Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Інформаційно-аналітичних технологій та менеджменту
(повна назва)
Кафедра Інформатики
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти перший (бакалаврський)

РОЗРОБКА МОВИ ПРОГРАМУВАННЯ JIT ДЛЯ ПОБУДОВИ НЕЙРОННИХ МЕРЕЖ
(тема)

Виконав:
студент 4 курсу, групи ІТІНФ-19-2
Коденко Я. Р.
(прізвище, ініціали)

Спеціальності 122 Комп'ютерні науки
(код і повна назва спеціальності)

Тип програми освітньо-професійна

Освітня програма Інформатика
(повна назва освітньої програми)

Керівник проф. Машталір С. В.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри _____ Кобилін О.А.
(підпис) (прізвище, ініціали)

2023 р.

Харківський національний університет радіоелектроніки

Факультет Інформаційно-аналітичних технологій та менеджменту
(повна назва)

Кафедра Інформатики
(повна назва)

Рівень вищої освіти перший (бакалаврський)

Спеціальність 122 Комп'ютерні науки
(код і повна назва)

Тип програми освітньо-професійна

Освітня програма Інформатика
(повна назва освітньої програми)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«____»_____ 2023 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Коденкові Ярославу Романовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Розробка мови програмування JIT для побудови нейронних мереж.

затверджена наказом університету від 15 травня 2023 року № 474 Ст

2. Термін подання студентом роботи до екзаменаційної комісії 02 червня 2023 р.

3. Вихідні дані до роботи науково-методична та науково-технічна література, матеріали конференцій, дані інтернет-мережі, бібліотека Boost.Spirit, бібліотеки LLVM, засіб для автоматизації зборки CMake, пакетний менеджер VCKPG, бібліотека MS.GSL, мова програмування C++ 23.

4. Перелік питань, що потрібно опрацювати в роботі
1. Огляд існуючих методів та алгоритмів для розробки компіляторів.
2. Опис формальної мови для подальшої побудови парсера.
3.Програмна реалізація компілятора.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри) UML діаграми компонентів програмного застосунку, модель даних для обробки.

6. Консультанти розділів роботи (п.6 включається до завдання за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата
Консультант з дотримання діючих стандартів та норм	Доцент Творошенко І.С.		

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	Отримання завдання на кваліфікаційну роботу	10.04.2023	
2	Аналіз завдання, підбір літератури	11.04.23-17.04.23	
3	Аналіз літератури з досліджуваної проблеми	18.04.23-20.04.23	
4	Аналіз технічних засобів	21.04.23-30.04.23	
5	Розробка методу	01.05.23-14.05.23	
6	Програмна реалізація	15.05.23-23.05.23	
7	Оформлення пояснювальної записки	24.05.23-26.05.23	
8	Перевірка на плагіат	27.05.23	
9	Рецензування	28.05.23	
10	Підготовка презентації та доповіді	29.05.23-30.05.23	
11	Занесення роботи в електронний архів	31.05.23	
12	Попередній захист кваліфікаційної роботи	11.06.23	

Дата видачі завдання 10 квітня 2023 р.

Студент _____
(підпис)

Керівник роботи _____ _____проф. Машталір С. В._____
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ/ABSTRACT

Пояснювальна записка до кваліфікаційної роботи: 61 с., 9 рис., 40 джерел.

ПАРСИНГ, АБСТРАКТНЕ СИНТАКСИЧНЕ ДЕРЕВО, АРХІТЕКТУРА КОМПІЛЯТОРА, ГРАМАТИКА ЯКА РОЗБИВАЄ ВИРАЗИ.

Об’єктом роботи є алгоритми перетворення початкового коду на машинний код.

Метою роботи є розробка застосунка для обробки тексту та генерації машинного коду.

Проаналізовані вимоги до мови програмування. Описана формальна мова за допомогою PEG. Проаналізовані засоби програмування для створення компіляторів, альтернативи були порівняні. Досліджена граматика PEG та алгоритми синтаксичного аналізу. Досліджені програмні застосунки для створення мов програмування.

У результаті роботи була розроблена мова програмування та програмна реалізація компілятора.

PARSING, ABSTRACT SYNTAX TREE, COMPILER’S ARCITECTURE, PARSING EXPRESSION GRAMMARS.

The objective of the work is the source code’s transformation into machine code.

The aim of the work is the development of an application that processes text and generates machine code.

Requirements for the programming language were analyzed. A formal language was described with PEG. Programming tools for compiler development were explored and the alternatives were compared. PEG grammar and parsing algorithms were studied.

As a result of the work, a programming language and a compiler were developed.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

АСТ – абстрактне синтаксичне дерево

PEG – Parsing Expression Grammar (граматика яка розбиває вирази)

API – Application Programming Interface (інтерфейс для програмування застосунку)

Парсинг – процес синтаксичного аналізу початкового коду

Фронтенд компілятора – архітектурна частина компілятора, яка перетворює початковий код в платформо незалежну форму для подальшої обробки бекендом

Бекенд компілятора – архітектурна частина компілятора, яка перетворює результат роботи фронтенда в машинний код цільової платформи

ВСТУП

Мови програмування поділяються мови загального призначення та предметно орієнтовані.

Мови програмування загального призначення дозволяють розробляти різноманітні застосунки без обмеження предметної області. Проблема полягає в тому, що вирішення задач деякої галузі на мовах загального призначення може супроводжуватися невиправдано великими затратами часу та людських ресурсів. Предметно орієнтовані мови програмування дозволяють формалізувати та чіткіше виразити певний тип проблеми та її рішення.

Найпопулярніша мова програмування для робіт та досліджень, які пов’язані з нейронними мережами та штучним інтелектом, – це Python. Проблема в тому, що можливості мови нейронних мереж необхідні в багатьох областях, де використання мови Python може бути пов’язаним з певними складнощами. Мови C та C++, які дозволяють використовувати популярні фреймворки для машинного навчання, значно складніші за мову Python як у використанні так і в вивчанні. Через це виникає необхідність у простій мові програмування, яка дозволить легко розробляти нейронні мережі, але при цьому не потребуватиме повільного інтерпретатора Python.

Актуальність роботи полягає у необхідності розробки та дослідження застосунків, які використовують нейронні мережі та штучний інтелект. Мова програмування, яка розробляється, дозволить легко інтегрувати функції штучного інтелекту до нативних застосунків. Така інтеграція необхідна, адже дослідження відео та аудіо джерел вимагають високу швидкість роботи системи у реальному часі, що не завжди можливо досягти за допомогою повільних інтерпретованих мов програмування.

1 ОГЛЯД ОСНОВНИХ МЕТОДІВ РОЗРОБКИ КОМПІЛЯТОРА

1.1 Типи мов програмування

Мови програмування можна поділити на дві групи за форматом розповсюдження програмного застосунка. Це – компільовані та інтерпретовані програмування. Програмні застосунки на компільованих мовах програмування розповсюджуються у формі об’єктних файлів з машинними інструкціями. На платформі Linux такі файли мають формат ELF (Executable and Linkable Format), а на платформі Windows у формі PE (Portable Executable) – це файли з розширенням назви файлу «.exe». до цієї категорії можна віднести мови, які транслюються у певний платформо-незалежний код для віртуальної платформи. Приклади: C++, Rust, Java, C#. Програмні застосунки на інтерпретованих мовах програмування розповсюджуються у вигляді початкового коду програми. Приклади: JavaScript, Python.

Важливо зазначити, що сучасні інтерпретовані мови програмування під час виконання компілюються у платформо-незалежний байт-код, після цього вони виконуються так само як і мови Java або C#. Це зроблено з метою поліпшення часу виконання застосунку. При такому підході об’єктний чи машинний код зберігається в оперативній пам’яті та може опціонально кешуватися на накопичувачі у тимчасових файлах. Як правило, це трохи збільшує час старту програми та об’єм необхідної пам’яті, проте приріст у швидкості виконання компенсує недоліки. Через таку оптимізацію архітектура компілятора у мов програмування різних типів може бути дещо схожа.

1.2 Загальна структура компілятора

Типовий компілятор структурно можна поділити на три частини: фронтенд, оптимізатор, бекенд [1–7].

Задача фронтенду полягає у перетворенні коду програми на абстрактне синтактичне дерево – AST. AST – це структура даних, яка дозволяє представити код програми для подальшої обробки. Слід зазначити, що часто перед генерацією AST може генеруватися конкретне синтаксичне дерево. На відміну від AST, така структура даних може зберігати дані, які не важливі для подальшої роботи компілятора. Наприклад, коментарі, дужки у виразах, коми, тощо. Ця інформація може бути потрібна для проведення статичного аналізу, генерації стилістичних зауважень, автоматичного форматування коду.

Також на етапі роботи фронтенду компілятор повідомить користувача інформацію про синтаксичні помилки у коді програмі.

Оптимізатор аналізує AST та робить необхідні перетворення для покращення часу роботи програми та, іноді, зменшення розміру програми на накопичувачі. Це може включати обчислення деяких виразів на етапі компіляції, інлайнінг функцій, видалення невикористаного коду, тощо. При ідеальному дизайні компілятора оптимізатор не повинен залежати від цільової платформи, проте дуже часто на практиці це не зовсім так. У результаті своєї роботи оптимізатор може повертати як просто оптимізоване AST, так і якусь іншу структуру даних для опрацьовування бекендом. Така структура може бути у формі якоїсь низькорівневої абстрактної мови програмування. Це може бути необхідно для передачі даних бекенду про можливість використання специфічних інструкцій. Наприклад, для використання векторних інструкцій на платформах з їх підтримкою.

Задача бекенду полягає у генерації машинного коду. Цільова платформа може буте реальною, наприклад x86 або ARM, або віртуальною, наприклад JVM або CLR.

1.3 Огляд реалізації фронтедну

1.3.1 Парсинг

Перший етап роботи фронтенду мови називається парсинг. Існують декілька алгоритмів та методів для його реалізації. Найпростіший з них – рекурсивний спуск. Слід зазначити, що можливість використання одного чи іншого алгоритму залежить від типу граматики мови. Рекурсивний спуск неможливо застосувати до будь-яких контекстно-незалежних граматик, хоча з практичної точки зору цей алгоритм доволі корисний. Справа в тому, що дуже часто складна для реалізації граматика може бути складною для використання кінцевим користувачем. Це зменшує практичність мови програмування.

Парсинг можна реалізувати власноруч, а можна використовувати існуючи інструменти для генерації парсерів. Серед таких інструментів GNU Bison, Boost.Spirit, тощо. Такі інструменти відомі як компілятори для компіляторів (compiler-compiler). GNU Bison – це інструмент, яких дозволяє описати формальну граматику у формі EBNF для генерації парсеру. Boost.Spirit – це бібліотека для мови C++ з групи бібліотек Boost. За допомогою Boost.Spirit можливо отримати доволі швидкий парсер з рекурсивним спуском, який генерує структури на мові C++ [8–14].

Більшість помилок синтаксису буде отримано після парсингу. В результаті роботи своєї роботи парсер повертає AST.

1.3.2 Аналіз AST

Після генерації AST може бути необхідно зробити додаткові перевірки коректності згенерованої структури, або згенерувати додаткову інформацію для генерації машинного коду. Незважаючи на коректність коду програми з точки зору синтаксису, користувач може допустити інші помилки. Наприклад, помилки типізації. Під час аналізу компілятор проходить вузли AST та додає додаткові маркери. Наприклад, у випадку мови C++ це може бути додавання

до виразів позначень LValue, RValue, PRValue, XValue, тощо. У мові програмування C++ ці позначення важливі для перевірки коректності виразів, перетворення і дедукції типів, та інших механізмів мови, таких як семантика переміщення [15-23].

У найпростіших мовах програмування з динамічної типізацію цей етап може бути сильно спрощений. Частина перевірок при такому підході буде реалізована під час виконання програми.

1.4 Огляд реалізації оптимізатору

Оптимізація повинна виконуватися з урахуванням можливих архітектурних особливостей цільової платформи. Треба розуміти, що поведінка різних платформ може суттєво відрізнятися. Ці особливості можуть включати необхідність вирівнювання адрес даних, апаратну підтримку чисел з плаваючою точкою, тощо. Наприклад, на деяких процесорах платформи MIPS, які часто використовуються у мережевому обладнані, відсутня апаратна підтримка чисел з плаваючою точкою.

Через залежність процесу оптимізації від платформи, на якій буде виконуватися програма, оптимізація може проводитися на різних етапах компіляції. Наприклад, після обробки фронтендом коду, оптимізатор аналізує отриманий результат та робить платформо-незалежну оптимізацію. Також фронтенд або оптимізатор можуть залишати певні метадані на елементах код. Наприклад, у мові програмування Rust, в певний момент часу може існувати тільки один не константний референс на змінну. Через це фронтент може позначити змінну, як «noalias». Це позначення дозволяє бекенду реалізувати всі операції зі змінною в регістрах без необхідності робити копію на стеку.

1.5 Огляд реалізації бекенду

Хоча можливо реалізувати бекенд самостійно, більш прагматичним буде використати існуючі засоби для генерації машинного коду. Це дозволить спростити дизайн компілятора та покращити якість кінцевого продукту. Найбільш популярні на сьогоднішній день – GCC та LLVM.

Перша успішна реалізація модульної структури була GCC. Нажаль, архітектура GCC доволі монолітна, тому елементи компіляторів з цієї групи важко використовувати у якості бібліотек для розробки інший застосунків, наприклад, статичних аналізаторів.

Найбільш зручним фреймворком для розробки компіляторів на сьогоднішній день є LLVM. Цей набір бібліотек розроблявся з метою отримати універсальний набір інструментів для підтримки великої кількості платформ та інструментів. LLVM в змозі здійснити оптимізацію коду під цільову платформу, тому розробка додаткового оптимізатора не потрібна [4, 5].

1.6 Огляд способів реалізації підтримки нейронних мереж

Програмування нейронних мереж та штучного інтелекта вимагає багатьох паралельних математичних перетворень над вхідними даними. У якості основної обчислювальної одиниці у таких системах може виступати як центральний процесор, так і відеокарта. Так, наприклад, найбільш потужний шаховий рушій Stockfish використовує нейронну мережу NNUE (Efficiently updatable neural network) для оцінки позиції. При цьому усі обчислення використовують центральний процесор. У той же час один із основних конкурентів Stockfish, шаховий рушій Leela Chess Zero, проводить усі обчислення на відеокарті. Незважаючи на це, сила обох шахових рушіїв знаходиться приблизно на одному рівні [24–32].

Вибір відеокарти або центрального процесора для обчислень базується на обмеженнях конкретної задачі. Таким чином, для широти практичного

використання у розробляємій мові програмування повинна бути реалізована відповідна підтримка для багатьох пристроїв.

Реалізувати всі необхідні механізми можливо як з нуля, так і використовуючи вже існуючи фреймворки для машинного навчання.

Для створення механізмів для роботи з нейронними мережами необхідно мати інструмент для програмування розподілених обчислень. Таким інструментом може стати SYCL. SYCL – це модель для програмування гетерогенних пристроїв за допомогою C++17, яка стандартизується Khronos Group. За допомогою SYCL можливо розподіляти обчислення на різні пристрої, у тому числі комбінувати роботу центрального процесора та відеокарти.

Проблема реалізації усіх механізмів роботи з нейронними мережами полягає у обмеження ресурсів для розробки. Альтернативою цьому способу виступає використання існуючих фреймворків. Серед них найвідомішими є PyTorch та TensorFlow. Головним мінусом такого підходу є залежність від форматів та механізмів роботи фреймворків.

У даній роботі основною мовою програмування виступає C++, тому перевагу віддається PyTorch за дуже зручний та простий API для роботи з C++.

1.7 Постановка задачі

Таким чином, завдяки LLVM та бібліотек для генерації парсингу, для створення компілятора для нової мови програмування необхідно реалізувати фронтенд для обробки коду та передачі відповідних даних до LLVM.

Об’єктом роботи є алгоритми перетворення початкового коду на машинний код.

Метою роботи є розробка застосунка для обробки тексту та генерації машинного коду.

Для досягнення мети необхідно вирішити такі завдання:

- провести аналіз алгоритмів парсингу;
- розробити граматику мови програмування;
- реалізувати алгоритм для парсинга створеної граматики;
- реалізувати механізм для перетворення АСТ в машинний код.

2 МАТЕМАТИЧНІ МОДЕЛІ МОВИ ПРОГРАМУВАННЯ

2.1 PEG граматика

Перед створенням застосунку необхідно розробити формальну модель мови програмування. Це необхідно для планування архітектури застосунка та подальшого тестування.

Існує декілька способів опису формальної мови. В рамках даної роботи буде використана граматика PEG (parsing expression grammar) – граматика, яка розбиває вирази [33, 34]. PEG складається з:

- кінцевої множини N не термінальних символів;
- кінцевої множини F термінальних символів;
- кінцевої множини P правил парсингу;
- виразу e_S який називається початковим виразом.

Кожне правило $p \in P$ має форму

$$n \leftarrow e, \tag{2.1}$$

де n – не термінальних символ $n \in N$;

e – вираз.

Кожен вираз – ієрархічний вираз.

Нехай, маємо вирази e, e_1, e_2. Тоді новий вираз можемо побудувати за допомогою наступних операцій:

- послідовність: $e_1 e_2$;
- вибір: $e_1 | e_2$;
- нуль або більше: $e*$;
- один або більше: $e+$;
- не обов’язково: $e?$;
- предикат «і»: $\&e$;

– предикат «ні»: $!e$.

2.1.1 Рекурсивний спуск

Один з найпростіший та найкорисніших алгоритмів для парсингу є рекурсивний спуск. Цей алгоритм використовується разом з PEG, адже кожне правило являє собою рекурсивну функцію в алгоритмі рекурсивного спуску.

Атомарний вираз у PEG – це термінальний символ. Термінальні символи не визиватимуть рекурсію.

Наведемо приклад парсингу простої граматики.

Expr ← Sum

Sum ← number ‘+’ number

Лістинг 2.1 Приклад рекурсивного спуску для заданої граматики:

```
accept(Expr, TokenIterator):
    if accept(Sum, TokenIterator)
        return True
    return False

accept (Sum, TokenIterator):
    if not accept(Number, TokenIterator)
        return false
    if TokenIterator.Next != ‘+’
        return false
    if not accept(Number, TokenIterator.Next )
        return false
  return true'
```

2.1.2 PEG граматика у бібліотеці Boost.SpiritX3

Бібліотека для мови програмування C++ Boost.SpiritX3 дозволяє швидко створювати парсери за допомогою опису граматики мови за допомогою перевантаження операторів у мові C++ [35–40]. Такий спосіб запису граматики нагадує PEG, але записаний задом наперед, коли оператори унарні. Також ця бібліотека надає готові правила для парсингу чисел, символів, строк, тощо. Наприклад, для парсингу пари чисел з плаваючою точкою слід використовувати наступне правило.

Лістинг 2.2 Створення правила для парсингу двух чисел:

```
auto rule = x3::double_ >> x3::double_;
```

2.2 Розробка вимог до мови програмування

Мова програмування, яка розробляється в даній роботі, повинна надавати користувачеві змогу проводити математичні обчислення, писати складні алгоритми та моделювати нейронні мережі. З огляду на ці вимоги, мова програмування повинна:

– підтримувати запис математичних виразів;

– підтримувати створення процедур;

– підтримувати створення нейронних мереж;

– мати можливість друкувати результати обчислень на екран або зберігати в файл;

– мати можливість зберігати натреновану нейронну мережу в файл для подальшого використання.

2.3 Опис граматики мови

Вся граматика мови буде побудована за допомогою множини правил для парсингу окремих її виразів.

2.3.1 Базовий функціонал мови

Типова програма на мові програмування матиме наступний вигляд.

Лістинг 2.3 Приклад програми:

```
const compilation_date = _COMP_DATE;

func say_hello(name: str) -> void {
    echo(“Hello”);
    echo(name);
}

void main() -> void {
    let name = “N”;
    echo(to_string(compilation_date));
    say_hello(name);
}
```

Для опису подібного синтаксису створемо набір правил.

Будь яка програма може бути описана виразом $GlobalExprList$. Цей вираз складається з виразів $GlobalExpr$, які дозволено використовувати поза межами будь-яких функцій. $GlobalExpr$ дозволятиме користувачеві об’являти константи та записувати функції. Таким чином, маємо:

$$GlobalExprList \leftarrow GlobalExpr*, \tag{2.2}$$

$$GlobalExpr \leftarrow FuncDef \mid (ConstDef\ ';'). \tag{2.3}$$

Правило $ConstDef$ матиме наступний вигляд:

$$ConstDef \leftarrow \text{'const'}\ Identifier\ '='\ ValueExpr, \tag{2.4}$$

де $Identifier$ – правило для парсингу ідентифікаторів;

$ValueExpr$ – правило для парсингу виразів, які повертають значення.

Нехай ідентифікатор – це послідовність, яка починається з символу латинського алфавіту, та містить або літери латинського алфавіту, або цифри, або знаки ‘_’. Тоді правило $Identifier$ буде мати наступний вигляд:

$$Identifier \leftarrow Alpha\ (Alnum \mid '\_')*, \tag{2.5}$$

$$Num \leftarrow [0-9], \tag{2.6}$$

$$Alpha \leftarrow [A-Z][a-z], \tag{2.7}$$

$$Alnum \leftarrow Alpha \mid Num. \tag{2.8}$$

Нехай, маємо правило $double$ для парсингу чисел з плаваючою точкою. Тоді правило $ValueExpression$ для парсингу математичних виразів може бути записане наступним чином

$$ValueExpression \leftarrow AdditiveExpr, \tag{2.9}$$

$$AdditiveExpr \leftarrow \quad (2.10)$$
$$MultExpr\big(('+'\ MultExpr)\ \big|('-'\ MultExpr))*,$$

$$MultExpr \leftarrow \quad (2.11)$$
$$UnaryExpr\ \big(('*'\ UnaryExpr)\ \big|('/'\ UnaryExpr)\big),$$

$$UnaryExpr \leftarrow \quad (2.12)$$
$$PrimExpr\ \big|\ ('+'\ PrimExpr)\big|\ ('-'\ PrimExpr)\big),$$

$$PrimExpr \quad (2.13)$$
$$\leftarrow\ double\ |\ ('('\ ValueExpression\ ')')\ |\ FunctionCall\ |\ Identifier.$$

Правило $FunctionCallExpr$ задає синтаксис для виклику функцій:

$$FunctionCallExpr \leftarrow \quad (2.14)$$
$$Identifier\ '('\ ArgumentList\ ')',$$

$$ArgumentList\ \leftarrow (ValueExpr\ (','\ ValueExpr)*\ )?. \quad (2.15)$$

Для парсингу функцій використовуватимемо наступне правило:

$$\text{Func}Def \quad (2.16)$$
$$\leftarrow\ 'func'Identifier\ '('\ ParamList\ ')'\ '\rightarrow'\ Identifier\ CompoundExpr$$
$$,$$

$$ParamList \leftarrow (\ SingleParam\ (','\ SingleParam)*)?, \quad (2.17)$$

$$SingleParam\ \leftarrow Identifier\ ':'\ Identifier, \quad (2.18)$$

$$CompoundExpr \leftarrow \text{'\{'} \ (CompoundExprElement \ \text{';'}) * \ \text{'\}'}, \tag{2.19}$$

$$CompoundExprElement \ \leftarrow ConstDef \mid VariableDef \mid FunctionCall \mid AssignExpr, \tag{2.20}$$

$$VariableDef \ \leftarrow \ 'let' \ Identifier \ \text{'='} \ ValueExpr, \tag{2.21}$$

$$AssignExpr \ \leftarrow Identifier \ \text{'='} \ ValueExpr. \tag{2.22}$$

2.3.2 Додаткові елементи граматики

Вже описана граматика дозволить створити прикладну мову програмування для опису найпростіших сценаріїв. Проблема полягає в тому, що цього буде не достатньо для комфортної роботи з нейронними мережами. З огляду на це, введемо додаткові правила для роботи з масивами, умовні вирази та цикли. Адже нейронні мережі можуть бути представлені у вигляді масивів. Наведемо приклад.

Лістинг 2.4 Приклад створення масиву:

```
let my_array = [3, 4.0, sin(0)];
```

Лістинг 2.5 Приклад умовного оператора:

```
if (3 == get_value()) {
    echo(“Got the value”);
}

if (3 == get_value_another_value()) {
    echo(“Got another value”);
```

} else { echo(“Another value was missed”); }

Лістинг 2.6 Приклад циклу:

```
while ( i < 10) {
    echo(to_string(i));
    i = i + 1;
}
```

Таким чином, дані елементи матимуть наступні правила:

$$ArrayDef \leftarrow \text{'['}\ ValueExpr\ (\text{','}\ ValueExpr) * \text{']'}, \tag{2.23}$$

$$VariableDef \leftarrow \text{'let'}\ Identifier\ \text{'='}\ ValueExpr \mid ArrayDef, \tag{2.24}$$

$$IfStmt \leftarrow \text{'if'}\ \text{'('}\ BooleanExpr\ \text{')'}\ CompoundExpr\ ElseStmt?, \tag{2.25}$$

$$ElseStmt \leftarrow \text{'else'}\ CompoundExpr, \tag{2.26}$$

$$WhileExpr \leftarrow \text{'while'}\ \text{'('}\ BooleanExpr\ \text{')'}\ CompoundExpr, \tag{2.27}$$

$$BooleanExpr \leftarrow PrimBoolExpr\ (BoolOp\ PrimBoolExpr)?, \tag{2.28}$$

$$BoolOp \leftarrow \text{'\&\&'} \mid \text{'||'}, \tag{2.29}$$

$$PrimBoolExpr \leftarrow \text{'!'}?\ CmpExpr, \tag{2.30}$$

$$CmpExpr \leftarrow ValueExpr\ (CmpOp\ ValueExpr)?, \tag{2.31}$$

$$CmpOp \leftarrow\ '<'/\ '>'/\ '='/\ '\leq'/\ '\geq', \tag{2.32}$$

Таким чином правило $CompounExpr$ треба оновити:

$$CompoundExpr \leftarrow \tag{2.33}$$
$$'\{'\ \big((CompoundExprElement\ ';'\ )\ |\ WhileExpr\ |If\ Stmt\big) *$$
$$'\}'.$$

2.4 Опис моделі даних для збереження АСТ

Кожен етап роботи компілятора потребує зручного представлення даних задля швидкої та коректної обробки. АСТ буде представляти тип, яких не має своїх операції та просто передає необхідні дані іншим класам для подальшої обробки.

Взаємодію АСТ з іншими класами описує шаблон проєктування «Відвідувач». За допомогою цього шаблону ми будемо в змозі не створювати зайвих залежностей між модулями програми.

Так, на приклад, генерація машинного коду не потребує можливості надрукувати АСТ в консоль. АСТ матиме наступну структуру до рівня функції (рис. 2.1).

На глобальному рівні структура АСТ доволі прямолінійна та не має рекурсивних залежностей. На рівні функції ситуація відрізняється, бо правила побудови виразів мають рекурсивний характер. Структура АСТ на цьому рівні може бути описана структурним шаблоном «Компонувальник».

Як видно з рисунку 2.2, елементи АСТ мають дуже багато рекурсивних зв’язків. На практиці, у мові C++, це вирішуються за допомогою попередньої декларації [10].

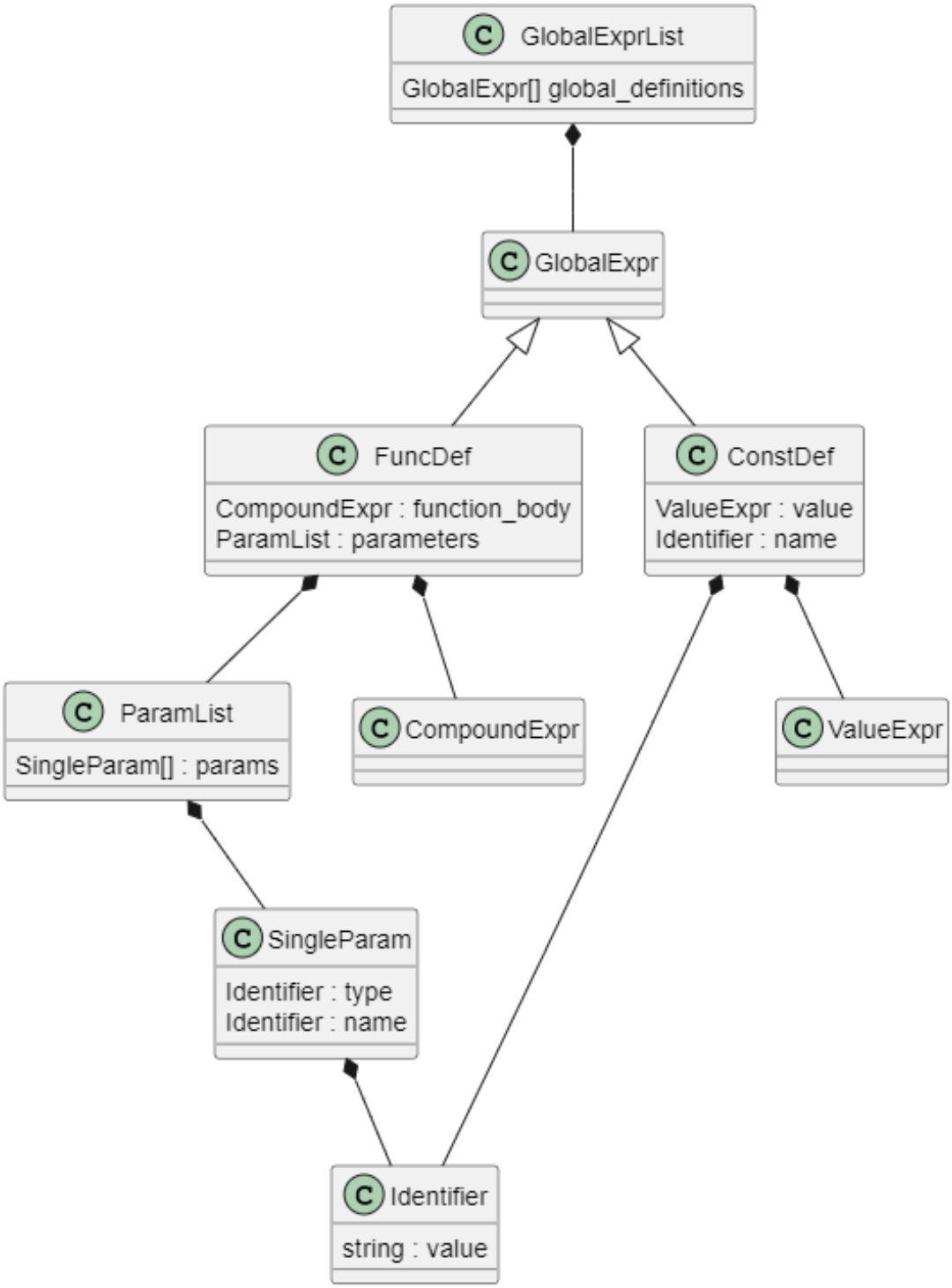

Рисунок 2.1 – Архітектура АСТ до рівня функцій

Бібліотека Boost.Spirit.X3 надає допоміжний шаблон boost::spirit::x3::forward_ast<T> для таких випадків.

Аналогічним чином задаємо інші елементи АСТ (рис. 2.3).

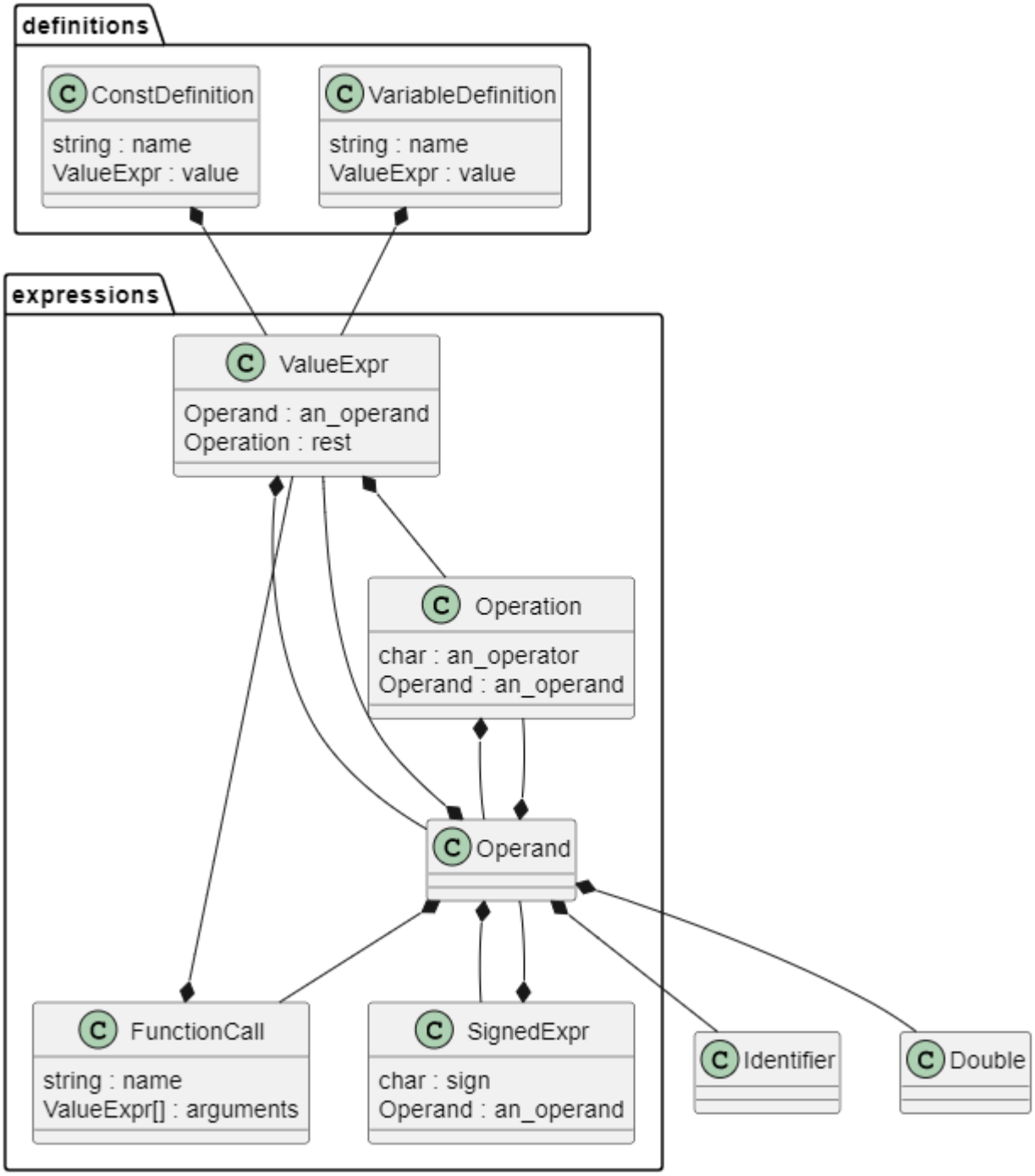

Рисунок 2.2 – Структура АСТ для виразів

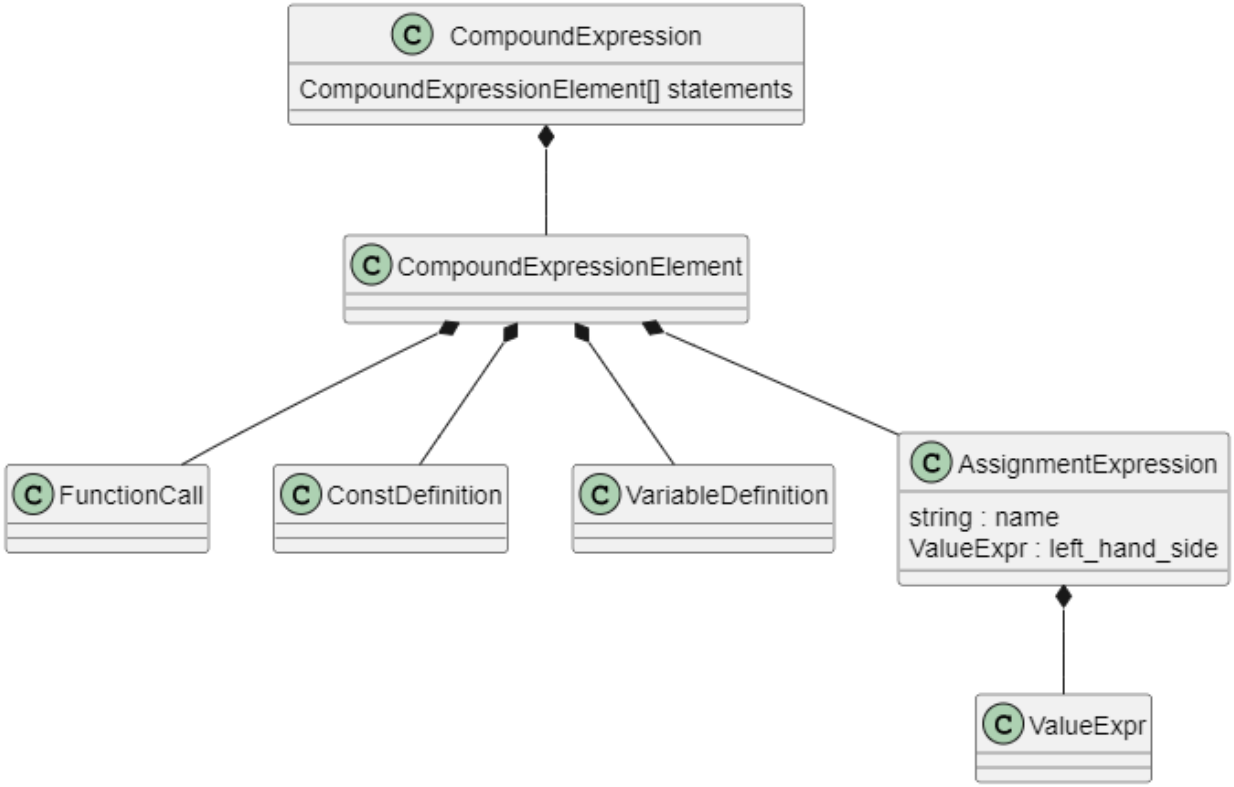

Рисунок 2.3 –Тіло функції

АСТ для представлення операцій потоку керування використовуватимуться наступні структури (рис. 2.4).

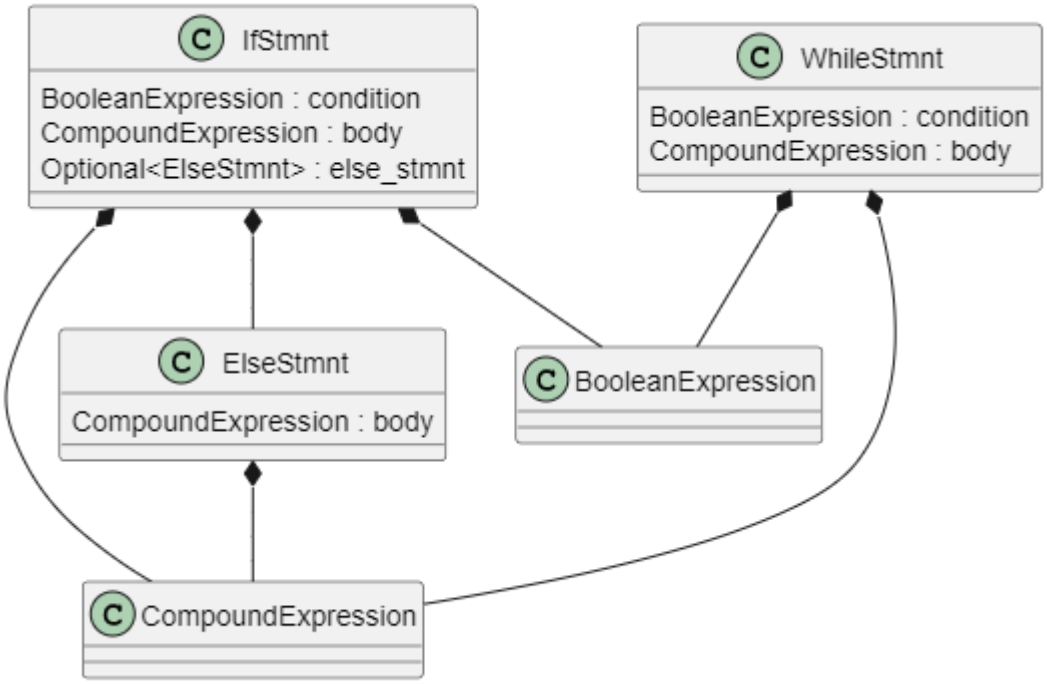

Рисунок 2.4 – АСТ для потоку керування

Умовні вирази слід об’єднати зі звичайними виразами задля гнучкості розробляємої моделі. Тим не менш, в рамках вимог до кінцевого застосунку має сенс спростити дизайн та обробляти умовні вирази як окремий тим виразів. Наступну структуру даних (рис. 2.5) можна використовувати для збереження умовних виразів.

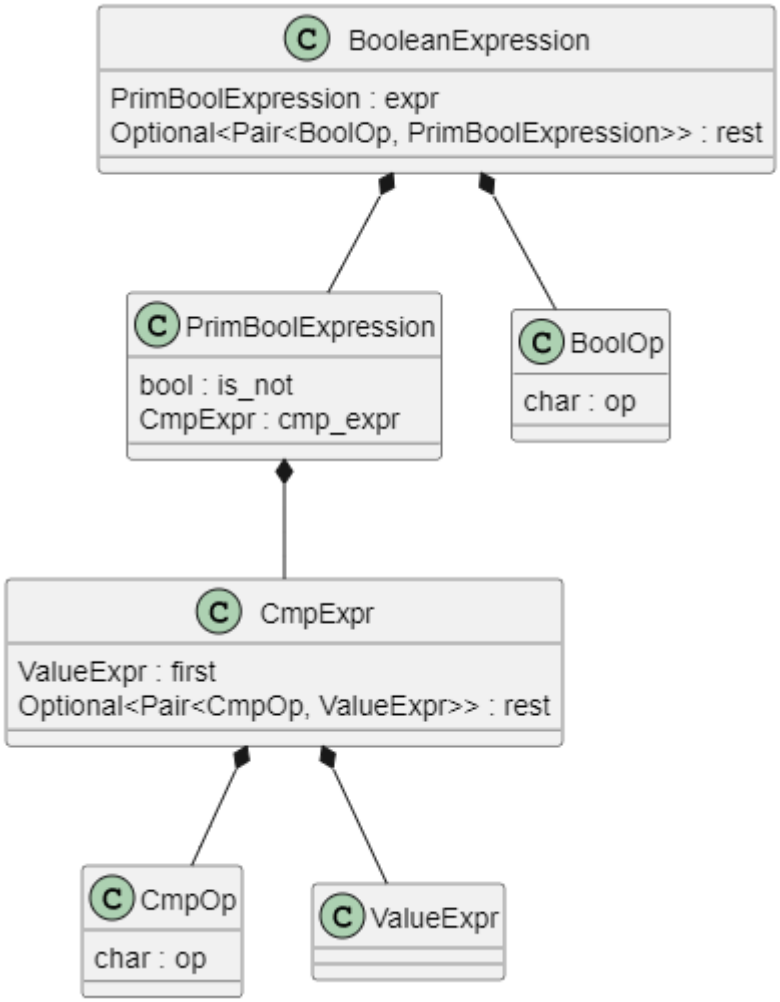

Рисунок 2.5 – Умовні вирази

Загальна діаграма АСТ матиме наступний вигляд (рис. 2.6).

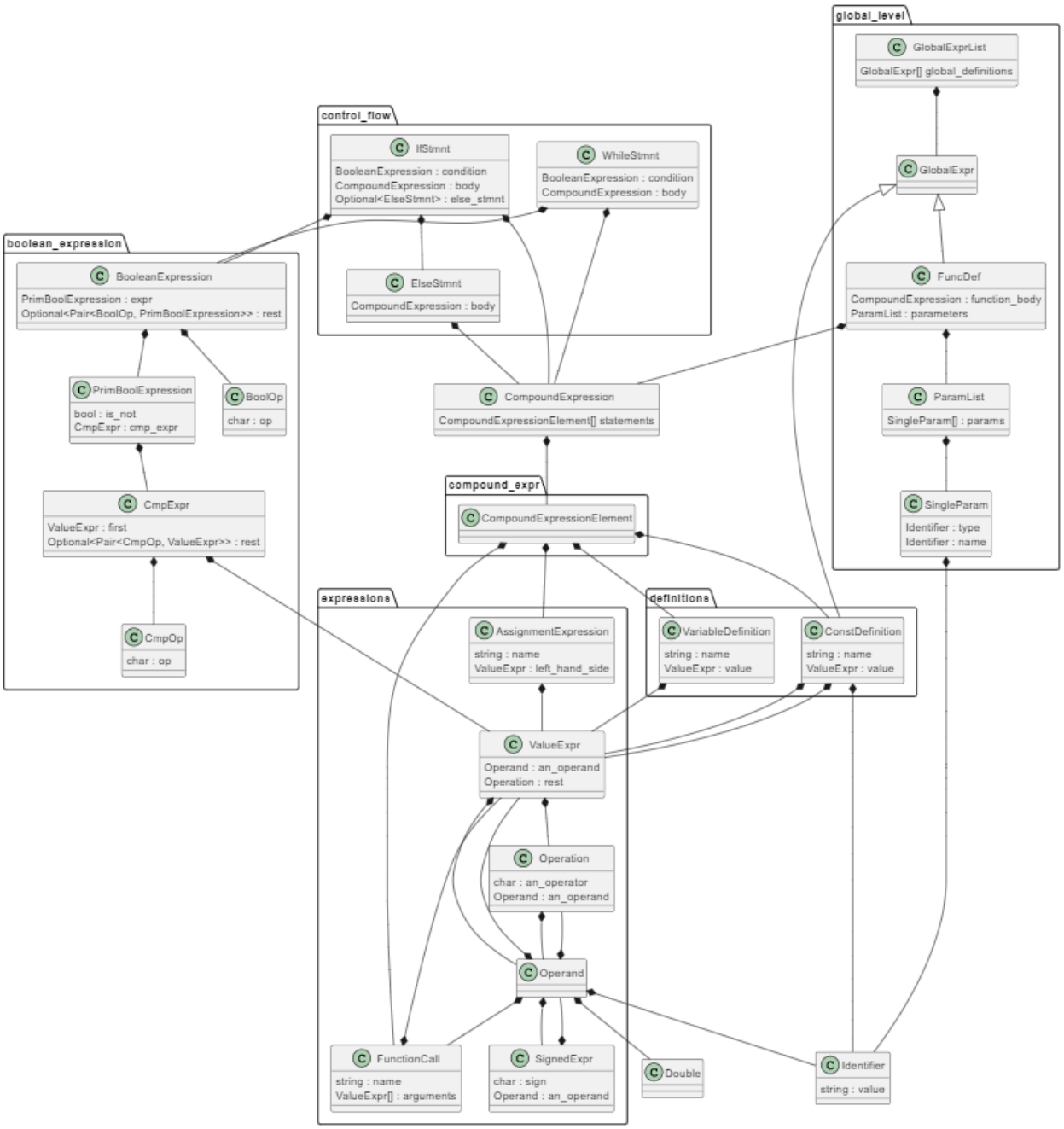

Рисунок 2.6 – Діаграма АСТ

2.5 Подальша обробка АСТ

Перед генерацією машинного коду необхідно зробити ще декілька перетворень над АСТ. Поточна модель не несе ніякої інформації щодо типів виразів, коректності виразів присвоювання та можливість виконання операцій над змінними в залежності від їх типів. Таким чином на даному етапі

необхідно реалізувати обхід дерева з подальшим обчисленням необхідної інформації з метою побудувати нове дерево, яке буде використано для генерації машинного коду. Нам необхідно створити декілька різноманітних дескрипторів для типів, які будуть в подальшому додані до AST.

Почнемо з функцій. Будь яка сігнатура функції складається з ім’я, списку аргументів та типом, який вона повертає. З огляду на це, треба створити декілька вбудованих типів. Це будуть *void, real* та *string,* де *real* – це 8 байтне число з плаваючою точкою. Дескриптор функції буде зберігати сігнатуру та тіло функції.

Аналогічним чином поступимо зі змінними та константами. При проході AST під час об’явлення змінних та констант алгоритм буде рекурсивно обходити вираз з присвоюванням та визначати тип.

На цьому етапі можемо зустріти перевизначення змінних. Існує декілька способів обробки таких ситуацій. Різні мови програмування вибирають різні та іноді діаметрально протилежні стратегії. Так, наприклад, мова програмування Rust дозволяє перевизначати змінні у рамках однієї функції безліч разів. При цьому старі змінні продовжують «жити» у пам’яті доки не вийдемо з функції. Мова програмування C# йде протилежним шляхом – вона взагалі не дозволяє перевизначення змінних у дочірніх виразах. Щось середнє відбувається у мовах C та C++. Там змінні можна перевизначати тільки у дочерніх зонах видимості, а ні як не в поточних.

Стратегія мов C та C++ – найлегша з точки зору реалізації компілятора. Подібний механізм можливо реалізувати за допомогою деревовидної структури ScopedNameResolver (рис. 2.7), де кожен потомок буде мати вказівник на батьківський клас. Структура буде зберігати дані про ідентифікатор.

При запросі на пошук даних про це ім’я структура спочатку буде шукати дані в себе, у разі помилки буде звертатися до батьківського класу (рис. 2.8).

Механізм роботи з класом ScopedNameResolver описується на рисунку 2.9.

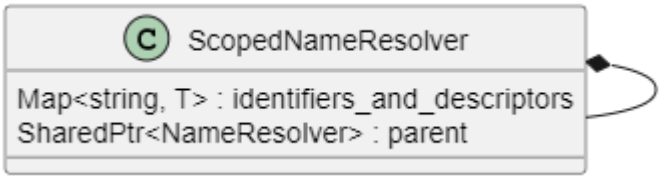

Рисунок 2.7 – Клас для зберігання ідентифікаторів

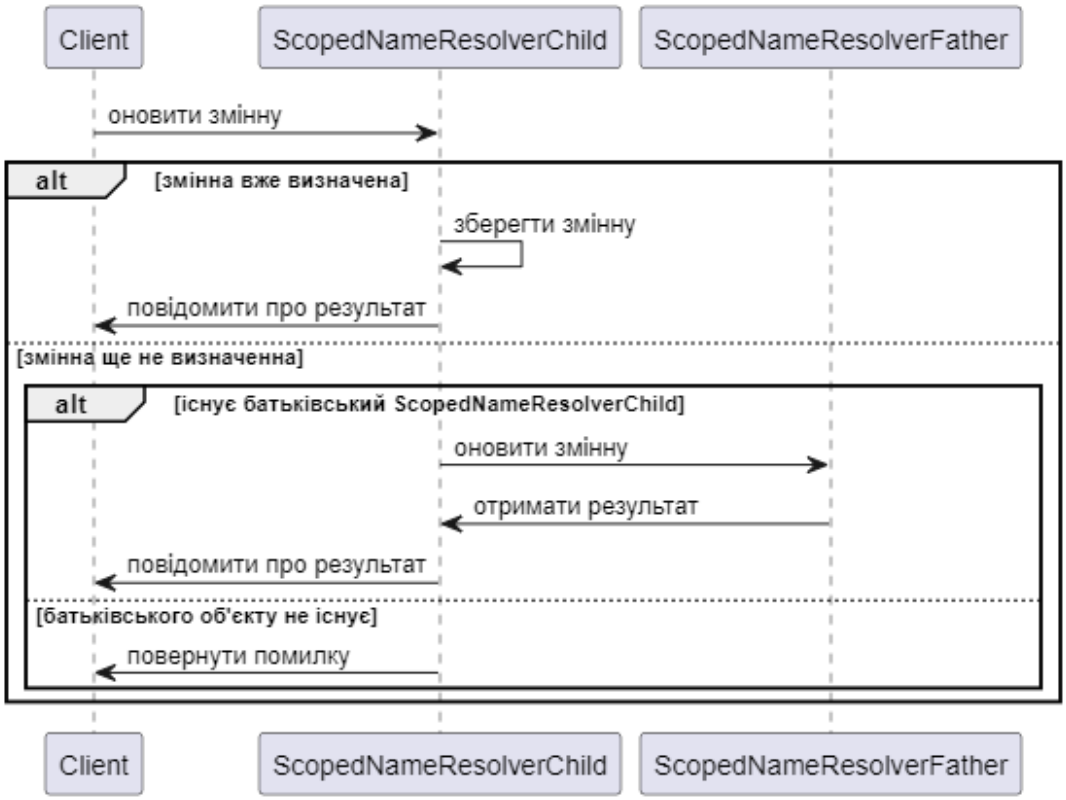

Рисунок 2.8 – Процес оновлення нової змінної

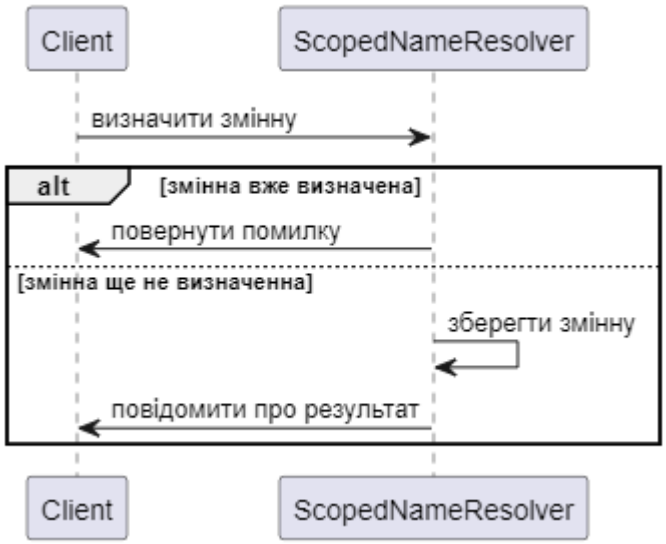

Рисунок 2.9 – Процес збереження нової змінної

Аналогічним чином можливо реалізувати механізм перевизначення констант та, за потреби, функцій. Слід зазначити, що в рамках даної роботи механізм перевизначення функцій не буде реалізований, адже він не визначений у вимогах.

3 КОМП'ЮТЕРНА МОДЕЛЬ КОМПІЛЯТОРА

3.1 Обґрунтування вибору середовища програмної реалізації

У рамках кваліфікаційної роботи був розроблен компілятор для мови програмування. Для реалізації була обрана мова C++23. Цей вибір зумовлен великою кількістю інструментів та бібліотек для даної мови програмування.

Основними бібліотеками під час розробки застосунка стали Boost.Spirit.X3 та LLVM. Обидві бібліотеки мають величезний розмір та свою нюанси для налаштування. Для отримання найбільш гнучкої конфігурації проекту був використаний застосунок CMake. CMake дозволяє генерувати різноманітні сценарії зборки для різних платформ. На відміну від Solution у Visual Studio, CMake може бути використаний для будь-яких платформ. Це може значно спростити процес портування застосунка з однієї платформи на іншу.

Разом з CMake був використаний пакетний менеджер Vcpkg, який розробляється компанією Microsoft. Пакетні менеджери дозволяють швидко керувати різноманітними бібліотеками та легко підключати їх до інсуючих проектів. Для мови програмування C++ та інструмента CMake існують два основних пакетних менеджера. Це Conan та CMake. Кожен з них має свої переваги та недоліки, проте обидва є кросплатформеними. Незважаючи на те, що Conan більш гнучкіший, Vcpkg значно легший у налаштуванні. Якоїсь кардинальної різниці у рамках даної роботи між ними не має.

Для тестування декілька крупних бібліотек та інструментів. Перед динамічним тестуванням слід проводити статичний аналіз. Для статичного аналізу інсує доволі багато інструментів, таких як Coverity, Sonar, ClangAnalyzer, тощо. З них повністю безкоштовним є ClangAnalyzer. Він використовувався у якості плагіну для IDE під час розробки. У якості альтернативи йому міг би виступати вбудований аналізатор для VisualStudio.

Для .юніт тестування використовувався фреймворк Catch2.

Від фазінгу було вирішено відмовитися, адже до застосунка немає ніяких вимог з боку безпеки. Тим не менш, має сенс згадати основні рішення для фазінгу в рамках мови C++. Основні бібліотеки – це AmericanFuzzyLop, FuzzTest та libFuzzer.

3.2 Програмна реалізація парсера

Для написання парсера була використана бібліотека Boost.Spirit.X3.

Одна з проблем використання цієї бібліотеки полягає в тому, що вона базується на використанні шаблонів, макросів та різноманітних «трюків» для генерації парсерів. Це, в свою чергу, створює певні проблеми під час розробки.

У разі будь-якої помилки під час використання можна отримати безліч помилок як з боку лінковщика, так і з боку компілятора. Також Boost.Spirit використовує Boost.Fusion для роботи з користувацькими структурами даних. В цілому, ця бібліотека може вимагати, щоб всі елементи парсеру були описані в одній одиниці трансляції. Це призводе до певних особливостей у структурі проекту структури проєкту [6].

3.2.1 Структура AST

Найбільш комплексний елемент в структурі AST – це Operand. Operand може бути як звичайним літералом, так і ідентифікатором або, навіть, складним виразом. При цьому вираз може складатися з декількох операндів. Така складна рекурсія вирішується за допомогою попереднього об’явлення типів.

Лістинг 3.1 Структура Operand:

```
struct expression;
struct signed_expr;
struct assignment_expression;
struct function_call;

struct operand : boost::spirit::x3::variant<
    double,
    std::string,
    boost::spirit::x3::forward_ast<expression>,
    boost::spirit::x3::forward_ast<signed_expr>,
    boost::spirit::x3::forward_ast<function_call>
>
{
  using base_type::base_type;
  using base_type::operator=;
};
```

Допоміжна структура *boost::spirit::x3::forward_as* необхідна для коректної компіляції об’єкту класу *variant.* Директиви *using* вимагаються бібліотекою для перетворення типів. Структура expression представляє послідовність операцій над операндом.

Лістинг 3.2 Структура expression:

```
struct expression
{
  operand              first;
  std::vector<operation> rest;
};
```

Структура operation представляє операцію та операнд. Ця структура буде використовуватися лише як частина expression.

Лістинг 3.3 Структура operation:

```
struct operation
{
 char    an_operator;
 operand an_operand;
};
```

Також, в окремій категорії маємо signed_expr та function_call. Незважаючи на зовнішню схожість signed_expr та operation, вони мають різне семантичне значення та signed_expr, на відміну від operation, може використовуватися самостійно, а не як частина expression.

Лістинг 3.4 Структура signed_expr:

```
struct signed_expr
{
 char    sign;
 operand an_operand;
};
```

Структура function_call описуватиме виклик функції.

Лістинг 3.5 Структура function_call:

```
struct function_call
{
 std::string          name;
 std::vector<expression> arguments;
};
```

Після створення структур необхідно надати можливість бібліотеці працювати з користувацькими типами. Це реалізується за допомогою макроса BOOST_FUSION_ADAPT_STRUCT.

Лістинг 3.6 Використання макроса зі структурою function_call:

```
BOOST_FUSION_ADAPT_STRUCT(
    empty_compiler::ast::function_call,
    (std::string, name)
    (std::vector<empty_compiler::ast::expression>, arguments)
)
```

Аналогічним чином створюються інші елементи АСТ. Архітектура АСТ була детально описана в минулих розділах даної роботи.

3.2.2 Написання правил для парсингу

В рамках фреймворку Boost.Spirit.X3 граматика описується цілком і повністю за допомогою правил. Кожне правило може мати або не мати певний атрибут. Цей атрибут є тип змінної, яка буде створена у разі успішного парсингу.

Для того, щоб була пожливість використовувати правила, їх треба об’явити. Це робиться наступним чином.

Лістинг 3.7 Об’явлення правила для парсингу виразів:

```
struct expression_class;
using expression_type = boost::spirit::x3::rule<expression_class, ast::expression>;
BOOST_SPIRIT_DECLARE(expression_type);
```

Слід звернути увагу на те, яким чином називаються expression_class, expression_type, тощо. Це пов’язано з конвенцією найменування в бібліотеці. Ця конвенція важлива під час використання макросів, тому її необхідно дотримуватися.

Після об’явлення правила необхідно зробити його визначення. Це робиться наступним чином.

Лістинг 3.8 Визначення правила для парсингу виразів:

```
expression_type const expression = "expression";

auto const expression_def = additive_expression;
auto const additive_expression_def =
    multiplicative_expression >> *(
        (char_('+') > multiplicative_expression)
      | (char_('-') > multiplicative_expression)
    );
auto const multiplicative_expression_def =
    unary_expression >> *(
        (char_('*') > unary_expression)
      | (char_('/') > unary_expression)
    );
auto const unary_expression_def =
    primary_expression
    | (char_('+') > primary_expression)
    | (char_('-') > primary_expression)
;
```

auto const primary_expression_def = double_ | ('(' > expression > ')') | function_call | identifier;

auto const function_call_def = get_identifier_parser() >> ('(' > (expression % ',') > ')');

В цілому, реалізація правил аналогічна правилам у формі PEG.

Усі інші правила пишуться подібним чином.

3.2.3 Подальша обробка AST

Для обробки багатьох елементів AST, зокрема операнду, краще реалізувати шаблон «Відвідувач». Це зробити дуже легко, достатньо перевантажити оператори виклику функція для всіх необхідний типів. Наприклад.

Лістинг 3.9 Приклад реалізації відвідувача за допомогою лямбда-функцій:

```
template<class... Ts>
struct overloaded : Ts... { using Ts::operator()...; };
template<class... Ts>
overloaded(Ts...) -> overloaded<Ts...>;
auto const my_visitor = overloaded{
    [](ast::signed_expr const &expr){},
    [](ast::expression const &expr){},
    [](ast::function_call const &expr){},
    [](std::string const &expr){},
    [](double const &expr){},
   };
```

boost::apply_visitor(my_visitor, operand);

Таким чином буде проходити подальша генерація коду.

3.3 Генерація машинного коду

Для генерації машинного коду буде використовуватися LLVM.

LLVM використовує IR (intermediate representation) для платформонезалежного представлення коду. Фактично IR – це щось середнє між асемблером та мовою С. На відміну від асемблера конкретної платформи, IR може бути скомпільоване на будь-яку платформу. До того, інструкція в IR може мати багато варіацій, щоб у розробника мови програмування не було обмежень функціоналу. Під час компіляції IR в асемблер цільової платформи LLVM може перетворити та спростити певні інструкції для оптимізації під конкретну платформу. Слід зазначити, що існують бекенди LLVM для генерації коду під різноманітні GPU [4, 5, 11].

Для роботи з LLVM нам знадобляться об’єкти типів LLVMContext, IRBuilder та Module. LLVMContext зберігає певні внутрішні структури, які необхідні для роботи інший об’єктів. IRBuilder потрібен для генерації IR коду. Module зберігає в собі функції, глобальні змінні, тощо.

Функція створюється за допомогою Function::Create.

Лістинг 3.10 Приклад створення функції:

```
FunctionType* ft = FunctionType::get(Type::getDoubleTy(*context), false);
Function* f = Function::Create(ft, Function::ExternalLinkage, name, module_ptr);
```

Для додавання функціоналу до функції необхідно створити базовий блок. Базовий блок – це частина функції, яка не має розгалужень та має одну точки входу та виходу. Базовий блок створюється наступним чином.

Лістинг 3.11 Приклад створення базового блоку:

```
BasicBlock* blk = BasicBlock::Create(*context, block_name);
```

Подальша робота з LLVM відбувається аналогічним чином.

LLVM надає інструменти для створення компіляторів не тільки для реальних машин, а і для віртуальних. Згенерований IR може бути скомпільовний та виконаний під час роботи програми. Для цього використовуватимуться об’єкти декількох типів. ExecutionSession – представляє працюючу програму, JITDylib – представляє динамічну JIT бібліотеку, JITTargetMachineBuilder – описує цільову машину, MangleAndInterner – проводить зміну імен у рамках ExecutionSession, DataLayout – клас для отримання даних про структуру динних на цільовій машині, наприклад порядок байт, тощо.

3.4 Підтримка нейронних мереж та спосіб володіння ресурсами

Головна мета нашою мови програмування – надати користувачеві зручний інструмент для побудови нейронних мереж. Мова програмування Python, хоча і є дуже зручним інструментом, має великий недолік – її швидкість виконання в деяких випадках може бути незадовільною. Альтернативою є мова C++, але її комплексність створює проблеми при використанні на практиці.

Робота з нейронними мережами буде виконуватися за допомогою вбудованих функція.

Для реалізації функціоналу пропонується бібліотека Pytorch. Ця бібліотека написана на мові C++ та має зручне API [39]. Для реалізації підтримки нейронних мереж залишиться тільки написати обгортку для цієї

бібліотеки. Використання PyTorch також надасть функціонал для зберігання натренованої моделі.

Для створення модуля PyTorch на мові C++ треба створити класс та наслідувати його від класу torch::nn::Module.

Лістинг 3.12 Створення модуля PyTorch на мові C++:

```
class MyModule : public torch::nn::Module {};
auto module = MyModule{};
```

Нехай треба, щоб модель мала певні параметри. Тоді це можна зробити наступним чином.

Лістинг 3.13 Створення та реєстрація параметрів у модулі:

```
class MyModule : public torch::nn::Module {
    MyModule(int N, int M) {
        W = register_parameter(“W”, torch::randn({N, M});
        b = register_parameter(“b”, torch::randn(M));
    }

    torch::Tensor W, b;
};
auto module = MyModule{3, 4};
```

Для підтримки прямого поширення клас може реалізувати метод forward.

Лістинг 3.14 Приклад реалізації прямого поширення:

```
class MyModule : public torch::nn::Module {
    torch::Tensor W, b;
    MyModule(int N, int M) {
```

```
        W = register_parameter(“W”, torch::randn({N, M});
        b = register_parameter(“b”, torch::randn(M));
    }
    torch::Tensor forward(torch::Tensor input) {
        return torch::addmm(b, input, W);
    }
};
```

Для того, щоб виконати нейронну мережу достатньо викликати метод forward.

Лістинг 3.15 Приклад виконання мережі:

```
auto net = MYModule(3,4);
auto&& output = net.forward(torch::ones({3, 4}));
```

Для використання PyTorch у нашій мові програмування необхідно зробити обгортку з інтерфейсом для мови C. Справа в тому, що імена у мові C++ перетворюються під час компіляції. Також слід мати на увазі, що наша мова програмування не підтримуватиме класів із мови C++, тому не зможемо реалізувати пряму підтримку конструкторів, деструкторів, тощо. Створення та взаємодія з PyTorch буде відбуватися у процедурному стилі. Також слід мати на увазі процес управління ресурсами. Фактично розробляєма мова повинна використовуватися як допоміжний інструмент під час розробки, або як інструмент для написання невеликих скриптів для швидкого відлагодження та експериментування. З огляду на це, при використанні мови користувач не повинен турбуватися про управління ресурсами PyTorch. Таким чином, для володіння ресурсами перед виконанням першої інструкції програмного модуля користувача буде створюватися таблиця з ресурсами, яка і буде відповідати за зберігання та очищення зайнятої пам’яті. У якості альтернативи

такому підходу можливо реалізувати один із наступних механізмів: RAII або рахування посилань.

RAII використовується у мовах C++ та Rust для володіння пам’яттю. У даній моделі кожен об’єкт має час життя та механізм звільнення ресурсів. На мові C++ це деструктори, на мові Rust це трейт drop.

Рахування посилань використовуються у мові Python, хоча і Rust, і C++ мають спеціальні класи-обгортки для реалізації даного механізму. Це механізм полягає у тому, що кожен об’єкт матиме лічильник власного використання, яких змінюватиметься при його копіюванні. При цьому об’єкт звільнятиме свої ресурси тільки коли лічильник досягне нуля.

Так як у якості допоміжною бібліотека використовується PyTorch, який написаний на мові C++ та покладається на використання shared_ptr, можливе використання обох підходів наступним чином. Буде створюватися таблиця ресурсів, описана раніше, але тепер вона зберігатиме лічильник. Реалізуємо функції для збільшення та зменшення лічильників об’єкта. Потім, під час генерації об’єктного коду, виклики цих функцій будуть вставлятися автоматично. Для того, щоб таблиця підтримувала будь-який тип мови C++ та була єдиною для всіх треба створити поліморфний клас table_resource, який використовуватиме «type punning» та «pimpl» – ідіоми мови C++. Ідіома «type punning» потрібна для додавання поліморфізму не поліморфним об’єктам, «pimpl» – для використання ще не створених типів.

Лістинг 3.16 Клас table_resource:

```
template <class T>
class table_resource;

class any_table_resource {
public:
    template <class T, class… Args>
    any_table_resource(Args&& … args);
```

```
    virtual table_resource() noexcept = default;

private:
    std::unique_ptr<any_table_resource> pimpl_;
};

template <class T, class… Args>
    any_table_resource ::any_table_resource(Args&& … args)
    : piml_{
        std::make_unique<table_resource_impl<T>>(
            std::forward<Args>(args)…
    )} {}


template <class T>
class table_resource : public any_table_resource{
    public:
        template < class …Args>
        table_resource_impl(Args&&… args)
        : resource_(std::forward<U>(args) … ) {}

        constexpr
        T& get() noexcept
        {
            return resource_;
        }

        T const& get() const noexcept
        {
            return resource_;
```

```
        }

    private:
        T resource_;
};
```

Як видно із коду, при створенні екземпляра, об’єкт класу table_resource створюватиме свого шаблонного нащадка, який зберігатиме ресурс. Так як типи C++ реалізують механізм RAII, цього фрагменту коду буде достатньо, щоб звільнити зайняті ресурси під час знищення table_resource.

Таблиця матиме наступний вигляд.

Лістинг 3.17 Клас rc_table:

```
class tc_table {
    std::unordered_map<std::unique_ptr<table_resource>, int> resources_;

    public:
        template <class T>
        table_resource<T>*
        add_resource(std::unique_ptr<table_resource<T>> r)
        {
            auto ptr = r.get();
            any_table_resource* ar = r.release();
            add_resource(ar);
            return ptr;
        }

        any_table_resource*
        add_resource(std::unique_ptr<any_table_resouce> r)
```

```
    {
        auto ptr = r.get();
    #ifdef _DEBUG
        if (try_increment_count(ptr))
            throw std::runtime_error("Invalid object ownership);
    #endif

        resources_.emplace_back(std::move(r), 1);
        return ptr;
    }

    bool decrement_count(any_table_resource* r)
    {
        auto item = resources_.find(r);
        if (item == resources_.end()) return false;
        item->second--;
        if (item->second == 0) {
            resources.erase(item);
            return true;
        }
        return false;
    }

    void increment_count(any_table_resource* r)
    {
        if (!try_increment_count(r))
            throw std::runtime_error("Unregistered resource");
    }

private:
```

```
        void try_increment_count(any_table_resource* r)
        {
            auto item = resources_.find(r);
            if (item == resources_.end()) {
                return false;
            }
            item->second++;
            return true;
        }

};
```

Слід зазначити, що дана реалізація не працюватиме у разі багатопотокового звернення до таблиці. Хоча на даному етапі розробляєма мова програмування не підтримує багатопотоковість, її підтримка може бути додана у майбутньому. Через це треба уважно продумати механізм роботи таблиці та її інваріанти в багатопотоковому середовищі, адже незручний може викликати багато проблем у майбутньому.

По-перше, на даному етапі таблиця повертає вказівник та table_resource, який ні як не захищений від багатопотокового доступа. Це не повинно викликати проблеми, адже зараз єдина задача table_resource – надати користувачеві змогу отримати до ресурсу доступ. Вже на рівні користувача розробляємого інтерфейсу повинна визначатися семантика багатопоточного доступу, адже не кожен об’єкт потребує якісь додаткові механізми синхронізації.

По-друге, задача збільшувати та зменшувати лічильник лягатиме на користувача. Такий підхід додає ризик виникнення багів через людський фактор, але він необхідний, адже таке керування лічильником потрібне для реалізації механізмів роботи з пам’яттю у нашої мови. У разі потреби можна

буде додати клас-обгортку для вказівників на table_resource для більш зручного використання.

Для правильної реалізації механізмів синхронізації потрібно розуміти характер сценаріїв доступу до ресурсів. Наприклад, зрозуміти що буде виконуватися частіше: створення нового та видалення ресурсу, або створення копії. Це пов’язано з тим, щоб зрозуміти, чи треба створювати окремі м’ютекси для читання та запису, або буде достатньо одного м’ютекса для обох операцій.

Також треба проаналізувати, чи є можливість замінити роботу м’ютексів на використання атомарних змінних. Якщо використання атомарних змінних можливе, як це вплине на роботу застосунка. Справа в тому, що використання атомарних змін не завжди є швидкішим за використання м’ютексів. Швидкість роботи атомарних змінних дуже сильно залежить від моделі пам’яті пристрою та операції доступу: читання та запис. Припускається, що операція запису при використанні нашої таблиці буде найчастішою. Кожен запис атомарної змінної повинен викликати синхронізацію кешів процесора. Слід розуміти ця операція доволі повільна на деяких платформах, хоча на платформі x86 на це можна не звертати уваги, адже атомарні операції читання та запису на x86 не дуже відрізняються по швидкості.

Також слід зрозуміти, де треба реалізовувати мехнізм синхронізації – у межах таблиці ресурсів, чи поза ними. Щоб дати відповідь на це питання, слід навести наступний приклад.

Лістинг 3.18 Створення та знищення мереж:

```
func raii_demonstration() -> void{
    let nn1 = Net()
    let nn2 = Net()
}
```

При виході з функції буде два звернення до таблиці, тому робити окремі виклики до м’ютекса не добре. З огляду на це, приходе думка винести виклик механізму синхронізації за межі класу rc_table, проте є інше рішення.

Для ідіоматійного інтерфейсу використання слід додати таблиці метод для групового очищення ресурсів. Є декілька варіантів реалізації: за допомогою масивів, за допомогою змінною кількості аргументів, та за допомогою шаблонів. Порівняємо всі методи.

Лістинг 3.19 Знищення ресурсів за допомогою масивів:

```
void batch_cleanup(std::span<table_resource const*> rs)
{
    auto lg = std::lock_guard{table_mutex_};
    for (auto&& res : rs)
        try_free_resource(res);
    }
}
```

Для використання такого інтерфейсу нам доведеться виділяти де-небудь масив з вказівниками на ресурси. Ми можемо робити це або на стеку, або динамічно в кучі. Динамічне виділення пам’яті може бути дуже повільним, адже в найгіршому випадку вона може бути значно повільніша блокування м’ютекса. Варіант з алокацію пам’яті на стеці занадто швидший, адже виконується завдяки зміни показника на кінець стека, хоча він теоретично може завдати проблеми при використанні на деяких платформах.

Лістинг 3.20 Знищення ресурсів за допомогою шаблонів:

```
template  <class Current, class… Rest>
void batch_cleanup(Current c, Rest&& … rest)
{
    auto lg = std::lock_guard{table_mutex_};
```

```
    batch_cleanup_impl(c, rest…);
}

void batch_cleanup_impl()
{ }


template <class Current, class… Rest>
void batch_cleanup_impl(Current c, Rest&& … rest)
{

    static_assert(std::is_same_v<
          std::decay_t<Current>, any_table_resource*>
          );
  try_free_resource(c);
  batch_cleanup_impl(rest…);
}
```

Даний спосіб найліпше підходить для використання у коді на мові C++. Проблема полягає в тому, що реалізувати виклики цих функцій для розробляємої мови дуже важко і подібна реалізація може бути залежною від платформи та компілятора. Якщо буде вирішено дотримуватися цього підходу, то доведеться під час компіляції інстаніювати ці функції для кожного очищення ресурсів у нашому коді. LLVM та Clang можуть допомогти вирішити цю проблему, проте комплексність компілятора розробляємої мови значно зросте.

Лістинг 3.21 Знищення ресурсів за функції зі змінною кількістю аргументів:

```
void batch_cleanup(std::size_t count, …)
```

```
{
    If (count == 0) return;
    auto lg = std::lock_guard{table_mutex_};
    va_list args;
    va_start(args, count);
    for (auto i : std::views::iota(0, count)) {
        auto res = va_arg(args, any_table_resource*);
        try_free_resource(res);
    }
    va_end(args);
}
```

Підтримку подібного інтерфейсу буде доволі легко реалізувати в рамках розробляємої мови. Головний мінус такого підходу – вища можливість виникнення бага при підтримці коду.

Враховуючи зазначене, варіант з функцією зі змінною кількістю аргументів є найкращим. Наведемо повний код rc_table.

Лістинг 3.22 Фінальний варіант rc_table:

```
class tc_table {
    std::mutex table_mutex_;
    std::unordered_map<std::unique_ptr<table_resource>, int> resources_;

    public:
        template <class T>
        table_resource<T>*
        add_resource(std::unique_ptr<table_resource<T>> r)
        {
            auto ptr = r.get();
            any_table_resource* ar = r.release();
            add_resource(ar);
```

```
        return ptr;
}

any_table_resource*
add_resource(std::unique_ptr<any_table_resouce> r)
{
        auto ptr = r.get();
#ifdef _DEBUG
        if (try_increment_count(ptr))
                throw std::runtime_error("Invalid object ownership);
#endif

        resources_.emplace_back(std::move(r), 1);
        return ptr;
}

bool decrement_count(any_table_resource* r)
{
        auto item = resources_.find(r);
        if (item == resources_.end()) return false;
        item->second--;
        if (item->second == 0) {
                resources.erase(item);
                return true;
        }
        return false;
}

void increment_count(any_table_resource* r)
{
```

```
            if (!try_increment_count(r))
                throw std::runtime_error("Unregistered resource");
        }

        void batch_cleanup(std::size_t count, …)
        {
            if (count == 0) return;
            auto lg = std::lock_guard{table_mutex_};
            va_list args;
        va_start(args, count);
        for (auto i : std::views::iota(0, count)) {
            auto res = va_arg(args, any_table_resource*);
            decrement_count (res);
        }
        va_end(args);
    }


     private:
        void try_increment_count(any_table_resource* r)
        {
            auto item = resources_.find(r);
            if (item == resources_.end()) {
                return false;
            }
            item->second++;
            return true;
        }

};
```

Після генерації машинного коду, об’єктний файл написаний на розробляємій мові буде автоматично лінкуватися з допоміжною бібліотекою, яка і реалізуватиме взаємодію з PyTorch та механізми володіння пам’яттю.

3.5 Тестування розробленої моделі

Для тестування був використаний фреймворк Catch2. Він має гарну підтримку як зі сторони розробників, так і зі сторони користувачів. Більшість IDE має змогу працювати з ним.

Тест на Catch2 створюється наступним чином.

Лістинг 3.23 Приклад тесту Catch2:

```
TEST_CASE("Benchmark Fibonacci", "[!benchmark]") {
   REQUIRE(fibonacci(5) == 5);

   REQUIRE(fibonacci(20) == 6'765);
   BENCHMARK("fibonacci 20") {
      return fibonacci(20);
   };

   REQUIRE(fibonacci(25) == 75'025);
   BENCHMARK("fibonacci 25") {
      return fibonacci(25);
   };
}
```

Як видно з прикладу, Catch2 також дозволяє тестувати швидкість роботи застосунка.

Для тестування розробленого застосунку необхідно винести весь функціонал в окрему бібліотеку. Клієнтами цієї бібліотеку будуть тестовий клієнт та компілятор.

Таким чином на необхідно створити одну бібліотеку та два виконуваних файлів. За допомогою CMake це робиться наступним чином [40].

Лістинг 3.24 Створення цілей для CMake:

```
add_executable(empty_compiler main.cpp) # наш компілятор
add_executable(test_runner tests.cpp) # тестовий клієнт
add_library(empty_compiler_lib …файли з кодом…) # бібліотека
```

Необхідно вказати шляхи для підключення заголовків для бібліотеки та об’явити це публічним, щоб ця залежність стосувалася і клієнтів.

Лістинг 3.25 Додавання шляхів для підключення заголовків:

```
target_include_directories(empty_compiler_lib PUBLIC “./include” PRIVATE “${SOME_PRIVATE_HEADERS}”)
```

Catch2 реалізує свій власний main, тому нам не потрібно робити це самотужки.

Лінкування бібліотеки відбувається наступним чином.

Лістинг 3.26 Лінкування бібліотеки:

```
target_link_libraries(empty_compiler PRIVATE empty_compiler_lib)
target_link_libraries(test_runner PRIVATE empty_compiler_lib Catch2::Catch2WithMain)
```

ВИСНОВКИ

У рамках кваліфікаційної роботи був розроблений і реалізований компілятор для нової мови програмування.

Парсер, який був створений за допомогою Boost.Spirit.X3 працює з достатньою швидкістю. LLVM в змозі провести множину оптимізацій над кодом.

Завдяки заздалегідь сформованій формальній моделі мови вдалося розробити модульну та гнучку архітектуру компілятора. Правильне використання інструментів розробки дозволило побудувати проєкт, який можливо легко перенести на інші платформи.

LLVM у якості основи для компілятора дозволила створити можливість для підтримки багатьох платформ у якості цільової платформи для компіляції, таких як мобільні пристрої або веббраузери з підтримкою WASM.

У поточній реалізації мова використовуватиме PyTorch у якості бекенда для роботи з нейронними мережами, проте ця залежність ніяк не повинна бути спостерігаєма користувачем. У разі необхідності можливо замінити PyTorch на будь-який інший фреймворк або створити свою альтернативу.

До недоліків створеної системи можна віднести доволі високі вимоги до обладнання для розробки через залежність від дуже великих бібліотек. Ти не менш, це ці залежності не повинні дуже впливати на кінцевий продукт. Також поточно реалізація мови передбачає лише процедурну парадигму програмування. Проте, враховуючи призначення мови, це не повинно стати проблемою для практичного використання.

Завдяки Boost.Spirit.X3 є можливість для швидкого подальшого розширення синтиксиса та можливостей мови.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Williams, A. (2012). C++ concurrency in action. London.
2. Parsing expression grammar. URL: https://en.wikipedia.org/wiki/Parsing_expression_grammar (дата звернення 05.20.2023).
3. Regular Expression Matching Can Be Simple And Fast. URL: https://swtch.com/~rsc/regexp/regexp1.html (дата звернення 20.05.2023).
4. LLVM Tutorial. URL: https://llvm.org/docs/tutorial/ (дата звернення 20.05.2023).
5. The Architecture of Open Source Applications (Volume 1) LLVM. URL: https://aosabook.org/en/v1/llvm.html (дата звернення 20.05.2023).
6. Spirit X3 3.10. URL: https://www.boost.org/doc/libs/master/libs/spirit/doc/x3/html/index.html (дата звернення 20.05.2023).
7. Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2020). Compilers: principles, techniques and tools.
8. Meyers, S. (2014). Effective modern C++: 42 specific ways to improve your use of C++ 11 and C++ 14. " O'Reilly Media, Inc.".
9. Stroustrup, B., & Sutter, H. (2018). C++ core guidelines. Web. Last accessed February.
10. Alexandrescu, A. (2001). Modern C++ design: generic programming and design patterns applied. Addison-Wesley.
11. Lattner, C., & Adve, V. (2004, March). LLVM: A compilation framework for lifelong program analysis & transformation. In International symposium on code generation and optimization, 2004. CGO 2004. (pp. 75-86). IEEE.
12. Meyers, S. (2005). Effective C++: 55 specific ways to improve your programs and designs. Pearson Education.
13. Lattner, C. (2008, May). LLVM and Clang: Next generation compiler technology. In The BSD conference (Vol. 5, pp. 1-20).

14. Hoare, C. A. (1973). Hints on programming language design. STANFORD UNIV CA DEPT OF COMPUTER SCIENCE.

15. Lakos, J. (1996). Large-scale C++ software design. Reading, MA, 173, 217-271.

16. Ford, B. (2004, January). Parsing expression grammars: a recognition-based syntactic foundation. In Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages (pp. 111-122).

17. Fullerton, R. R., & McWatters, C. S. (2001). The production performance benefits from JIT implementation. Journal of operations management, 19(1), 81-96.

18. Knuth, D. E., Morris, Jr, J. H., & Pratt, V. R. (1977). Fast pattern matching in strings. SIAM journal on computing, 6(2), 323-350.

19. Aho, A. V., Johnson, S. C., & Ullman, J. D. (1976, January). Code generation for expressions with common subexpressions. In Proceedings of the 3rd ACM SIGACT-SIGPLAN symposium on Principles on programming languages (pp. 19-31).

20. Aho, A. V., & Johnson, S. C. (1976). Optimal code generation for expression trees. Journal of the ACM (JACM), 23(3), 488-501.

21. Aho, A. V., Ganapathi, M., & Tjiang, S. W. (1989). Code generation using tree matching and dynamic programming. ACM Transactions on Programming Languages and Systems (TOPLAS), 11(4), 491-516.

22. Emanuelsson, P., & Nilsson, U. (2008). A comparative study of industrial static analysis tools. Electronic notes in theoretical computer science, 217, 5-21.

23. Waite, W. M., & Goos, G. (2012). Compiler construction. Springer Science & Business Media.

24. Cooper, K. D., & Torczon, L. (2011). Engineering a compiler. Elsevier.

25. Appel, A. W. (2004). Modern compiler implementation in C. Cambridge university press.

26. Powers, B., Tench, D., Berger, E. D., & McGregor, A. (2019, June). Mesh: Compacting memory management for C/C++ applications. In Proceedings of the

40th ACM SIGPLAN Conference on Programming Language Design and Implementation (pp. 333-346).

27. Johnson, S. C. (1975). Yacc: Yet another compiler-compiler (Vol. 32). Murray Hill, NJ: Bell Laboratories.

28. Fog, A. (2022). Optimizing software in C++. URL: https://www.agner.org/optimize/optimizing_cpp.pdf (дата звернення 20.05.2023).

29. Fog, A. (2022). Calling conventions for different C++ compilers and operating systems. URL: https://www.agner.org/optimize/calling_conventions.pdf (дата звернення 20.05.2023).

30. Fog, A. (2023). Optimizing subroutines in assembly language. URL: https://www.agner.org/optimize/optimizing_assembly.pdf (дата звернення 20.05.2023).

31. Fog, A. (2023). The microarchitecture of Intel, AMD and VIA CPUs. URL: https://www.agner.org/optimize/microarchitecture.pdf (дата звернення 20.05.2023).

32. Bilonoh, B., & Mashtalir, S. (2020, August). Parallel multi-head dot product attention for video summarization. In 2020 IEEE Third International Conference on Data Stream Mining & Processing (DSMP) (pp. 158-162). IEEE.

33. Nasu, Y. (2018). Efficiently updatable neural-network-based evaluation functions for computer shogi. The 28th World Computer Shogi Championship Appeal Document, 185.

34. Efficiently updatable neural network. URL: https://en.wikipedia.org/wiki/Efficiently_updatable_neural_network (дата звернення 20.05.2023).

35. NNUE Pytorch. URL: https://github.com/glinscott/nnue-pytorch/blob/master/docs/nnue.md (дата звернення 20.05.2023).

36. Stockfish 12. Blog. URL: https://stockfishchess.org/blog/2020/stockfish-12/ (дата звернення 20.05.2023).

37. Leela Chess Zero. URL: https://lczero.org/ (дата звернення 20.05.2023).

38. SYCL Overview. URL: https://www.khronos.org/sycl/ (дата звернення 20.05.2023).

39. PyTorch Documentation. URL: https://pytorch.org/docs/stable/index.html/ (дата звернення 20.05.2023).

40. CMake Documentation. URL: https://cmake.org/documentation/ (дата звернення 20.05.2023).