного одноразового присвоєння);
TAC – Three Address Code (трьохадресний код);
VM – Virtual Machine (віртуальна машина);

ВСТУП

Сучасні програмні системи дедалі частіше потребують ефективних механізмів виконання коду, які поєднують гнучкість інтерпретації з продуктивністю компіляції. У цьому контексті особливого значення набувають оптимізуючі інтерпретатори, що використовують як статичні, так і динамічні методи оптимізації для підвищення швидкодії виконання програм без зміни їхньої семантики.

Актуальність дослідження зумовлена широким використанням віртуальних машин і байт-кодових інтерпретаторів у сучасних мовах програмування, де значна частина витрат продуктивності припадає на етап інтерпретації та диспетчеризації інструкцій. Застосування проміжного представлення (IR), а також сучасних технік оптимізації, таких як SSA-форма, спеціалізація байт-коду та інлайнове кешування, дозволяє суттєво зменшити ці накладні витрати.

Метою роботи є дослідження та проєктування архітектури оптимізуючого інтерпретатора, який поєднує статичні та динамічні оптимізації проміжного представлення для підвищення ефективності виконання програм.

Для досягнення поставленої мети було визначено такі завдання:

- проаналізувати структуру компіляторів та інтерпретаторів і роль проміжного представлення;
- дослідити основні методи статичної та динамічної оптимізації;
- розробити архітектуру оптимізуючого інтерпретатора з підтримкою SSA-представлення;
- реалізувати та протестувати набір статичних оптимізацій;
- проаналізувати динамічні оптимізації на прикладі CPython.

1. АНАЛІЗ МЕТОДІВ ОПТИМІЗАЦІЇ

1.1 Огляд оптимізації коду

Перед визначенням поняття оптимізація, варто розглянути типову структуру компілятора та інтерпретатора.

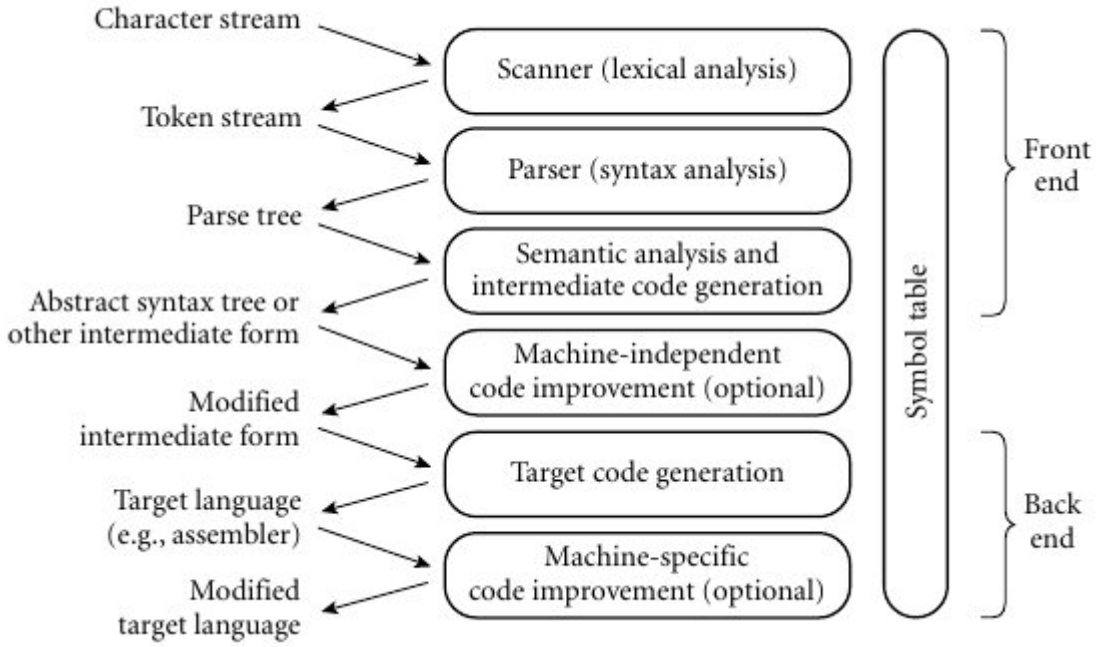

Рисунок 1.1 – Структурована модель компілятора

Компілятор – це програма, метою якою є трансляція вихідного коду на одній мові програмування в іншу, цільову мову. Зазвичай цільовою мовою є машинний код, компілятор перетворює код джерела на еквівалентний машинний код, структурує цей код в виконуваний файл (програму). В результаті цього отримуємо програму, яку можна запускати незалежно від компілятора.

Результуюча програма є націленою на певну архітектуру набору команд, підтримувану компілятором. Наприклад, у багатьох компіляторів є різні версії, кожна з яких підтримує різні архітектури – x86, ARM, SPARC. Винятком цьому є крос-компілятори – такі, які націлені на декілька платформ одночасно. Архітектура набору команд (ISA) являє собою абстрактний інтерфейс між апаратним забезпеченням процесора та програмним кодом, що на ньому виконується. Вона визначає модель поведінки процесора, яку спостерігає програміст або компілятор на рівні машинних інструкцій. Програми, написані високорівневими мовами, транслюються в машинний код, що відповідає конкретній ISA, після чого процесор виконує його через декодування та виконання відповідних інструкцій. Незважаючи на відмінності

в мікроархітектурній реалізації, процесори з однаковою ISA здатні виконувати той самий бінарний код, що забезпечує апаратну незалежність програм і їхню переносимість [1].

Компілятор має такі етапи:

- Сканер (лексичний аналіз): обробка вхідного тексту для формування потоку токенів. Токен – це найменша дозволена одиниця мови програмування, представляє собою рядок/слово, яке визнається мовою;
- Парсер (синтаксичний аналіз): парсер перетворює потік токенів в ієрархічну структуру дерева. Це дерево є проміжним представленням програми, структурно та функціонально еквівалентне до програмного коду;
- Семантичний аналіз та IR-генерація: на цьому етапі побудоване дерево аналізується на семантичні помилки. Після аналізатора виконується генерація еквівалентного IR (Intermediate Representation – проміжне представлення) коду;
- Машинно-незалежні оптимізації: модифікація проміжного представлення в такий спосіб, який сприятиме пришвидшенню програми;
- Генерація цільового коду;
- Машинно-залежні оптимізації: оптимізації, які враховують цільову архітектуру.

Етап машинно-незалежних оптимізацій – це статичні оптимізації. Вони можуть бути виконані на етапу аналізу програми, й не потребують додаткового контексту, який відомий лише під час виконання програми.

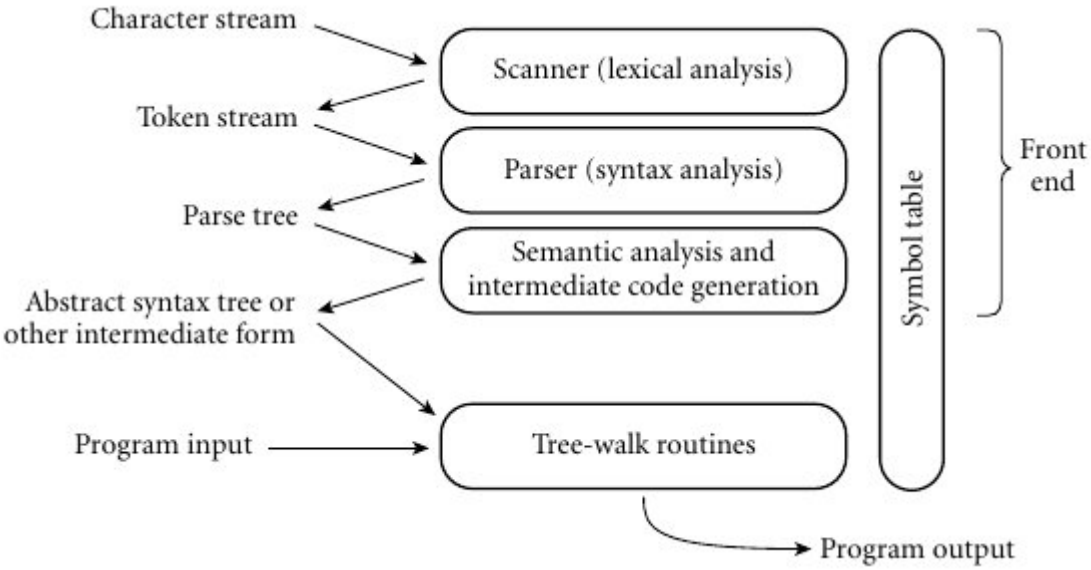

Рисунок 1.2 – Структура інтерпретатора

Інтерпретатор, з іншого боку, не транслює код в цільову мову чи виконуваний файл. Він виконує код одразу. Це дозволяє уникнути фаз,

пов’язаних з формуванням цільового коду. Натомість в інтерпретаторі їх замінює фаза, яка виконує код проміжного представлення.

Фази сканера, парсера та семантичного аналізу прийнято вважати фронт-ендом компілятора/інтерпретатора. Ця частина є спільною для обох структур.

Фази, пов’язані з генерацією цільового коду, архітектурно-залежні оптимізації та генерації, вважаються бек-ендом. Відповідно, в інтерпретаторах ця частина відсутня.

Зазвичай між фронт-ендом та бек-ендом поміщують мідл-енд – фази, що відповідають за оптимізацію проміжного представлення.

Дана робота розглядатиме оптимізації інтерпретатора як такі, що виконуються в мідл-енді.

Оптимізація коду – це процес удосконалення програмного коду з метою підвищення його ефективності за швидкодією, використанням пам’яті та інших обчислювальних ресурсів без зміни функціональної поведінки програми. До ключових аспектів оптимізації коду належать [2]:

- підвищення продуктивності: оптимізований код виконується швидше та потребує менше обчислювальних ресурсів;
- зменшення розміру коду: компактніші програми простіше розповсюджувати та розгортати;
- контроль часу компіляції: процес оптимізації не повинен суттєво збільшувати тривалість компіляції;
- збереження коректності: усі перетворення коду мають зберігати вихідну семантику та логіку програми.

Як зазначалося раніше, оптимізації поділяються на статичні та динамічні. Основна відмінність між ними полягає в етапі виконання та обсязі доступного контексту. Додатково вони відрізняються цілями, методами та характеристиками [3]:

- статичні оптимізації спрямовані на підвищення продуктивності шляхом аналізу коду та структур даних;
- динамічні оптимізації підвищують ефективність виконання через аналіз поведінки програми під час запуску та її адаптацію;
- основні методи статичної оптимізації включають перетворення коду, оптимізацію структур даних і розгортання циклів;
- для динамічних оптимізацій характерні JIT-компіляція, профілювання та адаптивні техніки;
- перевагою статичних методів є відсутність накладних витрат під час виконання та можливість заздалегідь застосувати глобальні покращення;
- динамічні методи, навпаки, дозволяють адаптуватися до реальних умов виконання, але потребують додаткових ресурсів під час роботи програми;
- недоліком статичних оптимізацій є відсутність інформації про runtime-поведінку;

– недоліком динамічних оптимізацій є накладні витрати та потенційна неефективність у деяких сценаріях.

1.2 Методи статичної оптимізації

Серед методів статичної оптимізації розглянемо наступні:

– DCE;
– Згортання констант;
– Поширення констант;
– Peephole;
– Strength reduction.

DCE (Dead Code Elimination – усунення “мертвого” коду) – є однією з найпоширеніших практично оптимізацій, результат якої можна доволі легко побачити.

Видалення мертвого коду (DCE) є однією з базових оптимізацій компілятора, метою якої є усунення інструкцій, що не впливають на спостережувану поведінку програми. Застосування DCE сприяє зменшенню обсягу програмного коду, зниженню витрат ресурсів під час виконання та скороченню кількості зайвих обчислень. Також ця оптимізація може спрощувати структуру програми, підвищуючи ефективність подальших етапів оптимізації. До мертвого коду відносять як недосяжні ділянки програми, так і обчислення, результати яких ніколи не використовуються [4].

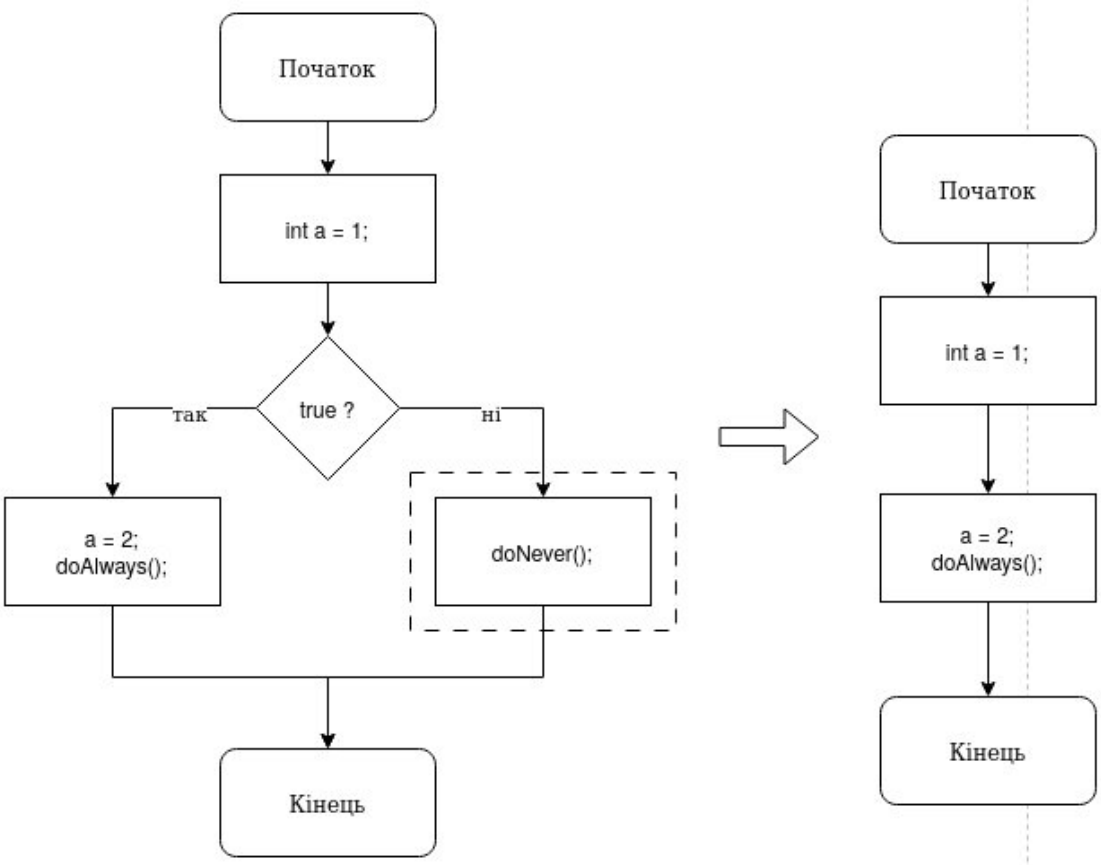

Рисунок 1.3 – Схема оптимізації програми з мертвим кодом

На схемі пунктиром обведено блоки, які містять мертвий код.

В лістингу 1.1 наведено код мовою C, який можна оптимізувати за допомогою DCE.

Лістинг 1.1 – Приклад DCE

```
int a = 1;
if (true) {
    a = 2;
    doAlways();
}
else {
    doNever();
}
```

Оскільки умова перевірки в блоці if завжди є істинною, то блок else ніколи не виконається, тобто є мертвим кодом. За допомогою DCE, цей код, очевидно непотрібний, не потрапить в кінцеву програму.

Менш очевидним з точки зору оптимізатора є ініціалізація з присвоєнням змінної a.

По-перше, тому що це інший тип твердження – в блоці if використовується лише присвоєння.

По-друге, тому що цей мертвий код стає очевидним лише під час аналізу DCE по умовних блоках – оскільки в умовному переході є зміна значення a, при тому що попереднє значення не було ніде використаним до цього. Хоча й в альтернативному блоці немає зміни значення, проте цей блок ніколи не виконується.

Оптимізатору важко аналізувати такий код “як є”, а деякі оптимізації й не є можливими або дуже важко підтримуються. Тому на вхід оптимізатора подають IR – проміжне представлення програми. В ньому код може виглядати зовсім інакше, не мати “синтаксичного цукру”, проте бути функціонально еквівалентним.

Проміжне представлення (IR) — це внутрішня форма подання програми, яку компілятор використовує для аналізу, оптимізації та генерації коду. Під час трансляції може застосовуватися одне або кілька IR різних рівнів абстракції. Усі подальші перетворення виконуються над IR, а його властивості безпосередньо впливають на доступні компілятору методи аналізу й оптимізації. Використання проміжного представлення дозволяє накопичувати інформацію про програму на одних проходах і використовувати її на інших, що сприяє генерації ефективнішого коду [5].

Наприклад, процес виконання C# – від аналізу коду до отримання результатів програми, передбачає декілька IR, кожна з яких релевантна на своїй фазі:

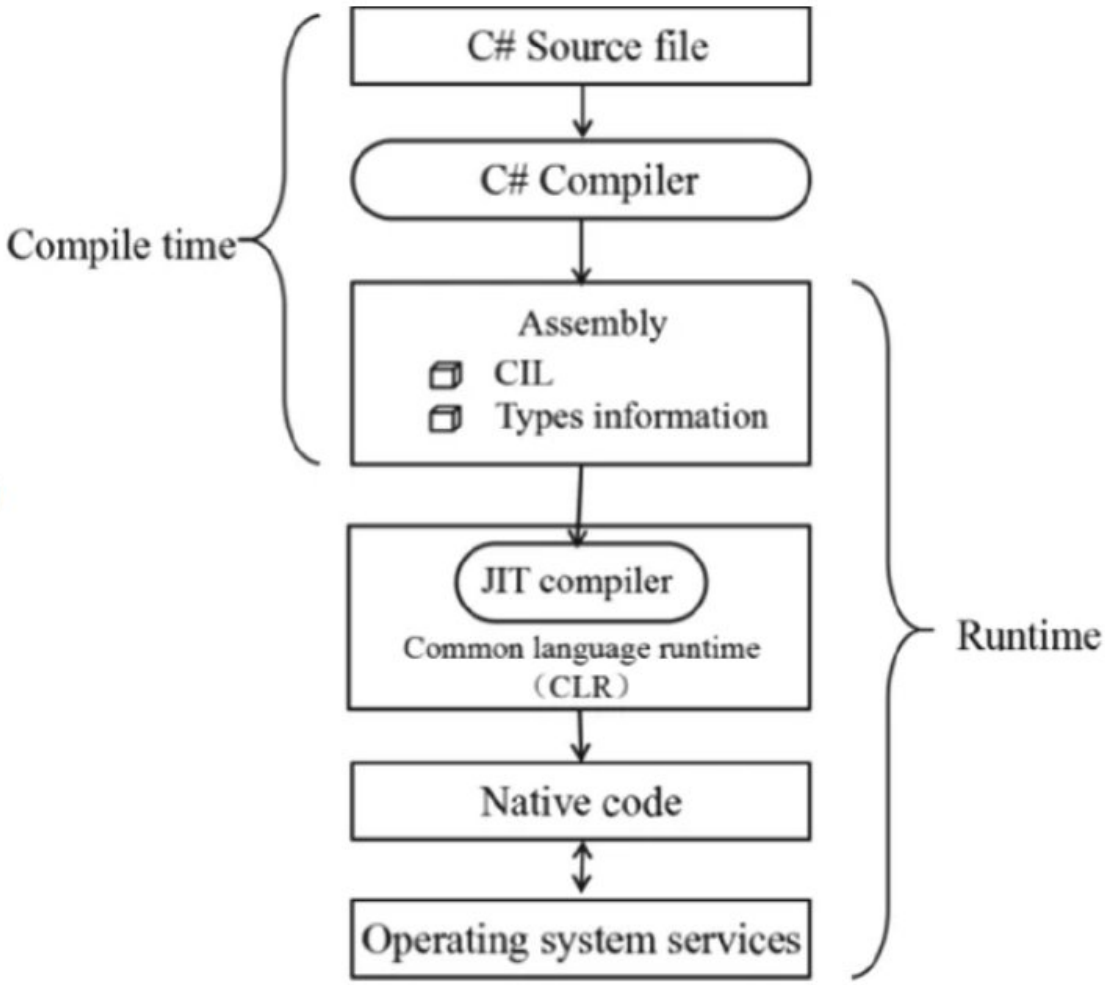

Рисунок 1.4 – Процес виконання C# програми

Варто зазначити, що IR не є концепцією, використовуваною виключно в компіляторі. IR може бути застосованим й в інтерпретаторі – всюди, де це є потрібним.

Наступна статична оптимізація – **згортання констант**. Згортання констант дозволяє обчислити результат виразу, який складається лише з констант, під час аналізу програми. Це безпосередньо впливає на час виконання програми, оскільки вже оптимізована програма не матиме цих зайвих обчислень.

Згортання констант – це метод оптимізації, при якому вирази обчислюються заздалегідь з метою економії часу виконання. Вирази, що дають у результаті константу, обчислюються під час компіляції, а отримані значення зберігаються у відповідних змінних. Цей метод також дозволяє зменшити розмір коду та застосовується для скорочення часу виконання, допомагає в ефективному управлінні пам'яттю.

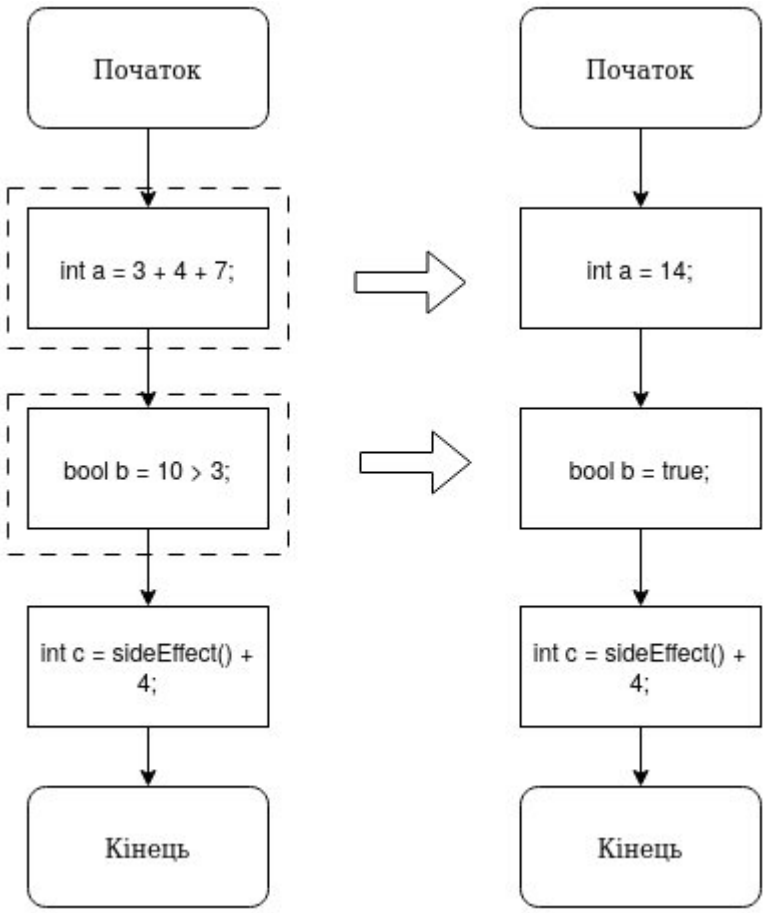

Рисунок 1.5 – Схема оптимізації програми з згортанням констант

На схемі пунктиром обведено блок, в якому можна згорнути константи.

Лістинг 1.2 – Приклад згортання констант

```
int a = 3 + 4 + 7;
bool b = 10 > 4;
int c = sideEffect() + 4;
```

Внаслідок згортання констант змінна a набуде лише одного значення, яке є результатом операцій з константами на ньому – значення 14. Так само щодо b – змінна набуде значення true. Як й всі інші статичні оптимізації, згортання констант відбувається до виконання коду – тому в кінцевій програмі значення вже буде оптимізованим.

Важливим обмеженням згортання констант є те, що воно не може застосовуватись там, де є ризик побічних ефектів. На прикладі це стосується присвоєння до змінної c.

Побічним ефектом називають зміну стану програми, яка відбувається під час виконання операції. Типовими прикладами є запис у пам’ять, виділення пам’яті, операції введення-виведення та виникнення винятків [6].

Присвоєння змінної c можна було б згорнути тоді, коли функція sideEffect не створювала побічних ефектів. В будь-якому разі, для даного випадку ця оптимізація потребує аналізу не лише виразу присвоєння, а й функцій, результати викликів яких є операндами виразу.

Оптимізатори зазвичай мають два підходи до вирішення цієї проблеми.

Перший підхід – не оптимізовувати вирази, які потенційно можуть мати побічні ефекти. В такому разі оптимізація або взагалі не виконується, або виконується на іншому рівні іншим оптимізатором, який бере до уваги вміст функцій.

Другий підхід – виконувати повний аналіз операндів на побічні ефекти, й лише після доведення факту того, що жоден з операндів не має побічних ефектів, застосовувати оптимізацію.

Близькою до згортання констант є оптимізація **поширення констант**. Ця оптимізація підставляє значення замість змінних там, де вони є константами. Це дозволяє уникнути зайвих операцій над цими змінними.

Поширення констант ґрунтується на аналізі досяжних визначень. Якщо в усіх точках, з яких змінна може отримати своє значення, їй присвоюється одна й та сама константа, то таку змінну можна замінити безпосередньо цією константою. Іншими словами, коли значення змінної однозначно відоме під час аналізу програми, усі її використання можуть бути замінені відповідним константним значенням. Такі підстановки виконуються з урахуванням графа потоку керування та можуть поширюватися на всі досяжні використання змінної [7].

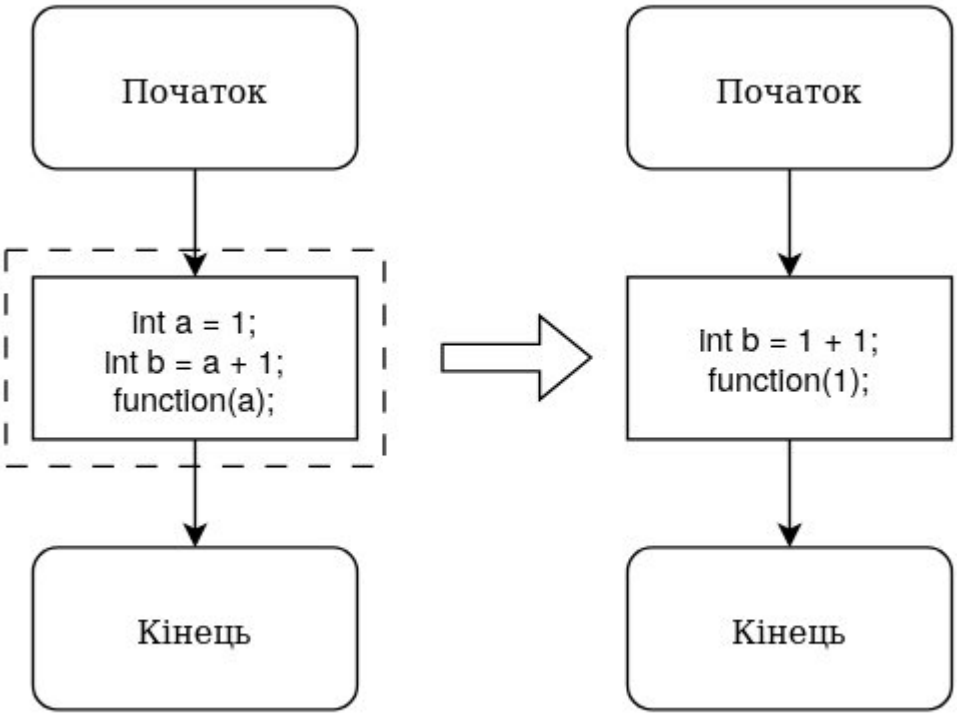

Рисунок 1.6 – Схема оптимізації програми з поширенням констант

Лістинг 1.3 – Приклад поширення констант

```
int a = 1;
int b = a + 1;
function(a);
```

В даному прикладі оптимізатор може побачити, що змінна a є незмінною на цій ділянці, й поширити її значення там, де вона використовується, замінивши використання самої змінної на значення, яке їй задане.

З викликом функцій може бути одна особливість – якщо функція, в якій за аргумент передано змінну, яку можна оптимізувати, приймає цей аргумент за посиланням, то оптимізація є шкідливою в такому разі.

Як вже згадувалося, ця оптимізація близько пов’язана з розгортанням констант. На цьому прикладі це можна побачити – якщо спершу виконати оптимізацію поширення констант, то вираз присвоєння змінній b перетвориться в вираз, який містить лише константні оператори.

В такому разі можна застосувати оптимізацію згортання констант, яка оптимізує присвоєння змінній b.

Наступна оптимізація – **“peephole”**. Вона працює з ланцюжками інструкцій, зазвичай у IR/асемблерному коді, й має на меті спрощення цих ланцюгів.

Peephole-оптимізація полягає в аналізі коротких послідовностей інструкцій з метою виявлення локальних покращень. Вона працює шляхом ковзного «вікна», яке проходить по коду та шукає оптимізаційні шаблони. При їх виявленні відповідні ділянки коду замінюються більш ефективними інструкціями [8].

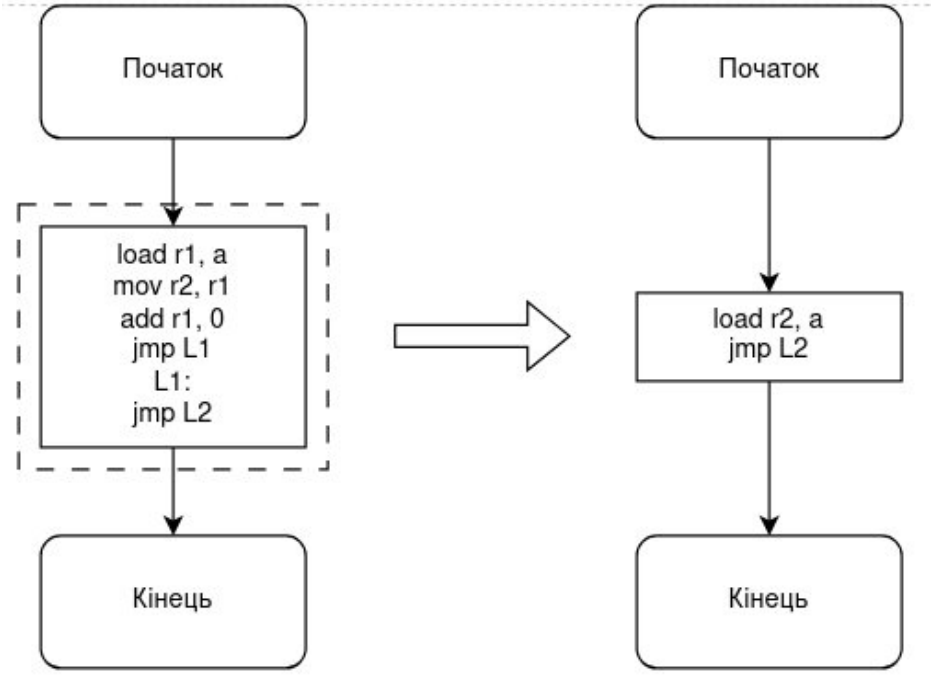

Рисунок 1.7 – Схема оптимізації програми з peephole

Лістинг 1.4 – Приклад peephole

```
load r1, a
mov r2, r1
add r1, 0
jmp L1
```

```
L1:
jmp L2
```

Приклад лістингу навмисно зроблено у вигляді IR, а не високорівневої мови, тому що оптимізація ланцюгів інструкцій краще видна з IR. Сама peephole-оптимізація є низькорівневою й працює на рівні IR/асемблерного коду.

Peephole оглядає локальні ділянки коду на можливість оптимізацій. Ця локальна ділянка є рухомою, тобто рухається далі по коду.

Лістинг 1.5 – Приклад peephole, після оптимізації

```
load r2, a
jmp L2
```

В прикладі було оптимізовано зайве завантаження в r1 та перехід по мітці L1, оскільки на початку мітки одразу йде перехід до L2.

Strength reduction (зниження вартості операцій) – це оптимізація, що передбачає заміну дорогих з обчислювальної точки зору операцій на еквівалентні, але більш ефективні інструкції цільової архітектури. Ідея полягає в тому, що деякі низькорівневі операції виконуються значно швидше за інші та можуть слугувати ефективною реалізацією складніших виразів. Наприклад, обчислення x^2 доцільніше реалізувати як множення x * x, ніж як виклик загальної функції піднесення до степеня. Множення або ділення на степені двійки зазвичай замінюють побітовими зсувами, а ділення з плаваючою комою на константу — множенням на обернену константу, що, як правило, є ефективнішим з точки зору виконання [9].

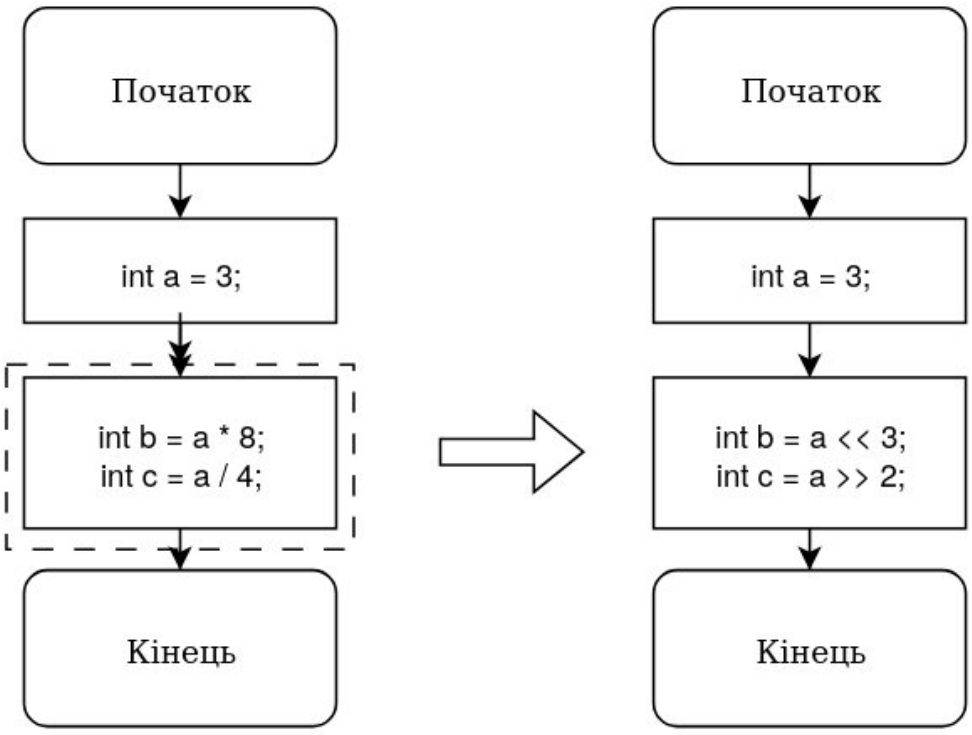

Рисунок 1.8 – Схема оптимізації програми з пониженням складності

Лістинг 1.7 – Приклад пониження складності

```
int a = 3;
int b = a * 8;
int c = a / 4;
int d = a + a + a + a;
```

Окрім заміни множення на побітові зсуви, strength reduction також включає заміну ділення на ступені двійки на операції зсуву вправо, а також оптимізацію повторюваних додавань.

1.3 Методи динамічної оптимізації

Серед методів динамічної оптимізації розглянемо наступні:

- Інлайнове кешування;
- Спеціалізація байткоду;
- Суперінструкції.

Байт-код — це проміжне представлення програми у вигляді об’єктного коду, яке інтерпретатор перетворює на машинні інструкції, придатні для виконання процесором. Зазвичай інтерпретатор реалізується як віртуальна машина, що виконує трансляцію байт-коду відповідно до конкретної цільової платформи. Використання байт-коду дозволяє виконувати компіляцію вихідного коду лише один раз, після чого він може запускатися на різних платформах за рахунок відповідного інтерпретатора, який перетворює його у машинний код, зрозумілий операційній системі та процесору [10].

Віртуальна машина виконує програму, подану як послідовність інструкцій. Кожна інструкція визначається операційним кодом (opcode), який є числовим ідентифікатором; зазвичай він займає один байт, що й зумовило термін «байт-код». Деякі інструкції містять додаткові операнди, які розташовуються після opcode у потоці виконання. Принцип роботи віртуальної машини подібний до роботи процесора [11]:

- з пам’яті зчитується наступна інструкція та виконується її декодування;
- завантажуються операнди, обчислюється результат і оновлюється стан системи;
- цей цикл повторюється для наступних інструкцій.

Інлайнове кешування (inline caching, IC) – це техніка динамічної оптимізації, за якої інтерпретатор зберігає результати дорогих операцій безпосередньо в місці їх виконання. Спочатку виконується повільний «базовий» шлях, під час якого аналізується структура об’єкта. Отримана інформація надалі кешується в точці доступу, що дозволяє уникати повторного аналізу під час наступних звернень. Основна ідея полягає у створенні швидкого шляху виконання для доступу до властивостей об’єкта, який використовується тоді, коли припущення щодо його типу та структури залишаються коректними. Таким чином, інформація про вже зустрінуті

об’єкти зберігається безпосередньо в механізмі доступу до даних, що й визначає назву підходу. У розширених варіантах інлайнове кешування може підтримувати кілька альтернативних швидких шляхів, що відповідає поліморфному режиму роботи [12].

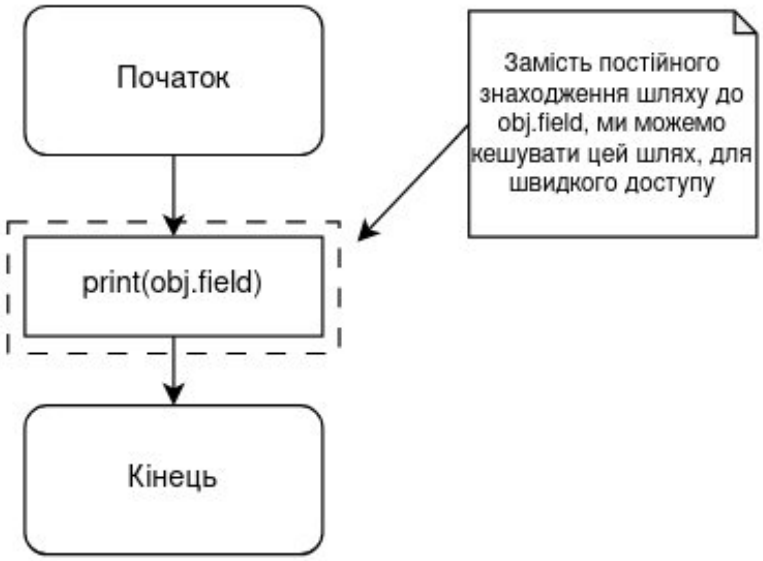

Рисунок 1.9 – Схема оптимізації програми з інлайновим кешуванням

Інлайнове кешування залежить від інформації про структуру об’єкта, інформацію про який намагаємося кешувати. Наприклад, якщо інтерпретатор в момент виклику функції print з аргументом obj.field очікує лише один конкретний тип, то й шлях буде єдиним. В такому разі відбувається мономорфне кешування – для одного типу.

Проте рано чи пізно виникатиме потреба в доступу до об’єкта, тип якого є динамічним. Наприклад, якщо б функція print приймала довільний тип, й очікувала в ньому декілька фактичних типів. В такому разі, інлайнове кешування повинно містити в собі різні знання про різні структури – тобто бути поліморфним. Важливо відрізняти поліморфний кеш від мегаморфного – поліморфний зберігає декілька часто використовуваних шляхів, а мегаморфний містить занадто багато різних шляхів для різних типів. В такому разі виграшу по швидкодії майже немає, а кеш, наповнений усіма можливими шляхами займає значно більше пам’яті.

При оптимізації **спеціалізації байткоду** інтерпретатор переписує або замінює універсальні інструкції на більш вузькі й швидкі варіанти під час виконання програми, використовуючи фактичні типи та значення, які спостерігаються в рантаймі.

Спеціалізація байткоду є технікою динамічної оптимізації, що підвищує швидкість виконання програми шляхом модифікації інструкцій відповідно до фактичних умов рантайму. Загальні інструкції замінюються на більш вузькоспеціалізовані та ефективні варіанти, які краще відповідають конкретним спостережуваним ситуаціям під час виконання. У межах цього підходу кожна інструкція може самостійно оновлювати свою реалізацію,

використовуючи вбудовані механізми кешування для накопичення та застосування інформації про виконання [13].

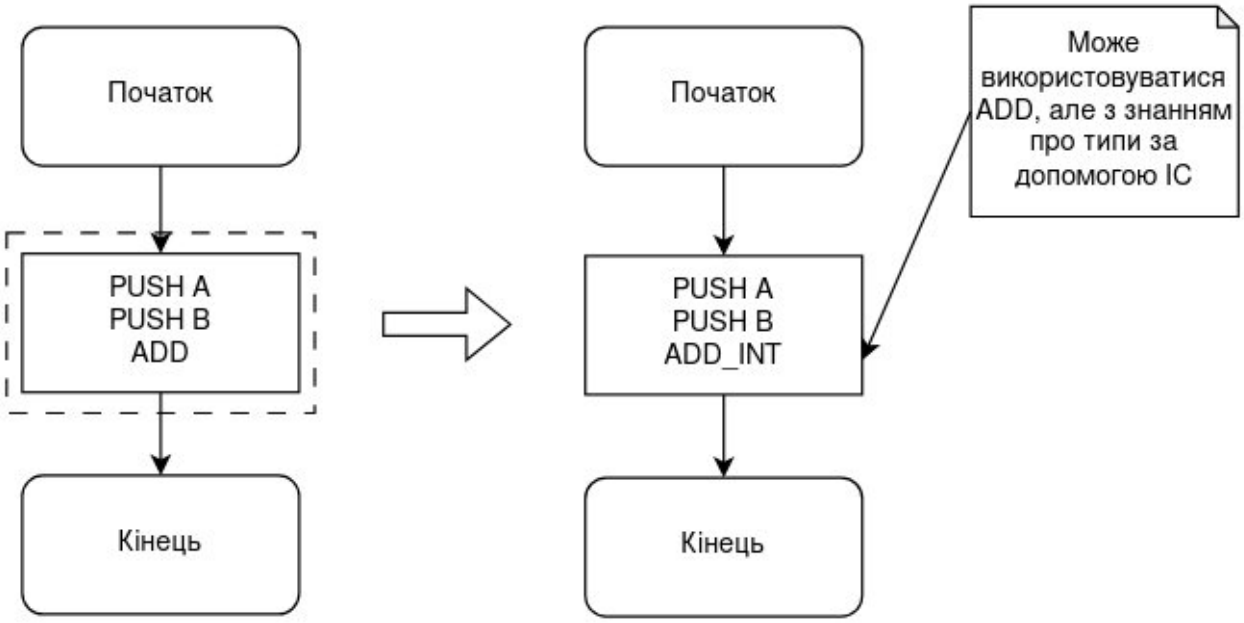

Рисунок 1.10 – Схема оптимізації програми з спеціалізацією байткоду

Спеціалізація байткоду може існувати у двох видах:

– Переписування байткоду: коли узагальнена інструкція замінюється більш конкретною, у прикладі з ADD в ADD_INT;
– Інлайнове кешування: коли сама байткод-інструкція залишається незмінною, але всередині неї використовується IC (наприклад, кешуються типи операндів і відповідний швидкий шлях виконання), що дозволяє уникати повторного визначення необхідної реалізації операції.

Суперінструкції є динамічною оптимізацією, при якій послідовності часто виконуваних інструкцій байткоду об’єднуються в одну спеціалізовану інструкцію. Це дозволяє зменшити кількість циклів інтерпретації, скоротивши накладні витрати на цикл обробки інструкції та підвищивши ефективність виконання гарячих ділянок коду.

Суперінструкції зменшують кількість виконуваних операцій шляхом об’єднання кількох базових інструкцій в одну. Наприклад, додавання константи до змінної зазвичай вимагає послідовного виконання завантаження значення, завантаження константи, операції додавання та запису результату. У випадку суперінструкції ці кроки замінюються однією узагальненою операцією, що виконує їх одразу. Водночас підхід обмежений розміром простору opcode та залежністю ефективного набору суперінструкцій від характеру навантаження [14].

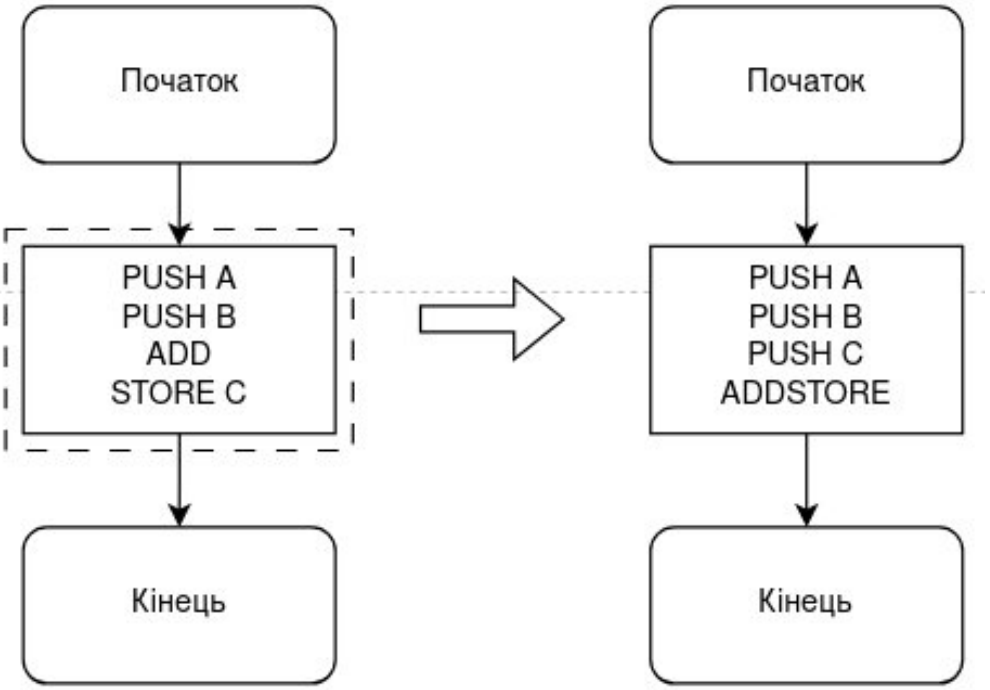

Рисунок 1.11 – Схема оптимізації програми з суперінструкціями

Суперінструкції дозволяють скоротити кількість переходів через інтерпретаторний цикл, зменшити кількість непрямих переходів і покращити локальність виконання коду. У результаті гарячі ділянки програми виконуються ефективніше за рахунок зменшення загальних накладних витрат диспетчеризації, що є одним із основних джерел втрат продуктивності в інтерпретаторах.

Диспетчеризація інструкцій – це механізм, що визначає, яку операцію слід виконати під час інтерпретації байт-коду. На кожному кроці інтерпретатор зчитує opcode з потоку інструкцій і переходить до відповідного обробника, який реалізує його семантику. Оскільки цей процес повторюється для кожної інструкції, він суттєво впливає на продуктивність віртуальної машини, а швидкість декодування та переходів напряму визначає загальну швидкодію інтерпретації [15].

1.3 Види IR

IR можна розділити на два загальні класи:
- На основі дерева (tree-based);
- На основі інструкцій (instruction-based).

IR на основі дерева представлений як дерево операцій, де вузли є операціями, а їх листя – операндами. Прикладом такої IR є AST. AST – це результат етапу парсингу. В типовому інтерпретаторі/компіляторі парсер продукує AST, який передається в наступні фази, й з якого формується IR нижчих рівнів – ці IR вже на основі інструкцій.

В абстрактному синтаксичному дереві (AST) кожен внутрішній вузол відповідає операції, тоді як його дочірні вузли представляють операнди цієї операції. Загалом будь-яку мовну конструкцію можна відобразити у вигляді дерева, де відповідний оператор задається як вузол, а його семантично значущі частини інтерпретуються як операнди [9].

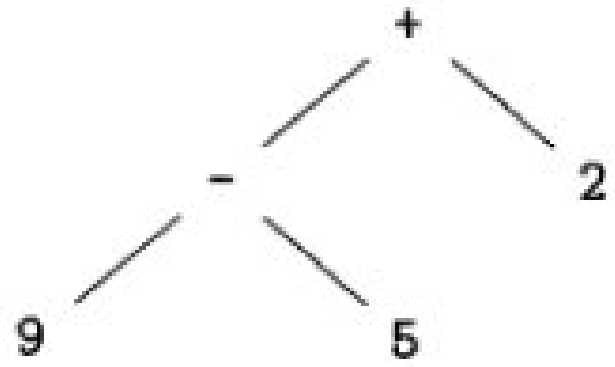

Рисунок 1.12 – AST для виразу 9-5+2

Варто зазначити, що програмний код може бути повністю представлений у вигляді AST – зазвичай так й відбувається. Парсер формує AST – це дерево зазвичай не містить семантичної інформації. Проте на його основі можна побудувати типізований AST – такий, що містить семантичну інформацію для вузлів, для яких вона потрібна.

На нижчих рівнях використовуються IR на основі інструкцій. Поширеною IR на основі інструкцій є TAC (Three Address Code – IR з трьохадресними кодами). В цій IR всі інструкції мають не більше трьох операндів.

TAC (трьохадресний код) – це проміжне представлення, яке використовується в компіляторах для спрощення складних виразів шляхом їх розбиття на послідовність простих інструкцій. Кожна така інструкція має не більше трьох адрес: дві для операндів і одну для результату. Отримані значення зазвичай зберігаються в тимчасових змінних. Завдяки цьому TAC явно фіксує порядок виконання операцій, що полегшує подальші оптимізації та генерацію машинного коду [16].

Лістинг 1.8 – Приклад TAC для виразу a*4-(-b+5)

```
t1 = a*4
t2 = -b
t3 = t2+5
t4 = t1-t3
```

Хоча й TAC буквально означає трьохадресний код, не всі операції повинні мати рівно 3 операнди – в прикладі це можна побачити на присвоєння унарного мінуса по b до тимчасового регістру t2. Проте будь-яка операція не може мати більше 3 операндів.

Іншою реалізацією IR на основі інструкцій є стекова IR (stack-based). Це проміжне представлення, де обчислення виконується через стек, а не через явні регістри чи змінні. В ній операнди не кодуються явно, а беруться з вершини стеку.

Стекове проміжне представлення (IR) організовує обчислення через стекову модель виконання. Назва походить від того, що інструкції в такій моделі переважно працюють зі стеком: вони поміщають значення на стек і знімають їх для обчислень. Такий підхід забезпечує компактність коду, оскільки більшість інструкцій не потребує явних операндів. Водночас це ускладнює аналіз програми та реалізацію оптимізацій [17].

Лістинг 1.9 – Приклад стекової IR для виразу a*4-(-b+5)

```
PUSH A
PUSH 4
MUL
STORE T1
PUSH B
NEG
STORE T2
PUSH T2
PUSH 5
ADD
STORE T3
PUSH T1
PUSH T3
SUB
STORE T4
```

1.4 Висновки

У даному розділі розглянуто загальну структуру компіляторів та інтерпретаторів, а також відмінності між їхніми моделями виконання. Показано роль використання проміжного представлення (IR) як основи для аналізу та оптимізацій.

Описано основні класи оптимізацій: статичні (DCE, згортання та поширення констант, peephole, strength reduction) і динамічні (інлайнове кешування, спеціалізація байткоду, суперінструкції), які відповідно застосовуються до або під час виконання програми.

Також наведено основні форми IR: деревоподібні (AST), інструкційні (TAC) та стекові представлення, що використовуються на різних рівнях абстракції.

Визначено, що ефективність виконання програм значною мірою залежить від вибору проміжного представлення та застосованих оптимізацій на різних етапах обробки коду.

2. ПРОЄКТУВАННЯ ІНТЕРПРЕТАТОРА

2.1 Структура оптимізуючого інтерпретатора

У попередньому розділі було розглянуто загальну структуру інтерпретатора. У межах даної роботи пропонується архітектура оптимізуючого інтерпретатора, яка інтегрує механізми статичних оптимізацій на рівні проміжного представлення та забезпечує можливість їх композиційного застосування.

Ключовим архітектурним рішенням є вибір моделі виконання програми, оскільки саме вона визначає форму проміжного представлення та допустимі трансформації. У сучасних реалізаціях інтерпретаторів та віртуальних машин зазвичай розглядають три основні підходи:

- модель виконання дерева;
- байт-кодова модель;
- JIT-компіляція до нативного коду.

Спочатку оберемо модель виконання коду. Серед основних варто виділити: модель проходження дерева, байт-кодову модель, JIT-компіляція.

Модель виконання безпосередньо дерева синтаксису є найбільш інтуїтивною та простою для реалізації. Вона передбачає виконання програми шляхом рекурсивного обходу вузлів AST без введення проміжного рівня представлення.

Попри простоту, така модель має суттєві обмеження з точки зору оптимізації:

- функціональність фронтенду зміщується в бік виконання (перетин парсингу, аналізу та інтерпретації);
- кожен вузол потребує динамічної диспетчеризації, що знижує продуктивність;
- відсутня зручна форма для побудови SSA та аналізу потоків даних, що критично для статичних оптимізацій.

Таким чином, AST-модель не забезпечує достатнього рівня структурованості для оптимізуючого інтерпретатора.

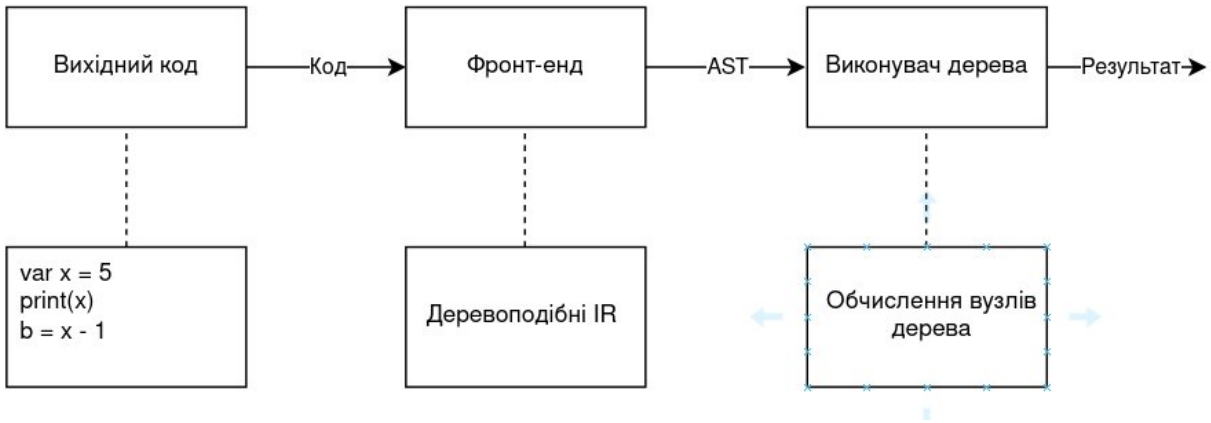

Рисунок 2.1 – Спрощена схема інтерпретатора-виконувача дерева

Байт-кодова модель передбачає компіляцію вихідного коду у проміжне представлення у вигляді інструкцій (байт-кодкод або TAC-подібний IR), які виконуються віртуальною машиною.

На відміну від AST-підходу, ця модель вводить чітке розділення етапів:

- фронтенд: синтаксичний та семантичний аналіз;
- мідл-енд: побудова та оптимізація IR;
- бекенд (VM): виконання інструкцій.

Це розділення є критично важливим для оптимізуючого інтерпретатора, оскільки дозволяє:

- застосовувати статичні оптимізації до IR незалежно від виконання;
- повторно виконувати один і той самий байт-код без повторної компіляції;
- вводити структуровані аналізи потоків даних.

Спрощена схема інтерпретатора байт-коду (рис. 2.2):

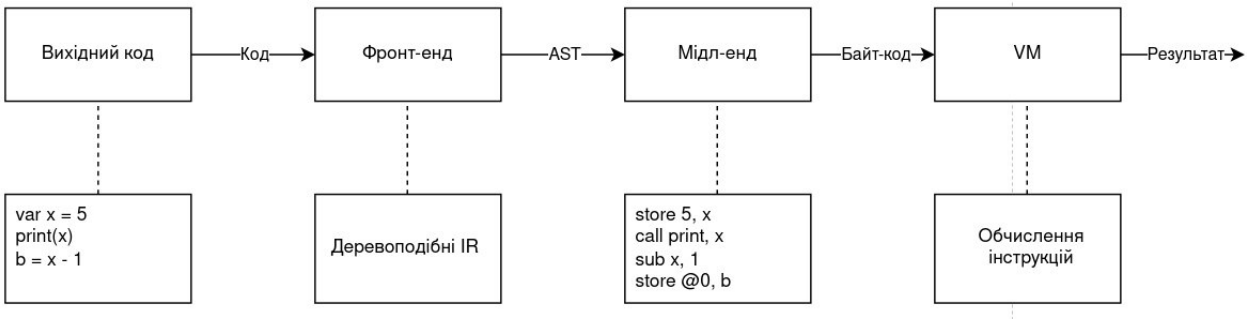

Рисунок 2.2 – Спрощена схема інтерпретатора байт-коду

Інтерпретатор з JIT-компіляцією є розвитком байт-кодової моделі, у якій гарячі ділянки коду додатково компілюються в нативний машинний код.

Попри високу продуктивність, JIT суттєво ускладнює систему, оскільки вимагає:

- профілювання виконання;

- аналізу стабільності типів;
- генерації та оптимізації нативного коду.

У межах даної роботи JIT не використовується, оскільки метою є дослідження та проєктування оптимізаційного пайплайну на рівні IR, а не побудова нативного виконання.

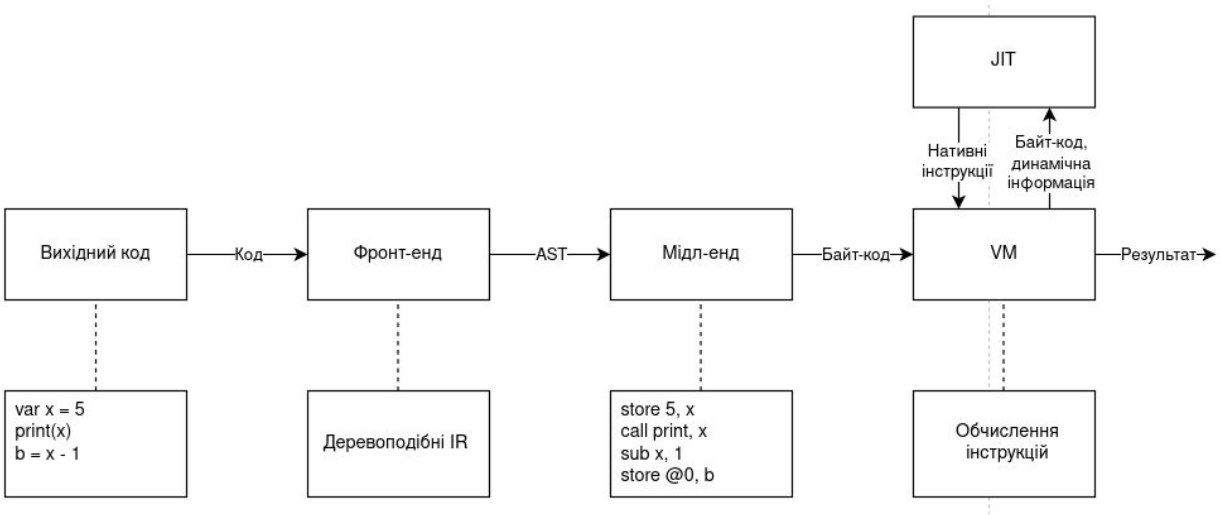

Рисунок 2.3 – Спрощена схема інтерпретатора байт-коду з JIT-компіляцією

З урахуванням вимог до реалізації оптимізацій, обрано байт-кодову модель виконання як базову. Вона забезпечує достатній рівень абстракції для:

- побудови SSA;
- реалізації def-use аналізу;
- застосування каскаду статичних оптимізацій;
- подальшого дослідження динамічних оптимізацій на рівні VM.

Ця модель й буде вважатися структурою оптимізуючого інтерпретатора в даній роботі.

Важливо підкреслити, що інструкції байт-коду не відображають апаратні інструкції процесора, а є семантично визначеним набором операцій, який виконується віртуальною машиною відповідно до специфікації.

2.2 Підготовка до статичних оптимізацій: мова програмування та її IR

В роботі буде використано прототип мови програмування, що має такі особливості:

- Конструкції умовного переходу: if-else;
- Цикли: while;
- Області видимості: окрема на кожен блок;
- Змінні: динамічно типізовані з частковими вказівками типу, затінення дозволене;
- Використання виразів у якості тверджень: обмежене;
- Вбудовані типи: числовий, булевий.

Сніппет 2.1 – Приклад коду

```
var x = 5
whl (x > 0)
{
    print(x)
    x = x - 1
}

fn print(var x) {
    x = 3
    ret x
}

~ example of a class definition
class Example {
    fn change(Number val) {
        var x = 3
        if (x > 0)
            a = val
        b = x
        3+5

        ret a
    }

    var a
}
```

Будь-яке оголошення змінної визначається словом “var”.

Як вже зазначалося, використання виразів у якості тверджень є обмеженим для спрощення оптимізацій, тому що в інакшому разі потрібно було б враховувати побічні ефекти, які можуть бути у таких виразів. Тому в прикладі вираз “3+5”, що використовується як твердження, буде просто проігноровано під час побудови IR.

Інші особливості мови перебувають поза увагою, тому що вони не стосуються згаданих статичних оптимізацій.

2.3 Підготовка до статичних оптимізацій: створення SSA-форми та Def-Use ланцюгів

Статичні оптимізації базуються на аналізу потоків даних. Наприклад, оптимізатор, виконавши згаданий аналіз, знає, які змінні використовувалися в коді, які значення не використовувалися, які ділянки коду або інструкції не є потрібними, тому що не мають ефекту або не є досяжними.

Саме на інструкціях базуватиметься аналіз потоків даних в інтерпретаторі. Відповідно, використовуватимемо для проміжного представлення цих інструкцій форму трьох-адресного коду (TAC).

Проте сам по собі TAC не є достатнім для статичних оптимізацій, адже можливості аналізу потоків даних на ньому є мінімальними. Це проміжне представлення потрібно перетворити в SSA-форму.

Форма Static Single Assignment (SSA) є різновидом проміжного представлення, у якому кожне визначення змінної створює нову версію цієї змінної. У результаті кожне значення має єдину точку визначення, а зв’язки між визначеннями та використаннями стають явними. Для об’єднання значень, що надходять із різних шляхів виконання програми, використовуються спеціальні φ-функції [18].

φ-функції не є справжніми інструкціями процесора або віртуальної машини. Вони використовуються лише на етапі аналізу та оптимізації програмного коду. Їх головне призначення полягає в об'єднанні кількох SSA-версій однієї змінної після розгалужень або на вході до циклів. У реалізованому оптимізаторі φ-функції також використовуються під час аналізу циклів та виконання глобального поширення констант.

Програма перебуває у SSA-формі тоді, коли кожна змінна отримує значення лише один раз, а кожне використання значення може бути однозначно пов’язане з його визначенням. Завдяки цьому значно спрощуються аналіз потоків даних, побудова def-use ланцюгів та реалізація багатьох оптимізацій.

Головною перевагою SSA є те, що вона робить потоки даних явними. Після перетворення в SSA більшість аналізів зводяться до роботи з прямими зв'язками між визначеннями та використаннями значень. Саме тому багато сучасних компіляторів використовують SSA як основне проміжне представлення для виконання оптимізаційних проходів.

Граф потоку керування (CFG) є поданням програми у вигляді графа, де вершини відповідають базовим блокам, а ребра відображають можливі переходи керування між ними під час виконання програми. Таким чином CFG описує всі потенційні шляхи виконання програми та використовується як основа для аналізу потоків даних і оптимізації [5].

Сніпет 2.2 – Приклад коду для CFG

```
var x = 5

whl (x > 0)
{
    if (x > 2)
        print(x)

    x = x - 1
}

ret x
```

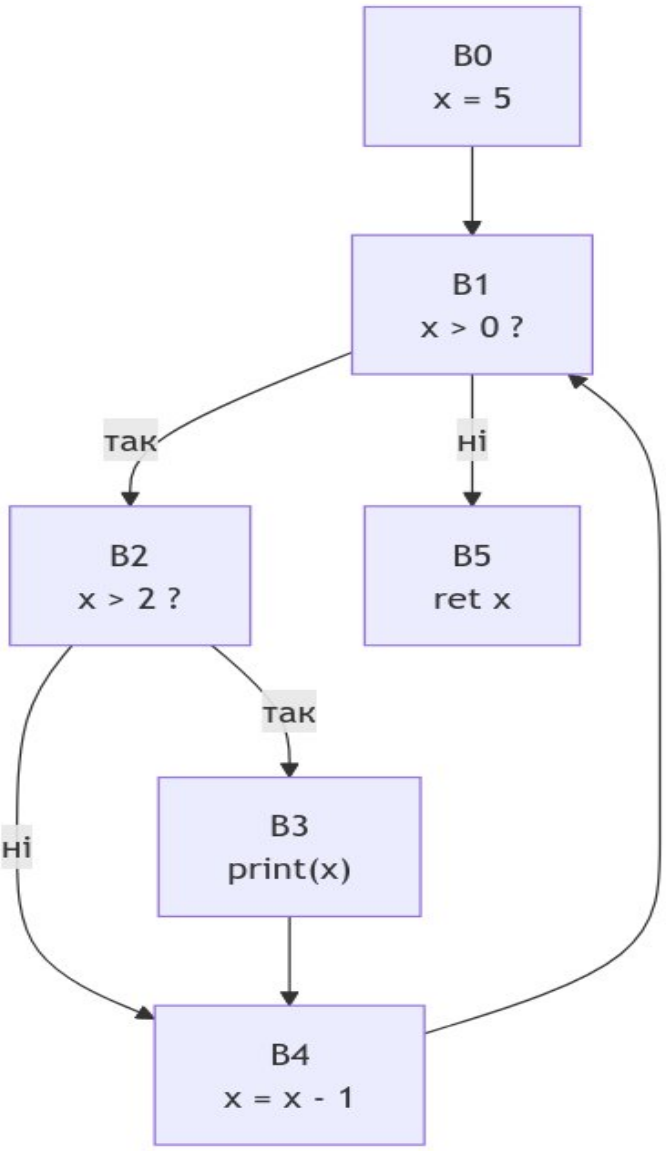

Рисунок 2.4 – CFG до прикладу коду

Граф потоку керування є фундаментальною структурою для більшості оптимізаційних алгоритмів. Саме він дозволяє визначити, які ділянки програми можуть бути виконані після інших ділянок, де відбуваються розгалуження та об'єднання шляхів виконання, а також які блоки можуть бути недосяжними. Без CFG побудова SSA-форми є практично неможливою, оскільки вставка φ-функцій безпосередньо залежить від структури переходів між блоками.

Інтеграція phi-функцій із графом потоку керування визначається структурою розгалужень у CFG. Кожна phi-функція відповідає точці злиття потоків керування і залежить від набору попередніх блоків, що формують вхід до поточного вузла [19].

CFG складається з блоків інструкцій та ребер, які представляють собою переходи між блоками.

Блоком називають максимальну послідовність інструкцій, яка має єдину точку входу та єдину точку виходу. Керування може потрапити в блок лише через його першу інструкцію, а залишити блок – лише після виконання останньої інструкції.

Використання блоків дозволяє розглядати програму не як окремі інструкції, а як більші структурні одиниці. Це суттєво спрощує аналіз, оскільки всередині блоку порядок виконання завжди лінійний і не містить внутрішніх переходів. У графі потоку керування саме базові блоки зазвичай використовуються як вершини графа.

Після побудови SSA наступним кроком йде побудова Def-Use ланцюгів.

Def-Use ланцюги дозволяють для кожної інструкції, що породжує значення, знати наступне:

- Інформація про інструкцію: операнди, породжуване значення;
- Інформація про використання інструкції: перелік інших інструкцій з зазначенням номеру операнда, під яким використовується породжуване значення.

2.3 Статичні оптимізації: поширення/згортання констант

Поширення копій і констант належить до базових статичних оптимізацій і використовується для спрощення проміжного представлення після перетворень, які змінюють граф потоків даних або набір інструкцій. Ідея цієї оптимізації полягає в тому, щоб замінити використання проміжного значення безпосереднім використанням його джерела, якщо таке джерело відоме та не змінюється на даному шляху виконання. У контексті SSA-форма подібні перетворення є особливо зручними, оскільки кожне визначення значення має однозначний зв’язок із місцем його породження.

Поширення копій полягає в тому, що якщо деяка інструкція лише передає або копіює вже наявне значення, то її результат можна підмінити самим джерелом. Після цього сама інструкція може стати непотрібною і бути позначеною як мертва. У реалізованому оптимізаторі таке поширення виконується через ланцюги Def-Use: коли інструкція визначає нове значення, усі її користувачі оновлюються так, щоб посилатися безпосередньо на оригінальний операнд. Це зменшує кількість проміжних інструкцій і полегшує подальші проходи оптимізації.

Поширення констант є окремим випадком поширення копій, у якому замість проміжного SSA-значення в користувачів підставляється безпосередньо константа. Такий підхід дозволяє зняти частину обчислень ще до згортання констант, оскільки після переписування ланцюгів використання часто виявляються нові операції, що повністю складаються з відомих констант.

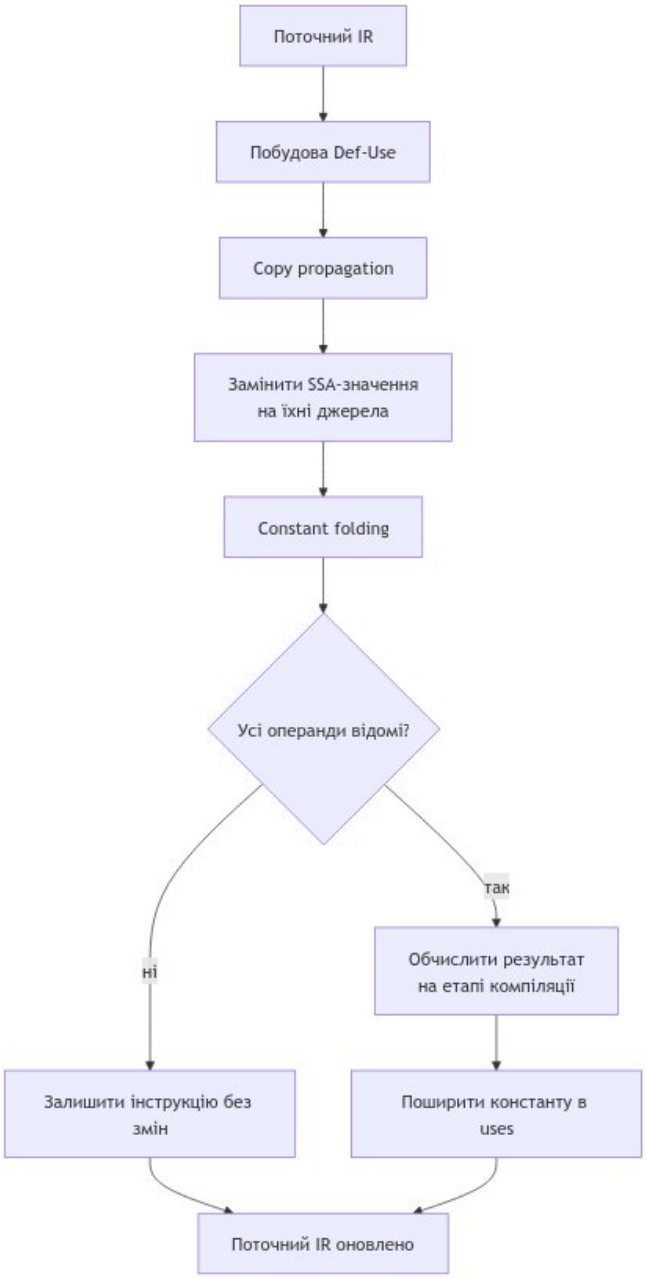

Рисунок 2.5 – Схема поширення/згортання констант

Згортання констант виконується після або разом із поширенням і полягає в тому, що кожна операція, у якої всі операнди є відомими константами, обчислюється на етапі компіляції, а її результат замінює вихідну інструкцію. Таким чином, замість виконання арифметики або логіки під час роботи програми обчислення переноситься на етап оптимізації.

Для спрощення реалізації згортання констант у системі передбачено окремі функції згортання залежно від типів операндів. Це дозволяє уникнути великої кількості умовних конструкцій усередині одного великого обробника і зробити систему розширюваною. Для унарних операторів використовується та сама схема, але другий операнд у відповідному виклику не бере участі в обчисленні. У такій архітектурі достатньо окремо описати згортання для числових і булевих значень, а конкретна операція вибирається за кодом інструкції.

Для двох чисел будуть визначені такі операції згортання:

- Додавання, віднімання, множення, ділення, ділення за модулем;
- Бітовий зсув вліво/вправо, бітові І, АБО, виключне АБО;
- Більше, менше, більше або дорівнює, менше або дорівнює, рівність, нерівність, логічні І та АБО;
- Логічне заперечення та арифметичний мінус.

Для двох булевих значень будуть визначені такі операції:

- Більше, менше, більше або дорівнює, менше або дорівнює, рівність, нерівність, логічні І та АБО, логічне заперечення.

Окремі унарні операції, зокрема арифметичний мінус або логічне заперечення, також можуть бути згорнуті, якщо їхній операнд відомий на етапі оптимізації.

Зазвичай таблиця згортання реалізується у вигляді багатовимірного масиву або іншої структури швидкого доступу, де за типом операції та типами операндів можна миттєво отримати відповідну функцію згортання. Такий підхід є зручним для компілятора, оскільки розділяє механізм вибору правила та саме правило обчислення.

2.4 Статичні оптимізації: strength reduction

Зменшення складності виразів (strength reduction) полягає в заміні операцій, які потребують відносно більшої вартості виконання, на еквівалентні, але дешевші форми. У компіляторах така оптимізація застосовується насамперед тоді, коли заміна не змінює семантику програми, але спрощує її виконання на цільовій віртуальній машині або на фізичному процесорі.

У цій роботі strength reduction використовується у вузькому, але практично корисному варіанті. Для операцій множення і ділення розглядаються лише ті випадки, коли один з операндів є додатним степенем двійки. За таких умов множення може бути замінене зсувом вліво, а ділення – зсувом вправо на відповідну кількість бітів. Це зменшує обсяг обчислень, оскільки бітовий зсув, як правило, простіший за арифметичну операцію множення або ділення.

Окремо реалізуються алгебраїчні спрощення з нейтральними та поглинаючими елементами. Такі правила не завжди змінюють “силу” операції у вузькому математичному сенсі, однак вони суттєво скорочують кількість інструкцій і тому природно поєднуються з strength reduction у межах одного проходу.

Наприклад, вираз x + 0 може бути замінений на x, оскільки додавання нуля не змінює значення; вираз x * 1 також зводиться до x, оскільки множення на одиницю не має ефекту; а вираз x * 0 завжди замінюється на 0, оскільки нуль є поглинаючим елементом для множення.

Крім того, подібні правила можуть застосовуватися до булевих операцій. У таких випадках оптимізація не лише зменшує кількість інструкцій, а й допомагає простіше аналізувати подальші умови переходів, оскільки результат логічного виразу стає очевидним уже на етапі компіляції.

Загалом цей прохід доцільно розглядати як комбінований, який виконує одночасно алгебраїчні спрощення та зменшення складності операцій. У практичній реалізації він є найбільш ефективним у зв'язці з поширенням констант: після заміни операндів на відомі значення з'являються нові можливості для спрощення, а після спрощення – нові кандидати на видалення як мертвий код.

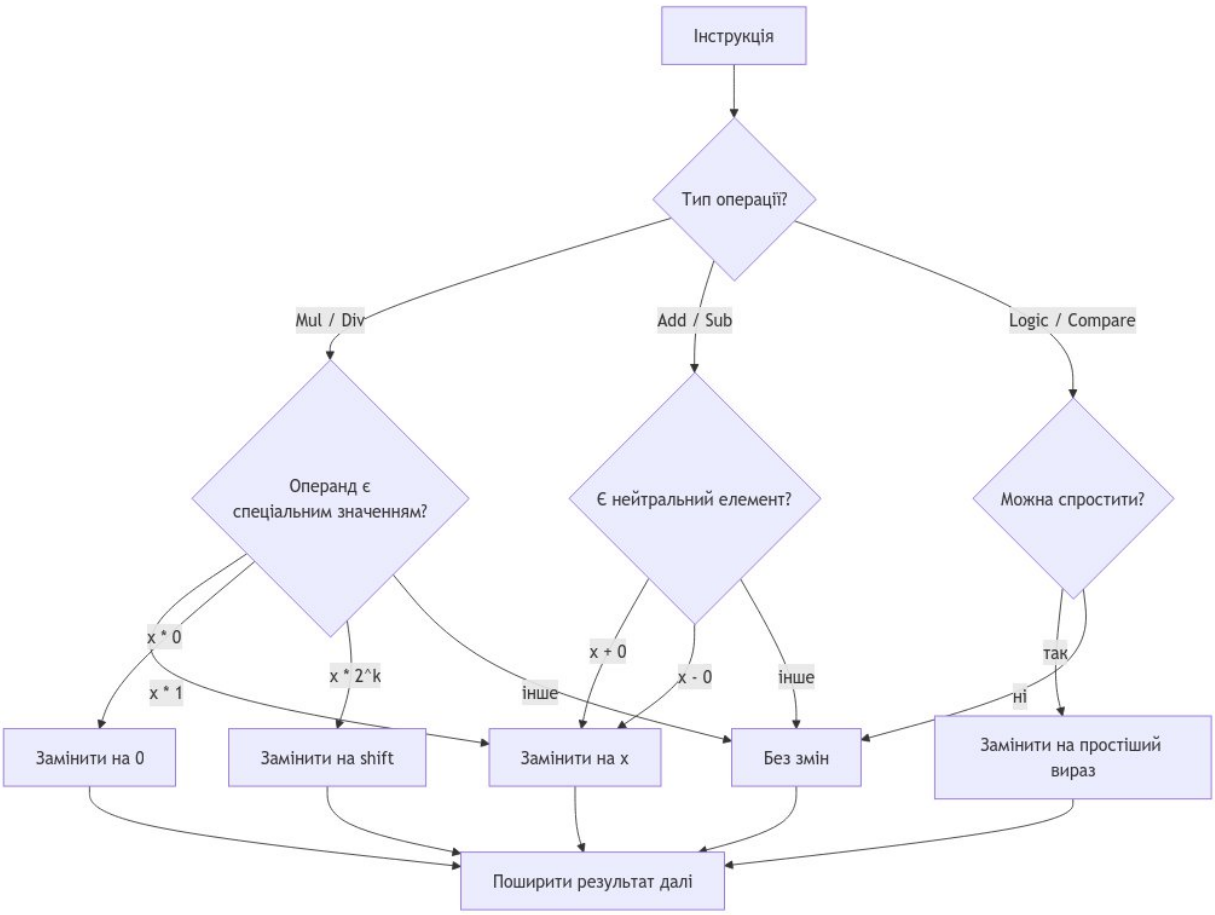

Рисунок 2.6 – Схема оптимізації пониження складності та алгебраїчного спрощення

2.5 Статичні оптимізації: peephole

Peephole-оптимізації застосовуються як локальні перетворення над короткими послідовностями інструкцій. На відміну від глобальних оптимізацій, вони не потребують повного аналізу всього графа програми і працюють на невеликих фрагментах коду, зазвичай у межах одного базового блока або на стику двох суміжних блоків. Саме тому їх зручно використовувати після базових SSA-перетворень, коли структура проміжного коду вже достатньо стабільна.

У цій роботі peephole використовується насамперед для спрощення потоку керування. До таких локальних перетворень належать спрощення умовних переходів, коли умова переходу вже відома на етапі компіляції, а також усунення ланцюжків переходів, у яких один блок безпосередньо передає керування іншому порожньому або технічному блоку. У результаті проміжне представлення стає компактнішим, а наступні етапи оптимізації отримують простіший граф потоку керування.

Окреме значення peephole-оптимізацій полягає в тому, що вони часто створюють додаткові можливості для подальшого поширення констант, спрощення φ-функцій та видалення мертвого коду. Саме тому даний pass доцільно запускати не одноразово, а як частину циклу локальних перетворень до досягнення стабільного стану проміжного представлення.

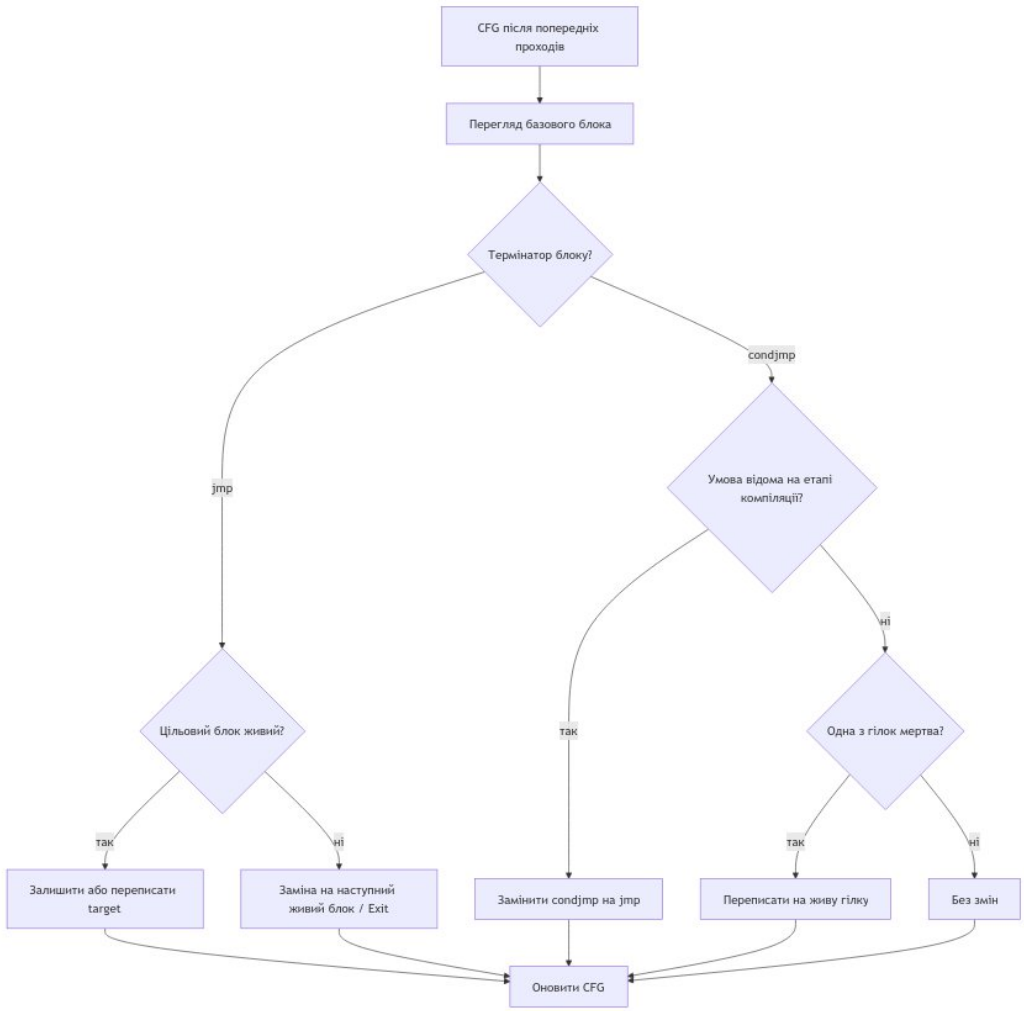

Рисунок 2.7 – Схема оптимізації peephole спрощення умовних переходів

2.6 Статичні оптимізації: DCE

Видалення мертвого коду (Dead Code Elimination, DCE) є одним із ключових етапів оптимізації проміжного представлення. Його мета полягає у видаленні інструкцій, результати яких ніколи не використовуються, а також у видаленні ділянок коду, які стають недосяжними після інших оптимізацій. У

SSA-формі DCE є особливо ефективним, оскільки зв’язки між визначеннями і використаннями значень явно представлені у Def-Use ланцюгах.

У реалізованій системі DCE виконується на двох рівнях.

Перший рівень – це рівень інструкцій. На цьому етапі проходу видаляються всі інструкції, для яких немає живих користувачів, а також інструкції, що не впливають на результат виконання програми. До таких інструкцій належать, зокрема, проміжні обчислення, які були витіснені поширенням констант або копій, а також вузли, що стали зайвими після згортання виразів.

Другий рівень – це рівень базових блоків. Після видалення окремих інструкцій деякі блоки можуть стати порожніми або втратити доступність з точки входу. Такі блоки вже не впливають на виконання програми і можуть бути відкинуті з графа потоку керування. Однак видалення блоків повинно виконуватися обережно, оскільки воно може впливати на φ-функції та інші елементи SSA-структури. Саме тому DCE у системі тісно пов’язаний із наступним етапом repair, який відновлює узгодженість між CFG та SSA-представленням.

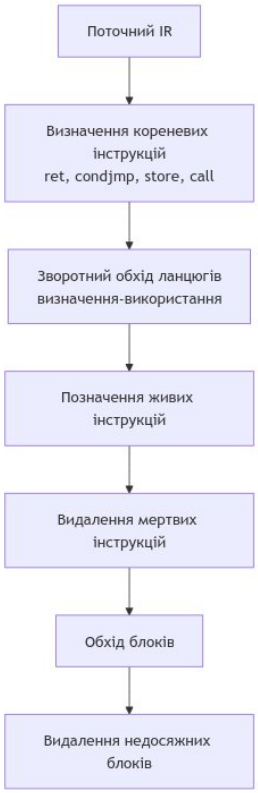

Рисунок 2.8 – Схема оптимізації DCE

Практично DCE доцільно виконувати як ітераційний процес: видалення мертвих інструкцій може відкрити нові можливості для видалення інших інструкцій або блоків. Через це повне очищення проміжного представлення зазвичай потребує кількох проходів до досягнення фіксованої точки. Саме такий підхід забезпечує стабільність результату та зменшує кількість зайвих інструкцій перед завершальними етапами оптимізації.

2.7 Статичні оптимізації: Стабілізація SSA та загальна структура оптимізатора

Після побудови проміжного представлення у формі SSA та виконання базових локальних перетворень оптимізатор переходить до найважливішого етапу – стабілізації проміжного коду. Під стабілізацією у даному контексті розуміється не лише зменшення кількості інструкцій, а й приведення SSA-структури до узгодженого стану після кожного проходу, що змінює значення, переходи або доступність блоків. Для оптимізуючого інтерпретатора це особливо важливо, оскільки саме проміжне представлення є “робочою мовою” системи: воно має бути достатньо компактним для ефективного виконання, але водночас достатньо точним для подальших перетворень.

На відміну від простого інтерпретатора, який виконує AST безпосередньо, або від повноцінного JIT-компілятора, що переводить код у нативні інструкції, оптимізуючий інтерпретатор працює у проміжній зоні між аналізом і виконанням. Це означає, що оптимізації тут не є допоміжною надбудовою, а становлять центральну частину архітектури. Кожен прохід може змінити не лише окремі операції, а й взаємозв’язки між значеннями, тому після будь-якого суттєвого перетворення необхідно повторно узгодити Def-Use ланцюги, φ-функції та граф потоку керування. Саме тому в структурі оптимізатора окремо виділяються етапи repair та DCE, а також багаторазовий цикл локальних проходів до досягнення стабільного стану.

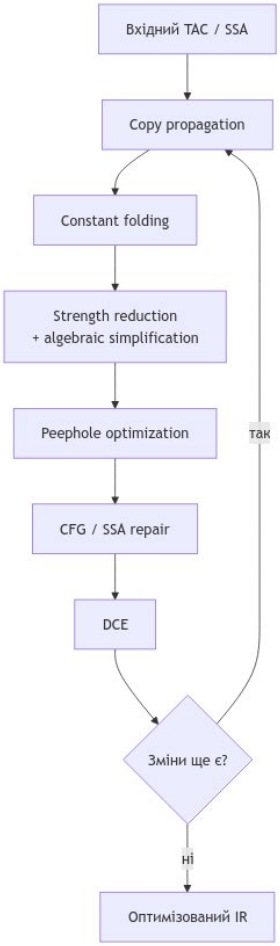

Рисунок 2.9 – Схема перетворення та оптимізації IR

Першою в циклі виконується поширення копій. Це базовий, але дуже важливий етап, оскільки він зменшує кількість непрямих посилань на значення та робить інші оптимізації точнішими. У SSA-формі копії й константи поширюються особливо природно: кожне значення має єдине визначення, тому заміна посилань у Def-Use ланцюгу не потребує складного аналізу всього коду. Саме поширення копій часто відкриває нові можливості для згортання констант, оскільки після підстановки джерела значення у багато інструкцій замість змінних з’являються безпосередні константні операнди.

Далі виконується згортання констант. У контексті оптимізуючого інтерпретатора цей етап має особливе значення, оскільки дозволяє перенести частину обчислень із фази виконання на фазу оптимізації. Якщо обидва операнди операції відомі під час компіляції, результат обчислюється заздалегідь, а вихідна інструкція замінюється на значення-результат. Це зменшує як кількість виконуваних інструкцій, так і кількість переходів через інтерпретаторний цикл VM. Для системи, де кожна інструкція породжує певні накладні витрати на диспетчеризацію, така заміна дає відчутний ефект навіть тоді, коли сам вираз здається простим.

Третім етапом у загальному циклі є strength reduction, тобто зменшення складності виразів. У межах даної роботи ця оптимізація реалізована у практично корисному, але обмеженому вигляді: вона застосовується лише до тих випадків, де заміна дійсно зменшує вартість обчислення або усуває зайву інструкцію. Частина таких правил має алгебраїчний характер і використовує нейтральні чи поглинаючі елементи операцій; інша частина стосується операцій множення або ділення на степені двійки. У середовищі байт-кодового інтерпретатора важливо не просто переписати вираз у формально еквівалентний, а дійсно зменшити кількість виконуваних кроків, і тому в strength reduction застосовуються лише ті перетворення, які дають реальний виграш на рівні інтерпретації.

Після локальних арифметичних перетворень виконується peephole-оптимізація, спрямована на спрощення потоку керування. У цій роботі вона використовується не як універсальний аналізатор, а як набір локальних перетворень над термінаторами блоків. Саме тут спрощуються умовні переходи з константною умовою, а також усуваються або скорочуються ланцюги переходів через порожні блоки.

Для оптимізуючого інтерпретатора це особливо важливо, оскільки зайві блоки і переходи безпосередньо збільшують накладні витрати на виконання програми. Крім того, локальна зміна CFG майже завжди впливає на SSA-структуру, а отже після peephole потрібно окремо відновити узгодженість між графом переходів і φ-функціями.

Саме з цієї причини у схемі оптимізатора окремо виділено етап SSA/CFG repair. Його призначення – не виконувати нові оптимізації, а стабілізувати структуру після попередніх перетворень. Після спрощення умовних переходів, видалення блоків або перепрошивки ребер термінатори та φ-функції повинні знову відповідати актуальному графу потоку керування. Якщо цього не зробити, SSA втрачає свої інваріанти: φ-функції можуть посилатися на

неіснуючих попередників, а блоки – на недосяжні цілі переходів. У рамках даного проєкту repair виступає як технічний, але критично необхідний механізм, який дозволяє поєднати агресивні локальні оптимізації з коректною SSA-формою.

Після repair виконується видалення мертвого коду. Цей етап завершального очищення важливий тому, що попередні проходи майже завжди створюють додаткові можливості для усунення інструкцій і блоків.

Наприклад, поширення констант може зробити частину обчислень непотрібними, strength reduction – замінити складну операцію на простішу і тим самим відкрити нові шляхи для спрощення, а peephole – зробити цілими недосяжні гілки CFG. DCE у такій архітектурі є не одноразовим “прибиранням”, а логічним завершенням кожного циклу оптимізації, після якого оптимізатор перевіряє, чи з’явилися нові можливості для подальших спрощень.

З цієї причини загальна структура оптимізатора організована як ітеративний цикл. Спочатку виконуються операції, які змінюють значення в межах SSA-графа: поширення копій, згортання констант і strength reduction. Після цього виконується peephole, який може змінити потік керування. Далі запускається repair, який відновлює узгодженість проміжного представлення, і вже після цього – DCE, що видаляє мертві інструкції та недосяжні блоки. Якщо результат одного циклу відкриває нові можливості для спрощення, оптимізатор повертається до початку ланцюга. Якщо нових змін немає, процес завершується, а на виході залишається стабілізоване проміжне представлення, придатне для подальшого виконання віртуальною машиною.

Такий підхід є особливо доречним саме для оптимізуючого інтерпретатора. У ньому важливо не лише отримати коректний результат, але й забезпечити, щоб проміжний код був достатньо простим для ефективної інтерпретації. В даному випадку проміжне представлення є кінцевою формою перед виконанням. Отже, чим краще воно спрощене, тим менші витрати нестиме сам механізм інтерпретації. Саме тому в даній роботі оптимізаційний цикл побудовано навколо стабілізації SSA, а не навколо генерації машинного коду: система повинна не просто аналізувати програму, а безпосередньо покращувати її форму для подальшого виконання у віртуальній машині.

2.8 Динамічні оптимізації

У рамках даної роботи динамічні оптимізації не розглядаються як окрема реалізаційна частина власного інтерпретатора, а виступають як експериментальна модель, що демонструє можливий напрям розвитку оптимізуючого інтерпретатора. Це важливо, тому що архітектура, запропонована в роботі, не обмежується лише статичними перетвореннями: вона передбачає, що інтерпретатор у перспективі може поєднувати попередню оптимізацію проміжного коду з подальшою адаптацією під реальний профіль виконання.

З цієї причини CPython у роботі використовується як готове середовище для дослідження того, як динамічна адаптація впливає на байт-код. Такий підхід дозволяє виділити три основні механізми: інлайнове кешування, спеціалізацію байт-коду та суперінструкції. Усі вони мають спільну мету – зменшити вартість виконання гарячих ділянок програми без зміни її семантики.

Такий підхід є обґрунтованим, оскільки метою роботи є проєктування архітектури оптимізуючого інтерпретатора та аналіз механізмів, які можуть у ньому застосовуватися. CPython у цьому контексті виступає як практичний приклад середовища виконання, що реалізує адаптивні оптимізації на рівні байт-коду.

CPython є еталонною реалізацією мови Python, написаною мовою C. Це найбільш поширена реалізація Python, яка використовується як базовий стандарт для порівняння з іншими реалізаціями мови. CPython виконує компіляцію Python-коду у проміжний байт-код, який надалі інтерпретується його віртуальною машиною. Ця реалізація забезпечує стандартне середовище виконання Python, включаючи стандартну бібліотеку та вбудовані функції [20].

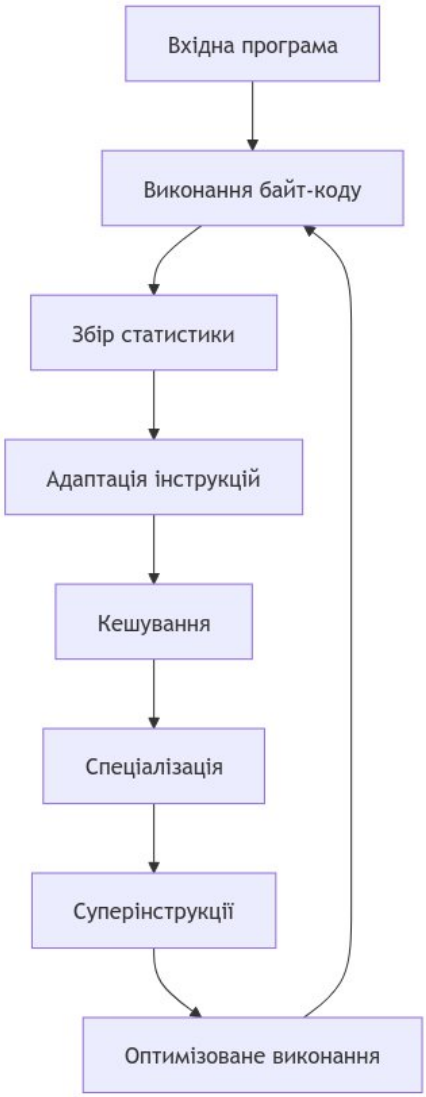

Рисунок 2.10 – Схема динамічних оптимізацій та адаптації в CPython

Використання CPython дозволяє розглядати динамічні оптимізації у природному для них середовищі: байт-код виконується віртуальною машиною, яка збирає статистику виконання, аналізує повторювані шаблони та замінює універсальні інструкції на спеціалізовані. Така модель добре узгоджується з концепцією оптимізуючого інтерпретатора, оскільки в обох випадках проміжне представлення не є статичним і незмінним: воно може переписуватися під час виконання або після накопичення інформації про поведінку програми.

З практичної точки зору він зручний ще й тим, що його динамічні оптимізації можна спостерігати без модифікації самого інтерпретатора. Для цього використовується модуль dis, який дозволяє виводити байт-код до накопичення статистики та після warm-up, а також показувати значення кешів для окремих інструкцій.

У контексті віртуальних машин warm-up описується як початковий період виконання програми, протягом якого система ще не досягає стабільного рівня продуктивності. У цей час механізми виконання, зокрема інтерпретатор або JIT-компілятор, поступово накопичують інформацію про поведінку програми, що дозволяє виконувати подальші оптимізації. Після завершення цього періоду система переходить у стан стабільної продуктивності (steady-state), у якому оптимізовані версії коду вже застосовуються до ділянок програми, що виконуються найчастіше [21].

У межах даної роботи CPython розглядається не як альтернатива запропонованій архітектурі, а як її експериментальний аналог для динамічної частини. Це дозволяє окремо дослідити той клас оптимізацій, який не є статичним і не реалізується на етапі побудови SSA або CFG, а виникає як результат виконання програми в реальному часі. Саме тому динамічні оптимізації у цій роботі виділено в окрему підсистему аналізу, незалежну від статичного оптимізатора.

2.9 Інлайнове кешування

Інлайнове кешування є однією з ключових динамічних оптимізацій CPython і основою для більшості спеціалізацій байт-коду. Його сутність полягає в тому, що поряд із самою інструкцією зберігається додаткова службова інформація, яка накопичується під час виконання і допомагає інтерпретатору швидко приймати рішення без повторного виконання дорогих перевірок. Таким чином інструкція поступово отримує контекст виконання, потрібний для адаптації до конкретних об’єктів і структур даних.

На відміну від статичних оптимізацій, де трансформація виконується один раз на етапі компіляції в проміжне представлення, кешування накопичує інформацію поступово. Це дозволяє CPython відрізняти стабільні гарячі ділянки коду від випадкових, а отже застосовувати спеціалізацію лише там, де вона справді дає виграш.

Рисунок 2.11 – Схема інлайнового кешування

Кеші в CPython зберігають не тільки кількість виконань, але й метадані, пов’язані з об’єктами, типами, словниками атрибутів та функціями. Саме тому після warm-up можна спостерігати, що значення cache counter зростають, а поруч із байт-кодом з’являються додаткові поля, які дозволяють перевірити стабільність середовища виконання. Для дослідження це зручно, бо можна порівняти початковий стан інструкції з її адаптованим станом і побачити, як накопичення статистики приводить до появи спеціалізованих варіантів.

Для архітектури оптимізуючого інтерпретатора інлайнове кешування є показовим, оскільки демонструє проміжний рівень між чистою інтерпретацією та агресивною компіляцією. Воно не потребує генерації машинного коду, але вже дозволяє значно зменшити вартість повторюваних операцій. Саме тому цей механізм доцільно розглядати як один із базових елементів динамічно адаптивної виконувальної моделі.

2.10 Спеціалізація байт-коду

Спеціалізація байт-коду є динамічною оптимізацією, у межах якої універсальні інструкції замінюються на більш вузькі версії, орієнтовані на конкретні типи об’єктів, структуру даних або спосіб доступу. На початковому етапі виконання інтерпретатор працює з загальними інструкціями, які повинні враховувати різні можливі випадки. Після накопичення статистики виконання CPython може визначити, що певна операція стабільно виконується над об’єктами одного й того самого типу, і замінити універсальну інструкцію на спеціалізовану.

Рисунок 2.12 – Схема спеціалізації байт-коду

Спеціалізація байт-коду розглядається як механізм зменшення вартості інтерпретації. Універсальна інструкція потребує більше перевірок під час виконання: вона повинна враховувати типи операндів, можливі виключення, особливості доступу до атрибутів чи елементів контейнерів.

Спеціалізована ж інструкція може пропустити значну частину цих перевірок, якщо попередній профіль виконання показав стабільність об’єктів і структури даних. У результаті скорочується як кількість умовних перевірок, так і кількість непрямих переходів у віртуальній машині.

Для інтерпретатора це особливо важливо, оскільки основна частина вартості виконання байт-коду пов’язана не з самим обчисленням, а з диспетчеризацією інструкцій. Тому заміна універсальних операцій на спеціалізовані аналоги створює ефект, аналогічний локальній оптимізації: сама програма не змінює семантику, але шлях її виконання стає коротшим і стабільнішим. Саме ця властивість робить спеціалізацію байт-коду природним кандидатом для дослідження в межах оптимізуючого інтерпретатора.

Доцільно трактувати спеціалізацію як адаптацію байт-коду до гарячих ділянок виконання. Якщо певна інструкція багаторазово виконується з однаковими типами операндів, вона поступово переходить від універсального варіанта до спеціалізованого. Це відповідає загальній ідеї оптимізуючого інтерпретатора: система не обов’язково повинна бути максимально швидкою з першого запуску, але повинна вміти адаптуватися до реального режиму використання.

2.11 Суперінструкції

Суперінструкції є наступним логічним рівнем динамічної оптимізації після спеціалізації та кешування. Якщо спеціалізація зменшує вартість окремої інструкції, а кешування зменшує кількість повторних перевірок, то суперінструкції скорочують кількість самих переходів між інструкціями. Їх суть полягає в об’єднанні кількох часто використовуваних сусідніх інструкцій у єдиний ефективний шлях виконання.

Суперінструкції важливі тому, що інтерпретатор витрачає значну частину часу саме на цикл диспетчеризації: вибір інструкції, перехід до її обробника, підготовку операндів і повернення до головного циклу. Коли дві або більше інструкції стабільно виконуються одна за одною, їх можна трактувати як єдиний шаблон виконання. У результаті скорочується кількість “коротких” переходів усередині віртуальної машини, а отже зменшуються накладні витрати інтерпретації.

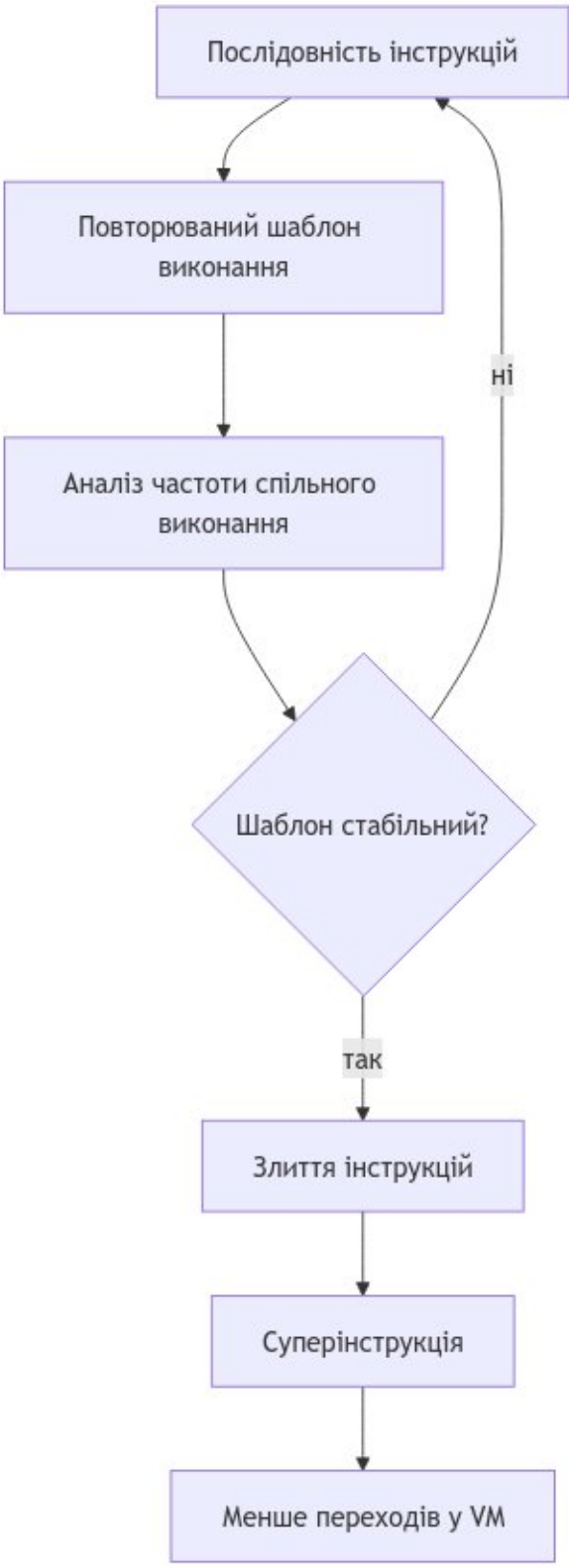

Рисунок 2.13 – Схема суперінструкцій

У межах дослідження це можна спостерігати на прикладах доступу до атрибутів, індексації списків і викликів методів. В усіх цих випадках CPython після warm-up переходить від послідовності універсальних інструкцій до більш щільних і спеціалізованих варіантів, які фактично реалізують злитий сценарій виконання. Це особливо добре узгоджується з ідеєю оптимізуючого

інтерпретатора, оскільки таке злиття не змінює семантику програми, але робить механізм виконання компактнішим і швидшим.

Для проєктованої архітектури важливо, що суперінструкції є природним продовженням попередніх етапів оптимізації. Якщо статичний оптимізатор прагне зменшити кількість інструкцій у SSA/CFG, то динамічний механізм суперінструкцій робить аналогічну операцію вже під час виконання, але на рівні байт-коду. Таким чином, уся система – як статична, так і динамічна її частина – працює в одному напрямку: зменшити обсяг проміжних переходів і наблизити форму виконання програми до найбільш ефективного вигляду.

2.12 Висновки

У цьому розділі було спроєктовано архітектуру оптимізуючого інтерпретатора та визначено набір статичних і динамічних оптимізацій, що використовуватимуться в подальшому дослідженні.

Для статичної оптимізації було обрано SSA-представлення та розроблено послідовність проходів, яка включає поширення копій і констант, згортання констант, strength reduction, peephole-оптимізації, відновлення узгодженості SSA та видалення мертвого коду. Запропонована структура дозволяє виконувати оптимізації ітеративно до досягнення стабільного стану проміжного представлення.

Для дослідження динамічних оптимізацій було обрано інтерпретатор CPython, який реалізує механізми інлайнового кешування, спеціалізації байт-коду та суперінструкцій. Розглянуті механізми відповідають сучасним підходам до побудови адаптивних інтерпретаторів та можуть бути експериментально досліджені без розробки власної динамічної оптимізаційної підсистеми.

Таким чином, у розділі сформовано теоретичну та архітектурну основу для подальшої реалізації статичного оптимізатора і дослідження динамічних оптимізацій на прикладі CPython.

3. ПЕРЕВІРКА ОПТИМІЗАЦІЙ

3.1 Перевірка статичних оптимізацій

Вихідний код статичного оптимізатора наведено у GitHub-репозиторії за посиланням [22].

Для перевірки статичних оптимізацій було підготовлено набір тестових програм мовою розроблюваного інтерпретатора. Кожен тест навмисно сконструйовано так, щоб він демонстрував дію окремого проходу або взаємодію кількох проходів. Усі приклади були прогнані через оптимізатор, після чого порівнювалися проміжне представлення та кількість інструкцій у базових блоках.

Перший тест перевіряє здатність оптимізатора обчислювати вирази з відомими операндами ще до етапу виконання. У вхідному коді присутні арифметичні, логічні та комбіновані вирази, які не залежать від динамічних даних програми. Це дозволяє перевірити, чи здатний оптимізатор згорнути не лише окремі прості операції, а й цілі ланцюги обчислень.

Сніппет 3.1 – Тестовий код

```
var a = 10 + 3
var b = 2 * (4 + 1)
var c = (8 / 2) + (6 - 1)
var d = (true && false) || true
ret a + b + c + d
```

Сніппет 3.2 – Вивід тесту згортання констант

```
BEFORE opt:

block 0:
@0 = add 10.000000, 3.000000
@1 = load @0
store @1, a0_0
@2 = add 4.000000, 1.000000
@3 = mul 2.000000, @2
@4 = load @3
store @4, b1_0
@5 = div 8.000000, 2.000000
@6 = sub 6.000000, 1.000000
@7 = add @5, @6
@8 = load @7
store @8, c2_0
@9 = logicand true, true
```

```
@10 = logicor @9, true
@11 = load @10
store @11, d3_0
@12 = load a0_0
@13 = load b1_0
@14 = add @12, @13
@15 = load c2_0
@16 = add @14, @15
@17 = load d3_0
@18 = add @16, @17
ret @18

AFTER opt:

block 0:
@18 = add 32.000000, true
ret @18
```

Після проходу згортання констант більшість проміжних інструкцій зникає, а значення виразів замінюються на константи. У результаті під час подальших проходів оптимізатор працює вже не з деревом арифметичних операцій, а з компактним представленням, у якому частина результатів відома наперед. Це добре демонструє основну перевагу згортання констант: перенесення обчислень із фази виконання на фазу оптимізації.

Другий тест побудовано як ланцюг простих присвоєнь і завантажень значення, що спочатку зберігається у змінній x, а далі послідовно передається через y, z та t. Така структура дозволяє перевірити, чи вміє оптимізатор не лише розпізнавати локальну константу, а й доводити її через декілька рівнів проміжних значень.

Сніппет 3.1 – Тестовий код

```
var x = 5
var y = x
var z = y
var t = z + 1
if (t > 5) {
    ret t
} else
    ret 0
```

Сніппет 3.2 – Вивід тесту поширення констант

```
BEFORE opt:

block 0:
@0 = load 5.000000
store @0, x0_0
@1 = load x0_0
```

```
@2 = load @1
store @2, y1_0
@3 = load y1_0
@4 = load @3
store @4, z2_0
@5 = load z2_0
@6 = add @5, 1.000000
@7 = load @6
store @7, t3_0
jmp blk1

block 1:
@8 = load t3_0
@9 = gt @8, 5.000000
condjmp @9, blk2, blk3

block 2:
@10 = load t3_0
ret @10
jmp blk4

block 3:
ret 0.000000
jmp blk4

block 4:

AFTER opt:

block 2:
ret 6.000000
```

Після поширення констант значення 5 переходить через усі ланцюжки копій і стає відомим у місці умовного переходу. Це дає змогу оптимізатору спростити умову t > 5 до константного результату та видалити недосяжну гілку. У підсумку в оптимізованому коді залишається лише шлях виконання, який відповідає істинному результату умови, а заключне значення функції стає відомим на етапі компіляції.

Третій тест призначено для перевірки DCE і містить ланцюг обчислень, проміжні результати якого ніколи не використовуються. Окрім цього, у тесті присутня операція динамічного читання та виклик функції, які мають побічний ефект і тому не можуть бути видалені без додаткового аналізу.

Сніппет 3.3 – Тестовий код

```
var x = 5
var y = x + 1
var z = y + 2
var w = z + 3
```

```
print(x)
ret x
```

Сніппет 3.4 – Вивід тесту DCE

```
BEFORE opt:

block 0:
@0 = load 5.000000
store @0, x0_0
@1 = load x0_0
@2 = add @1, 1.000000
@3 = load @2
store @3, y1_0
@4 = load y1_0
@5 = add @4, 2.000000
@6 = load @5
store @6, z2_0
@7 = load z2_0
@8 = add @7, 3.000000
@9 = load @8
store @9, w3_0
@10 = load x0_0
param @10
@11 = dynread dyn ltrl4
@12 = call @11
@13 = load x0_0
ret @13

AFTER opt:

block 0:
param 5.000000
@11 = dynread dyn ltrl4
@12 = call @11
ret 5.000000
```

Після проходу DCE усі обчислення, що не впливають на результат програми, зникають разом із проміжними store/load-ланцюгами. Водночас інструкції з побічним ефектом зберігаються, оскільки вони можуть бути важливими для семантики виконання. Це підтверджує, що реалізований DCE коректно відрізняє справді мертвий код від операцій, які необхідно залишити навіть за відсутності прямого використання їхнього результату.

Четвертий тест було спеціально побудовано як вкладену конструкцію з константною зовнішньою умовою та однаковими результатами в обох гілках внутрішнього розгалуження. Крім того, змінна x ініціалізується значенням -1,

що дозволяє спостерігати взаємодію локальних спрощень із константним поширенням.

Сніппет 3.5 – Тестовий код

```
var x = -1
if (true) {
    if (x > 0) {
        ret x
    } else
        ret x
} else {
    ret 0
}
```

Сніппет 3.6 – Вивід тесту Peephole

```
BEFORE opt:

block 0:
@0 = aneg 1.000000
@1 = load @0
store @1, x0_0
jmp blk1

block 1:
condjmp true, blk2, blk3

block 2:
jmp blk5

block 3:
ret 0.000000
jmp blk4

block 4:

block 5:
@2 = load x0_0
@3 = gt @2, 0.000000
condjmp @3, blk6, blk7

block 6:
@4 = load x0_0
ret @4
jmp blk8

block 7:
@5 = load x0_0
ret @5
```

```
jmp blk8

block 8:
jmp blk4

AFTER opt:

block 7:
ret -1.000000
```

Після peephole-оптимізації весь фрагмент скорочується до одного інструкційного шляху з поверненням -1. Це означає, що оптимізатор не лише прибрав зайву умовну гілку, а й спростив потік керування настільки, що весь вкладений розгалужений код виявився непотрібним. Такий результат добре демонструє, що peephole у цій реалізації працює не тільки як локальне переписування інструкцій, а й як механізм спрощення CFG з подальшим очищенням порожніх блоків.

П’ятий тест перевіряє одразу кілька типових випадків: множення на степінь двійки, ділення на степінь двійки, множення на 1 та множення на 0. Такий приклад є зручним, оскільки він поєднує справжнє strength reduction із алгебраїчними спрощеннями, які в практичному оптимізаторі природно реалізуються разом.

Сніппет 3.7 – Тестовий код

```
var a = n * 8
var b = a / 4
var c = b * 1
var d = c * 0
ret a + b + c + d
```

Сніппет 3.8 – Вивід тесту Strength reduction

```
BEFORE opt:

block 0:
@0 = dynread dyn ltrl0
@1 = mul @0, 8.000000
@2 = load @1
store @2, a0_0
@3 = load a0_0
@4 = div @3, 4.000000
@5 = load @4
store @5, b1_0
@6 = load b1_0
@7 = mul @6, 1.000000
@8 = load @7
store @8, c2_0
@9 = load c2_0
```

```
@10 = mul @9, 0.000000
@11 = load @10
store @11, d3_0
@12 = load a0_0
@13 = load b1_0
@14 = add @12, @13
@15 = load c2_0
@16 = add @14, @15
@17 = load d3_0
@18 = add @16, @17
ret @18

AFTER opt:

block 0:
@0 = dynread dyn ltrl0
@1 = shiftl @0, 3.000000
@4 = shiftr @1, 2.000000
@14 = add @1, @4
@16 = add @14, @4
@18 = add @16, 0.000000
ret @18
```

Після проходу оптимізації операція множення на 8 перетворюється на зсув вліво, а ділення на 4 – на зсув вправо. Одночасно множення на 1 і 0 усувається або зводиться до значення, яке не потребує додаткового обчислення. У результаті добре видно, що сила операцій зменшується: замість дорогих арифметичних інструкцій використовуються простіші бітові або тривіальні форми. Це особливо важливо для інтерпретатора, де кожна інструкція має власну вартість диспетчеризації.

Останній тест поєднує одразу кілька оптимізацій у межах одного прикладу.

Сніппет 3.9 – Тестовий код

```
var x = 5
var y = x
var a = 10 + 3
var b = a * 1
if (true) {
    var c = b * 8
    print(c)
} else {
    var d = b + 0
    print(d)
}
var e = y + 0
ret e
```

Сніппет 3.10 – Вивід фінального тесту

```
BEFORE opt:

block 0:
@0 = load 5.000000
store @0, x0_0
@1 = load x0_0
@2 = load @1
store @2, y1_0
@3 = add 10.000000, 3.000000
@4 = load @3
store @4, a2_0
@5 = load a2_0
@6 = mul @5, 1.000000
@7 = load @6
store @7, b3_0
jmp blk1

block 1:
condjmp true, blk2, blk3

block 2:
@8 = load b3_0
@9 = mul @8, 8.000000
@10 = load @9
store @10, c4_0
@11 = load c4_0
param @11
@12 = dynread dyn ltrl5
@13 = call @12
jmp blk4

block 3:
@14 = load b3_0
@15 = add @14, 0.000000
@16 = load @15
store @16, d5_0
@17 = load d5_0
param @17
@18 = dynread dyn ltrl5
@19 = call @18
jmp blk4

block 4:
@20 = load y1_0
@21 = add @20, 0.000000
```

```
@22 = load @21
store @22, e6_0
@23 = load e6_0
ret @23

AFTER opt:

block 2:
param 104.000000
@12 = dynread dyn ltrl5
@13 = call @12
jmp blk4

block 4:
ret 5.000000
```

Спочатку присутнє поширення констант через копіювання значень, далі виконується згортання констант для виразу 10 + 3, потім застосовується strength reduction для множення на 8, а також peephole-оптимізація для спрощення умовного переходу. Після цього DCE видаляє недосяжну гілку та всі проміжні інструкції, які втратили використання.

3.2 Перевірка динамічних оптимізацій

Як вже було зазначено в попередньому розділі, дослідження динамічних оптимізацій виконувалося на інтерпретаторі CPython.

Основною особливістю сучасного CPython є використання адаптивного байт-коду. На відміну від традиційної інтерпретації, де всі інструкції мають універсальний характер, CPython накопичує статистику виконання та замінює окремі інструкції їх спеціалізованими версіями. Для перевірки цього механізму необхідно проаналізувати байт-код до накопичення статистики виконання та після багаторазового запуску однакових ділянок програми.

Для дослідження оптимізацій розробимо тестову програму мовою Python, яка виконуватиме деяку роботу та дозволятиме дивитися інструкції байт-коду.

Сніппет 3.11 – Тестова програма мовою Python

```
import dis
import sys

class Point:
    def __init__(self, x):
        self.x = x

    def add(self, dx):
        return self.x + dx

points = [Point(i) for i in range(1000)]
```

```
values = list(range(1000))

def read_attr(seq):
    total = 0
    for p in seq:
        total += p.x
    return total

def list_index(seq):
    total = 0
    for i in range(len(seq)):
        total += seq[i]
    return total

def method_call(seq):
    total = 0
    for p in seq:
        total += p.add(1)
    return total

def show(fn, arg, name):
    print(f"\n=== {name}: baseline ===")
    dis.dis(fn, adaptive=False, show_caches=True)
    for _ in range(50000):
        fn(arg)
    print(f"\n=== {name}: after warm-up ===")
    dis.dis(fn, adaptive=True, show_caches=True)

if hasattr(sys, "_stats_clear"):
    sys._stats_clear()

show(read_attr, points, "read_attr")
show(list_index, values, "list_index")
show(method_call, points, "method_call")

if hasattr(sys, "_stats_dump"):
    sys._stats_dump()
```

Програма призначена для дослідження поведінки Python-виконання на рівні байткоду та аналізу впливу оптимізацій інтерпретатора на різні типи операцій доступу до даних. У ньому визначено просту структуру Point, а також три характерні сценарії обчислень: доступ до атрибутів об’єкта, індексація списку та виклик методу об’єкта.

Кожен сценарій реалізує окремий патерн доступу до даних, який по-різному взаємодіє з механізмами віртуальної машини Python. Функція read_attr моделює часті звернення до атрибутів об’єктів, list_index — доступ до

елементів послідовності за індексом, а method_call — виклик методів, що додатково включає динамічний диспетчинг.

Для аналізу використовується модуль dis, який дозволяє отримати байткодове представлення функцій до та після “розігріву” (warm-up). Додатково виконується багаторазове прогонювання функцій, щоб активувати динамічні механізми оптимізації інтерпретатора. Це дає змогу спостерігати зміну байткоду та використання кешів інструкцій у режимі адаптивного виконання.

Таким чином, програма використовується як експериментальна тестова база для порівняння різних типів доступу до даних та оцінки ефекту динамічних оптимізацій, зокрема спеціалізації байткоду, інлайнового кешування та механізмів суперінструкцій у CPython.

3.2 Перевірка динамічних оптимізацій: спеціалізація байт-коду

Як вже було зазначено в попередньому розділі, дослідження динамічних оптимізацій виконується на інтерпретаторі CPython.

Дослідження виконувалось на основі порівняння двох станів інтерпретатора: BEFORE (до накопичення статистики виконання) та AFTER (після накопичення статистики і активації адаптивної спеціалізації). У стані BEFORE байткод представлений у формі універсальних інструкцій, які не враховують конкретні типи об’єктів і виконуються через загальні механізми диспетчеризації. У стані AFTER спостерігається заміна частини інструкцій на спеціалізовані версії, які враховують типи та структуру об’єктів, а також зменшують кількість непрямих переходів.

Розглянемо стан BEFORE.

У початковому стані доступ до атрибутів виконується через універсальну інструкцію LOAD_ATTR, без урахування конкретної структури об’єкта. Ітерація реалізується через загальні механізми GET_ITER та FOR_ITER. Арифметична операція виконується через BINARY_OP без спеціалізації.

Сніппет 3.12 – Інструкції в стані BEFORE

```
=== read_attr: BEFORE ===
14 RESUME 0
CACHE 0 (counter: 0)

15 LOAD_SMALL_INT 0
STORE_FAST 1 (total)

16 LOAD_FAST 0 (seq)
GET_ITER 0
CACHE 0 (counter: 0)
L1: FOR_ITER 21 (to L2)
STORE_FAST 2 (p)
```

```
17 LOAD_FAST_BORROW_LOAD_FAST_BORROW 18 (total, p)
LOAD_ATTR 0 (x)
BINARY_OP 13 (+=)
STORE_FAST 1 (total)

18 L2: END_FOR
POP_ITER

19 LOAD_FAST_BORROW 1 (total)
RETURN_VALUE
```

Отримані результати демонструють, що спеціалізація в CPython виконується не на рівні функції в цілому, а на рівні окремих інструкцій байт-коду.

Сніппет 3.13 – Інструкції в стані AFTER

```
=== read_attr: AFTER ===
14 RESUME_CHECK 0
CACHE 0 (counter: 416)

15 LOAD_SMALL_INT 0
STORE_FAST 1 (total)

16 LOAD_FAST 0 (seq)
GET_ITER_VIRTUAL 0
FOR_ITER_LIST 21 (to L2)

17 LOAD_FAST_BORROW_LOAD_FAST_BORROW 18 (total, p)
LOAD_ATTR_INSTANCE_VALUE 0 (x)
BINARY_OP_ADD_INT 13 (+=)
STORE_FAST 1 (total)

18 L2: END_FOR
POP_ITER

19 LOAD_FAST_BORROW 1 (total)
RETURN_VALUE
```

Порівняння результатів демонструє декілька незалежних механізмів спеціалізації. По-перше, інструкція LOAD_ATTR замінюється на LOAD_ATTR_INSTANCE_VALUE. Універсальна версія повинна виконувати пошук атрибута через загальний механізм доступу до об'єктів Python, тоді як спеціалізована версія використовує інформацію про стабільну структуру екземплярів класу Point.

По-друге, цикл переходить від універсальної схеми GET_ITER + FOR_ITER до спеціалізованої комбінації GET_ITER_VIRTUAL + FOR_ITER_LIST, орієнтованої на ітерацію списків. Це дозволяє уникнути частини перевірок, характерних для довільних ітераторів.

По-третє, арифметична операція перетворюється з універсальної BINARY_OP на BINARY_OP_ADD_INT. Оскільки під час виконання інтерпретатор спостерігає виключно цілочисельні значення, необхідність повторної перевірки типів відпадає.

Також перевіримо спеціалізацію індексації списку.

У початковому стані індексація списку реалізується через загальні механізми range + len + CALL, а доступ до елементів виконується через універсальний BINARY_OP ([]). Ітерація не спеціалізована.

Сніппет 3.14 – Інструкції в стані BEFORE

```
=== list_index: BEFORE ===
20 LOAD_GLOBAL 1 (range + NULL)
LOAD_GLOBAL 3 (len + NULL)
LOAD_FAST_BORROW 0 (seq)
CALL 1

21 GET_ITER 0
L1: FOR_ITER 18 (to L2)
STORE_FAST 2 (i)

22 LOAD_FAST_BORROW_LOAD_FAST_BORROW 16 (total, seq)
LOAD_FAST_BORROW 2 (i)
BINARY_OP 26 ([])

23 BINARY_OP 13 (+=)
STORE_FAST 1 (total)
```

Після warm-up застосовуються спеціалізовані інструкції: LOAD_GLOBAL_BUILTIN, CALL_LEN, CALL_BUILTIN_CLASS, FOR_ITER_RANGE та BINARY_OP_SUBSCR_LIST_INT.

Сніппет 3.15 – Інструкції в стані AFTER

```
=== list_index: AFTER ===
20 LOAD_GLOBAL_BUILTIN 1 (range + NULL)
LOAD_GLOBAL_BUILTIN 3 (len + NULL)
LOAD_FAST_BORROW 0 (seq)
CALL_LEN 1

21 CALL_BUILTIN_CLASS 1
GET_ITER 0
L1: FOR_ITER_RANGE 18 (to L2)

22 LOAD_FAST_BORROW_LOAD_FAST_BORROW 16 (total, seq)
LOAD_FAST_BORROW 2 (i)
BINARY_OP_SUBSCR_LIST_INT 26 ([])

23 BINARY_OP_ADD_INT 13 (+=)
STORE_FAST 1 (total)
```

Особливий інтерес становить спеціалізація циклу. Інтерпретатор розпізнає ітерацію по об'єкту range та замінює універсальний механізм обходу на інструкцію FOR_ITER_RANGE, яка безпосередньо працює зі структурою об'єкта range. Аналогічно доступ до елементів списку через індекс спеціалізується до BINARY_OP_SUBSCR_LIST_INT, що усуває необхідність виконання загального алгоритму індексації для довільних об'єктів.

Дослідимо спеціалізацію викликів методу.

Виклики методів традиційно належать до найбільш витратних операцій в інтерпретованих мовах, оскільки потребують пошуку атрибута, створення зв'язаного методу та перевірки коректності аргументів. Після накопичення статистики CPython визначає, що виклик здійснюється для одного й того самого типу об'єктів і з незмінною кількістю параметрів.

Метод викликається через LOAD_ATTR + CALL. У стані BEFORE відсутня інформація про стабільність функції та сигнатуру виклику.

Сніппет 3.16 – Інструкції в стані BEFORE

```
=== method_call: BEFORE ===
26 LOAD_ATTR 1 (add + NULL|self)
LOAD_SMALL_INT 1
CALL 1
BINARY_OP 13 (+=)
```

Після warm-up використовується спеціалізований доступ до методу та виклик з фіксованою сигнатурою CALL_PY_EXACT_ARGS. Додається inline cache метаданих.

Сніппет 3.17 – Інструкції в стані AFTER

```
=== method_call: AFTER ===
26 LOAD_ATTR_METHOD_WITH_VALUES 1 (add + NULL|self)
LOAD_SMALL_INT 1
CALL_PY_EXACT_ARGS 1
BINARY_OP_ADD_INT 13 (+=)
```

У результаті використовується спеціалізована інструкція CALL_PY_EXACT_ARGS, яка розрахована на виклики з фіксованою сигнатурою. Це дозволяє скоротити кількість перевірок, що виконуються перед фактичним переходом до коду функції.

Аналіз усіх трьох сценаріїв показує, що адаптивна спеціалізація в CPython охоплює декілька рівнів виконання програми: доступ до атрибутів, арифметичні операції, ітерацію колекцій, індексацію та виклики функцій. В усіх випадках універсальні інструкції замінюються версіями, орієнтованими на конкретні типи даних або шаблони виконання.

Отримані результати підтверджують, що сучасний CPython використовує профілювально-орієнтований підхід до оптимізації, за якого рішення про спеціалізацію приймаються безпосередньо під час виконання програми на основі накопиченої статистики.

3.3 Перевірка динамічних оптимізацій: інлайнове кешування

У всіх попередніх випадках ключову роль відіграє механізм CACHE, який спочатку містить нульові значення, а після warm-up накопичує статистику виконання (counter, version, keys_version, func_version). Це дозволяє інтерпретатору підтверджувати стабільність типів і переходити до спеціалізованих інструкцій.

CACHE використовується для збереження метаданих про попередні виконання opcode, включаючи:

- counter (кількість виконань інструкції),
- version (версія структури типу або об’єкта),
- keys_version (версія словника атрибутів),
- func_version (версія функції або методу).

Сніппет 3.18 – Еволюція кешу

```
BEFORE:
CACHE 0 (counter: 0)

AFTER:
CACHE 0 (counter: 416)
CACHE 0 (version: 131156)
CACHE 0 (keys_version: 24)
CACHE 0 (func_version: 746)
```

Отримані значення кешу демонструють, що після достатньої кількості виконань інтерпретатор накопичує інформацію про стабільність структури об'єктів.

Зокрема, параметри version та keys_version використовуються для перевірки того, що схема розміщення атрибутів не змінилася з моменту спеціалізації інструкції.

Якщо структура об'єкта залишається незмінною, інтерпретатор може виконувати прямий доступ до необхідного поля без повторного пошуку в словниках атрибутів. У випадку зміни структури кеш автоматично втрачає актуальність, а спеціалізована інструкція може бути повернута до універсальної форми.

Таким чином, інлайнове кешування виконує дві функції одночасно: накопичує статистику для прийняття рішень щодо спеціалізації та забезпечує перевірку коректності вже виконаних оптимізацій.

3.5 Перевірка динамічних оптимізацій: суперінструкції

На відміну від попередніх оптимізацій, суперінструкції не завжди безпосередньо відображаються в дизасемблері як окремі opcode. Натомість їх існування можна встановити опосередковано через появу послідовностей спеціалізованих інструкцій, які інтерпретатор виконує як єдиний оптимізований шлях виконання.

Перевіримо формування суперінструкцій на прикладі злиття доступу до атрибутів та арифметики.

У стані BEFORE операція доступу до атрибута та додавання виконуються як дві окремі інструкції: LOAD_ATTR + BINARY_OP. Це означає, що між ними відбувається повний цикл dispatch та робота стеку VM.

Сніппет 3.19 – Інструкції в стані BEFORE

```
=== read_attr: BEFORE ===
LOAD_ATTR 0 (x)
BINARY_OP 13 (+=)
```

Після warm-up CPython об’єднує ці операції в суперінструкцію, яка виконує одразу доступ до значення атрибута та його додавання до акумулятора. Це проявляється як LOAD_ATTR_INSTANCE_VALUE + BINARY_OP_ADD_INT, але на рівні виконання вони можуть бути злиті в єдиний execution path.

Сніппет 3.20 – Інструкції в стані AFTER

```
=== read_attr: AFTER ===
LOAD_ATTR_INSTANCE_VALUE 0 (x)
BINARY_OP_ADD_INT 13 (+=)
```

У випадку індексації списку до оптимізації кожна ітерація циклу складається з окремих інструкцій: FOR_ITER + BINARY_OP. Це означає, що кожен цикл включає щонайменше два цикли диспетчеризації.

Сніппет 3.21 – Інструкції в стані BEFORE

```
=== list_index: BEFORE ===
FOR_ITER
BINARY_OP ([])
```

Після застосування суперінструкцій CPython формує комбінований шлях виконання FOR_ITER_RANGE + BINARY_OP_SUBSCR_LIST_INT, який усуває проміжні переходи між отриманням індексу та доступом до елемента списку.

Сніппет 3.22 – Інструкції в стані AFTER

```
=== list_index: AFTER ===
FOR_ITER_RANGE
BINARY_OP_SUBSCR_LIST_INT
```

Важливим ефектом є зменшення кількості операцій у межах одного циклу, оскільки індекс та доступ до елемента обробляються як єдина операція.

Найбільш виражений ефект суперінструкцій спостерігається у викликах методів. У стані BEFORE виконання складається з двох окремих фаз: отримання методу через LOAD_ATTR та виклик через CALL

Сніппет 3.23 – Інструкції в стані BEFORE

```
=== method_call: BEFORE ===
LOAD_ATTR add
```

```
CALL 1
```

Після оптимізації ці етапи частково зливаються у LOAD_ATTR_METHOD_WITH_VALUES + CALL_PY_EXACT_ARGS, де відбувається кешування дескриптора методу та фіксація сигнатури виклику. Хоча формально це не повністю одна інструкція, логічно це є fused execution path, де мінімізуються накладні витрати на підготовку виклику.

Сніппет 3.24 – Інструкції в стані AFTER

```
=== method_call: AFTER ===
LOAD_ATTR_METHOD_WITH_VALUES add
CALL_PY_EXACT_ARGS 1
```

Отримані результати не дозволяють безпосередньо спостерігати внутрішні механізми формування суперінструкцій, проте демонструють появу характерних шаблонів спеціалізованого байт-коду, які використовуються інтерпретатором для зменшення кількості циклів диспетчеризації.

Аналіз отриманих результатів показує, що суперінструкції є логічним продовженням механізму спеціалізації байт-коду та інлайнового кешування. Якщо спеціалізація зменшує вартість окремих операцій, а кешування зменшує вартість повторних перевірок, то суперінструкції зменшують кількість самих переходів між операціями.

3.6 Висновки

У розділі було проведено перевірку запропонованих статичних і динамічних оптимізацій.

Результати тестування статичного оптимізатора підтвердили коректність роботи проходів поширення та згортання констант, strength reduction, peephole-оптимізацій і видалення мертвого коду. Інтеграційний тест показав, що їх послідовне застосування дозволяє досягати додаткових спрощень проміжного представлення програми.

Дослідження динамічних оптимізацій на прикладі CPython підтвердило ефективність адаптивної спеціалізації байткоду, інлайнового кешування та суперінструкцій. Аналіз байткоду до та після накопичення статистики виконання продемонстрував зміну універсальних інструкцій на спеціалізовані та скорочення накладних витрат інтерпретації.

Отримані результати підтверджують працездатність запропонованої архітектури оптимізуючого інтерпретатора та доцільність поєднання статичних і динамічних оптимізацій.

ВИСНОВКИ

У роботі було досліджено архітектуру оптимізуючого інтерпретатора та принципи побудови статичних і динамічних оптимізацій для проміжного представлення програми. У ході роботи було проаналізовано основні моделі виконання коду, роль проміжного представлення в компіляторах та інтерпретаторах, а також місце SSA-форм, CFG і def-use аналізу в процесі оптимізації.

На основі проведеного аналізу було спроєктовано архітектуру оптимізуючого інтерпретатора, орієнтовану на байт-кодову модель виконання та SSA-подання проміжного коду. Для статичної частини системи було визначено набір оптимізацій, що включає поширення копій і констант, згортання констант, strength reduction, peephole-оптимізації, відновлення узгодженості SSA та видалення мертвого коду. Така структура дозволила організувати послідовний ітеративний pipeline, у якому результати одного проходу створюють умови для наступних спрощень.

Окремо було досліджено динамічні оптимізації на прикладі CPython як готового адаптивного середовища виконання. Розгляд інлайнового кешування, спеціалізації байт-коду та суперінструкцій показав, що ефективність інтерпретації можна підвищувати не лише за рахунок статичного спрощення коду, а й завдяки адаптації до реального профілю виконання програми.

Практична частина роботи підтвердила працездатність запропонованого підходу. Тестування статичних оптимізацій показало коректність їхнього застосування та взаємодії між собою, а аналіз динамічних механізмів у CPython продемонстрував зміну байт-коду після warm-up і зменшення накладних витрат інтерпретації. Отримані результати підтверджують, що поєднання статичних і динамічних оптимізацій є доцільним підходом для побудови ефективного оптимізуючого інтерпретатора та може бути основою для подальшого розвитку систем подібного класу.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1) ARM. Instruction Set Architecture (ISA). URL: https://www.arm.com/glossary/isa (дата звернення: 31.05.2026).
2) GeeksforGeeks. Code Optimization in Compiler Design. URL: https://www.geeksforgeeks.org/compiler-design/code-optimization-in-compiler-design/ (дата звернення: 31.05.2026).
3) EasyExamNotes. Static vs Dynamic Optimization. URL: https://easyexamnotes.com/explain-static-vs-dynamic-optimization/ (дата звернення: 31.05.2026).
4) Dead-code elimination. URL: https://everything.explained.today/Dead-code_elimination/ (дата звернення: 31.05.2026).
5) Cooper K. D., Torczon L. Engineering a Compiler. 3rd ed. Elsevier, 2022.
6) Anatomy of Programming Languages. URL: https://www.cs.utexas.edu/~wcook/anatomy/anatomy.htm (дата звернення: 31.05.2026).
7) GeeksforGeeks. Constant Propagation in Compiler Design. URL: https://www.geeksforgeeks.org/compiler-design/constant-propagation-in-complier-design/ (дата звернення: 31.05.2026).
8) Peephole Optimization. URL: https://www.sciencedirect.com/science/chapter/monograph/abs/pii/B9780128154120000176 (дата звернення: 31.05.2026).
9) Aho A. V. et al. Compilers: Principles, Techniques, and Tools. 2nd ed. Addison-Wesley, 2006.
10) TechTarget. Bytecode Definition. URL: https://www.techtarget.com/whatis/definition/bytecode (дата звернення: 31.05.2026).
11) EPFL CS420. Interpreters and Virtual Machines. URL: https://cs420.epfl.ch/archive/24/c/11_interp-vms.html (дата звернення: 31.05.2026).
12) Inline Caches in Virtual Machines. URL: https://mrale.ph/blog/2012/06/03/explaining-js-vms-in-js-inline-caches.html (дата звернення: 31.05.2026).
13) Python CPython Developers. Interpreter internals. URL: https://github.com/python/cpython/blob/main/InternalDocs/interpreter.md (дата звернення: 31.05.2026).
14) Matz A. Dissertation on virtual machine optimizations. Dispatch Techniques. URL: https://www.cs.toronto.edu/~matz/dissertation/matzDissertation-latex2html/node6.html (дата звернення: 31.05.2026).

15) Crafting Interpreters. A Virtual Machine. URL: https://craftinginterpreters.com/a-virtual-machine.html (дата звернення: 31.05.2026).
16) GeeksforGeeks. Three Address Code in Compiler Design. URL: https://www.geeksforgeeks.org/compiler-design/three-address-code-compiler/ (дата звернення: 31.05.2026).
17) Intermediate Representations. URL: https://compilerprogramming.github.io/intermediate-representations.html (дата звернення: 31.05.2026).
18) Muchnick S. S. Advanced Compiler Design and Implementation. Morgan Kaufmann, 1997.
19) Cytron R. et al. Efficiently Computing Static Single Assignment Form. URL: https://www.cs.utexas.edu/~pingali/CS380C/2010/papers/ssaCytron.pdf (дата звернення: 31.05.2026).
20) GeeksforGeeks. Python vs CPython. URL: https://www.geeksforgeeks.org/python/python-vs-cpython/ (дата звернення: 31.05.2026).
21) Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, Laurence Tratt. Virtual Machine Warmup Blows Hot and Cold. URL: https://arxiv.org/pdf/1602.00602 (дата звернення: 31.05.2026).
22) Репозиторій з кодом проєкту на GitHub. URL: https://github.com/anotherwinter/unnamedlang/tree/ir (дата звернення: 31.05.2026).