**КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ**
**ІМЕНІ ТАРАСА ШЕВЧЕНКА**
Факультет комп'ютерних наук та кібернетики
Кафедра інтелектуальних програмних систем

# Кваліфікаційна робота

**на здобуття освітнього рівня бакалавра**
за спеціальністю 121 Інженерія програмного забезпечення
на тему:

**КОМПІЛЯТОР ПРЕДМЕТНО-ОРІЄНТОВАНОЇ МОВИ ДЛЯ РОБОТИ З МАТРИЦЯМИ**

Виконав студент 4-го курсу
Тимофій ШАРАЄВ _____
(підпис)

Науковий керівник:
доцент, кандидат фізико-математичних наук
Олексій ЧЕНЦОВ _____
(підпис)

Засвідчую, що в цій роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Роботу розглянуто й допущено до захисту
на засіданні кафедри інтелектуальних
програмних систем
«____»_____2021 р.,
протокол № ___
Завідувач кафедри
Олександр ПРОВОТАР _____
(підпис)

Київ – 2021

# РЕФЕРАТ

Кількість сторінок: 41, 7 ілюстрацій, 2 таблиці, 14 джерел посилань

Ключові слова:

КОМПІЛЯТОР, КОНТЕКСТНО-ВІЛЬНА ГРАМАТИКА, ЛЕКСЕР, ЛЕКСИЧНИЙ АНАЛІЗ, МЕТОД РЕКУРСИВНОГО СПУСКУ, МОВА АСЕМБЛЕРА, СИНТАКСИЧНИЙ АНАЛІЗ, МАТРИЦІ, C, ПРОМІЖНА МОВА

Об’єкт дослідження: процес розробки компілятора предметно-орієнтованої мови для роботи з матрицями. Предметом роботи є мова збагачена елементами матричних обчислень.

Мета роботи: на прикладі знайомої предметної області дослідити підходи до збагачення базової програмної мови елементами предметно-орієнтованої мови.

Методи та інструменти розроблення: використані метод рекурсивного спуску для синтаксичного аналізу, GNU Scientific Library, C, Python, GCC

Рекомендації щодо використання роботи: для дослідження інструментів та технологій розробки предметно-орієнтованої мови.

Висновки: було побудовано транслятор мови, збагаченої елементами матричних обчислень,у проміжний код на мові C.

# ЗМІСТ

# ВСТУП

**Оцінка сучасного стану об’єкта розробки.** На сьогоднішній день існує великий попит на програмне забезпечення різних видів, адже кожна взаємодія з машинами та комп’ютерами потребує використання деякого засобу, яким зможе оперувати як людина, так і бути зрозумілим комп’ютеру. Таким засобом є мова програмування — текст, який створює людина, а спеціалізовані програми транслюють цей текст у послідовність команд процесору. Серед мов програмування розрізняють мови високого (C++, Java, Scala та інші) та низького рівня (наприклад, мова асемблера).

Спеціалізовані програми для транслювання мови високого рівня у низькорівневу мову або взагалі двійковий код називають компіляторами. Існує також особливий вид програм, який не транслює, а одразу виконує код програми на мові високого рівня — інтерпретатор. Головна відмінність цих програм є в тому, що компілятори оброблюють весь код програми на машинний код без її виконання, коли інтерпретатор здійснює пооператорну обробку в машинний код та одразу виконує отримані інструкції. Кожного дня створюються нові мови програмування, створюються нові підходи, парадигми до написання програми. Також змінюються архітектури процесорів, на яких виконуються програми в зв’язку з вдосконаленням технологій, або зміни напрямку індустрії, як це сталося з поступовим підйомом чипів процесорів ARM на фоні x86. Тому, для підтримки та вдосконалення існуючого програмного забезпечення та обладнання постійно оновлюються, або навіть створюються компілятори та інтерпретатори нових та вже існуючих мов.

**Актуальність роботи та підстави для її виконання.** Мови програмування є інструментом для створення майже всього програмного забезпечення у світі на цей час. Поява мов програмування більш високого рівня абстракції має свої плюси: зменшення часу на розробку коду наприклад, і дозволяє розробляти складні програми не переймаючись про те, що

відбувається на рівні апаратури, так і мінуси, наприклад помітне уповільнення роботи програми та збільшення ресурсів, які вона займає. Тому для отримання швидкої та невимогливої програми у даній предметній області впирається у наявність гарно сконструйованих та ефективних предметно-орієнтованих мов програмування.

Існують специфічні задачі, вирішення яких ускладнюється або недостатньою продуктивністю коду, який згенерував компілятор, або переобтяжений синтаксис мови, якою вирішують задачі даного типу. В контексті розв'язання задач з використанням матричних розрахунків має сенс використати ефективність коду написаного на мові С. На жаль, її синтаксис деколи стає на заваді швидко створити рішення задачі, необхідної для виконання. Тому дана робота є максимально актуальною для розробників і науковців, яким потрібна ефективність мови С та простота вдосконаленого синтаксису розробленої предметно-орієнтованої мови.

**Об'єкт, методи й засоби дослідження або розроблення.** Об'єктом розробки є компілятор мови, націленої на роботу з матрицями.

Під час розробки був створений лексичний аналізатор, який оброблював код програми, створюючи послідовність лексем на їх основі, синтаксичний аналізатор на основі методу рекурсивного спуску та відповідна для нього контекстно-вільна граматика, який створює абстрактне синтаксичне дерево.

Для породження машинного коду за матричними виразами було використано бібліотеку GNU Scientific Library та інструментарій GNU (GNU toolchain).

**Можливі сфери застосування.** Розроблена система може бути використана для створення програм для обчислень над матрицями. Відкритий код дозволяє робити зміни в ядрі мові та пришвидшувати роботу програми у окремих випадках.

**Мета й завдання роботи.** Мета даної роботи на прикладі знайомої предметної області дослідити підходи до збагачення базової програмної мови

елементами предметно-орієнтованої мови. Були поставлені такі завдання: обрати C-сумісну мову у якості базової мови; специфікувати її синтаксис та описати семантику та реалізувати компілятор з неї на мову асемблера x86; проаналізувати способи збагачення мов програмування; на основі синтаксично-семантичного підходу побудувати транслятор мови, збагаченої елементами матричних обчислень, у проміжний код на мові програмування C.

# РОЗДІЛ 1 СУЧАСНИЙ СТАН РОЗРОБКИ КОМПІЛЯТОРІВ

## 1.1 Історія появи компіляторів

Програмне забезпечення для ранніх комп'ютерів було написано переважно мовою асемблера. Зазвичай простіше використовувати мову високого рівня, а програми, написані за допомогою мови високого рівня, можуть повторно використовуватися на комп’ютерах з різною архітектурою. Незважаючи на це, компілятори не були поширені, оскільки вони генерували код, який не працював так само добре, як спеціально написаний код асемблера. При цьому обмежений об'єм пам'яті ранніх комп'ютерів створив значні технічні обмеження для створення перших компіляторів

Перші компілятори були створені у якості дослідницьких робіт та не були тим, що є компілятором у сучасному розумінні цього слова. Вони мали за собою реалізації таких структурних частин як завантажувач або компонувальник. Першими компіляторами у сучасному розумінні були FORTRAN, розроблений у 1952 році, та ALGOL 5 у 1958 р.

До 1960 р. був створений розширений комілятор Fortran, ALTAC. Першою відомою мовою високого рівня, яка працювала на різних платформах була COBOL. У демонстрації в грудні 1960 р. Програма COBOL була складена і виконана як на UNIVAC II, так і на RCA 501.

## 1.2 Лексери та парсери

### 1.2.1 Лексери

Робота лексера – отримати з послідовності символів послідовність токенів.  Даний процес називається лексингом або токенізацією. Токен – рядок з присвоєним та ідентифікованим сенсом. Він складається з назви токена та за значення токена. Назва токену – категорія лексичної одиниці. Поширені назви токенів:

1. Ідентифікатор – назви змінних, функцій які обирає той, хто пише код;
2. Ключове слово – зарезервовані мовою програмування слова, що вже мають значення;
3. Сепаратори – пунктуаційні символи мови програмування;
4. Оператори – символи, що виконують операції над їх аргументами та повертають результат;
5. Літерали – числа, рядки, логічні значення тощо;
6. Коментарі – невиконувані частини коду.

### 1.2.2 Парсери

Парсинг або синтаксичний аналіз – процес аналізу узгодження рядку символів з правилами деякої формальної граматики.

Парсери можна розбити на 2 основні групи: с низхідним та висхідним синтаксичним розбором.

Парсери з низхідним розбором починають роботу з верхівки синтаксичного дерева та рухаються вниз по дереву, використовуючи продукції граматики. Серед таких парсерів є DCG парсери, LL-парсери, парсери рекурсивного спуску та парсер Ерлі. Хоча LL-парсери та парсери рекурсивного спуску можуть працювати експоненційно довго на деякій граматиці, але мають лінійний час роботи на класі однозначних контекстно-вільних граматиках.

Парсери з висхідним розбором спочатку розпізнають текст на найнижчому рівні синтаксичного дерева та намагаються піднятися вверх за допомогою бектрекінгу або, найчастіше, операцій зсуву та згортки.

Бектрекінг відбувається за допомогою перебором обернених продукцій граматики.

Алгоритм зсуву-згортки відбувається з двох частин. Зсув – додання поточного символу до стеку парсера. Згортка – обернене перетворення

продукції верхівки стека парсера. Найрозповсюджені парсери такого типу – LR-парсери, що будуються на детермінованих контекстно вільних граматиках.

### 1.3 Проміжне представлення

Проміжне представлення (IR) - це структура даних або код, який використовується компілятором або віртуальною машиною для представлення початкового коду. IR розроблений так, щоб над ним можна було просто використовувати для подальших перетворень як оптимізація та створення переклад на іншу мову. Проміжне представлення має бути точним, а саме правильно представляти початковий код з мінімальною втратою інформації. Проміжне представлення може бути у декількох формах: структура даних яка зберігається у пам’яті або ж код, який може бути прочитаний програмою, який ще називаються проміжною мовою.

Будь яка мова яка орієнтована на віртуальну машину може бути розглянута як проміжною мовою. Прикладами таких мов:

- Байт-код Java
- Common Intermediate Language – проміжна мова продуктів компанії Microsoft, який є спільним між усіма компіляторами .NET Framework
- Parrot intermediate representation – проміжна мова для мов з динамічною типізацією
- Прекомпільований код MATLAB

Також у двох програмних засобів для створення компіляторів LLVM та GNU Compiler Collection є власні проміжні представлення, а саме GENERIC та GIMPLE у GNU Compiler Collection і LLVM IR для LLVM. Ці представлення є універсальними при роботі з кожним інструментом в рамках цих систем

### 1.4 Ahead of time компілятори

Компіляція перед виконанням – підхід до компіляції програм мов високого рівня в основі якого лежить перетворення програми в більш низьку

мову програмування до її запуску, що дозволяє зменшити кількість роботи, необхідної під час виконання.

Найчастіше AOT компіляція асоціюється з мовами високого рівня як C, C++, Fortran, Rust або ж з проміжними представленнями як байт-код Java або .NET Framework Common Intermediate Language у відповідний до системи машинний код.

Основною перевагою AOT компіляторів є зменшення накладних витрат часу при виконанні роботи. Деякі мови програмування використовують JIT компілювання. Це компілює проміжний код у машинний код для власного запуску, під час виконання проміжного коду, що може сповільнити роботу програми. AOT компіляція усуває необхідність у цьому кроці, створюючи машинний код перед виконанням, а не під час виконання.

У більшості ситуацій програми та бібліотеки скомпільовані за допомогою AOT можна зменшити витрати середовища виконання та зекономити ресурси машини, на якій вона виконується.

Компілятори AOT можуть виконувати складні та розширені оптимізації коду, що в більшості випадків JIT вважатиметься занадто дорогим. На відміну від цього, AOT, як правило, не може виконати деякі можливі оптимізації в JIT, такі як PGO (Profile-guided optimizations), розповсюдження псевдо-констант або вкладання непрямо-віртуальних функцій. AOT повинен скомпілювати до цільової архітектури, тоді як JIT може скомпілювати код, щоб найкращим чином використовувати фактичний процесор, на якому він працює, навіть роки після випуску програмного забезпечення.

Крім того, JIT компілятори можуть робити припущення щодо коду та створювати деякі оптимізації на їх основі. Ці оптимізації можуть бути прибрані, якщо припущення виявляться невірними. Це призводить до уповільнення роботи програмного забезпечення, поки не будуть використані нові оптимізації. AOT компілятор не має доступу до таких оптимізації, і йому потрібно набагато більше інформації під час етапу компіляції.

### 1.5 Just-in-time компілятори

Компіляція “на льоту” (англійською just-in-time compilation) - це техніка компіляції програм, базована на попередній інтерпретації та оптимізації згенерованого профілю виконання деякої програми. JIT-компіляція є підвидом динамічної компіляції, що дозволяє використання технік адаптивної оптимізації, таких як динамічна рекомпіляція, використання інтерпретатором мікроархітектурних оптимізацій, використання так званого PGO для виконання левової частини оптимізацій (оптимізації за профілем виконання). Техніка JIT-компіляції дуже розповсюджена у компіляторах і рантаймах динамічних мов програмування, оскільки системи компіляції “на льоту” можуть сконструювати пізньо-зв'язуванні типи даних, і, таким чином, гарантувати безпеку. Скомпільований код зберігається в пам'ять та одразу виконується.

Ще однією, досить цікавою особливістю техніки компіляції “на льоту” є розподілення процесу компіляції між декількома компіляторами. Справа у тому, що накладні витрати на більш потужну перекомпіляцію можуть перевищувати витрати на актуальне навантаження віртуальної машини. Іншими словами, досить часто банально немає потреби застосовувати більш потужні оптимізації. Тому, існує розподіл на low-tier, іноді mid-tier та high-tier компілятори. Чим вищий компілятор у ієрархії, тим потужніше оптимізації застосовуються. Критерії застосування того чи іншого компілятора залежать від реалізації конкретної віртуальної машини, але найчастіше це частота визову метода і його розмір у проміжному представленні.

Розглянемо приклади застосувань “компіляторів на льоту” у промислових проектах.

**JVM (HotSpot)**

Класичним прикладом застосування техніки компіляції “на льоту” є віртуальна машина Java, більш відома під назвою HotSpot. Слід зазначити, що назва JVM є лише узагальненням для сімейства віртуальних машин, що

реалізовані згідно з специфікації даної віртуальної машини. Спочатку AOT компілятор javac компілює текст програми (а саме класи) на мові Java у спеціальне проміжне представлення, яке називається байт-кодом. Після цього стартує процес віртуальної машини, який приймає на вхід набір скомпільованих класів у вигляді байт-коду, інтерпретує байт-код і починає вибіркову оптимізацію методів згідно із навантаженням.

Початкові оптимізації виконує lower-tier компілятор C1, також відомий як “клієнтський” компілятор. Даний інструмент орієнтується на швидкість компіляції коду, і не застосовує “важких” оптимізацій у процесі компіляції.

Більш працезатратні оптимізації виконує компілятор C2, задля чого використовується особливий вид проміжного представлення програми Sea Of Nodes. Даний підхід спрощує застосування таких оптимізацій, як згортка та розповсюдження констант, а також видалення “мертвого” коду.

Також застосовується Escape Analysis і наступна за ним оптимізація Scalar Replacement.

**V8**

V8 - це віртуальна машина, що використовується для виконання програми, написаною мовою JavaScript. Застосовується у браузерному двигуну Chromium і його розширеннях, а також у серверних двигунах Node.js і Deno.

У V8 використовується зв’язка “інтерпретатор - low-tier компілятор - high-tier компілятор”. У ролі інтерпретатора виступає Ignition, який виконує перший прохід по байт-коду і генерує перший профіль. Далі, в залежності від характеру навантаження застосовується або low-tier компілятор Sparkplug, або більш потужний Turbofun, який використовує техніки, подібні до технік у C2, але даний компілятор спеціалізується на компіляції динамічної мови програмування, тому має свої особливості, як, наприклад, передбачення типу аргументів функцій.

**Spidermonkey**

Spidermonkey - це ще одна віртуальна машина, що використовується для виконання програми, написаною мовою JavaScript. Застосовується у браузері Mozilla Firefox, а також у модифікаціях серверного двигуна Node.js.

У Spidermonkey використовується зв'язка "інтерпретатор - low-tier компілятор - high-tier компілятор". Інтерпретатор виконує перший прохід по байт-коду і генерує перший профіль. Далі, в залежності від характеру навантаження застосовується або low-tier компілятор Baseline, або більш потужний IonMonkey, який використовує техніки, подібні до технік у Turbofan, але даний компілятор також має свої особливості. Наприклад, він використовує три форми проміжного представлення, які залежать від рівня абстракції щодо поточної фази компіляції. Щодо форми представлення, то це класичний CFG із модифікаціями SSA-форм на різних рівнях абстракції.

**JavaScriptCore (JSC)**

JavaScriptCore - це ще одна віртуальна машина, що використовується для виконання програми, написаною мовою JavaScript. Застосовується у браузері Safari, у всіх браузерах на iOS, а також у програмах, написаних за допомогою React Native.

У JSC використовується зв'язка "інтерпретатор - low-tier компілятор - mid-tier компілятор - high-tier компілятор". Інтерпретатор виконує перший прохід по байт-коду і генерує перший профіль. Далі, в залежності від характеру навантаження застосовується або low-tier компілятор Baseline, або більш потужний DFG, який використовує техніки, подібні до технік у Turbofan.

Якщо є потреба ще більше заоптимізувати програму, то використовується заснований на LLVM компілятор FTL (Faster Than Light). Даний двигун цікавий тим, що використовується аж 3 JIT-компілятора, що дозволяє йому найгнучкішим і найпродуктивнішим компілятором на даний момент.

### 1.6 Класифікація мов програмування

Мови програмування можуть бути розділені на декілька поколінь. Поділ мов почав поширюватись з появою високорівневих мов програмування та мав 3 покоління: мови програмування машинного рівня, мови асемблера, мови програмування високого рівня. Але згодом з'явилася класифікація на 5 поколінь:

1. Мови машинного рівня.
2. Мови асемблера.
3. Мови високого рівня.
4. Предметно-орієнтовані мови.
5. Мови програмування, спеціалізовані на вирішенні проблем із використанням обмежень наданих програмі, а не за допомогою алгоритму розробленим програмістом.

Ціллю роботи є створення саме предметно-орієнтованої мови для роботи з матрицями. Прикладами предметно-орієнтованих мов є HTML як мова розмітки гіпертексту, MATLAB та GNU Octave як мови для матричного програмування, Mathematica та Maple як мови для символьних обчислень та SQL для реляційних баз даних. Всі ці мови є ефективними та зручними у використанні в своїй предметній області. При цьому MATLAB та Octave мають ту саму предметну область та мають схожий між собою синтаксис.

# РОЗДІЛ 2 ПОЧАТКОВА МОВА

У минулому була створена мова що є підмножиною С та орієнтована на операції над цілими числами.

## 2.1 Опис мови

У минулому була створена мова, яка є дуже схожою на мову С з деякими обмеженнями. Мова має лише один тип - цілі числа. Також у мові присутні більшість операцій, які доступні для цілих чисел у мові С та інші конструкції мови як операція умовного переходу та цикли.

## 2.2 Лексичний аналіз

### 2.2.1 Задача лексичного аналізатора

Залишилось визначити правила для лексера. В даній роботі для було вибрано формат мови С. Зважаючи на обмеження лише на цілочислені значення та стандарту С отримуємо наступні обмеження.

### 2.2.2 Ідентифікатори

Для ідентифікаторів є наступні правила:

Ідентифікатор може складатись з літер англійського алфавіту як нижнього так і верхнього регістру, цифр та нижніх підкреслювань «_».

Першим символом має бути або літера або нижнє підкреслення.

Ідентифікатори не мають бути ключовими словами.

Обмежень на довжину немає, але деякі компілятори не сприймають більше 31 символи.

Ключові слова

З типів буде лише реалізація типу *int*.

Інші ключові слова що були реалізовані: do, while, for, if, else, break, continue, return.

### 2.2.3 Сепаратори

У С сепараторами є символи «;», «(», «)», «{», «}». Також сепараторами виступають пробільні символи. Пробільні символи використовуються для відокремлення токенів між собою, «;» для визначення речень, «{» та «}» для блоків.

### 2.2.4 Оператори

+ - * / = == < <= > >= && || ! & | ^ ~ << >>

### 2.2.5 Літерали

Літералами є лише цілі числа.

Отже тепер встановлено список правил для лексера для отримання токенів. Наступний крок – парсинг, або синтаксичний аналіз.

## 2.3 Синтаксичний аналізатор

### 2.3.1 Задача синтаксичного аналізу

В основі парсеру мови

Контекстно-вільна граматика – граматика що задається четвіркою $G = (V, \Sigma, R, S)$, де

$V$ – скінченна множина нетермінальних символів

$\Sigma$ – скінченна множина термінальних символів

$R$ – скінченна множина продукцій граматики $A \longrightarrow \gamma$, де у лівій частині знаходиться єдиний нетермінал, а справа будь які символи граматики.

$S$ – початкові символи граматики

Продукції граматики буде записано у розширеній нотації Бекуса-Наура. Зразок продукції у розширеній нотації Бекуса-Наура :

<нетермінал> ::= вираз

Де нетермінали записуються в кутових дужках, а вираз – послідовність символів які повністю перелічують нетермінал з лівої частини формули. Різні можливі послідовності розділені символом “|”, послідовність символів, яка є не

обов’язковою записується у квадратних дужках, а послідовність виразів, яка може бути 0 або більше разів підряд записується у фігурних дужках.

### 2.3.2 Розробка граматики

Код програми складається з функцій

<program> ::= {<function>}

Функція складається з <type> як типу, який вона повертає, переліку її аргументів та тіла функції, яке складається з блоку команд після цього

<function> ::= <type> <id> “(“ [ <type> <id> { "," <type> <id> } ] ")" “{“ <block> “}”

У початковому ядрі мові існує 2 типи: ціле число та порожній тип. Тип цілого числа використовується при створенні змінних, що вказує на те, що буде зберігатися в даній змінній та при створенні функцій що визначає тип, який буде повертати функція. Пустий тип потрібен для створення процедур, тобто функцій які не повертають значення. При розширенні мови дана продукція може поповнюватися новими типами даних.

<type> ::= “int” | “void”

Блок команд складається з переліку команд у фігурних дужках.

<block> ::= { “{” <block_item> “}” }

Команди блоку можна розділити на твердження та оголошення змінних. Оголошення змінної є окремою від твердження продукцією через різницю у їх семантиці та подальшого створення коду на основі їх продукцій. Також оголошення закінчується крапкою з комою так як оголошення всередині циклу “for” є необов’язковим, але

<block_item> = <statement> | <declaration> “;”

Твердження відповідають за операції над змінними та можуть бути різних типів. Їх можна поділити на повернення типу за допомогою ключового слова “return”, операцію присвоєння та просто вираз. Також твердження може бути блоком, що дозволяє створювати області видимості змінних або ж додавати блок команд до умовних операцій та циклів. Ще один тверджень – умовні

операції, які можуть бути з твердженнями при виконанні умови так і при хибній умові. Твердження можуть бути різними циклами, а саме “for” та “while” з умовою до та після тіла циклу. Останні типи команд – операції контролю циклу а саме “break” та “continue”.

<statement> ::= “return” <expression> “;” |

<id> “=” <expression> “;” |

<expression> “;” |

<block> |

“if” “(“ <expression> “)” <statement> [“else” <statement>] |

“while” “(“ <expression> “)” <statement> |

“for” “(“ [ <declaration> ] “;”[ <expression> ] “;” [ <expression> ]“)” <statement> |

“do” <statement> “while” “(“ <expression> “)” “;”|

“break” “;” |

“continue” “;”

Оголошення змінних складаються з типу змінної, назви змінної та необов’язковою ініціалізацією змінної за допомогою операції присвоєння.

<declaration> = <type> <id> [“=” <expression> ]

Вирази мови відповідають за операції. Кожна з операцій має свій пріоритет. Для реалізації пріоритету в рамках контекстно-вільної граматики на основі рекурсивного спуску операції з більшим пріоритетом будуть глибше у рекурсії. Також не всі операції є асоціативними і виконуються зліва направо тому алгоритм рекурсивного спуску має бути ліворекурсивним, що дозволяє правильно виконувати арифметичні без попереднього перегляду наступних лексем.

У таблиці 1 вказаний пріоритет операцій, при чому операції з вищим пріоритетом мають виконуватись спочатку. Всі операції крім унарних (позначені як (у.)) виконуються зліва направо.

Таблиця 1.1 Пріоритет операцій.

| : ? | \|\| | && | \| | ^ | & | ==, != | <,<=,>,>=,== | <<,>> | +,- | *,/,% | +, -, !, ~ (у.) |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

Далі розписані продукції для кожної з операцій. Так використовуючи рекурсивний спуск спочатку відбувається спуск до операції з нижчим пріоритетом, а операції з вищим пріоритетом обчислюються глибше в рекурсії.

<expression> ::= <conditional_expression>

<conditional_expression> ::= <logical_or_expression> {? <logical_or_expression> : <logical_or_expression>}

<logical_or_expression> ::= <logical_and_expression> {“||” <logical_and_expression>}

<logical_and_expression> ::= < bitwise_or_expression > {“&&” <bitwise_or_expression >}

<bitwise_or_expression> ::= < bitwise_xor_expression > {“|” <bitwise_xor_expression >}

<bitwise_xor_expression> ::= < bitwise_and_expression > {“^” <bitwise_and_expression >}

<bitwise_and_expression> ::= < equality _expression > {“&” < equality _expression >}

<equality _expression> ::= < relational_expression > {(“==” | “!=”) <relational_expression >}

<relational_expression> ::= < bitwise_shift_expression > { (“<” | “<=” | “>” | “>=” ) <bitwise_shift_expression>}

<bitwise_shift_expression> ::= < additive_expression > {(“>>” | “<<”) <additive_expression >}

<additive_expression> ::= < multiplicative_expression > (“+” | “-”) <multiplicative_expression >}

<multiplicative_expression> ::= < unary_expression > {(“*” | “/” | “%”) <unary_expression >}

<unary_expression> ::= (“+”|”-”|”!”|”~”) <unary_expression> |
<bracket_expression >

При такій граматиці операції без явного вказання пріоритету за допомогою дужок будуть виконуватися вірно.

Також можна змінити порядок виконання операцій використовуючи круглі дужки, які мають найвищий пріоритет з усіх виразів

<bracket_expression> ::= <factor> | “(” <expression> “)”

Нетермінал <factor> поєднує у собі термінали що відповідають дані, які використовуються у мові. В даному випадку це ідентифікатори та цілочисельні літерали.

<factor> ::= <id> | <integer>

### 2.3.3 Помилки синтаксичного аналізатора

Дана граматика дозволяє створення деяких послідовностей, які не передбачені мовою і замість того щоб ускладнювати правила граматики та додати обмеження які не передбачені у розширеною нотацією Бекуса-Наура потрібно додати деякі обмеження.

1. Змінні не можуть бути типу void
2. Команди “break” та “continue” можуть бути лише всередині циклів
3. Заборонена повторне оголошення змінної в рамках одної області видимості
4. Заборонене використання змінних, які не були оголошені.

Всі ці випадки будуть видавати граматичну помилку компілятора.

### 2.3.4 Створення абстрактного синтаксичного дерева

Абстрактне синтаксичне дерево буде створене наступним чином: для кожної продукції граматики буде створений вузол дерева. Всі нетеремінальні символи, які є в означенні продукції будуть мати свої вузли та будуть дітьми даного вузла. Для операцій з однаковим пріоритетом вузли розбиваються так, щоб бінарні оператори мали двох дітей а унарні – одного.

## 2.4 Бекенд компілятора

### 2.4.1 Створення коду асемблера

Для того щоб код програми був виконуваним його потрібно перетворити у код асемблера. Код асемблера – мова програмування низького рівня команди якої мають високу відповідність між інструкціями мови та інструкціями машинного коду.

Для перетворення абстрактного синтаксичного дерева у код асемблера для кожного вузла дерева обчислюються його діти після чого виконується відповідна до типу вузла генерація коду. Вузли дерева можна поділити на арифметичні, побітові, логічні операції, операції порівняння, літерали, ідентифікатори а також операції умовного переходу та циклів.

Створення коду асемблера відбувалося за допомогою використання регістрів даних для виконання операцій над числами та стеку для збереження змінних.

### 2.4.2 Арифметичні операції

Генерація коду для унарних операцій відбувається обчисленням її аргументу який є сином вузла у абстрактному синтаксичному дереві.

Для бінарних операцій обчислюється ліва частина виразу, потім права частина виразу і після цього виконується генерація коду для відповідної функції. У коді асемблера при виконанні бінарних функцій у більшості бінарних операцій з цілими числами потрібно перемістити перший аргумент у регістр EAX, другий у регістр ECX після чого виконати відповідну команду асемблера. Для асоціативних операцій регістри у які розміщені аргументи не мають значення.

Операції ділення та остачі є окремим випадком арифметичних операцій. Для них ділене, яке є першим аргументом записується у 2 регістри EDX EAX як одне восьмибайтове число

### 2.4.3 Операції порівняння

Для операцій порівняння в мові асемблера виконується операція cmpl після якої змінюються регістри стану ZF та CF та відбувається занулення регістру EAX. Після цього відповідно до операції та обчислених регістрів стану обчислюється результат.

### 2.4.4 Побітові операції

Побітові операції схожі по створенню коду до арифметичних операцій але мають логічний тип даних як результат. Побітові операції в рамках створеної граматики є асоціативними тому порядок у якому вони поміщаються в регістри EAX та ECX не має значення.

### 2.4.5 Логічні операції

Логічними операціями є операції диз'юнкції та кон'юнкції. Логічні операції мають декілька аргументів при чому для отримання результату перевірка всіх аргументів не обов'язкова. Для отримання 1 для диз'юнкції потрібний 1 аргумент, що дорівнює одиниці, інакше результат буде 0. Аналогічно для кон'юнкції результат буде 0 якщо один з аргументів дорівнює 0 і 1 в іншому випадку.

Для того щоб не перевіряти всі аргументи можна скористатись мітками. Для кожного аргументу логічної операції буде проводитись порівняння з нулем, після чого буде або стрибок до закінчення операції або перевірка наступного аргументу.

### 2.4.6 Операція умовного переходу

Операцію умовного переходу можна розділити на 3 частини. Умова переходу, блок команд при правильності умови та блок команд якщо умова хибна. При цьому лише перший блок  є обов'язковим. Для переходу між блоками використовуються мітки для направлення у правильну ділянку інструкцій яка є відповідною до блоку команд.

### 2.4.7 Цикли

Для всіх циклів на початку має зазначатись мітка початку циклу, щоб переміщуватись до нього у разі виконання умови. Для for мітка має бути після коду першого виразу

При створенні коду умова буде спочатку циклу для for та while. Тому для них після коду умови має бути перехід до кінця циклу при не виконанні умови.

Для циклу for можна зберегти токени для третього виразу з умови циклу та синтаксично аналізувати його після тіла циклу для зменшення постійних переходів між мітками.

В кінці циклів for та while потрібно додати перехід до початку циклу та мітка кінця.

Для циклу do while потрібно додати окрім мітки початку – перехід до нього після коду постумови про її виконанні та мітка кінця циклу для операції break;

На початку обробки циклу парсером також глобально зберігається мітки початку та кінця циклу. При обробці терміналів break та continue відбувається перехід до кінця та початку циклу відповідно.

### 2.4.8 Створення та доступ до змінних

При створенні змінної її назва зберігається в хеш таблиці, а значення зберігаються у пам’яті. Є 2 типи збереження значення: на стеку та у купі. Для збереження на стеку значення змінної переміщюється на стек після чого зберігається його положення у стеку на хеш таблиці. Для збереження у купі виконується системний виклик для виділення пам’яті після чого значення змінної переміщюється до виділеної пам’яті а адреса в купі зберігається в хеш таблиці. В рамках роботи було обрано зберігання значень на стеку через невеликі вимоги до кількості збережених даних та швидкість доступу до даних на стеці. Для збереження даних на стеці потрібно постійно зберігати його розмір та звільнювати його при виході з певної області видимості.

### 2.4.9 Області видимості

Кожний блок створює нову область видимості, в якій можуть бути створені нові змінні, які існуватимуть лише в рамках цього блоку. Також якщо існує змінна з деякою назвою але оголошена у поза даною областю видимості у новому блоці можна створити нову змінну з такою же назвою, яка буде незалежною від тієї, що знаходиться поза блоком. Тому додаємо множину оголошених змінних саме у цій області видимості, для уникнення повтору змінних. При вході у нову область зберігається словник змінних, множина оголошених змінних та лічильник стеку. Після цього множина оголошених змінних стає порожньою. Після виходу з області потрібно відновити стек програми та повернути збережені дані. Для повернення стеку обчислюємо значення $n = saved\_stack\_index - stack\_index$ і оновлюємо верхівку стеку щоб видалити змінні оголошення всередині області видимості

**Приклад**

Приклади створення коду на мові асемблеру на основі коду програми:

```
1  int main()
2  {
3      return 0;
4  }
```

Рис. 2.1 Проста програма, що повертає 0

```
1      .globl main
2  main:
3      push %ebp
4      movl %esp, %ebp
5      movl $0, %eax
6      movl %ebp, %esp
7      pop %ebp
8      ret
```

Рис. 2.2 Породжений код для програми, що повертає 0

У даній програмі спочатку виконуються інструкції, що відповідають початку функції у третьому та четвертому рядках. Потім у регістр EAX записується нуль у 5 рядку. Так як повернення значення означає вихід з функції то виконуються інструкції, що відповідають закінченню функції у шостому та сьомому рядках. Після цього виконується інструкція ret яка повертає значення регістру EAX

```
1   int main()
2   {
3       int a = 5;
4       int ans = 1;
5       for (int i = 0; i < a; i += 1)
6       {
7           ans *= (i+1);
8       }
9       return ans;
10  }
```

Рис. 2.3 Реалізація факторіалу на початковій мові

```
    .globl main
main:
    push %ebp
    movl %esp, %ebp
    movl $5, %eax
    pushl %eax
    movl $1, %eax
    pushl %eax
    movl $0, %eax
    pushl %eax
_loop_begin0:
    movl -12(%ebp), %eax
    push %eax
    movl -4(%ebp), %eax
    pop %ecx
    cmpl %eax, %ecx
    movl $0, %eax
    setl %al
    cmpl $0, %eax
    je _loop_end0
    movl -12(%ebp), %eax
    push %eax
    movl $1, %eax
    pop %ecx
    addl %ecx, %eax
    mull %eax, -8(%ebp)
    addl $0, %esp
    movl $1, %eax
    addl %eax, -12(%ebp)
    jmp _loop_begin0
_loop_end0:
    addl $4, %esp
    movl -8(%ebp), %eax
    movl %ebp, %esp
    pop %ebp
    ret

```

Рис 2.4 Реалізація факторіалу на мові асемблера.

# РОЗДІЛ 3 ЗБАГАЧЕННЯ МОВИ ЕЛЕМЕНТАМИ МАТРИЧНИХ ОБЧИСЛЕНЬ

## 3.1 Загальні міркування щодо збагачення базової мови

Збагачення мови додатковими конструкціями відбувається за допомогою трьох основних підходів: за допомогою бібліотеки розширення, чисто синтаксичний підхід (на основі «синтаксичного цукру») та змішаний синтаксично-семантичний підхід з модифікацією ядра мови.

Збагачення з використанням **бібліотеки предметної області** досягається засобами самої мови. Більшість мов програмування передбачають той чи інший механізм розширення. Такі механізми дозволяють описувати та реалізовувати додатковий інтерфейс. У першу чергу це конструкції для визначення нових типів користувача та функцій для роботи з ними (процедурна абстракція). Причому сам код бібліотеки може бути оформлений у вигляді зовнішніх модулів і для його використання може знадобитися підтримки на етапі зв’язування (linking) та запуску. Даний спосіб дозволяє розширити мову без втручання у її ядро, що відносно полегшує задачу. У деяких випадках (нативні методи) бібліотеки розширення можуть надавати оптимальніший машинний код для роботи у предметній області ніж це дозволяє початкова мова. Однак робота з сутностями предметної області, що виходить за межі можливостей бібліотеки, може виявитися неефективною або неможливою.

**Синтаксичний підхід** полягає у поповненні синтаксису мови деякими новими формами. Причому нічого суттєво нового ці форми не вносять. Вони направлені на створення зручнішої системи запису коду, яка має полегшувати його написання та сприйняття. Привнесений синтаксис прийнято називати синтаксичним цукром. «Синтаксичний цукор» – синтаксичний елемент програми який дублює функціонал вже існуючий елемент або механізм програми, але має інший спосіб описання. Нові синтаксичні елементи повністю зводяться до вже існуючих синтаксичних конструкцій та можуть бути замінені

на них без жодної втрати змісту. Прикладом синтаксичного цукру є оператор розадресації структури -> у мові С. При синтаксичному підході до збагачення мови створюються конструкції мови властиві для предметної області. Вони є розширенням базового синтаксису вихідної мови. При розборі коду розширеної мови у ньому виокремлюються елементи синтаксичного цукру та перетворюються на синтаксичні відповідники вихідної мови. Даний підхід додає синтаксис, що надає додаткову зручність у роботі з сутностями предметної області та є відносно простим (дешевим) у реалізації, оскільки не вимагає міркувань щодо логіки програми, достатньо лише замінювати нові конструкції на відомі. Слабким місце даного підходу є неможливість зміни семантики мови з метою підлаштування її під предметну область. Тому зазвичай при використанні цього підходу доцільно спиратися на вихідні мови більш розвиненої виразності (expressive power).

В основі **синтаксично-семантичного підходу** є зміна як синтаксису так і семантики початкової мови. Цей підхід має як можливість створення синтаксичного цукру, як у попередньому підході, так і можливість зміни семантики, яка може бути націлена на обхід обмежень синтаксису або ж на оптимізацію створеного коду. До прикладів таких розширень є підтримка лямбда-виразів у Java. Зміни у семантиці мови передбачають модифікацію її ядра. Загалом синтаксично-семантичний підхід може вимагати змін як у лексичні структурі мови (нові токени), так і у синтаксичній структурі (граматиці), так і у семантиці -- поява нових конструкцій/сутностей, що може потребувати внесення змін до бекенду компілятора. Позитивною рисою даного підходу є те, що він дозволяє використовувати мов програмування більш низького рівня, для яких необхідний контроль розподілу ресурсів. Недолік підходу полягає в тому, що семантичний аналіз високорівневої мови може бути ускладненим.

У даній роботі надано перевагу синтаксично-семантичному підходу до збагачення базової мови. Конструкції для роботи з матрицями представляються

як надбудова на базовою мовою, утворюючи додатковий верхній рівень. Синтаксис (а також лексика) мови поповнюється новими формами, що відповідають матричним обчисленням. Семантика існуючих та введених операцій на нових видах сутностей (векторах та матрицях) довизначається у термінах (або спираючись) відкритої бібліотеки підпрограм GSL. Мовне виконавче середовище при цьому доповнюється як бібліотечними методами так і інтерфейсним кодом до них.

Для того щоби конкретизувати дану задачу необхідно чітко уявляти спектр дій із матрицями, а також його підмножину, що буде реалізована у роботі. Непогане уявлення про дії з матрицями можна скласти з розділів теорії матриць, а також вивчаючи можливості, які надаються сучасними програмними пакетами орієнтованими на дану предметну область. Це є предметом розгляду наступного підрозділу.

### 3.2 Міркування щодо нотації для матричних обчислень

Так як потрібно додати нову конструкцію для нотації матриць та векторів та операцій над ними варто дослідити як саме вони описані в різних мовах та спеціалізованих пакетах. В основному є два підходи: реалізація через списки, де вектор записується як перелік чисел, а матриці – як перелік векторів, а інший поєднує матриці та вектори в одну структуру. Перший варіант є кращим коли вже є готові структури списків або вже прописана семантика операцій між матрицями та векторами окремо. Другий випадок є більш загальним, але потребує розгляду більшої кількості випадків. При розробці синтаксису був обраний варіант з розділенням матриць та векторів через використання бібліотеки GSL, де на етапі створення коду простіше знати чи є об’єкт скаляром, вектором або матрицею та полегшує розробку мови.

#### 3.2.1 Нотації для матричних обчислень у програмних пакетах

Нотації для матричних обчислень у програмних пакетах зазвичай дуже схожі на синтаксис структур списків або масивів. Наприклад структура матриць

бібліотеки NumPy є схожою на структури списків у Python. Це дозволяє мати хорошу інтегрованість з мовою і не перенасичує синтаксис мови. Такий синтаксис є має свої переваги у даному випадку, адже використовує вже готові конструкції мови, до якої звикли користувачі, зменшує витрати ресурсів на розробку власного синтаксису та робить структури мови легко сумісними з базовими типами мови.

### 3.2.2 Нотації для матричних обчислень у спеціалізованих мовах

У спеціалізованих мовах до матриць та векторів найчастіше підходять як до одного типу, тобто вектор це матриця, одна з розмірностей якої дорівнює одиниці. У мовах Matlab та Octave матриці записані у квадратних дужках, а вектори розділяються між собою крапкою з комою.

## 3.3 Оновлення граматики

### 3.3.1 Оновлення лексики

Оновлена граматика передбачатиме нові типи токенів, які відсутні у базовій лексиці. Тому потрібно оновити лексику новими лексемами, а саме сепараторами “`”, “<”, “>”. Також з’являтимуться ключові слова для унарних операцій “rank”, “det”, “tran”, “inv”, “double” і бінарний оператор “.”.

### 3.3.2 Декларація та ініціалізація векторів та матриць

Оновлення граматики мови орієнтоване на можливість створення на операцій над векторами та матрицями. Так мають бути передбачені конструкції для їх створення. Для цього потрібно додати нові типи та конструкції для побудови матриць. Тип для векторів буде мати назву vector а тип для матриць matrix. Структурно вектор є переліком чисел, а матриця – переліком векторів.

Так продукції для оголошення мають наступну структуру

<vector_declaration> ::= “vector” <id> [“=” <vector_expression>]

<matrix_declaration> ::= “matrix” <id> [“=” <matrix_expression>]

Вони мають таку ж саму структуру як і вже присутня продукція оголошення:

<declaration> = <type> <id> [“=” <expression> ]

Тому можна узагальнити продукції оголошення вектору та матриці до вже існуючої продукції. Для цього потрібно додати нові типи, оновивши продукцію типу.

<type> ::= “int” | “void” | “vector” | “matrix” | “double”

Також є залежність від продукцій <vector_expression> та <matrix_expression>. Вони відповідають за вирази що повертають вектор та матриці відповідно. Так як для узагальнення потрібно щоб вирази векторного та матричного типу були звичайними виразами. Для цього необхідно або додати їх оновивши продукції для виразів або ж перевіряти тип виразу на етапі парсингу. Перший варіант потребує створення нового типу виразу та знову оголошення продукцій операцій між всіма новими типами. Другий варіант дозволяє використовувати вже готову граматику, оновивши новими операціями, а перевірку на тип виразів робити додатково поверх граматики.

### 3.3.3 Операції над векторами

Операції над векторами можна поділити на операції між векторами, операції між вектором та скаляром та унарні операції. Так операціями між векторами є добуток векторів результатом якого є матриця, тобто матричний вираз, операції додавання та віднімання. Операції між вектором та скаляром складаються з операцій додавання, віднімання, множення та ділення вектора на скаляр.

Майже всі бінарні операції вже описані у граматиці, хоча для них треба уточнювати обмеження на типи виразів між якими вони можуть відбуватися. Винятком є операція множення для векторів так як існує скалярне та векторне множення векторів. Нехай для векторного множення буде використовуватись оператор “*” а для скалярного – новий оператор “dot”. Для цього потрібно оновити продукцію множення.

<multiplicative_expression> ::= < unary_expression >{(“*” | “/” | “%”)
<unary_expression >}

перетворюється на

<multiplicative_expression> ::= < unary_expression >

{(“*” | “/” | “%”| “dot”) <unary_expression >}

Унарні операції потрібно оновити додавши правила продукцій. Зараз продукція має вигляд:

<unary_expression> ::= (“+”|”-”|”!”|”~”) <unary_expression> |

<bracket_expression >

Унарні операцій які потрібно додати: “rank”, “det”, “tran”, “inv”. Тому нова продукція має вигляд

<unary_expression> ::= (“+”|”-”|”!”|”~”|“rank”|“det”|“tran”|“inv”)

<unary_expression> | <bracket_expression >

Приклад створення матриці:

vector a = `1, 2, 3`

matrix mtx = <`3,2,1`, a, a>;

При даному розширенні граматики операції над цілими числами та матрицями будуть граматично вірними, але не завжди існують. Тому потрібно описати що саме буде виконуватись при кожній з операцій та обмежити граматику якщо операції між типами не існує. Перелік операцій з новими типами наведено у таблиці.

| | |
|---|---|
| <vector>+<vector> | Перевірка на однакову розмірність і по елементне додавання чисел двох векторів. |
| <vector>+<int><br><int>+<vector> | До кожного елемента вектору додається число. |
| <vector>-<vector> | Перевірка на однакову розмірність і по елементне віднімання чисел двох векторів. |
| <vector>-<int> | До кожного елемента вектору відбувається операція віднімання з числом. |
| <vector>*<vector> | Перевірка на розмірність та векторне множення двох |

| | |
|---|---|
| | векторів. |
| <vector> . <vector> | Перевірка на розмірність та скалярне множення двох векторів |
| <vector>*<int><br><int>*<vector> | До кожного елемента вектору відбувається операція множення з числом. |
| <matrix>+<matrix> | Перевірка на однакову розмірність і по елементне додавання чисел двох матриць. |
| <matrix>+<int><br><int>+<matrix> | До кожного елемента матриці додається число. |
| <matrix>-<matrix> | Перевірка на однакову розмірність і по елементне віднімання чисел двох матриць. |
| <matrix>-<int> | До кожного елемента матриці відбувається операція віднімання з числом. |
| <matrix>*<matrix> | Перевірка на розмірність та множення двох матриць. |
| <matrix>*<int><br><int>*<matrix> | До кожного елемента матриці відбувається операція множення з числом. |
| tran <matrix> | Операція транспонування матриці. |
| det <matrix> | Визначник матриці |
| tran <matrix> | Транспонування матриці |
| inv <matrix> | Обернення матриці |

Таблиця 3.1 Нові операції

**Додаткові помилки граматики**

При даному розширенні граматика створює нові небажані помилки, для усунення яких потрібно ввести нові додаткові правила.

Операція присвоєння працює є вірною змінна, яка присвоюється і вираз є одного типу.

Операції з матрицями для яких не описано правила є невірними.

### 3.4 Зміни у семантиці

Синтаксичні зміни мови також несуть за собою семантичні зміни так як нові конструкції мають власне семантичне навантаження. Було вирішено відмовитись від безпосереднього породження коду асемблера для матричних виразів. Це було зроблено з кількох причин: 1) часові обмеження на виконання роботи; 2) неможливість оптимізації породжуваного коду (без принципової модифікації синтезуючої частини компілятора), зводить нанівець потенційні переваги підходу; 3) негнучкість – внесення модифікацій є дещо проблематичним, важким у супроводженні. Тому матричні конструкції попередньо переводяться у проміжну форму, що є надмножиною базової мови. Серед варіантів проміжного представлення розглядалися C та C++. Вибір C пояснюється тим, що це залишатиме можливість використання у цій якості базової мови (за умови її доопрацювання). З одного боку це розширює спектр платформ, на які може бути перенесена система, оскільки компілятор C є більш доступним. З іншого боку аналіз показує, що при використанні C завдання є цілком здійсненним. Певну семантику роботи з матричними сутностями можна забезпечити додатковим кодом на C, що буде слугувати інтерфейсом до бібліотечних викликів. Звісно, що в C++ деякі можливості були б доступні “з коробки”, але їх цілком можна забезпечити своїми силами без необхідності реалізації всього спектру можливостей C++, зворотнім боком якого є значне підвищення складності компілятора

Операції над векторами та матрицями повинні виконувати відповідні обчислення над ними. Для реалізації був використаний підхід з трансляцією викликів до відкритої бібліотеки GSL на етапі створенні коду.

GNU Scientific Library (GSL) – бібліотека направлена на числові обчислення та має широкий спектр математичних структур. GSL має підтримку підпрограм базової лінійної алгебри (BLAS) - набір основних операцій над векторами та матрицями, які можуть бути використані для створення

оптимізованої функціональності лінійної алгебри вищого рівня. Бібліотека забезпечує рівень низького рівня, який безпосередньо відповідає стандарту BLAS на мові C, (CBLAS), та інтерфейс більш високого рівня для операцій над векторами та матрицями GSL. Є 3 рівня операцій BLAS:

1. Перший рівень: векторні операції
2. Другий рівень: операції між матрицями та векторами
3. Третій рівень: Операції між матрицями

### 3.5 Оновлення створення коду

Основною задачею при оновлені створення коду є нові операції над матрицями та векторами, які є новими для цієї мови. Так як операції над матрицями та векторами тісно зв’язані з базовими синтаксичними формами, то потрібно проводити аналіз всього коду, а не лише частини, де з’являються нові конструкції. Тому потрібно аналізувати та створювати новий код для операцій, де присутні матриці та вектори або ж відтворювати код базової мови.

Код початкової мови має транслюватися у мову C. Це є простою задачею, позаяк базова мова по суті є підмножиною мови, код якої створюється. Оскільки у абстрактному синтаксичному дереві зберігається не кожна текстова деталь початкового коду, а лише структурна його частина, то буде створений код, що не повністю збігатиметься з вихідним. Відмінності стосуватимуться пробільних символи та сепараторів (див.рис.3.1).

Для більш простої роботи з матрицями була використана бібліотека GNU Scientific Library.

При створенні змінних має типів вектору та матриці створюється відповідна структура GNU Scientific Library. Ім’я цієї змінної буде співпадати з тим, що є в початковій мові. При цьому потрібно буде зберігати розміри матриці на рівні мови, щоб операції над матрицями виконувались правильно.

```
int main()
{
    int a = 4;
    int b = 2*4-5*6/2;
    int c;
    if(b > 0)
    {
        c = 3;
    }
    else
    {
        c = a+b;
    }

    int i = 1;
    while(i > 0 || 1)
    {
        i -= 3|2*4-3&3;
    }

    do
    {
        int i = 0;
        i = 3;
    } while(0);
    return 3;
}
```

```
int main (){
int  a  =  4 ;
int  b  =  2  *  4  -  5  *  6  /  2 ;
int  c ;
if( b  >  0 ){
 c  =  3 ;

}
else{
 c  =  a  +  b ;

}
int  i  =  1 ;
while( i  >  0  ||  1 ) {
 i  -=  3  |  2  *  4  -  3  &  3 ;

}
do{
int  i  =  0 ;
 i  =  3 ;

} while( 0 );
return  3 ;
}

```

Рис. 3.1 Порівняння початкового та створеного коду.

При операції над матрицями результат може бути як проміжним обчисленням так і зберігатися у пам’яті після цього. Для проміжних обчислень важливо контролювати щоб їх результат не залишався у пам’яті та звільняти відповідну ділянку.

Для контролю пам’яті при створенні структур внутрішніх типів GSL для матриць та векторів буде створювати обгортка, яка буде містити у собі мітку що вказує на те чи потрібно звільнити для неї пам’ять. На початку мітка вказує на видалення даних змінної, але якщо дані тимчасової змінної присвоюються постійній змінній то значення мітки має вказувати на збереження пам’яті. Після виконання кожної операції над структурами буде виконуватись операція звільнення пам’яті.

Також при створенні коду можливо, що всередині виразів були операції над векторами. Результат таких операцій обчислюється за допомогою відповідної до операції функції a, результат зберігається у тимчасовій змінній, яку простіше передавати як аргумент при генерації коду. Тому для кожного вузла абстрактного синтаксичного дерева спочатку обчислюється код, який створений його дітьми. Для векторного або матричного виразу генерація коду це створення нової змінної. Тому якщо серед дітей є такі вирази то вони додаються на початок коду, створеного у вузлі і зберігається назва тимчасової змінної, щоб був доступ до результату виразу.

Більшість функцій вже реалізована у бібліотеці GNU Scientific Library тому найважчою задачею були задача очистки сміття та контроль над матричними та векторними виразами.

### 3.6 Приклад трансляції невеликого фрагмента програми на збагаченій мові

Прикладом роботи програми буде обчислення скалярного множення двох векторів (рис. 3.2). Для початку створюються два вектори *a=<3,4,5>* та *b=<1,4,6>*. І після цього обчислюється їх скалярний добуток який записується у змінну *res*.

```
1   int main()
2   {
3       vector a = `3, 4, 5`;
4       vector b = `1, 4, 6`;
5
6       double res = a . b;
7   }
```

Рис. 3.2 Приклад коду предметно-орієнтованої мови для скалярного добутку

Після цього отримується наступний проміжний код на C (рис.3.3)

```
1   int main (){
2   vector_view_t vector_view_0 = vector_view_from_array(3, (double [])  3 ,  4 ,  5 ) ;
3   vector_t  a  = vector_from_view(vgp_clone(&vector_view_0.vector), true) ;
4   vector_view_t vector_view_1 = vector_view_from_array(3, (double [])  1 ,  4 ,  6 ) ;
5   vector_t  b  = vector_from_view(vgp_clone(&vector_view_1.vector), true) ;
6   scalar_result_t temp_var1 = vector_ddot( a ,  b ) ;
7   printf("status=%d, result=%g\n", res.status, res.result) ;
8   double  res  = temp_var1.res ;
9
10  }
```

Рис. 3.3 Отриманий код мови C для скалярного добутку

У третього та четвертого рядків відбувається операція створення вектора та його присвоєння до деякого константного значення. Спочатку обробляється права частина операції присвоєння, тому створюється відповідна тимчасова змінна типу vector_view_t. Після цього відбувається створення нової змінної типу vector_t з тим самим ім’ям що і у початковому коді. Значення цієї змінної ініціалізуєтсья як значення виразу у лівій частині операції присвоєння.

Наступний крок - створення коду для шостого рядка. Тут верхніми також є операції ініціалізації на присвоєння. У правій частині виразу знаходиться вираз, що відповідає скалярному добутку. Він обчислюється за допомогою функції vector_ddot результат якої записується у тимчасову скалярну змінну, а результат операції записується в консоль . Після цього відповідно до операцій ініціалізації та присвоєння виконується присвоєння.

Після цього закінчуються інструкції предметної мови та завершується створення команд у головній функції програми.

# ВИСНОВКИ

У даній роботі була поставлена мета створення предметно-орієнтованої мови для операцій над матрицями. Були досліджені методи створення мов програмування та обраний спосіб розширення існуючої мови на основі синтаксично-семантичного підходу з проміжним представленням у вигляді коду на мові С.

Було розроблено базову мову для дій з числами та основними структурними елементами для управління потоком виконання. Побудовано компілятор з цієї мови на мову асемблера для архітектури x86. Було запропоновано збагачення базової мови конструкціями для матричних обчислень. Розширення здійснено на основі синтаксично-семантичного підходу. У зв’язку із складністю реалізації довелось відмовитися від ідеї компіляції матричних виразів безпосередньо у мову асемблера. Замість цього було використано трансляцію матричних виразів у мову програмування С з використанням відкритої бібліотеки GNU Scientific Library для виконання матричних обчислень. Це з одного боку надає можливість породження ефективного коду за рахунок його оптимізації відомими програмними інструментами, а з іншого робить систему доволі гнучкою щодо її доповнення новими діями для роботи з матрицями. Було реалізовано програмну систему, що дозволяє компілювати програми з С-подібної мови, збагаченої виразами фрагменту матричної алгебри, у машинний код.

## ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ


1. Compilers Principles Techniques and Tools (2nd Edition) / [Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D.]. – 2007, с. 109-140 , с.191-259
2. Basic Linear Algebra Subprograms for Fortran Usage / [C. Lawson, R. Hanson, D. Kincaid, F. Krogh]. Vol.: 5 (1979), с. 308–325.
3. An Extended Set of Fortran Basic Linear Algebra Subprograms / [J.J. Dongarra, J. DuCroz, S. Hammarling, R. Hanson] Vol.: 14, No.: 1 (1988), с. 1–32.
4. A Set of Level 3 Basic Linear Algebra Subprograms / [J.J. Dongarra, I. Duff, J. DuCroz, S. Hammarling], Vol.: 16 (1990), с. 1–28.
5. Intel® 64 and IA-32 Architectures Software Developer's Manual [Електронний ресурс] - Intel – Режим доступу до ресурсу: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf
6. x86 and amd64 instruction reference [Електронний ресурс] - felixcloutier – Режим доступу до ресурсу: https://www.felixcloutier.com/x86/
7. Anatomy of a Compiler [Електронний ресурс] – Режим доступу до ресурсу: http://www.cs.man.ac.uk/~pjj/farrell/comp3.html
8. History of compiler construction [Електронний ресурс] – Режим доступу до ресурсу: https://wikimili.com/en/History_of_compiler_construction
9. Ahead-of-time (AOT) compilation [Електронний ресурс] – Google – Режим доступу до ресурсу: https://angular.io/guide/aot-compiler
10. Ahead of Time Compilation (AoT) [Електронний ресурс] – Michael Krimgen – 2020 – Режим доступу до ресурсу: https://www.baeldung.com/ahead-of-time-compilation

11. Just In Time Compilation [Електронний ресурс] – Режим доступу до ресурсу: http://eecs.ucf.edu/~dcm/Teaching/COT4810-Spring2011/Literature/JustInTimeCompilation.pdf
12. Intermediate Representation [Електронний ресурс] – Режим доступу до ресурсу: https://www.sciencedirect.com/topics/computer-science/intermediate-representation
13. Understanding V8's Bytecode [Електронний ресурс] – Режим доступу до ресурсу: https://medium.com/dailyjs/understanding-v8s-bytecode-317d46c94775
14. Profile-guided optimizations [Електронний ресурс] – Режим доступу до ресурсу: https://docs.microsoft.com/en-us/cpp/build/profile-guided-optimizations?view=msvc-160