МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ
Факультет кібербезпеки та програмної інженерії
Кафедра інженерії програмного забезпечення

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач кафедри

Олексій Горський
“____”_____2023 р.

КВАЛІФІКАЦІЙНА РОБОТА
(ПОЯСНЮВАЛЬНА ЗАПИСКА)

ВИПУСКНИКА ОСВІТНЬОГО СТУПЕНЯ
МАГІСТРА

Тема: “Методика розробки компілятора на мові Rust”

Виконавець: Тарасов Ігор Валерійович

Керівник: к.т.н доцент Шибицька Наталія Миколаївна

Нормоконтролер: ст.викл Гололобов Дмитро Олександрович

Київ 2023

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет кібербезпеки, комп'ютерної та програмної інженерії

Кафедра інженерії програмного забезпечення

Освітній ступінь магістр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

Форма навчання денна

ЗАТВЕРДЖУЮ

Завідувач кафедри

Олексій Горський

"___" _____ 2023 р

ЗАВДАННЯ

на виконання кваліфікаційної роботи студента

Тарасова Ігоря Валерійовича

1. Тема кваліфікаційної роботи: «Методика розробки компілятора на мові Rust» затверджена наказом ректора від 29.09.2023 р. № 1994/ст.
    1. Термін виконання проекту: з 02.10.2022 р. по 31.12.2023 р.
    2. Вихідні дані до роботи : код компілятора на мові програмування Rust, тестові приклади.
2. Зміст пояснювальної записки:
    1. Огляд та порівняльний аналіз аналогів об'єкта досліджень
    2. Теоретичні та експериментальні дослідження
    3. Модель предметної області та програмна реалізація.

4. Перелік обов'язкових слайдів презентації:
    1. Актуальність створення компілятора на мові Rust.
    2. Огляд та порівняльний аналіз аналогів об’єкта досліджень.
    3. Теоретичні та експериментальні дослідження.
    4. Схема роботи програми.
    5. Демонстрація роботи програми.
    6. Демонстрація роботи модулів програми.
5. Календарний план-графік.

№ пор.	Завдання	Термін виконання	Відмітка про виконання
1.	Розробка та затвердження графіка роботи.	01.09.2023-20.09.2023	Виконано
2.	Підготовка та написання 1 розділу. Відсилка керівнику	21.09.2023-22.09.2023	Виконано
3.	Підготовка та написання 2 розділу. Відсилка керівнику	22.09.2023-10.10.2023	Виконано
4.	Підготовка та написання 3 розділу. Відсилка керівнику	11.10.2023-08.11.2023	Виконано
5.	Редагування та друк пояснювальної записки, графічного матеріалу Відсилка ПЗ для перевірки на плагіат одним файлом.	08.11.2023-07.12.2023	Виконано
6.	Проходження нормо-контролю, перепліт пояснювальної записки. Отримання відгуку керівника. Підготовка презентації та тексту доповіді.	07.12.2023-10.13.2023	Виконано
7.	Передзахист. При наявності цього	11.12.2023-	Виконано

№ пор.	Завдання	Термін виконання	Відмітка про виконання
	приймається рішення про допуск к захисту дипломного проекту перед Екзаменаційною комісією. Що оформлюється протоколом засідання кафедри	15.12.2023	
8.	Отримання рецензії.	16.12.2023-17.12.2023	Виконано
9.	Здати секретарю ДЕК: ПЗ, ГМ, CD-R з електронними версіями ПЗ, ГМ, презентацію, відгук керівника, рецензію, довідку про успішність, 2 папки, 2 конверта)	18.12.2023-24.12.2023	Виконано
10.	Захист кваліфікаційної роботи перед ЕК	25.12.2023-31.12.2023	Виконано

Дата видачі завдання 02.10.2023 р.

Керівник кваліфікаційної роботи: к.т.н доцент Наталія ШИБИЦЬКА

Завдання прийняв до виконання: Ігор ТАРАСОВ

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи «Методика розробки компілятора на мові Rust»: 80 сторінок, 7 рисунків, 1 таблиця, 25 використаних джерел, 2 додатки.

КОМПІЛЯТОР, ЛЕКСИЧНИЙ АНАЛІЗ, СИНТАКСИЧНИЙ АНАЛІЗ, СЕМАНТИЧНИЙ АНАЛІЗ, ОПТИМІЗАЦІЯ КОДУ, ГЕНЕРАЦІЯ ВИХІДНОГО КОДУ, ТЕСТУВАННЯ ПРОГРАМ, ПРОДУКТИВНІСТЬ КОМПІЛЯТОРА, ЕФЕКТИВНІСТЬ RUST У ВЕЛИКИХ ПРОЕКТАХ, ПОРІВНЯННЯ КОМПІЛЯТОРІВ.

Об’єкт дослідження - процес розробки компілятора на мові програмування Rust.

Мета кваліфікаційної роботи - розробка методики для створення компілятора на мові програмування Rust.

Метод дослідження – аналіз літературних джерел, вивчення мови програмування Rust, розробку прототипу компілятора, проведення експериментальних досліджень та тестування для подальшого аналізу результатів та документації.

В результаті цієї роботи була успішно розроблена методика створення компілятора на мові програмування Rust. Аналіз літературних джерел і вивчення особливостей мови Rust дозволили зрозуміти ключові аспекти, які впливають на процес компіляції. Реалізований прототип компілятора включає етапи лексичного та синтаксичного аналізу, семантичного аналізу, генерації коду та оптимізації. Експериментальні дослідження та перевірка підтвердили коректність і високу продуктивність розробленого компілятора. Отримані результати детально задокументовані в пояснювальній записці, а висновки свідчать про досягнення поставлених цілей та окреслюють перспективи подальших досліджень у цьому напрямі.

ABSTRACT

Explanatory note to the diploma thesis "Methodology for the development of a compiler in the Rust language": 80 pages, 7 figures, 1 tables, 25 used sources, 2 appendices.

COMPILER, LEXICAL ANALYSIS, SYNTAX ANALYSIS, SEMANTIC ANALYSIS, CODE OPTIMIZATION, SOURCE CODE GENERATION, PROGRAM TESTING, COMPILER PERFORMANCE, RUST PERFORMANCE ON LARGE PROJECTS, COMPILER COMPARISON.

The object of research is the process of developing a compiler in the Rust programming language.

The aim of the thesis is to develop a methodology for creating a compiler in the Rust programming language.

The research method is the analysis of literary sources, the study of the Rust programming language, the development of a compiler prototype, conducting experimental research and testing for further analysis of the results and documentation.

As a result of this work, a method for creating a compiler in the Rust programming language was successfully developed. The analysis of literary sources and the study of features of the Rust language allowed us to understand the key aspects that affect the compilation process. The implemented prototype of the compiler includes stages of lexical and syntactic analysis, semantic analysis, code generation and optimization. Experimental studies and verification confirmed the correctness and high performance of the developed compiler. The obtained results are documented in detail in the explanatory note, and the conclusions indicate the achievement of the set goals and outline the prospects for further research in this direction.

Зміст

ПЕРЕЛІК ПРИЙНЯТИХ СКОРОЧЕНЬ

АСТ: абстрактне синтаксичне дерево

JIT: Just-in-time translator

WASM: WebAssembly

RAM: оперативна пам'ять

GPU: графічний процесор

ОС: Операційна система

API: інтерфейс прикладного програмування

HTTP: протокол передачі гіпертексту

URL: Уніфікований покажчик ресурсів

HTML: Мова розмітки гіпертексту

CSS: каскадні таблиці стилів

JS: JavaScript

IDE: інтегроване середовище розробки

SQL: мова структурованих запитів

JSON: нотація об’єктів JavaScript

XML: розширювана мова розмітки

FTP: протокол передачі файлів

LAN: Локальна мережа

WAN: глобальна мережа

VPN: віртуальна приватна мережа

IoT: Інтернет речей

API: інтерфейс прикладного програмування

SDK: комплект розробки програмного забезпечення

DNS: система доменних імен

HTTPS: безпечний протокол передачі гіпертексту

SSL: Рівень захищених сокетів

TLS: Безпека транспортного рівня

ASCII: Американський стандартний код для обміну інформацією

UTF-8: Формат перетворення Unicode – 8-бітний

CRUD: Створення, читання, оновлення, видалення (операції з базою даних)

REST: репрезентативний державний трансфер

ООП: об'єктно-орієнтоване програмування

ВСТУП

У середовищі мов програмування та розробки програмного забезпечення, що постійно розвивається, створення ефективних і надійних компіляторів залишається ключовим напрямком досліджень та інновацій. У цій роботі розглядається методологія розробки компілятора, спеціально розробленого для мови програмування Rust.

Rust, відомий своїм наголосом на безпеці, продуктивності та сучасному синтаксисі, отримав значну популярність у спільноті розробників програмного забезпечення. Оскільки попит на надійні інструменти компілятора продовжує зростати, дослідження ефективних методологій для створення компіляторів, адаптованих до Rust, стає дедалі актуальнішим. Цей вступ закладає основу для поглибленого вивчення ключових аспектів розробки компілятора Rust, охоплюючи теоретичні основи, практичну реалізацію та експериментальний аналіз.

Завдяки цьому дослідженню я прагну зробити внесок у розвиток технології компіляторів та запропонувати розуміння тонкощів адаптації таких інструментів до відмінних особливостей Rust, сприяючи глибшому розумінню взаємодії між дизайном мови та конструкцією компілятора.

1.1 Актуальність теми

Це дослідження заглиблюється в складну сферу побудови компілятора, приділяючи особливу увагу використанню мови програмування Rust для цієї мети. Мотивація, що лежить в основі цього дослідження, ґрунтується на унікальних характеристиках Rust, таких як акцент на безпеці пам’яті, безкоштовні абстракції та безпечний паралелізм. Мета полягає в тому, щоб дослідити доцільність і переваги використання Rust у розробці компіляторів, оцінивши його вплив на продуктивність продукту та розробників та безпеку.

Поточний стан досліджень щодо створення компіляторів мовою Rust

демонструє багатообіцяюче поєднання безпеки, продуктивності програмного засобу та розробника. Унікальні функції Rust, такі як право власності (ownership) та запозичення (borrow), пропонують переконливий підхід до безпеки пам’яті під час побудови компілятора, що відповідає сучасним вимогам до безпечного програмного забезпечення. Однак серед труднощів є помітна крива навчання для розробників, які не знайомі з концепціями Rust, потенційні обмеження в інструментах компілятора та запитання щодо витрат часу виконання. Проблеми сумісності та потреба в добре задокументованих ресурсах також мають місце. Поточна ситуація свідчить про те, що, незважаючи на те, що Rust має потенціал для вдосконалення технології компілятора, вирішення цих проблем є ключовим для реалізації його більш широкого впровадження та максимізації його переваг з точки зору безпеки та ефективності.

Стан наукових досліджень побудови компілятора мовою Rust сформовано завдяки внеску відомих дослідників і практиків. Серед ключових фігур у цій сфері – Керол Ніколс, Стів Клабнік і Грейдон Хоар, які зіграли ключову роль у розробці Rust і його застосуванні в системному програмуванні. Хоча існуюча література переважно стосується переваг Rust, залишається помітна прогалина у всебічних дослідженнях конкретних проблем, які виникають під час впровадження компіляторів у Rust. Обмежене дослідження проблем сумісності, крива навчання, пов’язана із системою власності Rust, і відсутність детальних досліджень накладних витрат часу виконання є областями, які вимагають подальшого вивчення. Поточна кількість літератури підкреслює необхідність глибокого аналізу та емпіричних досліджень для покращення нашого розуміння цих нюансів, надаючи більш повну картину проблем і можливостей у розробці компілятора за допомогою мови Rust.

Вибір дослідження створення компіляторів на мові Rust мотивований очевидною недорозвиненістю окремих аспектів цієї області. У той час як існуюча

література визнає переваги Rust у створенні компілятора, існує помітна прогалина у комплексних дослідженнях, що стосуються тонких проблем. Відносна нестача поглиблених досліджень, які вивчають криву навчання, пов’язану із системою власності Rust, потенційні проблеми взаємодії та вплив певних мовних функцій на накладні витрати часу виконання, залишили критичні запитання без відповіді. Визнання цих недосліджених аспектів підкреслює необхідність і важливість обраної теми дослідження, оскільки вона прагне заповнити ці прогалини, внести емпіричні висновки та забезпечити більш цілісне розуміння практичних наслідків і проблем, пов’язаних із розробкою компіляторів у Rust.

1.2 Об'єкт дослідження

Компілятор на мові Rust.

1.3 Предмет дослідження

Методи та засоби розробки програмного забезпечення.

1.4 Мета дипломного проекту

Дослідження та вдосконалення практик розробки програмного забезпечення шляхом поглибленого вивчення методів та інструментів. Це дослідження має на меті критично оцінити існуючі методології та технологічні рамки, що використовуються в розробці програмного забезпечення, визначити їх переваги та обмеження, а також запропонувати інноваційні підходи чи вдосконалення. Кінцева мета полягає в тому, щоб підвищити ефективність, надійність і зручність обслуговування програмних систем шляхом надання практичних ідей і рекомендацій для розробників і практиків галузі. Завдяки ретельному аналізу та експерименту дипломний проект має на меті внести цінні знання в цю сферу, сприяючи глибшому розумінню сучасних методів та інструментів розробки, одночасно вирішуючи поточні виклики та сфери, які потрібно вдосконалити.

1.5 Методи дослідження

Огляд літератури: Проведення комплексного огляду існуючих наукових

статей, книг і матеріалів конференцій для створення теоретичної основи та розуміння поточного стану знань у цій галузі.

Тематичні дослідження: Аналіз реальних проектів або організацій для проведення поглибленого аналізу процесів їх розробки, методологій і вибору інструментів, зосереджуючись на розумінні результатів і викликів.

Кількісний аналіз: використання статистичних методів для аналізу кількісних даних, таких як показники продуктивності, оцінки задоволеності користувачів або інші вимірні показники, пов’язані з ефективністю методів і інструментів розробки.

Якісний аналіз: використання якісних методів аналізу, таких як аналіз контенту або тематичне кодування, для інтерпретації текстових даних або даних інтерв’ю та виявлення шаблонів, тем або нових концепцій.

Експериментальне дослідження: проведення контрольованих експериментів для оцінки впливу конкретних методів або інструментів розробки на різні результати, такі як якість коду, часові рамки проекту або задоволеність розробників.

Спостережні дослідження: спостереження та документування процесів розробки в режимі реального часу, щоб отримати уявлення про повсякденні виклики та практики, пов’язані з конкретними методами та інструментами розробки.

Порівняльний аналіз: порівняння різних методологій та інструментів розробки з точки зору їхніх переваг, недоліків і загальної ефективності на основі попередньо визначених критеріїв.

Симуляція та моделювання: використання засобів моделювання або методів моделювання для моделювання та аналізу різних сценаріїв, пов’язаних із процесами розробки програмного забезпечення та використанням інструментів.

1.6 Технічні та програмні засоби

Інтегроване середовище розробки (IDE): Visual Studio Code

Система контролю версій (VCS): Git

Інструмент аналізу коду: rust-analyzer

Інструмент документації: Markdown

Платформа для співпраці: GitHub

1.7 Наукова новизна

Наукова новизна полягає у дослідженні і створенні простого, функціонального, безпечного для пам'яті і багатопоточності компілятора, який використовуватиме малу кількість ресурсів системи і матиме на меті швидку і легку компіляцію, але не оптимізацію.

1.8 Практична значимість проєкту

Практична значимість проєкту полягає у можливості використання створеного компілятора в екосистемі мови Rust, крім того через не велике споживання ресурсів системи можливе також використання у вбудованих системах.

1.9 Особистий внесок

Дослідження, розробка і аналіз роботи компілятора на мові Rust.

РОЗДІЛ 1. ОГЛЯД ТА ПОРІВНЯЛЬНИЙ АНАЛІЗ АНАЛОГІВ ОБ'ЄКТА ДОСЛІДЖЕНЬ

У цьому розділі буде проведено всебічний огляд і порівняльний аналіз існуючих компіляторів на мові Rust, наголошуючи на їх дизайні, продуктивності та функціях. Це дослідження слугуватиме еталоном для оцінки нового компілятора розробленого на мові Rust.

1.1 Структура порівняльного аналізу

1.1.1 Продуктивність

Час компіляції — це фаза розробки програмного забезпечення, на якій вихідний код програми транслюється компілятором у машинний код або проміжний код перед виконанням. На тривалість часу компіляції впливають різні фактори, включаючи складність коду, ефективність компілятора, продуктивність апаратного забезпечення, налаштування компіляції та інструменти збірки. Налагодження та випуск збірок, інкрементальна компіляція, паралельна компіляція та використання попередньо скомпільованих заголовків або механізмів кешування можуть впливати на ефективність процесу компіляції. Розробники часто прагнуть оптимізувати час компіляції для швидших ітерацій під час розробки, гарантуючи, що кінцевий виконуваний файл добре оптимізований для продуктивності під час збірки випуску.

Швидкість виконання та використання ресурсів є критичними аспектами продуктивності програми. Швидкість виконання означає, наскільки швидко програма виконує свої завдання, залежно від таких факторів, як ефективність алгоритму, оптимізація коду та архітектура апаратного забезпечення. З іншого боку, використання ресурсів передбачає ефективне використання системних ресурсів, таких як процесор, пам'ять, диск і мережа. Управління пам'яттю, оптимізовані операції введення-виведення та паралельність відіграють ключову роль у використанні ресурсів. Досягнення балансу між швидкістю виконання та

використанням ресурсів є важливим, враховуючи компроміси та особливості платформи. Прагнення до оптимальної продуктивності передбачає ретельний дизайн, інструменти профілювання та безперервні зусилля з удосконалення для підвищення швидкості реагування та ефективності програмного забезпечення.

1.1.2 Особливості

Мови програмування оснащені різними функціями, які формують спосіб написання та виконання коду. Ці функції впливають на читабельність коду, виразність і функціональність. Загальні аспекти включають можливість визначати змінні та керувати ними, керуючі структури для керування потоком програм, функції або методи для модульного коду та підтримку різних структур даних. Об’єктно-орієнтовані мови програмування надають класи та об’єкти для інкапсуляції та успадкування. Механізми обробки винятків керують помилками, а засоби керування пам’яттю обробляють розподіл і звільнення. Підтримка паралелізму та паралелізму, лямбда-вирази, замикання, зіставлення шаблонів і генерики сприяють виразності коду. Відображення дозволяє перевіряти під час виконання, а функції метапрограмування дозволяють маніпулювати кодом під час компіляції. Функції сумісності полегшують спілкування з кодом іншими мовами. Багаті стандартні бібліотеки забезпечують готові функції для стандартних завдань. Комбінація та обсяг цих функцій різниться залежно від мови програмування, впливаючи на процес розробки та кінцеву кодову базу.

Інтеграція стандартної бібліотеки та розширюваність є ключовими аспектами мови програмування, які впливають на легкість розробки та універсальність проектів програмного забезпечення. Надійна стандартна бібліотека надає набір готових модулів і функцій, що охоплюють широкий спектр поширених завдань, що значно прискорює процес розробки, зменшуючи потребу розробників впроваджувати основні функції з нуля. Інтеграція з добре розробленою стандартною бібліотекою покращує читабельність коду, зручність

обслуговування та гарантує, що розробники можуть покладатися на перевірені й оптимізовані компоненти коду. Одночасно розширюваність дозволяє розробникам розширювати та налаштовувати стандартну бібліотеку шляхом додавання власних модулів або функцій, адаптуючи мову до конкретних вимог проекту. Мова, яка успішно врівноважує інтеграцію стандартної бібліотеки та розширюваність, дозволяє розробникам використовувати існуючі рішення, одночасно адаптуючи та розширюючи функціональні можливості для задоволення унікальних потреб проекту, сприяючи ефективній та масштабованій розробці програмного забезпечення.

1.1.3 Оптимізації

Рівень оптимізації компілятора відіграє ключову роль у визначенні продуктивності програми під час виконання. Оптимізація компілятора включає різні прийоми, спрямовані на підвищення ефективності скомпільованого коду. Агресивні рівні оптимізації можуть призвести до більш раціоналізованого та швидшого виконання коду завдяки використанню таких методів, як розгортання циклу, вбудовування та постійне згортання. Проте вищі рівні оптимізації також можуть збільшити час компіляції та призвести до збільшення розміру виконуваних файлів. Вкрай важливо знайти правильний баланс між оптимізацією та пов’язаними з нею витратами. У деяких випадках надмірно агресивна оптимізація може призвести до зменшення віддачі або навіть до непередбачуваних наслідків, таких як довший час компіляції або збільшення використання пам’яті. Отже, розробникам необхідно ретельно адаптувати параметри оптимізації на основі конкретних вимог програми, враховуючи такі фактори, як швидкість виконання, розмір коду та використання ресурсів для оптимальної продуктивності під час виконання.

1.2 Існуючі компілятори

RustPython — це інтерпретатор Python, написаний мовою Rust. RustPython

можна вбудувати в програми Rust, щоб використовувати Python як мову сценаріїв для вашої програми, або його можна скомпілювати до WebAssembly, щоб запускати Python у браузері. RustPython є безкоштовним і відкритим.

Rustc - компілятор мови Rust. Так як rustc написаний на тій же мові яку він сам компілює гарантує високий рівень оптимізацій в коді компілятора, так як його пишуть самі розробники мови.

Inko - Мова програмування, компілятор якої написаний на Rust. Цікава мова, яка має велику кількість функціоналу і направлена на асинхронні обчислення.

1.3 Порівняльний аналіз

Порівняльний аналіз компіляторів включає складні дослідження, і на результати можуть впливати різні фактори, зокрема тип компілюваного коду, цільова платформа та застосована конкретна оптимізація. Ось загальний огляд того, як можна оцінити RustPython, rustc та Inko з точки зору продуктивності, функціоналу та оптимізації:

- Продуктивність:

RustPython: як інтерпретатор Python, RustPython може мати характеристики продуктивності, відмінні від компільованих мов, таких як Rust. Цей компілятор працює з AST і генерує байткод, який потім може виконати за допомогою JIT. Це доволі функціональний проект, який має на меті запуск коду будь де, навіть на WASM.

Rustc (компілятор Rust): Rust відомий своєю зосередженістю на продуктивності, а rustc прагне створити оптимізований машинний код. Модель власності Rust і абстракції з нульовою вартістю сприяють її перевагам у продуктивності.

Inko: Inko представляє унікальну модель паралелізму, розроблену для підвищення продуктивності. Порівняльний аналіз Inko включав би оцінку його функцій паралелізму, швидкості виконання та використання пам’яті. Порівняння

може врахувати, наскільки добре Inko працює в одночасних робочих навантаженнях порівняно з іншими мовами.

- Особливості:

RustPython: функції включають сумісність мови Python, підтримку стандартних бібліотек Python і легкість інтеграції з проектами Rust. Важливо оцінити, наскільки RustPython підтримує динамічні функції та екосистему Python.

Rustc (компілятор Rust): Rust наголошує на безпеці пам'яті, паралелізмі та продуктивності. Особливості включають систему власності та терміну служби, відповідність шаблону та надійну систему типів. Екосистема Rust включає менеджер пакунків (Cargo) і зосередженість на програмуванні на системному рівні.

Inko: функції в Inko включатимуть оцінку його моделі паралельного виконання на основі акторів, гарантій безпеки пам'яті та загальної виразності мови. Міркування включають те, наскільки добре Inko вирішує сучасні виклики програмування та його підтримку бібліотек і інструментів.

- Оптимізації:

RustPython: Інтерпретатори Python часто покладаються на оптимізацію часу виконання.

rustc (компілятор Rust): Rust наголошує на оптимізації під час компіляції, а rustc застосовує різні оптимізації, такі як вбудовування, постійне згортання та оптимізація циклів.

Inko: Inko представляє власний набір оптимізацій, особливо в контексті моделі актора.

Висновок

У дослідженні існуючих компіляторів на мові Rust та порівняльному аналізі, проведеному в цьому розділі, фокус змістився з безпосередньої функціональності розробленого компілятора до ширшої перспективи, яка надає першочергового

значення навчанню та зростанню як програмний інженер. Основною задачею створеного продукту має бути низький рівень використання ресурсів системи.

Хоча розроблений компілятор, можливо, не досягне бажаного рівня функціональності ближчим часом, значення полягає в накопиченому досвіді. Складне дослідження існуючих компіляторів та їх порівняльний аналіз забезпечили детальне розуміння проблем, притаманних розробці компіляторів, а також розуміння різних парадигм дизайну та стратегій оптимізації.

РОЗДІЛ 2 ТЕОРЕТИЧНІ ТА ЕКСПЕРИМЕНТАЛЬНІ ДОСЛІДЖЕННЯ

У цьому розділі описано взаємодію між теоретичними основами та практичними експериментами в контексті розробки компілятора мови Uniq в Rust. Поєднання теоретичних принципів і практичних експериментів є невід'ємною частиною розуміння тонкощів побудови компілятора та вдосконалення практичних навичок програмування.

2.1 Теоретична база

Розробка компілятора включає багатогранний процес, що охоплює різні етапи, кожен з яких має власний набір принципів, спрямованих на перетворення коду програмування високого рівня в машинний код або проміжний код. Початкова фаза, лексичний аналіз, розбирає вихідний код на лексеми або лексеми, призначаючи типи кожному. Після цього синтаксичний аналіз аналізує потік маркерів для побудови синтаксичного дерева або абстрактного синтаксичного дерева (AST), що представляє граматичну структуру вихідного коду. Далі йде семантичний аналіз, який забезпечує правильність коду з точки зору його передбачуваного значення, перевірку оголошень змінних перед використанням, сумісність типів та інших семантичних правил.

Наступним критичним кроком є генерація проміжного коду, де створюється абстрактне представлення вихідного коду, що полегшує оптимізацію та подальший переклад. Оптимізація компілятора, ключовий принцип, зосереджена на вдосконаленні проміжного коду для покращення продуктивності під час виконання, зменшення розміру коду та оптимізації використання ресурсів. Ця фаза включає такі техніки, як розгортання петлі, вбудовування та постійне згортання. Встановлення балансу між рівнями оптимізації має важливе значення, оскільки надмірно агресивна оптимізація може призвести до зменшення віддачі або небажаних наслідків, таких як збільшення часу компіляції або збільшення розміру виконуваних файлів.

Після оптимізації коду компілятор переходить до генерації коду, де оптимізований проміжний код транслюється в цільовий машинний код або інший проміжний код, залежно від платформи. Видача коду, останній крок, передбачає створення скомпільованого коду у формі виконуваного файлу або зв'язування його з іншими модулями для створення остаточного виконуваного файлу.

У процесі компіляції ефективна обробка помилок має вирішальне значення. Компілятор повинен надавати інформативні повідомлення про помилки, щоб допомогти розробникам у виявленні та виправленні проблем у коді. Крім того, для відстеження ідентифікаторів, змінних та їхніх атрибутів використовується керування таблицею символів. Стратегії керування пам'яттю реалізовано для забезпечення ефективного розподілу та звільнення пам'яті під час процесу компіляції.

Принципи проектування компілятора також враховують цільову машинну незалежність для досягнення переносимості, розділяючи машинно-залежні та машинно-незалежні аспекти компіляції. Деякі сучасні компілятори включають методи розпаралелювання для оптимізації виконання коду на багатоядерних процесорах. Динамічна компіляція — це ще одна функція, яка підтримується певними компіляторами, що дозволяє компілювати код під час виконання для своєчасного виконання (JIT). Крім того, компілятори можуть включати інформацію про налагодження в згенерований код, щоб допомогти розробникам у діагностиці та виправленні проблем під час процесу налагодження. Поступова компіляція — це стратегія, яка дозволяє повторно компілювати лише ті частини коду, які змінилися, оптимізуючи процес розробки та покращуючи швидкість ітерації. Розуміння та застосування цих принципів має важливе значення для створення компіляторів, які ефективно перекладають мови програмування високого рівня у виконуваний код, адаптований до цільової платформи.

Мова програмування Rust ґрунтується на наборі принципів і функцій,

призначених для пріоритетності безпеки пам’яті, безкоштовних абстракцій і паралельного програмування без шкоди для продуктивності. Розроблений Mozilla, Rust спеціально створений для системного програмування, де дуже важливий точний контроль апаратних ресурсів. За своєю суттю Rust представляє унікальну систему власності, яка керує керуванням пам’яттю. Кожне значення в Rust має визначеного «власника», і одночасно дозволено використовувати лише одного власника. Ця модель власності допомагає запобігти поширеним проблемам, таким як змагання за даними та помилки, пов’язані з пам’яттю, контролюючи, як доступ до даних і зміна.

Систему власності доповнює концепція запозичення, де право власності може бути тимчасово передано або спільно використано за допомогою посилань. Засіб перевірки запозичень, ключовий компонент компілятора Rust, забезпечує дотримання суворих правил для забезпечення безпечного використання посилань, запобігаючи таким проблемам, як завислі посилання або змагання даних у паралельних середовищах.

Тривалість життя є ще одним невід'ємним аспектом теорії Rust. Тривалість життя — це явні анотації, які позначають область, для якої посилання дійсні. Це забезпечує ясність у розумінні того, як довго посилання залишаються придатними для використання, і допомагає запобігти незначним помилкам, пов’язаним із посиланнями, які переживають дані, на які вони вказують.

Підхід Rust до паралелізму примітний своїм наголосом на запобіганні перегонів даних, не покладаючись на збирач сміття. Система власності гарантує, що змінні дані не можуть мати одночасний доступ для кількох потоків, забезпечуючи високий рівень безпеки під час паралельного програмування. Це досягається без шкоди для продуктивності, що узгоджується з прихильністю Rust до безкоштовних абстракцій.

Мова також підтримує безкоштовні абстракції, що означає, що функції мови

високого рівня не створюють непотрібних накладних витрат на виконання. Rust дозволяє розробникам писати виразний і безпечний код, не жертвуючи низькорівневим контролем, коли продуктивність критична. Такі функції, як зіставлення шаблонів і алгебраїчні типи даних, сприяють виразності коду, дозволяючи розробникам легко обробляти різні випадки та визначати складні структури даних.

Характеристики, ще одна ключова функція, забезпечують механізм для визначення спільної поведінки для різних типів, сприяючи повторному використанню коду та абстракції. Rust заохочує структурований паралелізм за допомогою таких конструкцій, як async і await, що робить асинхронне програмування безпечнішим і зрозумілішим.

Обробка помилок у Rust базується на типах Result і Option, що забезпечує явну обробку потенційних помилок. Цей підхід підвищує надійність коду, пояснюючи, коли функція може повернути помилку, таким чином зменшуючи ймовірність непередбачених проблем.

Нарешті, Rust визнає необхідність низькорівневих операцій у певних сценаріях. Мова забезпечує режим «Небезпечний Rust», який дозволяє розробникам обходити певні перевірки безпеки, коли це необхідно. Однак використання небезпечних функцій суворо контролюється, щоб пом’якшити небажані наслідки.

Таким чином, теорія Rust обертається навколо власності та запозичень, тривалості життя, абстракцій з нульовою вартістю та зосередженості на запобіганні типових помилок програмування. Ці принципи спільно сприяють досягненню мети Rust щодо забезпечення безпечного та продуктивного програмування на системному рівні, що робить його потужною мовою для створення ефективного, одночасного та надійного програмного забезпечення.

2.2 Експериментальне впровадження

2.2.1 Лексичний аналіз

Лексичний аналіз, перший етап у процесі компіляції, є вирішальним кроком у перетворенні необробленого вихідного коду в структурований і керований формат. Він передбачає розбиття вихідного коду на токени, які є найменшими одиницями значення в програмі. Токени охоплюють різні мовні конструкції, такі як ключові слова, ідентифікатори, літерали, оператори та знаки пунктуації. Процес токенізації спирається на попередньо визначені шаблони, які часто описуються за допомогою регулярних виразів, щоб розпізнавати та класифікувати лексеми — послідовності символів, які відповідають шаблонам для певного токена.

Пробіли та коментарі, хоча й присутні у вихідному коді, зазвичай не враховуються під час лексичного аналізу, оскільки вони не впливають на семантику програми. Однак інформація про номери рядків або позиції може зберігатися для звітування про помилки. Лексичні аналізатори обладнані для виявлення лексичних помилок і звітів про них, допомагаючи розробникам у виявленні проблем у коді на ранніх стадіях процесу компіляції.

Таблиці символів можуть бути заповнені під час лексичного аналізу, зберігаючи інформацію про ідентифікатори, які зустрічаються у вихідному коді, такі як їхні імена, типи та місця пам’яті. Ця інформація має вирішальне значення для наступних етапів процесу компіляції. Використання скінченних автоматів або алгоритмів зіставлення регулярних виразів є звичайним у реалізації лексичних аналізаторів, оскільки вони ефективно розпізнають і позначають лексеми на основі заданих шаблонів.

Лексеми, створені під час лексичного аналізу, пов’язуються з атрибутами, надаючи додаткову інформацію для подальших етапів компіляції. Наприклад, лексема числового літералу може містити фактичне числове значення як атрибут. Крім того, лексичні аналізатори відіграють певну роль в управлінні лексичним

визначенням, особливо в мовах, де обсяг ідентифікаторів визначається лексичними правилами.

Щоб автоматизувати створення лексичних аналізаторів, розробники часто використовують генератори лексерів або інструменти, які беруть специфікацію, часто у формі регулярних виразів, і генерують відповідний код лексера. Ця автоматизація спрощує процес розробки та допомагає забезпечити послідовність і точність токенізації вихідного коду. Таким чином, лексичний аналіз є основоположним кроком, який готує основу для наступних етапів компіляції, забезпечуючи структуроване представлення програми, яке компілятори можуть далі аналізувати на синтаксис і семантику.

2.2.2 Синтаксичний аналіз

Синтаксичний аналіз, також відомий як парсинг, є критичною фазою процесу компіляції, яка слідує за лексичним аналізом. Тоді як лексичний аналіз розбиває вихідний код на лексеми, синтаксичний аналіз зосереджується на структурі та граматиці програми. Основною метою є побудова ієрархічної структури, такої як синтаксичне дерево або абстрактне синтаксичне дерево (AST), яка представляє граматичні зв’язки між різними елементами коду.

Під час синтаксичного аналізу компілятор перевіряє, чи відповідає розташування лексем правилам граматики мови програмування. Ця фаза передбачає застосування формальної граматики, часто визначеної за допомогою контекстно-вільних граматик, для розпізнавання синтаксичної структури вихідного коду. Отримане синтаксичне дерево служить проміжним представленням, яке фіксує синтаксичну структуру програми.

Одним із ключових аспектів синтаксичного аналізу є виявлення синтаксичних помилок. Якщо компілятор стикається з кодом, який порушує правила граматики мови, він генерує значущі повідомлення про помилки, щоб допомогти розробникам виправити свій код. Синтаксична структура, визначена на

цьому етапі, служить основою для подальшого семантичного аналізу та етапів генерації коду.

Генератори синтаксичних аналізаторів або комбінатори синтаксичних аналізаторів є широко використовуваними інструментами для автоматизації впровадження синтаксичного аналізу. Ці інструменти приймають формальну граматичну специфікацію як вхідні дані та генерують код для аналізатора. Рекурсивний синтаксичний аналіз за спуском, синтаксичний аналіз знизу вгору та синтаксичний аналіз LR є одними з технік, які використовуються для створення синтаксичних аналізаторів для різних мов.

Синтаксичний аналіз важливий для розуміння зв’язків між різними компонентами програми, створюючи основу для подальшого аналізу та трансформації. Отримане синтаксичне дерево має вирішальне значення для наступних етапів компілятора, сприяючи загальному процесу перетворення вихідного коду високого рівня у виконувану форму. По суті, синтаксичний аналіз гарантує, що програма дотримується встановленого синтаксису мови та формує надійну основу для наступних етапів компіляції.

2.2.3 Семантичний аналіз

Семантичний аналіз є ключовим етапом у процесі компіляції, який слідує за синтаксичним аналізом. Тоді як синтаксичний аналіз зосереджується на структурі та граматиці програми, семантичний аналіз заглиблюється в значення та логіку коду. Основна мета полягає в тому, щоб програма відповідала семантиці мови, вирішуючи проблеми, які виходять за межі синтаксису та передбачають розуміння передбачуваної поведінки коду.

Під час семантичного аналізу компілятор перевіряє зв’язки між різними елементами програми, перевіряє логічну послідовність і забезпечує дотримання специфічних для мови правил щодо використання змінних, типів та інших конструкцій. Цей етап виходить за рамки синтаксичної правильності, перевіреної

на попередніх етапах, і спрямований на виявлення помилок, пов’язаних із передбачуваним значенням коду. Загальні аспекти семантичного аналізу включають перевірку типу, розв’язання області та виконання правил і обмежень, характерних для мови.

Перевірка типів є важливим компонентом семантичного аналізу, гарантуючи, що змінні та вирази відповідають вказаним типам даних. Компілятор перевіряє, чи виконуються операції над сумісними типами даних, допомагаючи запобігти пов’язаним із типом помилкам виконання. Крім того, семантичний аналіз передбачає визначення областей видимості змінних, гарантуючи, що змінні використовуються в правильному контексті на основі їх оголошень.

Іншим ключовим аспектом є виявлення семантичних помилок, таких як неініціалізовані змінні, несумісні призначення або порушення правил мови. Змістовні повідомлення про помилки генеруються, щоб допомогти розробникам зрозуміти та виправити ці проблеми.

Семантичний аналіз закладає основу для наступних етапів, таких як оптимізація та генерація коду. Компілятор генерує проміжне представлення програми, яке відображає як її синтаксичну структуру, так і семантичне значення. Потім це представлення використовується для створення ефективного та правильного машинного коду або іншого проміжного коду для виконання.

По суті, семантичний аналіз усуває розрив між синтаксичною структурою коду та його передбачуваним значенням, гарантуючи, що програма поводиться так, як очікувалося, і дотримується правил мови програмування. Цей етап відіграє вирішальну роль у підвищенні надійності та правильності скомпільованого коду, сприяючи досягненню загальної мети перетворення вихідного коду високого рівня у виконувані програми.

2.2.4 Генерація коду

Генерація коду є важливою фазою в процесі компіляції, яка відбувається

після лексичного та синтаксичного аналізу. Після того, як компілятор проаналізував структуру та значення вихідного коду, фаза генерації коду відповідає за переклад проміжного представлення високого рівня (часто абстрактного синтаксичного дерева або іншої форми проміжного коду) у цільовий машинний код або іншу виконувану форму.

Під час генерації коду компілятор перетворює абстрактне представлення в представлення нижчого рівня, яке можна виконати на цільовій архітектурі. Це передбачає відображення конструкцій високого рівня на відповідні машинні інструкції, регістри та адреси пам'яті. Методи оптимізації також можуть бути застосовані на цьому етапі для підвищення продуктивності згенерованого коду.

Згенерований код має на меті зберегти семантику вихідної програми з урахуванням тонкощів цільової машини. Компілятор повинен керувати розподілом і звільненням ресурсів, таких як регістри та пам'ять, а також виконувати переклад високорівневих конструкцій, таких як цикли та умови, в ефективні інструкції асемблера або машинного коду.

На цьому етапі може відбуватися оптимізація компілятора, щоб покращити продуктивність згенерованого коду під час виконання. Техніки оптимізації включають, серед іншого, усунення загальних підвиразів, розгортання циклу та вбудовування функцій. Ці оптимізації спрямовані на скорочення часу виконання, мінімізацію використання пам’яті та загалом підвищення ефективності скомпільованого коду.

Генерація коду сильно залежить від цільової архітектури, і компілятори часто мають бекенди, адаптовані для конкретних платформ. Згенерований код має відповідати набору інструкцій і моделі пам’яті цільової машини, щоб забезпечити сумісність і оптимальну продуктивність.

Підсумовуючи, генерація коду — це процес перекладу високорівневого незалежного від платформи представлення програми в машинний код або іншу

виконувану форму, яка може працювати на певній цільовій архітектурі. Цей етап має вирішальне значення для створення ефективного та правильного виконуваного коду з оптимізацією, спрямованою на покращення продуктивності кінцевої програми.

2.3 Інтеграція теорії та практики

2.3.1 Практичні проблеми та рішення

Написання компілятора є складним завданням, яке передбачає вирішення багатьох практичних завдань. Ось деякі типові проблеми, які виникають під час розробки компілятора, а також потенційні рішення:

1. Проблема: розробка ефективного аналізатора для аналізу синтаксису мови програмування та створення абстрактного синтаксичного дерева (AST) може бути складною.

   Рішення: можна використовувати генератори чи комбінатори синтаксичних аналізаторів, щоб автоматизувати генерацію коду синтаксичного аналізатора на основі формальної граматичної специфікації. Такі інструменти, як Bison, ANTLR або PLY, можуть допомогти у створенні парсерів.
2. Проблема: Потрібно переконатися, що код відповідає семантиці мови, включаючи перевірки типів на допустимі значення, це вимагає ретельного впровадження семантичного аналізу.

   Рішення: потрібно запровадити фазу надійного семантичного аналізу, яка перевірятиме невідповідності типів, забезпечить дотримання правил визначення обсягу та виявляє інші семантичні помилки. Будуть використані наявні інструменти або бібліотеки для алгоритмів перевірки типу.
3. Проблема: генерація проміжного представлення коду, яке зберігає семантику, водночас придатне для оптимізації та створення коду, є критичним кроком.

   Рішення: розробка і реалізація структур даних, таких як абстрактне

синтаксичне дерево (AST) або проміжний код, який фіксує структуру високого рівня програми. Це проміжне представлення повинно бути легко перекладено в машинний код.

4. Проблема: розробка ефективних стратегій оптимізації для покращення продуктивності згенерованого коду може бути складною та займати багато часу.

   Рішення: розробка базової оптимізації, як-от постійне згортання, і поступове впровадження більш просунутих методів. Використання наявних досліджень та інструментів для отримання вказівок щодо ефективних стратегій оптимізації.

5. Проблема: Створення ефективного та правильного машинного коду, адаптованого до конкретної цільової архітектури, створює проблеми, особливо для різноманітних платформ.

   Рішення: розробка бекенду, який переводить проміжне представлення в машинний код, враховуючи набір інструкцій цільової архітектури та модель пам’яті. Потрібно використати LLVM або подібні фреймворки для створення коду.

6. Проблема: Виявлення помилок і повідомлення про помилки зручним способом є важливими для налагодження та покращення роботи розробника.

   Рішення: запровадження надійного механізму обробки помилок, який надає інформативні повідомлення про помилки, включаючи детальну інформацію про місце та характер помилки. Це допоможе розробникам швидко виявляти та усувати проблеми у своєму коді.

7. Проблема: Забезпечення правильності компілятора та ефективне його налагодження може бути складним завданням.

   Рішення: розробка комплексного набору тестів, які охоплюють різні мовні функції та крайні випадки. Використання інструментів та методів

налагодження, щоб виявити та виправити проблеми в реалізації компілятора. Реалізуйте механізми журналювання та відстеження, щоб допомогти у налагодженні.

8. Проблема: відсутність документації та підтримки спільноти може перешкодити прийняттю та вдосконаленню компілятора.

   Рішення: надання вичерпної документації щодо використання компілятора, внутрішніх елементів і точок розширення. Створення спільноти навколо компілятора, заохочення співпраці та пошук відгуків від інших розробників.

Підсумовуючи, написання компілятора передбачає вирішення різноманітних завдань, і ключ до успіху полягає в ретельному плануванні, використанні існуючих інструментів і бібліотек і поступовому вдосконаленні компілятора через ітерації та відгуки від спільноти.

2.3.2 Ітеративний процес розробки

Розробка компілятора є складною роботою, яка часто супроводжується ітеративним процесом розробки. Цей підхід передбачає поступове створення та вдосконалення компілятора через кілька етапів, кожен з яких додає нові функції або покращує існуючі. Ось короткий огляд ітераційного процесу розробки для створення компілятора:

- Специфікація та дизайн

Визначення специфікацій мови програмування, яку підтримуватиме компілятор. Чітке окреслення синтаксису, семантики та особливостей. Проектування архітектури компілятора, включаючи різні етапи, такі як лексичний аналіз, розбір, семантичний аналіз і генерація коду.

- Лексичний аналіз і парсинг

Впровадження фази лексичного аналізу, щоб токенізувати вихідний код. Розробка синтаксичного аналізатору, який генерує базове абстрактне синтаксичне

дерево (AST). Спочатку потрібно зосередитися на підтримці основних мовних конструкцій.

- Семантичний аналіз

Удосконалити компілятор, реалізувавши семантичний аналіз для перевірки помилок типів, проблем із областю та інших семантичних правил. Поступове введення більш просунутих можливостей мови.

- Генерація проміжного коду

Введення генерації проміжного представлення (IR) для захоплення високорівневої структури програми. Це служить основою для наступних етапів оптимізації та генерації коду.

- Оптимізація

Застосування основних методів оптимізації для підвищення продуктивності згенерованого коду. Потрібно почати з простих оптимізацій, таких як постійне згортання.

- Генерація коду

Розробка бекенду генерації коду, який перетворює проміжне представлення в машинний код. Розгляд набіру інструкцій і моделі пам'яті цільової архітектури. Уточнення процесу створення коду для коректності та ефективності.

- Тестування та налагодження

Створення повного набіру тестів для перевірки правильності та функціональності компілятора. Використання засобів налагодження, щоб виявити та виправити проблеми в реалізації компілятора. Ітеративне вдосконалення компілятору на основі відгуків від тестування.

- Документація

Документування використання компілятора, внутрішніх елементів та точок розширення. Надання чіткої та вичерпної документації, щоб допомогти користувачам і розробникам зрозуміти функції та можливості компілятора.

- Розширення функіоналу

Поступове розширення підтримка мови та введення більш розширених функції. Реалізація підтримки додаткових типів даних, керуючих структур і мовних розширень. Кожна нова функція має узгоджуватися із загальним дизайном компілятора.

- Залучення спільноти

Створення спільноти навколо компілятора, заохочуючи співпрацю та шукаючи відгуки від користувачів та інших розробників. Потрібно розглянути можливість використання компілятора з відкритим вихідним кодом, щоб сприяти внеску спільноти та вдосконаленням.

- Доопрацювання та оптимізація

Потрібно постійно вдосконалювати внутрішні компоненти компілятора на основі відгуків користувачів і реального використання. Оптимізація продуктивності компілятора та ефективності пам’яті за допомогою постійних ітерацій.

Тож, дотримуючись ітераційного процесу розробки, розробники можуть керувати складністю конструкції компілятора, поступово створюючи надійний і багатофункціональний інструмент. Кожна ітерація приносить покращення, удосконалення та розширену підтримку мови, що з часом робить компілятор більш надійним і універсальним.

2.4 Оцінка та налагодження

2.4.1 Показники продуктивності

Оцінка продуктивності спеціального компілятора передбачає оцінку різних факторів, які впливають на його ефективність, зручність використання та якість створеного коду. Ось основні показники продуктивності для написання власного компілятора:

- Швидкість компіляції

Час, потрібний компілятору для перекладу вихідного коду у виконуваний машинний код, є критичним показником продуктивності. Ефективний лексичний і синтаксичний аналіз, а також спрощені методи оптимізації сприяють прискоренню часу компіляції.

- Якість створеного коду

Якість згенерованого машинного коду безпосередньо впливає на продуктивність виконання скомпільованої програми. Ефективні стратегії оптимізації, такі як постійне згортання, розгортання циклу та вбудовування, сприяють створенню оптимізованого та ефективного коду.

- Використання пам'яті

Моніторинг використання пам'яті компілятором є важливим, особливо для великих кодових баз. Ефективне управління пам'яттю на різних етапах компіляції гарантує, що компілятор може обробляти значні програми без споживання надмірної пам'яті.

- Сумісність цільової архітектури

Здатність компілятора генерувати машинний код, сумісний із цільовою архітектурою, має вирішальне значення. Переконайтеся, що компілятор оптимально використовує функції та інструкції цільового обладнання, що сприяє кращій продуктивності під час виконання.

- Підтримка мовних функцій

Те, якою мірою компілятор підтримує функції мови програмування, є ключовим показником ефективності. Повне мовне покриття гарантує, що розробники можуть використовувати весь спектр мовних конструкцій без шкоди для продуктивності.

- Обробка помилок і звітування

Ефективність обробки помилок і механізмів звітування впливає на

взаємодію з користувачем. Чіткі та інформативні повідомлення про помилки допомагають розробникам швидко виявляти та виправляти проблеми у своєму коді, скорочуючи час, витрачений на налагодження.

- Простота використання та документування

Зручність використання компілятора, включаючи простоту його інтерфейсу командного рядка та наявність повної документації, сприяє позитивному досвіду розробника. Добре задокументований компілятор допомагає користувачам зрозуміти його функції та можливості.

- Налаштування рівня оптимізації

Надання параметрів налаштування для рівнів оптимізації дозволяє розробникам збалансувати швидкість компіляції та агресивність оптимізації. Пропонування різних рівнів оптимізації забезпечує гнучкість на основі конкретних потреб проекту.

- Прийняття та внески спільноти

Рівень прийняття в спільноті розробників і отримання внесків можуть свідчити про ефективність компілятора. Активна спільнота припускає, що компілятор задовольняє потреби розробників і має потенціал для постійного вдосконалення.

- Підтримка налагодження

Наявність підтримки налагодження, включаючи можливість генерувати символи налагодження та інтегрувати зі стандартними інструментами налагодження, покращує досвід розробки та налагодження для користувачів.

Оцінка цих показників ефективності разом дає змогу зрозуміти загальну ефективність і якість спеціального компілятора. Баланс таких факторів, як швидкість компіляції, якість коду та взаємодія з користувачем, гарантує, що компілятор відповідає потребам розробників і забезпечує оптимізоване та надійне виведення.

2.4.2 Перевірка теоретичних припущень

Перевірка теоретичних припущень у написанні компілятора передбачає ретельну оцінку, щоб забезпечити узгодженість із встановленими принципами побудови компілятора. Цей процес починається з перевірки того, що специфікація мови, яка охоплює формальну граматику, правила синтаксису та семантику, точно представлена в реалізації компілятора. Правильність лексичного та синтаксичного аналізу перевіряється на основі догматів формальної теорії мови, яка підтверджує, що лексичний і парсер належним чином інтерпретують і структурують вихідний код. Потім семантичний аналіз оцінюється на відповідність теоретичним концепціям, що регулюють перевірку типів, область видимості та правила мови. Проміжне представлення (IR) перевірено на точне відображення високорівневої семантики, що відповідає теоретичним принципам проектування проміжного коду. Фаза генерації коду оцінюється на її відповідність принципам цільової архітектури, гарантуючи, що згенерований машинний код узгоджується з запланованою поведінкою вихідної програми. Методи оптимізації перевіряються на відповідність встановленим принципам, а механізми обробки помилок перевіряються на їхню ефективність у повідомленні про синтаксичні та семантичні помилки. Шляхом підтвердження теоретичної обґрунтованості та вивчення формальних методів верифікації, таких як формальні методи або методи доказів, весь компілятор перевіряється на дотримання основоположних принципів у створенні компілятора, вселяючи впевненість у його правильності, надійності та узгодженості з теоретичними рамками в цій галузі.

2.5 Здобуті уроки та майбутні наслідки

Розробка компілятора — це багатогранна робота, яка вимагає ретельного розгляду різних факторів. Зазвичай цей процес відбувається за ітеративною моделлю розробки, починаючи зі специфікації та дизайну мови програмування, яку він підтримуватиме. Компілятор проходить етапи лексичного та

синтаксичного аналізу, семантичного аналізу та генерації проміжного коду, кожен з яких базується на теорії формальної мови та принципах побудови компілятора.

Під час розробки продуктивність компілятора вимірюється за кількома ключовими показниками. На швидкість компіляції, вирішальний фактор, впливає ефективність лексичного та синтаксичного аналізу, а також методи оптимізації. Якість коду, з точки зору згенерованого машинного коду, безпосередньо впливає на виконання скомпільованої програми. Використання пам'яті, сумісність цільової архітектури та підтримка мовних функцій ще більше підвищують ефективність компілятора.

Перевірка теоретичних припущень є критичним кроком у забезпеченні коректності компілятора. Це передбачає узгодження мовних специфікацій із реалізованими граматичними правилами, підтвердження відповідності фаз синтаксичного аналізу та аналізу формальній теорії мови та перевірку семантичного аналізу на відповідність принципам перевірки типу та визначення обсягу. Проміжне представлення (IR) оцінюється на його точність семантиці високого рівня, а генерація коду перевіряється на відповідність принципам цільової архітектури.

Методи оптимізації, такі як постійне згортання та розгортання циклу, сприяють підвищенню ефективності створеного коду. Ефективні механізми обробки помилок, документація та залучення спільноти додатково впливають на зручність використання компілятора. Ітеративний процес розробки дозволяє вдосконалювати кожну фазу, зосереджуючись на підтримці мовних функцій, налагодженні та поточній оптимізації.

Таким чином, добре спроектований компілятор характеризується дотриманням мовних специфікацій, ефективним аналізом і фазами генерації коду, оптимізацією продуктивності та ретельною перевіркою на теоретичні принципи. Успіх компілятора в кінцевому підсумку вимірюється його здатністю виробляти

правильний, ефективний і надійний машинний код, який узгоджується з передбачуваною семантикою вихідної мови.

Висновок

У дослідженні теоретичних основ і експериментальному впровадженні в рамках розробки компілятора мови Uniq в Rust ця подорож була відзначена багатою взаємодією між теорією та практикою.

Розділ розпочався з глибокого занурення в теоретичні основи, що охоплюють принципи проектування компілятора та теоретичні основи Rust. Теоретична основа забезпечила дорожню карту для розуміння складнощів, притаманних створенню компілятора Uniq на Rust, заклавши основу для наступних практичних експериментів.

Зрештою, справжня міра успіху в цій главі полягає не лише у функціональності компілятора Uniq, але й у розвитку навичок, розумінні теоретичних основ і здатності перевести теорію на практику. Незважаючи на те, що цей шлях був позначений викликами та розвитком кодової бази, він значно сприяв досягненню головної мети — стати досвідченим і універсальним програмістом.

РОЗДІЛ 3. МОДЕЛЬ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПРОГРАМНА РЕАЛІЗАЦІЯ

У цьому розділі розглядається концептуальна основа предметної галузі, надаючи теоретичну модель, яка лежить в основі розробки компілятора мови Uniq мовою Rust. Посилання на репозиторій проекту https://github.com/ihor-tarasov/uniq

Він плавно переходить у практичну сферу, досліджуючи тонкощі впровадження програмного забезпечення, наголошуючи на гармонійній інтеграції теоретичних концепцій у матеріальний код.

3.1 Модель предметної області

3.1.1 Концептуальна основа

Для дослідження було розроблемо мову програмування Uniq, ця мова має бути доволі простою і динамічно типізованою. Для прикладу озглянемо просту реалізацію алгоритму сортування бульбашкою (Рис 3.1)

```
fn bubble_sort(a) {
    let sorted = false;
    while !sorted {
        sorted = true;
        for i = 0, i < len(a) - 1, i++ {
            if a[i] > a[i + 1] {
                let temp = a[i];
                a[i] = a[i + 1];
                a[i + 1] = temp;
                sorted = false;
            }
        }
    }
}
```

Рис 3.1 Функція сортування бульбашкою на мові Uniq

Процес аналізу коду розпочинається з читання вихідного файлу. Так як потрібно використовувати якомога менше ресурсів системи, було вирішено не

читати вихідний код програми написаної на мові Uniq цілком в оперативну пам’ять, а використовувати ітерацію по символах з кешуванням. Після читання іде лексичний аналіз, де вихідний код токенізується для ідентифікації основних елементів мови. На основі лексичного аналізу більшість сучасних мов будують абстрактне синтаксичне дерево, але так як Uniq повинен використовувати якомога менше ресурсів системи було зроблено рішення про відмову від АСД.

Далі слідує семантичний аналіз із забезпеченням дотримання специфічних для мови правил, проведенням перевірки типу та перевіркою обсягу. Цей етап має на меті переконатися, що код відповідає передбачуваному значенню мови, ідентифікуючи та повідомляючи про помилки, які виходять за рамки синтаксичних проблем.

Генерація проміжного коду створює проміжне представлення, яке фіксує суттєву семантику вихідного коду. Це подання служить мостом між вихідним кодом високого рівня та кінцевим машинним кодом.

Потім генерація коду перетворює проміжне представлення в байт код, адаптований до цільової віртуальної машини.

Протягом усього процесу ефективна обробка помилок і чіткі механізми звітування є вирішальними для полегшення налагодження та покращення взаємодії з користувачем. Документація відіграє важливу роль у спрямуванні розробників щодо використання компілятора та розуміння його внутрішніх функцій.

3.1.2 Принципи дизайну

Для того щоб результуючий байт код міг бути виконаний, віртуальній машині потрібно передати Чанк (Chunk). На вхід компілятору приходять байти з вихідного файлу або області пам’яті, де лежить вихідний код, як раніше зазначалося потрібно створити лексичний аналізатор для перетворення байтового представлення вихідного коду у потік токенів або лексем (Рис 3.2).

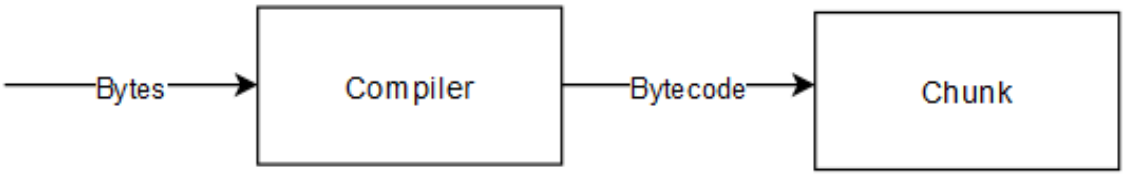

Рис 3.2 Процес перетворення вихідного коду в чанк з байткодом.

На початку чітка та вичерпна специфікація мови є наріжним каменем дизайну. Визначення синтаксису, семантики та особливостей мови програмування створює теоретичну основу для наступних етапів. Ця специфікація дає інформацію про вибір дизайну протягом усього процесу розробки, гарантуючи, що компілятор точно інтерпретує та перекладає передбачувані мовні конструкції.

Лексичний і синтаксичний аналіз, початкові етапи компілятора, дотримуються принципів, що ґрунтуються на формальній теорії мови. Лексер токенізує вихідний код, а аналізатор структурує його ієрархічно, дотримуючись заданих граматичних правил. Дизайн підкреслює модульність і гнучкість для адаптації варіацій мовних конструкцій і підтримки майбутніх мовних розширень.

Семантичний аналіз забезпечує виконання правил мови, проводить перевірку типу та забезпечує правильність визначення області. Проект визначає пріоритетність надійної та розширюваної таблиці символів для керування ідентифікаторами та їхніми атрибутами. Використання чітко визначених структур даних полегшує ефективне представлення та маніпулювання семантичною інформацією.

Генерація проміжного коду спрямована на чітке та виразне представлення семантики вихідного коду. У дизайні наголошується на створенні проміжного представлення , яке забезпечує баланс між простотою та виразністю.

Протягом усього процесу проектування обробка помилок розглядається як критичний компонент. Компілятор призначений для виявлення та звітування про

помилки у змістовний та зручний для користувача спосіб, допомагаючи розробникам у виявленні та виправленні проблем у коді.

Документація вплетена в принципи проектування, надаючи вичерпні вказівки щодо використання компілятора, внутрішніх елементів і точок розширення. Добре задокументований компілятор полегшує розуміння користувачами та заохочує внески спільноти розробників.

Використовується ітераційна модель розробки, що дозволяє поступово покращувати, виправляти помилки та включати розширені функції. Залучення спільноти, чи то через співпрацю з відкритим кодом, чи через відгуки користувачів, сприяє вдосконаленню та розвитку дизайну компілятора.

Підсумовуючи, принципи проектування компілятора включають чітку специфікацію мови, дотримання формальної теорії мови, модульність, гнучкість, ефективні структури даних, проміжне представлення коду, міркування цільової архітектури, методи оптимізації, надійну обробку помилок, ретельну документацію та ітераційну розробку. із залученням громади. Ці принципи разом формують компілятор, який є не тільки правильним і ефективним, але також адаптованим до мінливих потреб мови.

3.2 Архітектура та дизайн програмного забезпечення

3.2.1 Архітектура системи

Системна архітектура компілятора охоплює добре структурований дизайн, який ефективно організовує різні етапи, пов’язані з перекладом вихідного коду високого рівня у виконуваний машинний код. Архітектура має модульний і багаторівневий підхід, що забезпечує легкість розробки, обслуговування та розширення.

Для створення швидкого і компактного по відношенню до ресурсів системи таких як оперативна пам’ять, потрібно читати чатину даних, опрацьовувати її і записувати результат ще перед тим як почати працювати над наступною

частиною. Загальна схема архітектури ПЗ (Рис 3.2) повинна бути мінімалістичною і працювати з даними по мірі їх поступання.

3.2.2 Шаблони та парадигми проектування

Шаблони та парадигми проектування відіграють вирішальну роль у розробці компілятора, надаючи вказівки щодо структурування коду, управління складністю та сприяння ремонтопридатності. У створенні компіляторів зазвичай використовується кілька ключових шаблонів проектування та парадигм.

Шаблон відвідувач часто використовується для проходження складних структур даних, таких як абстрактні синтаксичні дерева (AST), без зміни самої структури. У контексті компілятора цей шаблон застосовується на етапі семантичного аналізу, дозволяючи компілятору відвідувати вузли в AST і виконувати різні операції, такі як перевірка типу або створення таблиці символів, не змінюючи базову структуру дерева.

Паттерн інтерпретатор застосовується на ранніх стадіях компілятора, зокрема під час лексичного та синтаксичного аналізу. Він передбачає створення інтерпретаторів для мовних конструкцій, що дозволяє компілятору інтерпретувати та перетворювати вихідний код у проміжне представлення. Цей шаблон допомагає перетворити конструкції мови високого рівня у форму, яка піддається подальшій оптимізації та фазам генерації коду.

Шаблон спостерігач є цінним у сценаріях, коли зміни в одній частині компілятора потрібно поширювати на інші компоненти. Наприклад, під час фази оптимізації шаблон спостерігача можна використовувати для сповіщення різних модулів оптимізації про зміни в проміжному представленні, запускаючи оновлення та повторні оцінки для підтримки узгодженості.

Шаблон білдер часто використовується для побудови складних структур даних, таких як AST. Під час синтаксичного аналізу шаблон конструктора дозволяє покроково будувати вузли AST, забезпечуючи гнучкість у розміщенні

різних мовних конструкцій. Цей шаблон спрощує процес побудови складних ієрархічних структур, одночасно відокремлюючи процес побудови від фактичного представлення.

Шаблон стратегія є цінним у контексті методів оптимізації, застосованих у середньому кінці компілятора. Різні стратегії оптимізації, такі як постійне згортання або розгортання циклу, можуть бути інкапсульовані як взаємозамінні алгоритми. Компілятор може динамічно вибирати та застосовувати ці стратегії на основі конкретних вимог або характеристик вихідного коду.

Шаблон декоратор можна застосувати для введення додаткових функцій або перетворень до компонентів компілятора. Наприклад, під час генерації коду декоратори можна використовувати для покращення поведінки конкретних модулів генерації коду, не змінюючи їх основні функції. Це сприяє гнучкому та розширюваному дизайну.

З точки зору парадигм, парадигма об’єктно-орієнтованого програмування (ООП) зазвичай застосовується в проектуванні компілятора через її акцент на інкапсуляцію, успадкування та поліморфізм. Ці принципи добре узгоджуються з модульною та ієрархічною природою компонентів компілятора. Крім того, парадигми функціонального програмування, такі як незмінність і функції вищого порядку, все більше інтегруються, особливо на етапах оптимізації, пропонуючи стислі та виразні способи маніпулювання та перетворення даних.

Підсумовуючи, застосування шаблонів проектування, включаючи шаблони Visitor, Interpreter, Observer, Builder, Strategy та Decorator, разом із включенням ООП та парадигм функціонального програмування, сприяє створенню добре структурованого, модульного та зручного для обслуговування дизайну компілятора. Ці шаблони та парадигми допомагають керувати складністю, властивою конструкції компілятора, забезпечуючи гнучкість і розширюваність у міру розвитку компілятора.

3.3 Реалізація коду

3.3.1 Лексичний аналіз

```rust
pub struct Lexer<'a, R> {
    read: &'a mut R,
    current: Option<u8>,
    offset: usize,
}
```

Рис 3.3 Лексер

Як і зазначалося раніше лексичний аналізатор повинен читати байти з вхідного потоку read (Рис 3.3), цей шаблон потім буде обмежений стандартним потоком вводу. Усі байти тимчасово будуть зберігатися в полі current (Рис 3.3) і після приходу нових даних будуть замінені ними.

```rust
pub fn lex(&mut self, buf: &mut Vec<u8>) -> std::io::Result<Token> {
    if let Some(c) = self.current {
        match c {
            b'+' => self.double_plus(),
            b'-' => self.double_minus(),
            b'*' => self.double_asterisk(),
            b'/' => self.double_slash(),
            b'(' => self.single(Token::LeftParen),
            b')' => self.single(Token::RightParen),
            b'{' => self.single(Token::LeftBrace),
            b'}' => self.single(Token::RightBrace),
            b'[' => self.single(Token::LeftBracket),
            b']' => self.single(Token::RightBracket),
            b',' => self.single(Token::Comma),
            b'|' => self.single(Token::VerticalBar),
            b';' => self.single(Token::Semicolon),
            b'!' => self.double_exclamation(),
            b'=' => self.double_equal(),
            b'<' => self.double_less(),
            b'>' => self.double_greater(),
            b'a'..=b'z' | b'A'..=b'Z' | b'_' => self.identifier(buf),
            b'0'..=b'9' => self.integer(buf),
            _ => self.single(Token::Unknown),
        }
    } else {
        Ok(Token::End)
    }
}
```

Рис 3.4 Процес перетворення вхідних вихідного коду на токени.

Для обробки вихідного потоку замало просто зберігати один теперішній байт, якщо прийдеться аналізувати більші лексеми то одного байта не вистачить, тому було створено буфер, в якому зберігатиметься певне накопичення даних для подальшого аналізу.

3.3.2 Аналіз синтаксису

Екземпяр компілятора повинен зберігати в собі чанк та лексер (Рис 3.5) і працювати як будівельник, в який ми можемо складати різні модулі написаної програми на Uniq (Рис 3.6).

```
pub struct Compiler<'a> {
    token: Token,
    range: Range<usize>,
    chunk: Chunk,
    buffer: Vec<u8>,
    address_stack: Vec<u32>,
    source_id: usize,
    function: Function,
    globals: Block,
    natives: &'a Natives,
    cycles: Cycles,
    function_addresses: HashMap<Box<[u8]>, u32>,
}
```

Рис 3.5 Структура компілятора

```rust
pub fn compile<R>(&mut self, source_id: usize, read: &mut R) -> Res
where
    R: std::io::Read,
{
    if self.chunk.len() != 0 {
        self.chunk.pop();
    }

    self.source_id = source_id;
    let mut lexer = Lexer::new(read)?;
    self.lex(&mut lexer)?;

    loop {
        match self.token {
            Token::End => break,
            Token::Fn => self.function(&mut lexer)?,
            Token::Let => {
                self.let_stat(&mut lexer, true, true)?;
                self.chunk.push(opcode::DROP)?;
            }
            _ => {
                self.expression(&mut lexer, true)?;
                self.expect(Token::Semicolon)?;
                self.lex(&mut lexer)?; // Skip ';'.
                self.chunk.push(opcode::DROP)?;
            }
        }
    }

    self.chunk.push(opcode::RET)?;

    Ok(())
}
```

Рис 3.6 Функція для компіляції окремого модуля мови Uniq

Структура компілятора є фундаментальним компонентом реалізації компілятора або інтерпретатора. Він керує різними аспектами процесу компіляції, включаючи обробку маркерів, відстеження діапазону вихідного коду, керування фрагментами байт-коду та загальний стан компіляції.

Структура містить такі поля, як маркер, що представляє поточний маркер; діапазон, що вказує на діапазон вихідного коду, пов’язаний з маркером; фрагмент, що керує скомпільованим байт-кодом; буфер, буфер тимчасового зберігання; address_stack, стек для відстеження адрес під час компіляції; source_id, ідентифікатор вихідного коду; функція, що представляє поточну функцію; глобали, керування глобальними змінними та константами; рідні, посилання на власні функції, доступні компілятору; цикли, відстеження кількості циклів

компіляції; і function_addresses, зіставлення імен функцій з відповідними адресами.

Структура компілятора створюється на початку процесу компіляції. Його полями маніпулюють, коли компілятор обробляє маркери, генерує байт-код і керує станом компіляції. Він служить центральним центром для координації різних аспектів процесу компіляції, включаючи обробку маркерів, генерацію коду, керування адресами, обробку функцій, керування глобальними змінними, власний доступ до функцій, відстеження циклів і відображення адрес функції.

Залежності для структури компілятора включають структури даних Token, Range, Chunk, Function, Block, Natives, Cycles і HashMap.

У практичному використанні екземпляр компілятора створюється з відповідними параметрами ініціалізації, а метод компіляції викликається з вихідним кодом як аргументом для ініціювання процесу компіляції.

Необхідно врахувати, щоб параметри ініціалізації відповідали реалізації компілятора чи інтерпретатора. Крім того, обробка помилок і граничних випадків під час компіляції має вирішальне значення для надійної поведінки.

Таким чином, структура компілятора відіграє вирішальну роль у процесі компіляції, інкапсулюючи стан, полегшуючи генерацію коду та взаємодіючи з іншими компонентами для створення виконуваного коду з вхідних даних джерела.

Структура яка описує токен:

```
#[derive(Clone, Copy, PartialEq)]
pub enum Token {
    Integer,
    Real,
    Identifier,
    True, // 'true'
    False, // 'false'
    If, // 'if'
    Else, // 'else'
    Let, // 'let'
    While, // 'while'
```

```
    For, // 'for'
    Return, // 'return'
    Break, // 'break'
    Continue, // 'continue'
    And, // 'and'
    Or, // 'or'
    Fn, // 'fn'
    VerticalBar, // '|'
    Comma, // ','
    LeftParen, // '('
    RightParen, // ')'
    LeftBrace, // '{'
    RightBrace, // '}'
    LeftBracket, // '['
    RightBracket, // ']'
    Semicolon, // ';'
    Plus, // '+'
    Minus, // '-'
    Asterisk, // '*'
    Slash, // '/'
    PlusEqual, // '+='
    MinusEqual, // '-='
    AsteriskEqual, // '*='
    SlashEqual, // '/='
    Equal, // '='
    Exclamation, // '!'
    PlusPlus, // '++'
    MinusMinus, // '--'
    ExclamationEqual, // '!='
    EqualEqual, // '=='
    Less, // '<'
    Greater, // '>'
    GreaterEqual, // '>='
    LessEqual, // '<='
    Unknown,
    End,
}
```

Перелік Token визначає різні лексичні лексеми, які використовуються на етапі лексичного аналізу компілятора чи інтерпретатора. Кожен варіант представляє певний тип маркера, наприклад ключові слова, оператори та спеціальні символи. Enum отримано за допомогою ознак Clone, Copy і PartialEq, що дозволяє клонувати, копіювати та порівнювати рівність для екземплярів Token

enum.

Варіанти переліку Token включають Integer, Real, Identifier, True, False, ключові слова (If, Else, Let, While, For, Return, Break, Continue, And, Or, Fn), спеціальні символи (VerticalBar, Comma, LeftParen). , RightParen, LeftBrace, RightBrace, LeftBracket, RightBracket, крапка з комою) і оператори (Плюс, Мінус, Зірочка, Слеш, і т.д.). Варіант Unknown представляє невідомий або нерозпізнаний токен, а End представляє кінець вихідного коду вхідних даних.

Екземпляри Token enum використовуються для представлення окремих компонентів вихідного коду під час лексичного аналізу. Цей перелік полегшує ідентифікацію та класифікацію різних синтаксичних елементів, дозволяючи наступним фазам компілятора чи інтерпретатора належним чином обробляти вихідний код.

Підводячи підсумок, Token enum забезпечує структуроване та категоризоване представлення лексичних елементів у вихідному коді, необхідне для розбору та аналізу коду під час процесу компіляції чи інтерпретації.

Приклад компіляції банірних виразів:

```
impl<'a> Compiler<'a> {
    fn binary<R>(&mut self, lexer: &mut Lexer<R>, precedence: u8,
is_global: bool) -> Res
    where
        R: std::io::Read,
    {
        loop {
            let current = get_precedence(self.token);

            if current < precedence {
                break Ok(());
            }

            let opcode = match self.token {
                Token::Plus => opcode::ADD,
                Token::Minus => opcode::SUB,
                Token::Asterisk => opcode::MUL,
                Token::Slash => opcode::DIV,
```

```
                Token::EqualEqual => opcode::EQ,
                Token::ExclamationEqual => opcode::NE,
                Token::Less => opcode::LS,
                Token::Greater => opcode::GR,
                Token::LessEqual => opcode::LE,
                Token::GreaterEqual => opcode::GE,
                _ => unreachable!(),
            };

            let range = self.range.clone();

            self.lex(lexer)?;
            self.unary(lexer, is_global)?;

            let next = get_precedence(self.token);

            if current < next {
                self.binary(lexer, current + 1, is_global)?;
            }

            self.chunk.push_pos(Pos {
                range,
                source_id: self.source_id,
            });

            self.chunk.push(opcode)?;
        }
    }
}
```

Метод `binary` є частиною компілятора або інтерпретатора і відповідає за обробку бінарних операцій під час аналізу або процесу компіляції. Ось опис функції:

Метод `binary` — це рекурсивна функція, яка обробляє двійкові операції у вихідному коді. Він приймає змінне посилання на екземпляр компілятора (self), змінне посилання на лексер (lexer), відповідальний за токенізацію вихідного коду, поточний рівень пріоритету бінарної операції (precedence) і прапорець, який вказує, чи операція знаходиться на глобальна область (is_global).

Метод входить у цикл, який продовжується, доки пріоритет поточного маркера не стане меншим за вказаний пріоритет. Усередині циклу він перевіряє

пріоритет поточного маркера та розриває цикл, якщо він менший за вказаний пріоритет.

Потім метод зіставляє поточний маркер, щоб визначити відповідний код операції для двійкової операції. Поширені двійкові операції включають додавання, віднімання, множення, ділення, перевірку на рівність, перевірку на нерівність, менше, більше, менше або дорівнює та більше або дорівнює. Якщо поточний маркер не відповідає жодному з них, викликається макрос unreachable!(), який вказує, що код не повинен досягати цієї точки.

Після визначення коду операції метод клонує діапазон поточного токена, переміщує лексер до наступного токена та викликає унарний метод для обробки унарних операцій.

Потім метод перевіряє пріоритет наступного токена. Якщо пріоритет наступного токена вищий за поточний пріоритет, він рекурсивно викликає сам себе зі збільшеним рівнем пріоритету.

Нарешті, метод надсилає інформацію про позицію (діапазон та ідентифікатор джерела) до блоку байт-коду та додає визначений код операції до блоку.

Таким чином, метод `binary` рекурсивно обробляє двійкові операції у вихідному коді, обробляючи різні рівні пріоритету та генеруючи байт-код для кожної двійкової операції, що зустрічається.

В бінарних виразах для визначення пріоритетів операторів використовується функція `get_precedence`

```
fn get_precedence(token: Token) -> u8 {
    match token {
        Token::EqualEqual
        | Token::ExclamationEqual
        | Token::Greater
        | Token::Less
        | Token::LessEqual
        | Token::GreaterEqual => 2,
```

```
        Token::Plus | Token::Minus => 3,
        Token::Asterisk | Token::Slash => 4,
        _ => 0,
    }
}
```

Функція get_precedence є службовою функцією в компіляторі або інтерпретаторі. Він призначений для визначення рівня пріоритету заданого токена в контексті бінарних операцій. Функція приймає маркер як аргумент і повертає u8, що представляє рівень пріоритету, пов’язаний із цим маркером.

Функція використовує інструкцію відповідності, конструкцію потоку керування в Rust, щоб зіставляти вхідний токен за зразком із різними варіантами переліку Token. На основі зіставленого варіанту функція призначає певний рівень пріоритету:

Якщо маркер відповідає будь-якому з операторів відношення (EqualEqual, ExclamationEqual, Greater, Less, LessEqual, GreaterEqual), рівень пріоритету встановлюється на 2.

Якщо маркер відповідає будь-якому з операторів додавання (плюс, мінус), рівень пріоритету встановлюється на 3.

Якщо маркер збігається з будь-яким із мультиплікативних операторів (зірочка, коса риска), рівень пріоритету встановлюється на 4.

Для будь-якого іншого маркера, який не відповідає вказаним шаблонам, рівень пріоритету за умовчанням встановлено на 0.

По суті, функція класифікує різні типи токенів у групи на основі рівня пріоритету в контексті бінарних операцій. Ця категоризація допомагає визначити порядок оцінки під час аналізу або компіляції, забезпечуючи правильну обробку виразів.

Загалом функція get_precedence — це стисла й ефективна утиліта для призначення рівнів пріоритету токенам, сприяючи правильному аналізу та генерації байт-коду для двійкових операцій у компіляторі чи інтерпретаторі.

3.3.3 Генерація коду

Для генерації коду чанк повинен реалізувати певні допоміжні методи, а компілятор в свою чергу повинен викликати ці методи зі своїх працюючи рекурсивно, таким чином замість алокацій в кучі, АСД зберігатиметься в програмному стеці і то на момент обробки лишень певного виразу.

Байт-код — це низькорівневе представлення програмного коду, призначеного для ефективного виконання віртуальною машиною. Це проміжне представлення, яке знаходиться між вихідним кодом високого рівня та машинним кодом, що виконується апаратним забезпеченням комп'ютера. У віртуальних машинах байт-код зазвичай представляється як послідовність компактних інструкцій, незалежних від платформи. Ось основні аспекти того, як байт-код зазвичай представлений у віртуальних машинах:

Набір інструкцій:

Байт-код складається з набору інструкцій, які представляють основні операції або обчислення. Кожна інструкція є окремою одиницею роботи, яку може виконати віртуальна машина.

Код операції:

Кожна інструкція байт-коду ідентифікується кодом операції, який є числовим кодом, який представляє певну операцію. Код операції зазвичай є невеликим цілим числом або байтом, який унікально ідентифікує інструкцію.

Операнди:

Багато інструкцій байт-коду містять операнди, які є додатковими фрагментами даних, необхідними для роботи інструкції. Операнди можуть містити константи, адреси пам'яті або інші значення, необхідні для обчислень.

Інструкції фіксованої або змінної довжини:

Інструкції байт-коду можуть мати фіксовану або змінну довжину. Інструкції фіксованої довжини мають заздалегідь визначений розмір, що спрощує процес

декодування. Інструкції змінної довжини забезпечують більш компактне представлення, оскільки операнди можуть бути закодовані різної довжини.

Відображення в пам'яті:

Байт-код зберігається в пам'яті як безперервна послідовність байтів. Кожен байт відповідає коду операції або операнду. Розташування пам’яті таке, що віртуальна машина може легко проходити байт-код, послідовно витягуючи та виконуючи інструкції.

Незалежність від платформи:

Однією з ключових переваг байт-коду є його незалежність від платформи. Оскільки байт-код не є специфічним для будь-якої конкретної апаратної архітектури, його можна виконати на будь-якій машині, яка має сумісну віртуальну машину. Це робить його придатним для кросплатформної розробки.

Компактність і ефективність:

Байт-код розроблено таким чином, щоб бути компактним і ефективним для передачі, завантаження та виконання. Його стисле представлення дозволяє швидше завантажуватись і зменшити споживання пам’яті порівняно з оригінальним вихідним кодом.

Інтерпретація або компіляція:

Віртуальні машини можуть безпосередньо інтерпретувати байт-код або скомпілювати його в машинний код для виконання. Інтерпретація передбачає виконання інструкцій байт-коду одну за одною, тоді як компіляція переводить всю послідовність байт-коду в машинний код перед виконанням.

На основі стека або на основі реєстру:

Віртуальні машини можуть мати різні архітектури, наприклад стекову або регістрову. У стековій архітектурі операнди та результати часто надсилаються та витягуються зі стеку. В архітектурі, заснованій на регістрах, операнди зберігаються в регістрах.

Налагодження та профілювання:

Байт-код часто призначений для підтримки інструментів налагодження та профілювання. Інформація про налагодження може бути вбудована в байт-код, щоб відобразити інструкції байт-коду назад до відповідних рядків вихідного коду, допомагаючи в налагодженні.

Загалом байт-код забезпечує гнучкий і портативний спосіб представлення програмної логіки у формі, яку віртуальна машина легко перекладає на машинний код. Ця абстракція дозволяє виконувати код на різних платформах без необхідності повторної компіляції.

Кожен код операції відповідає певній операції або інструкції, яку може виконати віртуальна машина. Давайте опишемо кожну групу кодів операцій у параграфах:

Перша група кодів операцій складається з 1-байтових інструкцій. Ці коди операцій представляють фундаментальні та стислі операції, які може виконувати віртуальна машина. Вони включають такі інструкції, як RET (повернення), ADD (додавання), SUB (віднімання), MUL (множення) і логічні порівняння, такі як EQ (рівно), NE (не рівно), LS (менше), GR (більше), LE (менше або дорівнює) і GE (більше або дорівнює). Крім того, логічні значення TRUE і FALSE представлені кодами операцій 0x0B і 0x0C відповідно. Інші коди операцій у цій групі включають DROP (скидання значення зі стеку), VOID (представляє недійсність або відсутність операції), GET (отримує значення), SET (встановлює значення), LIST (представляє список або масив), INC (збільшення), DEC (зменшення), NOT (логічне HI) і NEG (заперечення).

Друга група кодів операцій розширюється до 2 байтів, що дозволяє використовувати більш складні або розширені інструкції. Приклади цієї групи включають INT1 (представляє 1-байтове ціле число), CALL (виклик функції), LD1 (завантажує значення), ST1 (зберігає значення), GL1 (завантажує глобальне

значення) і GS1 (зберігає глобальне значення). Значення).

Третя група кодів операцій потребує 3 байтів для представлення більш складних інструкцій. Примітні приклади: INT2 (представляє 2-байтове ціле число), JP2 (умовний перехід), JF2 (умовний перехід, якщо false), LD2 (завантажує 2-байтове значення), ST2 (зберігає 2-байтове значення), JT2 (переходить якщо істина), GL2 (завантажити 2-байтове глобальне значення) і GS2 (зберегти 2-байтове глобальне значення).

Четверта група кодів операцій розширюється до 5 байт, вміщуючи більш розширені інструкції. Примітні приклади включають JP4 (безумовний перехід), JF4 (безумовний перехід, якщо false), LD4 (завантажує 4-байтове значення), ST4 (зберігає 4-байтове значення), PTR (представляє вказівник), NAT (представляє рідний або специфічне для платформи значення), JT4 (безумовний стрибок, якщо істинно), GL4 (завантажити 4-байтове глобальне значення) та GS4 (зберігати 4-байтове глобальне значення).

Кінцева група кодів операцій охоплює 9 байтів, що вказує на потенційну складність відповідних інструкцій. Приклади включають INT8 (представляє 8-байтове ціле число) і REAL (представляє число з плаваючою комою).

Ці коди операцій разом визначають набір інструкцій байт-коду, дозволяючи віртуальній машині інтерпретувати та виконувати широкий спектр операцій під час обробки скомпільованого коду. Кожен код операції відповідає певній поведінці, що дозволяє віртуальній машині виконувати інструкції вищого рівня, закодовані у вихідному коді під час виконання програми.

Нижче наведено опис призначення кожного коду операції в наданому списку. Зауважте, що описи базуються на типових угодах у віртуальних машинах або наборах інструкцій байт-коду:

1 байт (0x00..=0x1F):

RET (0x00):

Опис: вказує на повернення з функції.

ADD (0x01):

Опис: додає два верхніх значення в стеку.

SUB (0x02):

Опис: віднімає найвище значення від другого значення в стеку.

MUL (0x03):

Опис: множить два верхніх значення в стеку.

DIV (0x04):

Опис: ділить перше значення на верхнє значення в стеку.

Еквалайзер (0x05):

Опис: перевіряє, чи рівні два верхні значення в стеку.

NE (0x06):

Опис: перевіряє, чи два верхніх значення в стеку нерівні.

LS (0x07):

Опис: перевіряє, чи друге значення менше верхнього значення в стеку.

GR (0x08):

Опис: перевіряє, чи значення другого до верхнього перевищує верхнє значення в стеку.

LE (0x09):

Опис: перевіряє, чи друге після верхнього значення менше або дорівнює

верхньому значенню в стеку.

GE (0x0A):

Опис: перевіряє, чи друге після верхнього значення більше або дорівнює верхньому значенню в стеку.

TRUE (0x0B):

Опис: надсилає логічне значення true в стек.

FALSE (0x0C):

Опис: надсилає логічне значення false у стек.

DROP (0x0D):

Опис: скидає (вилучає) верхнє значення зі стека.

VOID (0x0E):

Опис: являє собою відсутність операції (недійсність).

GET (0x0F):

Опис: отримує значення з пам’яті або структури даних.

SET (0x10):

Опис: встановлює значення в пам'яті або структурі даних.

LIST (0x11):

Опис: представляє список або масив.

INC (0x12):

Опис: збільшує верхнє значення в стеку.

DEC (0x13):

Опис: зменшує верхнє значення в стеку.

NOT (0x14):

Опис: виконує логічне НЕ для верхнього значення.

NEG (0x15):

Опис: заперечує (змінює знак) верхнього значення.

2-байтові коди операцій (0x20..=0x3F):

INT1 (0x20):

Опис: представляє 1-байтове ціле число.

CALL (0x21):

Опис: викликає функцію.

LD1 (0x22):

Опис: завантажує значення з пам'яті або структури даних (1-байт).

ST1 (0x23):

Опис: Зберігає значення в пам'яті або структурі даних (1-байт).

GL1 (0x24):

Опис: завантажує глобальне значення (1-байт).

GS1 (0x25):

Опис: глобальне зберігання значення (1 байт).

3-байтові коди операцій (0x40..=0x5F):

INT2 (0x40):

Опис: представляє 2-байтове ціле число.

JP2 (0x41):

Опис: безумовний перехід до вказаної адреси (2-байт).

JF2 (0x42):

Опис: Перехід до вказаної адреси, якщо верхнє значення в стеку false (2-байтове).

LD2 (0x43):

Опис: завантажує значення з пам'яті або структури даних (2-байт).

ST2 (0x44):

Опис: зберігає значення в пам'яті або структурі даних (2-байт).

JT2 (0x45):

Опис: переходить до вказаної адреси, якщо верхнє значення в стеку є істинним (2-байт).

GL2 (0x46):

Опис: завантажує глобальне значення (2-байтове).

GS2 (0x47):

Опис: глобальне зберігання значення (2-байт).

5-байтові коди операцій (0x60..=0x7F):

JP4 (0x60):

Опис: безумовний перехід до вказаної адреси (4-байт).

JF4 (0x61):

Опис: Перехід до вказаної адреси, якщо верхнє значення в стеку хибне (4-байтове).

LD4 (0x62):

Опис: завантажує значення з пам'яті або структури даних (4-байт).

ST4 (0x63):

Опис: Зберігає значення в пам'яті або структурі даних (4-байт).

PTR (0x64):

Опис: представляє покажчик.

NAT (0x65):

Опис: представляє нативне або специфічне для платформи значення.

JT4 (0x66):

Опис: переходить до вказаної адреси, якщо верхнє значення в стеку є

істинним (4-байт).

GL4 (0x67):

Опис: завантажує глобальне значення (4-байт).

GS4 (0x68):

Опис: глобальне зберігання значення (4-байт).

9-байтові коди операцій (0x80..=0x9F):

INT8 (0x80):

Опис: представляє 8-байтове ціле число.

REAL (0x81):

Опис: представляє число з плаваючою комою.

Ці описи коду операції забезпечують розуміння основних операцій і функцій, закодованих в інструкціях байт-коду, що дозволяє віртуальній машині виконувати скомпільований код. Конкретна поведінка може відрізнятися залежно від фактичної реалізації віртуальної машини або компілятора.

У віртуальній машині нативні(рідні) значення та функції відіграють вирішальну роль у взаємодії з базовою системою та забезпеченні ефективного доступу до функціональних можливостей конкретної платформи. Ось опис того, як нативні значення та функції можуть бути представлені та викликані у віртуальних машинах:

Нативні значення представляють фундаментальні типи даних або об’єкти, властиві платформі, такі як цілі числа, числа з плаваючою комою, покажчики або специфічні для платформи структури даних.

Вони часто представлені у спосіб, сумісний із внутрішніми типами даних віртуальної машини.

Абстракція даних:

Віртуальні машини забезпечують певний рівень абстракції для представлення рідних значень у спосіб, узгоджений на різних платформах.

Віртуальна машина може визначати власні внутрішні типи для інкапсуляції власних значень, забезпечуючи однакову обробку в межах віртуальної машини.

Нативні функції зазвичай реєструються у віртуальній машині під час ініціалізації. Цей процес реєстрації інформує віртуальну машину про існування та інтерфейс власних функцій.

Реєстрація може містити ім’я функції, типи аргументів, тип повернення та вказівник або посилання на фактичну рідну функцію на головній мові (наприклад, C або C++).

Функціональні підписи:

Віртуальні машини часто визначають стандартний спосіб представлення підпису власних функцій, включаючи кількість і типи аргументів і тип повернення.

Ця інформація є важливою для віртуальної машини для виконання перевірки типу та забезпечення належного виклику власних функцій.

Власні функції можна викликати безпосередньо з байт-коду або проміжного представлення програми.

Віртуальна машина містить інтерфейс для виклику рідного коду, передачі аргументів і отримання повернених значень.

Аргументи власних функцій можуть передаватися через стек, регістри або їх комбінацію, залежно від умов виклику віртуальної машини.

Повернуті значення подібним чином витягуються з нативної функції через призначений механізм.

Віртуальні машини повинні витончено обробляти помилки та винятки, викликані нативними функціями. Це передбачає перетворення власних помилок у винятки або коди помилок віртуальної машини.

Віртуальні машини часто містять механізми для перетворення між рідними типами та типами віртуальних машин. Це може передбачати перетворення цілих чисел, чисел з плаваючою комою або покажчиків на відповідні представлення віртуальної машини.

Якщо віртуальна машина використовує збір сміття (garbage collection), вона повинна враховувати нативні ресурси, які зберігаються рідними значеннями або функціями. Належна інтеграція гарантує послідовне керування ресурсами між віртуальною машиною та рідним кодом.

У багатьох віртуальних машинах рідні функції часто реалізуються такими мовами, як C або C++. Віртуальна машина надає C-сумісний інтерфейс, що дозволяє бездоганно інтегрувати власні функції, написані цими мовами.

Деякі віртуальні машини підтримують зовнішні функціональні інтерфейси, що дозволяє безперебійну інтеграцію з кодом, написаним іншими мовами. Ці інтерфейси часто містять механізми для виклику функцій і передачі даних між віртуальною машиною та зовнішнім кодом.

Віртуальні машини можуть підтримувати реєстрацію функцій зворотного виклику, що дозволяє викликати власний код із коду віртуальної машини та навпаки.

Таким чином, інтеграція нативних значень і функцій у віртуальних машинах передбачає ретельне представлення, реєстрацію та механізми виклику. Мета полягає в тому, щоб забезпечити безперебійний інтерфейс між віртуальною машиною та специфічним для платформи кодом, зберігаючи узгодженість і забезпечуючи належне керування пам’яттю та обробку помилок.

Структура Chunk:

```
pub struct Chunk {
    opcodes: Opcodes,
    ranges: HashMap<u32, Pos>,
}

impl Chunk {
    pub fn new() -> Self {
        Self {
            opcodes: Opcodes::new(),
            ranges: HashMap::new(),
        }
    }

    pub fn set_jf(&mut self, address: u32, value: u32) {
        if value <= 0xFFFF {
            self.opcodes[address] = opcode::JF2;
            (value as u16)
                .to_be_bytes()
                .iter()
                .cloned()
                .enumerate()
                .for_each(|(i, b)| self.opcodes[address + i as u32 + 1] =
b);
        } else {
            self.opcodes[address] = opcode::JF4;
            value
                .to_be_bytes()
                .iter()
                .cloned()
                .enumerate()
                .for_each(|(i, b)| self.opcodes[address + i as u32 + 1] =
b);
        }
    }

    pub fn set_jt(&mut self, address: u32, value: u32) {
        if value <= 0xFFFF {
            self.opcodes[address] = opcode::JT2;
            (value as u16)
                .to_be_bytes()
                .iter()
                .cloned()
                .enumerate()
                .for_each(|(i, b)| self.opcodes[address + i as u32 + 1] =
b);
```

```
        } else {
            self.opcodes[address] = opcode::JT4;
            value
                .to_be_bytes()
                .iter()
                .cloned()
                .enumerate()
                .for_each(|(i, b)| self.opcodes[address + i as u32 + 1] =
b);
        }
    }

    pub fn set_jp(&mut self, address: u32, value: u32) {
        if value <= 0xFFFF {
            self.opcodes[address] = opcode::JP2;
            (value as u16)
                .to_be_bytes()
                .iter()
                .cloned()
                .enumerate()
                .for_each(|(i, b)| self.opcodes[address + i as u32 + 1] =
b);
        } else {
            self.opcodes[address] = opcode::JP4;
            value
                .to_be_bytes()
                .iter()
                .cloned()
                .enumerate()
                .for_each(|(i, b)| self.opcodes[address + i as u32 + 1] =
b);
        }
    }

    pub fn integer(&mut self, value: &[u8]) -> Res {
        let value = std::str::from_utf8(value)?.parse::<u64>()?;
        if value <= 0xFF {
            self.opcodes.push(opcode::INT1)?;
            self.opcodes.push(value as u8)?;
        } else if value <= 0xFFFF {
            self.opcodes.push(opcode::INT2)?;
            self.opcodes.extend((value as u16).to_be_bytes())?;
        } else {
            self.opcodes.push(opcode::INT8)?;
            self.opcodes.extend(value.to_be_bytes())?;
```

```
        }
        Ok(())
    }

    pub fn real(&mut self, value: &[u8]) -> Res {
        let value = std::str::from_utf8(value)?.parse::<f64>()?;
        self.opcodes.push(opcode::REAL)?;
        self.opcodes.extend(value.to_be_bytes())
    }

    pub fn boolean(&mut self, value: bool) -> Res {
        self.opcodes
            .push(if value { opcode::TRUE } else { opcode::FALSE })
    }

    pub fn empty_address(&mut self) -> Res<u32> {
        let address = self.opcodes.len();
        self.opcodes.extend([0; 5])?;
        Ok(address)
    }

    pub fn len(&self) -> u32 {
        self.opcodes.len()
    }

    pub fn push(&mut self, opcode: u8) -> Res {
        self.opcodes.push(opcode)
    }

    pub fn call(&mut self, argc: u8) -> Res {
        self.opcodes.extend([opcode::CALL, argc])
    }

    pub fn store(&mut self, index: u32, is_global: bool) -> Res {
        if index <= u8::MAX as u32 {
            self.opcodes.extend([
                if is_global { opcode::GS1 } else { opcode::ST1 },
                index as u8,
            ])
        } else if index <= 0xFFFF {
            self.opcodes
                .push(if is_global { opcode::GS2 } else { opcode::ST2 })?;
            self.opcodes.extend((index as u16).to_be_bytes())
        } else {
            self.opcodes
```

```
                .push(if is_global { opcode::GS4 } else { opcode::ST4 })?;
            self.opcodes.extend(index.to_le_bytes())
        }
    }

    pub fn load(&mut self, index: u32, is_global: bool) -> Res {
        if index <= u8::MAX as u32 {
            self.opcodes.extend([
                if is_global { opcode::GL1 } else { opcode::LD1 },
                index as u8,
            ])
        } else if index <= 0xFFFF {
            self.opcodes
                .push(if is_global { opcode::GL2 } else { opcode::LD2 })?;
            self.opcodes.extend((index as u16).to_be_bytes())
        } else {
            self.opcodes
                .push(if is_global { opcode::GL4 } else { opcode::LD4 })?;
            self.opcodes.extend(index.to_le_bytes())
        }
    }

    pub fn ptr(&mut self, address: u32) -> Res {
        self.opcodes.push(opcode::PTR)?;
        self.opcodes.extend(address.to_be_bytes())
    }

    pub fn nat(&mut self, index: u32) -> Res {
        self.opcodes.push(opcode::NAT)?;
        self.opcodes.extend(index.to_be_bytes())
    }

    pub fn jump(&mut self, address: u32) -> Res {
        if address <= 0xFFFF {
            self.opcodes.push(opcode::JP2)?;
            self.opcodes.extend((address as u16).to_be_bytes())?;
            self.opcodes.extend([0; 2])
        } else {
            self.opcodes.push(opcode::JP4)?;
            self.opcodes.extend(address.to_be_bytes())
        }
    }

    pub fn start_function(&mut self, argc: u8) -> Res<u32> {
        self.opcodes.push(argc)?;
```

```
        let stack_size_address = self.opcodes.len();
        self.opcodes.extend([0; 4])?;
        Ok(stack_size_address)
    }

    pub fn write_u32_at(&mut self, address: u32, value: u32) {
        value
            .to_be_bytes()
            .iter()
            .cloned()
            .enumerate()
            .for_each(|(i, b)| self.opcodes[address + i as u32] = b);
    }

    pub fn push_pos(&mut self, pos: Pos) {
        self.ranges.insert(self.opcodes.len(), pos);
    }

    pub fn pop(&mut self) {
        self.opcodes.pop();
    }

    pub fn opcodes(&self) -> &[u8] {
        self.opcodes.as_slice()
    }

    pub fn pos(&self, address: u32) -> Option<&Pos> {
        self.ranges.get(&address)
    }
}
```

Структура Chunk є фундаментальним компонентом, призначеним для генерації байт-коду в компіляторі. Ця структура даних інкапсулює основні елементи, необхідні для складання послідовності інструкцій у формі кодів операцій. Основні поля структури Chunk включають коди операцій і діапазони, що полегшує зберігання і керування інструкціями байт-коду та відповідними позиціями, відповідно.

Поле кодів операцій:Поле кодів операцій типу Opcodes служить контейнером для зберігання інструкцій байт-коду. Ці коди операцій представляють операції низького рівня, які компілятор генерує для виконання

програми. Тип Opcodes є спеціальною реалізацією, імовірно, ефективно обробляє зберігання та маніпулювання інструкціями байт-коду.

Поле діапазонів: Поле діапазонів типу HashMap<u32, Pos> відповідає за зіставлення адрес із позиціями (Pos). Це відображення дозволяє асоціювати позиції (наприклад, номери рядків і стовпців вихідного коду) з конкретними адресами байт-кодів. Ця функція є цінною для налагодження та звітування про помилки, оскільки вона дозволяє компілятору відстежувати інструкції байт-коду до їхніх відповідних позицій у вихідному коді.

Структура Chunk надає різні методи взаємодії та маніпулювання байт-кодом:

Методи маніпулювання адресами:

empty_address: резервує адресу в байт-коді для майбутнього використання, повертаючи адресу.

len: отримує поточну довжину байт-коду.

Методи генерації коду операції:

Такі методи, як push, call, store, load, ptr, nat, jump, start_function, write_u32_at та інші, відповідають за створення конкретних кодів операцій і додавання їх до поля кодів операцій. Ці методи вміщують різноманітні операції, включаючи потік керування, виклики функцій, маніпулювання змінними тощо.

Методи позиційної інформації:

push_pos: пов’язує позицію (Pos) із поточною кінцевою позицією в байт-коді.

pop: видаляє останній код операції, що може бути корисним для відновлення помилок.

pos: Отримує позицію, пов’язану з заданою адресою байт-коду.

Структура Chunk демонструє чітко визначений інтерфейс для створення байт-коду, забезпечуючи гнучкість і розширюваність для компілятора. Він включає в себе ефективну обробку різних типів кодів операції, адресації та

позиційної інформації для забезпечення точної генерації байт-коду з можливістю налагодження.

3.4 Тестування та налагодження

Для тестування і налагодження компілятора було створено набір системних тестів. Вони включають в себе кілька функцій сортування написаних на мові Uniq і тест, який генерує випадкові дані і проганяє по цих даних функцію сортування.

Ось одна з функцій швидкого сортування написана мовою Uniq:

```
# A utility function to swap two elements
fn swap(arr, i, j) {
    let temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}

fn partition(arr, low, high) {
    # Choose the pivot
    let pivot = arr[high];

    # Index of smaller element and Indicate
    # the right position of pivot found so far
    let i = (low - 1);

    for j = low, j <= high, j++ {
        # If current element is smaller than the pivot
        if arr[j] < pivot {
            # Increment index of smaller element
            i++;
            swap(arr, i, j);
        }
    }
    swap(arr, i + 1, high);
    i + 1;
}

fn quick_sort(arr, low, high) {
    if low < high {
        # pi is partitioning index, arr[p]
        # is now at right place
        let pi = partition(arr, low, high);
```

```
        # Separately sort elements before
        # partition and after partition
        quick_sort(arr, low, pi - 1);
        quick_sort(arr, pi + 1, high);
    }
}
```

```
     Running `target\release\uniq.exe .\tests\bubble_sort.uniq .\tests\heap_sort.uniq .\tests\merge_sort.uniq .\tests\quicksort.uniq .\tes
ts\main.uniq`
[0.5, 2, 2, 5, 5.6, 7, 9, 10, 12, 12, 34, 45, 100]
[0.5, 2, 2, 5, 5.6, 7, 9, 10, 12, 12, 34, 45, 100]
[0.5, 2, 2, 5, 5.6, 7, 9, 10, 12, 12, 34, 45, 100]
[0.5, 2, 2, 5, 5.6, 7, 9, 10, 12, 12, 34, 45, 100]
```

Рис 3.7 Результат роботи тестових модулів з функціями сортування.

Написання модульних тестів для компілятора є важливим аспектом процесу розробки програмного забезпечення, гарантуючи, що кожен компонент компілятора функціонує належним чином. Модульні тести перевіряють правильність окремих блоків або модулів, забезпечуючи розробникам упевненість у надійності та точності поведінки компілятора. Процес передбачає створення тестових випадків, які охоплюють різні сценарії, граничні випадки та потенційні джерела помилок. Ось детальний опис кроків і міркувань, пов’язаних із написанням модульних тестів для тестування компілятора:

Модильні тести для компілятора повинні бути організовані таким чином, щоб охоплювати різні аспекти, включаючи лексичний аналіз, розбір, семантичний аналіз, генерацію коду та оптимізацію. Кожен основний компонент компілятора повинен мати власний набір тестів.

Висновок

У комплексному дослідженні написання компілятора пройдено ландшафт, який заглибився в теоретичні основи, практичні проблеми та принципи проектування, властиві такій складній справі. Теоретичні основи, засновані на теорії формальної мови та принципах побудови компілятора, підкреслюють

необхідність ретельного підходу до специфікації мови, лексичного та синтаксичного аналізу та семантичного розуміння. Ці теоретичні міркування готують основу для практичних аспектів розробки компілятора, де мова програмування Rust стає вагомим вибором.

У практичній сфері акцент мови Rust на безпеці пам’яті, безкоштовних абстракціях і надійній системі типів добре узгоджується з вимогами побудови компілятора. Використовуючи модель власності Rust, тривалість життя та потужні абстракції, компілятор, реалізований у Rust, може запропонувати баланс між продуктивністю та безпекою. Модель власності, зокрема, сприяє безпеці пам’яті без шкоди для ефективності виконання, критичного аспекту при створенні оптимізованого машинного коду.

Ітеративний процес розробки, який керується принципами та шаблонами проектування, такими як відвідувач, інтерпретатор і спостерігач, забезпечує систематичне та поступове вдосконалення компілятора. Цей підхід визнає складність завдання та дозволяє безперервно вдосконалюватись, реагуючи на мовні специфікації, відгуки користувачів і прогрес у побудові компілятора.

Вибір шаблонів проектування, включаючи стратегічне використання парадигм об’єктно-орієнтованого та функціонального програмування, підкреслює важливість гнучкої та модульної архітектури. Ці шаблони полегшують реалізацію важливих компонентів компілятора, таких як лексичні аналізатори, аналізатори, семантичні аналізатори та генератори коду, кожен з яких сприяє загальному успіху компілятора.

Залучення спільноти та документація стають ключовими елементами, що підвищують надійність і зручність компілятора. Співпраця з відкритим вихідним кодом і відгуки користувачів створюють активну екосистему навколо компілятора, заохочуючи колективні зусилля для вирішення проблем, впровадження нових функцій і гарантуючи, що компілятор залишається

актуальним і адаптованим.

Підсумовуючи, шлях до написання компілятора, особливо мовою Rust, є багатогранним дослідженням, яке охоплює теоретичні основи, проблеми практичного впровадження та міркування щодо дизайну. Об’єднання теоретичної точності, практичної реалізації та продуманих шаблонів проектування в межах виразного й ефективного ландшафту Rust є багатообіцяючою основою для створення компілятора, який відповідає сучасним принципам проектування мови. Еволюція такого компілятора відображає не тільки внутрішню складність мовного перекладу, але й стійкість керованого спільнотою, ітераційного та принципового підходу до побудови компілятора.

ВИСНОВКИ

Під час дослідження, проектування та впровадження компілятора мови Uniq в Rust ця робота подолала заплутані сфери теорії та практики, надаючи повне уявлення про багатогранний ландшафт розробки програмного забезпечення. Кульмінація цих зусиль не лише відображає виклики та успіхи побудови реальної системи, але й підкреслює ширший ріст та еволюцію програміста.

Теоретична основа, закладена в розумінні тонкощів мови Uniq, принципів проектування компілятора та теоретичних конструкцій Rust, заклала основу для продуманого та цілеспрямованого підходу до впровадження.

Окрім конкретного результату компілятора Uniq, ця робота охоплює ширший шлях особистого та професійного зростання. Виклики, з якими зіткнулися, розроблені рішення та отримане розуміння сприяли тонкому розумінню складності розробки програмного забезпечення. Ітеративний характер процесу відображає стійкість, необхідну для навігації в мінливому ландшафті програмування.

У підсумку ця робота означає не лише кінець конкретного проекту, але й крапку в безперервній подорожі до навичок програмування. Отримані уроки, подолані виклики та досліджені теоретичні основи створюють міцну платформу для майбутніх починань.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Crafting Interpreters: https://craftinginterpreters.com/
2. Документація Lua: https://www.lua.org/docs.html
3. Мова програмування C++

Книга, автор Bjarne Stroustrup

https://www.stroustrup.com/4th.html

4. Мова програмування Rust: https://doc.rust-lang.org/book/
5. Мова програмування C

Книга, автор Браян Керніган і Денніс Рітчі

6. Низькорівневе програмування:

Книга, автор Ігор Жирков

7. Writing compilers and interpreters

Книга, автор Рональд Мак

8. Компілятори: принципи, методи та інструменти

Книга, автор Альфред Аго, Джеффрі Ульман, і Раві Сеті

9. Реалізація сучасного компілятора в Java.

Книга, автор Аппель, А. В.

10. Розробка компілятора

Книга, автор Купер, К. Д., Торчзон, Л.

11. Створення компілятора

Книга, автор Фішер К. Н. та Леблан молодший Р. Дж.

12. Розширене проектування та реалізація компілятора

Книга, автор Muchnick, S. S.

13. Компілятор C з можливістю перенацілювання: проектування та реалізація

Книга, автор Купер К. Д. та Сімпсон Л.

14. Компіляція з продовженнями

Книга, автор Аппель, А. В.

15. Побудова компілятора

Книга, автор Вірт, Н.

16. Прагматика мови програмування

Книга, автор Скотт М. Л. та Уейт В. М.

17. Елементи програмування ML

Книга, автор Ульман, Дж. Д.

18. lex & yacc.

Книга, автор Левін, Дж. Р., Мейсон, Т., Браун, Д.

19. Розширене проектування та реалізація компілятора

Книга, автор Muchnick, S. S.

20. Супероптимізатор — погляд на найменшу програму

Книга, автор Massalin, Н.

21. Розробка компілятора: друге видання

Книга, автор Torczon, L., & Cooper, K. D.

22. Реалізація сучасного компілятора в ML

Книга, автор Аппель, А. В., Палсберг, Дж.

23. Прискорений курс зі складання x86 для реверсивних інженерів

Книга, автор Rosen, В. та Finkel, Н.

24. LLVM: структура компіляції для аналізу та трансформації програм протягом усього життя

Книга, автор Lattner, C., & Adve, V.

25. Онлайн-документація та специфікації для LLVM (https://llvm.org/docs/)