Хмельницький національний університет
Факультет інформаційних технологій
Кафедра комп’ютерної інженерії та інформаційних систем

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

Галузь знань __________12 – Інформаційні технології

Спеціальність _______123 –Комп’ютерна інженерія

на тему «Метод та засоби створення спеціалізованого двохпрохідного компілятора»

КвРКІ. 160115.20.19 ПЗ

Виконав: студент 2 курсу, група КІ2м-20-1 ________ Кривоносов Ю.А.
Підпис Ініціали, прізвище

Керівник __доктор техн. наук, професор ____ ________ Боровик О.В.
Науковий ступінь, вчене звання Підпис Ініціали, прізвище

До захисту допускаю:
Зав. кафедри КІІС, д.т.н., проф.
Т.О. Говорущенко
__________ 2022 р.

Хмельницький, 2022

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра КОМП'ЮТЕРНОЇ ІНЖЕНЕРІЇ ТА ІНФОРМАЦІЙНИХ СИСТЕМ

Освітній рівень МАГІСТР

Галузь знань 12 ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ

Спеціальність 123 КОМП'ЮТЕРНА ІНЖЕНЕРІЯ

Освітня програма ОСВІТНЬО-НАУКОВА ПРОГРАМА «КОМП'ЮТЕРНА ІНЖЕНЕРІЯ ТА ПРОГРАМУВАННЯ»

ЗАТВЕРДЖУЮ

Зав. кафедри Т.О.Говорущенко

"01" 09 2021 р.

ЗАВДАННЯ
НА ДИПЛОМНИЙ ПРОЕКТ (РОБОТУ)

Кривоносову Юрію Андрійовичу

Прізвище, ім'я, по батькові студента

1. Тема проекту (роботи) Метод та засоби створення спеціалізованого двохпрохідного компілятора

Керівник проекту (роботи) Боровик О.В., д.т.н., професор

Прізвище, ім'я, по батькові, науковий ступінь, вчене звання

Затверджена наказом ректора університету від 06.01.2022 р. № 1

2. Строк подання студентом проекту (роботи) на кафедру 03.05.2022 р.

3. Вихідні дані до проекту (роботи) Завдання на дипломне проектування

4. Зміст пояснювальної записки (перелік питань, які потрібно розробити)

Характеристика предметної області та постановка задачі

Аналіз алгоритмів складових компілятора та його оптимізації

Моделювання системи для розробки компілятора

Розробка програмно-технічного засобу та його тестування

5. Перелік графічного матеріалу (із зазначенням обов'язкових креслень)

6. Консультанти розділів дипломного проекту (роботи)

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Лисенко С.М., професор кафедри КІІС		
Антиплагіат	Нічепорук А.О., доцент кафедри КІІС		

7. Дата видачі завдання «_06_»________09__________2021р.

КАЛЕНДАРНИЙ ПЛАН

№з/п	Назва етапів (розділів) дипломного проекту (роботи)	Термін виконання етапів проекту (роботи)	Примітка
1	Вибір напряму дослідження та узгодження тематики ДРМ з керівником	05.09.2021	виконано
2	Ознайомлення з предметною областю; формулювання мети та задач дослідження; визначення об'єкта та предмета дослідження	05.10.2021	виконано
3	Робота над розділом 1 – аналіз відомих моделей, методів за темою; постановка задачі	05.11.2021	виконано
4	Робота над розділом 2 – розробка моделей для вирішення поставленої задачі	05.12.2021	виконано
5	Робота над науковою статтею	05.01.2022	виконано
6	Робота над розділом 3 – розробка методів для вирішення поставленої задачі	15.02.2022	виконано
7	Робота над розділом 4 – проектування та розробка ПЗ для вирішення поставленої задачі, експериментальна частина	05.04.2022	виконано
8	Оформлення пояснювальної записки згідно вимог	15.04.2022	виконано
9	Попередній захист ДРМ	18.04.2022	виконано
10	Захист ДРМ на засіданні ЕК	До 10.05.2022	

Студент ____________ Ю.А. Кривоносов
Підпис Ініціали, прізвище

Керівник проекту (роботи) ____________ Боровик О.В
Підпис Ініціали, прізвище

РЕФЕРАТ

Тема дипломної роботи: «Метод та засоби створення спеціалізованого двохпрохідного компілятора».

Автор роботи: Кривоносов Юрій Андрійович.

Керівник роботи: д.т.н., професор Боровик О.В.

Пояснювальна записка: 79с., 28 рис., 5 табл., 3 дод., 50 джерел.

КОМПІЛЯТОР, ОПТИМІЗАЦІЯ, ЛЕКСИЧНИЙ АНАЛІЗАТОР, ЛАНЦЮЖКОВИЙ АЛГОРИТМ, СИНТАКСИЧНИЙ АНАЛІЗАТОР, АЛГОРИТМ ЕРЛІ, LOOP FUSION

Об'єктом дослідження є спеціалізований двохпрохідний компілятор.

Предметом дослідження є методи та засоби створення спеціалізованого двохпрохідного компілятора.

Метою дослідження є розробка, оптимізація та модифікація спеціалізованого двохпрохідного компілятора.

Для розв'язання поставлених задач використовувалися методи лексичного аналізу, ланцюжковий метод вирішення колізій, синтаксичного аналізу, семантичного аналізу, модифікований метод «loop fusion».

Наукова новизна отриманих результатів:

– удосконалено метод оптимізації «loop fusion» спеціалізованого двохпрохідного компілятора на основі використання техніки розкрутки циклу;

– набула подальшого розвитку інформаційна технологія створення двохпрохідних компіляторів.

На основі проведених досліджень розроблена архітектура і компоненти спеціалізованого двохпрохідного компілятора.

Практична значимість отриманих результатів полягає у розробленому компіляторі, який дозволяє оптимізувати процес компіляції.

Під час виконання досліджень було опубліковано статтю на тему «Щодо актуальності задачі визначення оптимальних характеристик компілятора» в збірнику наукових праць за матеріалами XIII Всеукраїнської науково-практичної конференції «Актуальні проблеми комп'ютерних наук АПКН-2021». Хмельницький –2021. С.139-142.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ 5
ВСТУП.. 6
1 ХАРАКТЕРИСТИКА ПРЕДМЕТНОЇ ОБЛАСТІ «РОЗРОБКА ТА ВДОСКОНАЛЕННЯ МЕТОДІВ І ЗАСОБІВ СТВОРЕННЯ СПЕЦІАЛІЗОВАНОГО ДВОХПРОХІДНОГО КОМПІЛЯТОРА» 7
1.1 Аналіз предметної області 7
1.2 Аналіз існуючих методів оптимізації компіляції 14
1.3 Основні характеристики інтегрованих середовищ мови програмування C21
1.4 Постановка задачі. Розробка та вдосконалення методів і засобів створення спеціалізованого двохпрохідного компілятора.. 23
1.5 Висновки.. 24
2 АНАЛІЗ АЛГОРИТМІВ СКЛАДОВИХ КОМПІЛЯТОРА ТА ЙОГО ОПТИМІЗАЦІЇ.. 26
2.1 Загальні алгоритми фази аналізу компілятора 26
2.1.1 Алгоритм лексичного аналізу 26
2.1.2 Ланцюжковий алгоритм вирішення колізій хеш-таблиць 32
2.1.3 Алгоритм синтаксичного аналізу та алгоритм Ерлі.. 35
2.1.4 Алгоритм семантичного аналізу 37
2.2 Алгоритм генерації цільового коду.. 42
2.3 Загальна оптимізація компілятора 45
2.3.1 Оптимізація процесу компіляції.. 45
2.3.2 Загальні алгоритми оптимізації компіляторів 45
2.4 Модифікований алгоритм методу “loop fusion” 47
2.5 Висновки 51

3 МОДЕЛЮВАННЯ СИСТЕМИ ДЛЯ РОЗРОБКИ КОМПІЛЯТОРА 52

3.1 Визначення архітектури компілятора 52

3.2 Проектування складових компілятора 54

3.3 Проектування модифікованого методу оптимізації “loop fusion”.............. 60

3.4 Висновки 65

4 РОЗРОБКА ПРОГРАМНО-ТЕХНІЧНОГО ЗАСОБУ 66

4.1 Програмно-технічна реалізація компілятора 66

4.1.1 Виконавчий файл 66

4.1.2 Таблиця символів 67

4.1.3 Лексичний аналізатор 68

4.1.4 Синтаксичний аналізатор 69

4.1.5 Програмна реалізація модифікованого методу «loop fusion» 71

4.2 Обробка та аналіз тестових наборів даних 72

4.3 Оцінка ефективності розроблюваної системи 75

4.4 Висновки 78

ВИСНОВКИ 79

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ 81

ДОДАТОК А Лістинг програмного забезпечення спеціального двохпрохідного компілятора 87

ДОДАТОК Б Публікація 95

ДОДАТОК В Презентація 99

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

IR – проміжне представлення

LISP – сімейство мов програмування

GCC – набір компіляторів

SSA – форма подання графа потоку даних

ЦП – центральний процесор

IDE – інтегроване середовище

GLR - узагальнений висхідний аналізатор

LL – вид парсеру

LR – вид парсеру

FSF – фонд вільного програмного забезпечення

GNU GPL – ліцензія на вільне програмне забезпечення

GNU LGPL – безкоштовна ліцензія на програмне забезпечення

ВСТУП

Через швидкі темпи розвитку галузі інформаційних технологій глибоке розуміння основних принципів роботи різноманітних програмних засобів стає все більш актуальним розробка компіляторів. Велике різноманіття високорівневих програмних мов та фреймворків вимагає створення відповідних програмних засобів компіляції та інтерпретації коду. Архітектура цих засобів напряму впливає на швидкість та безпомилковість їх роботи. Тут швидкість є критично-важливою, адже в процесі розробки програмних продуктів компілювання виконується дуже часто і напряму впливає на його вартість.

В даній роботі розглянуто методи та засоби побудови спеціалізованого двохпрохідного компілятора з модифікованим методом оптимізації «loop fusion» на прикладі мови програмування С. Розробка виконувалась у середовищі Microsoft Visual Studio.

Серед ключових етапів розробки можна виділити:

- аналіз існуючих засобів створення та методів розробки компіляторів;
- аналіз існуючих методів оптимізації компіляторів;
- проектування програмного продукту на основі дослідження;
- програмну реалізацію системи;
- тести програмної реалізації системи;
- аналіз результатів.

Метою роботи є удосконалення методів та засобів розробки компілятора мови програмування, основних методів, моделей та інструментів реалізації. В роботі проведений аналіз предметної області, досліджені основні методи, моделі, інструменти реалізації та розробка компілятора мови програмування.

1 ХАРАКТЕРИСТИКА ПРЕДМЕТНОЇ ОБЛАСТІ «РОЗРОБКА ТА ВДОСКОНАЛЕННЯ МЕТОДІВ І ЗАСОБІВ СТВОРЕННЯ СПЕЦІАЛІЗОВАНОГО ДВОХПРОХІДНОГО КОМПІЛЯТОРА»

1.1 Аналіз предметної області

Компілятор - це комп'ютерна програма, яка перекладає вихідний код з мови програмування високого рівня (наприклад: Perl, Ruby, COBOL) на мову нижчого рівня (особливо мову асемблера та машинний код) для створення виконуваної програми. Процес перетворення мови програмування високого рівня в машинну мову відомий як компіляція. Компілятори не є єдиним мовним процесором, який використовується для перетворення вихідних програм, також існують інтерпретатори.

Інтерпретатор – це комп’ютерне програмне забезпечення яке здійснює пооператорну обробку, перетворення у машинний код та виконання програми або запиту, на відміну від компілятора, який транслює у машинні коди всю програму без її виконання. Процес перекладу впливає на дизайн комп'ютерних мов, що призводить до вибору переваг компіляції чи інтерпретації. Теоретично мова програмування може мати як компілятор, так і інтерпретатор, але на практиці мови програмування, як правило, асоціюються лише з одним компілятором або інтерпретатором.

Компілятор реалізує формальне перетворення з високорівневої вихідної програми в цільову програму низького рівня. Дизайн компілятора може визначати наскрізне рішення або брати участь у визначеній підмножині, яка взаємодіє з іншими інструментами компіляції, наприклад, препроцесорами, асемблерами, компонувальниками. Вимоги до проектування компілятора включають чітко визначені інтерфейси, такі як внутрішні – між компонентами компілятора, і зовнішні – між допоміжними наборами інструментів.

На перших порах на підхід до проектування компілятора безпосередньо впливали складність мови програмування, яка підлягала обробці, досвід осіб, які

розробляли його, і доступні ресурси. Обмеження ресурсів призводило до необхідності проходження вихідного коду більше одного разу.

Компілятор для відносно простої мови, написаної однією людиною, може бути єдиним монолітним програмним забезпеченням. Однак у міру ускладнення мови вихідного коду проект може бути розділений на ряд взаємозалежних фаз. Окремі фази забезпечують покращення дизайну, які зосереджують розробку на функціях у процесі компіляції. Тому компілятори поділяють за кількістю проходів на однопрохідні та багатопрохідні.

Класифікація компіляторів за кількістю проходів ґрунтується на обмеженості апаратних ресурсів комп'ютерів. Оскільки компіляція є досить ресурсозатратним процесом, тому ранні комп'ютери не мали достатньо пам'яті, щоб вмістити все в одну програму, яка б виконувала всю цю роботу. Таким чином, компілятори були розділені на менші програми, кожна з яких виконувала прохід, виконуючи певний необхідний аналіз та переклад.

Можливість компіляції за один прохід класично розглядається як перевага, оскільки вона спрощує роботу по написанню компілятора, а однопрохідні компілятори зазвичай виконують компіляцію швидше, ніж багатопрохідні компілятори. Таким чином, частково через обмеження ресурсів ранніх систем, багато ранніх мов були спеціально розроблені для того, щоб їх можна було зібрати за один прохід (наприклад, Pascal).

Недоліком компіляції за один прохід є те, що неможливо виконати багато складних оптимізацій, необхідних для створення високоякісного коду. Може бути важко підрахувати, скільки точно проходів робить оптимізуючий компілятор. Наприклад, різні етапи оптимізації можуть аналізувати певний вираз багато разів, але інший вираз проаналізувати лише один раз.

Також компілятори класифікують за принципом роботи:

1. Компілятори у зшитий код – компілятор який використовує техніку програмування, в якій код має форму, яка складається із викликів підпрограм. Такий компілятор має мова програмування Forth.

2. Інкрементальні компілятори – вид компіляторів, який компілює лише змінені частини програми, на відміну від звичайних компіляторів, які компілюють усі програмні модулі.

3. Передфінальний компілятор – компілятор, який відразу компілює у вихідну мову без реалізації проміжного коду. Такий компілятор реалізований для мови програмування Prolog.

4. Динамічний компілятор – компілятор, суть якого полягає в тому, що він включає компіляцію під час виконання програми, а не перед її виконанням. Цей процес може складатися з перекладу вихідного коду, але частіше за все це переклад машинного коду, який потім виконується безпосередньо. Система, що реалізує динамічний компілятор, зазвичай безперервно аналізує код, що виконується, і визначає частини коду, де прискорення, отримане від компіляції або перекомпіляції, переважить накладні витрати на компіляцію цього коду. Принциповим представником динамічної компіляції є середовище виконання мови Java.

5. Компілятор зі змінними цілями – компілятор, який відносно легко модифікувати для створення коду для різних архітектур набору інструкцій центрального процесору. Представниками даного типу компіляторів є P-компілятор та набір компіляторів GCC.

6. Компілятор розпаралелення – компілятор, який компілює програму, так щоб був можливий її запуск на паралельній архітектурі.

7. Компілятор компіляторів – інструмент програмування, який створює синтаксиний аналізатор, інтерпретатор або компілятор з певної форми формального опису мови програмування.

8. Векторизувальний – компілятор, який компілює вхідну програму в машинний код комп'ютерів оснащених векторним процесором.

Хоча звичайні типи компіляторів виводить машинний код, також існує багато інших типів компіляторів:

9. Компілятори від джерела до джерела — це тип компілятора, який приймає мову високого рівня як вхідні дані та виводить мову високого рівня.

Наприклад, компілятор автоматичного розпаралелювання часто приймає програму мови високого рівня як вхідні дані, а потім перетворює код і анотує його анотаціями паралельного коду або мовними конструкціями.

10. Апаратні компілятори (також відомі як інструменти синтезу) — це компілятори, вхідні дані яких є мовою опису апаратних засобів, а вихідними — опис конфігурації апаратного забезпечення у вигляді списку мереж або іншим чином.

Незалежно від точної кількості фаз у проекті компілятора, фази можна віднести до одного з трьох етапів (рисунок 1.1).

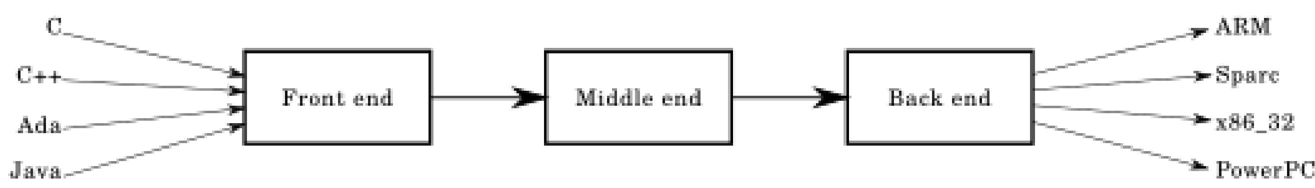


Рисунок 1.1 – Дизайн (структура) компілятора

1. Front-end (початковий етап) – сканує вхідні дані та перевіряє синтаксис і семантику відповідно до певної вихідної мови. Для статично типізованих мов він виконує перевірку типів даних, збираючи інформацію про них. Якщо вхідна програма синтаксично неправильна або має помилку типу, вона генерує повідомлення про помилку та/або попередження, зазвичай ідентифікуючи місце у вихідному коді, де була виявлена проблема. Початковий етап компілятора включає лексичний аналіз, аналіз синтаксису та семантичний аналіз. Розглянемо їх детальніше.

Лексичний аналіз виділяють як окремий етап для спрощення побудови компілятора. Лексичний аналіз – це процес перетворення вхідної послідовності символів у послідовність токенів. Наприклад рядок мовою Pascal $len := 3.14 * r;$ складається з таких токенів:

1. Ідентифікатор “len”.
2. Символ присвоєння “:=”.

3. Числова стала “3,14”.
4. Знак множення “*”.
5. Ідентифікатор “r”.
6. Роздільник операторів “;”.

Токенізація – це процес розмежування та класифікації розділів рядка вхідних символів. Отримані токени потім передаються іншій формі обробки. Процес можна вважати підзавданням розбору вхідних даних. Коли клас токенів представляє більше одного можливого токену, він зберігає достатньо інформації для відтворення оригінального токену, щоб їх можна було використовувати в семантичному аналізі. Синтаксичний аналізатор зазвичай отримує цю інформацію з лексера і зберігає її в абстрактному синтаксичному дереві. Це необхідно для того, щоб уникнути втрати інформації у випадку, коли номери також можуть бути дійсними ідентифікаторами.

Далі йде етап синтаксичний аналізу, який приймає вхідні дані і створює структуру даних – дерево аналізу, абстрактне синтаксичне дерево або іншу ієрархічну структуру, що дає структурне представлення вхідних даних, перевіряючи правильний синтаксис. Синтаксичному аналізатору часто передує окремий лексичний аналізатор, який створює токени з послідовності введених символів. Синтаксичний розбір доповнює шаблони, які виробляють відформатований вихід. Вони можуть застосовуватися до різних доменів, але часто з’являються разом, наприклад, пара scanf/printf або етапи введення та виведення компілятора.

Наприклад, вираз (a + b) * (c – d) може бути поданий у вигляді, що зображено на рисунку 1.2.

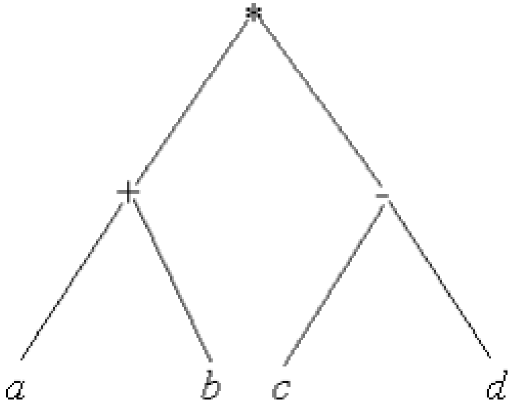


Рисунок 1.2 – Приклад синтаксичного дерева

Абстрактне синтаксичне дерево має кілька властивостей, які сприяють подальшим крокам процесу компіляції:

– абстрактне синтаксичне дерево можна редагувати та доповнювати такою інформацією, як властивості та анотації для кожного елемента, який він містить. Таке редагування та анотація неможливі з вихідним кодом програми, оскільки це означало б його зміну;

– порівняно з вихідним кодом, абстрактне синтаксичне дерево не містить неважливих розділових знаків і роздільників (дужок, крапки з комою);

– абстрактне синтаксичне дерево зазвичай містить додаткову інформацію про програму через послідовні етапи аналізу компілятором. Наприклад, він може зберігати положення кожного елемента у вихідному коді, дозволяючи компілятору друкувати корисні повідомлення про помилки.

Абстрактні синтаксичні дерева необхідні через притаманну природу мов програмування та їх документації. Часто мови за своєю природою неоднозначні. Щоб уникнути цієї неоднозначності, мови програмування часто вказуються як контекстно-вільна граматика (CFG). Однак часто є аспекти мов програмування, які CFG не може висловити, але є частиною мови та задокументовані в її специфікації. Це деталі, які потребують контексту, щоб визначити їх дійсність і поведінку. Наприклад, якщо мова дозволяє оголошувати нові типи, CFG не може передбачити ні назви таких типів, ні спосіб їх використання. Навіть якщо мова має заздалегідь

визначений набір типів, для забезпечення належного використання зазвичай потрібен певний контекст.

Отже, початковий етап (front-end) перетворює вхідну програму в проміжне представлення (IR) для подальшої обробки Middle-end (середній етап). Цей IR зазвичай є представленням програми нижнього рівня щодо вихідного коду.

2. Middle-end (середній етап) виконує оптимізацію IR, яка не залежить від цільової архітектури центрального процесору. Ця незалежність вихідного машинного коду призначена для того, щоб загальні оптимізації могли використовуватися спільно між різними версіями компілятора, що підтримують різні мови та процесори. Прикладами середньої оптимізації є видалення непотрібного (усунення мертвого коду) або недосяжного коду (аналіз досяжності), виявлення та поширення постійних значень (поширення константи), переміщення обчислень у місце, яке рідше виконується (наприклад, поза циклом), або спеціалізація обчислень на основі контексту, що в кінцевому підсумку створює «оптимізований» IR, який використовується сервером.

3. Back-end (кінцевий етап) отримує оптимізований IR від середнього. Він може виконувати більше аналізів, трансформацій та оптимізацій, які є специфічними для цільової архітектури центрального процесору. Кінцевий етап генерує цільовий ассемблерний код, виконуючи розподіл регістра в процесі. Внутрішня частина виконує планування інструкцій, щоб підтримувати зайнятість паралельних блоків виконання, заповнюючи слоти затримки (delay slots). Хоча більшість задач оптимізації є NP-складними, їх вирішення добре розроблені і в даний час реалізовані в компіляторах виробничої якості. Як правило, вихідним результатом серверної частини є машинний код, спеціалізований для конкретного процесора та операційної системи.

Фаза синтезу відповідає за специфічну оптимізацію архітектури ЦП та за генерацію вихідного коду.

Основними етапами даної фази є:

– машинозалежні оптимізації: оптимізації, які залежать від деталей архітектури ЦП, на які націлений компілятор;

– генерація коду: трансформована проміжна мова перекладається на мову виведення, як правило, рідну машинну мову системи. Це включає рішення щодо ресурсів і зберігання, наприклад, рішення, які змінні вписати в регістри та пам’ять, а також вибір і планування відповідних машинних інструкцій разом із пов’язаними з ними режимами адресації. Для полегшення налагодження також може знадобитися згенерувати дані налагодження.

1.2 Аналіз існуючих методів оптимізації компіляції

Оптимізація програми – це процес модифікації комп’ютерної програми таким чином, щоб програма працювала швидше, споживала менше ресурсів або взагалі функціонувала з більш високим рівнем ефективності. Завдання оптимізації може виконуватися автоматично деякими компіляторами мови програмування, навмисно за допомогою програми оптимізації або вручну програмістами, які покроково перебирають вихідний код .і намагаються внести конкретні покращення. Загалом, оптимізація програми виконується з певною метою, оскільки існує дуже мало загальних оптимізацій, які можна зробити до програми, які якимось чином не зменшують оптимізований стан іншої частини програми, тобто програма зазвичай може бути оптимізованими для швидкості або використання ресурсів, але зазвичай не обох. Одне з ускладнень, яке може виникнути з деякими типами оптимізації, полягає в тому, що багато мов програмування високого рівня забезпечують такий великий рівень абстракції між рідним кодом і мовою комп’ютера, що оптимізацію може бути важко або неможливо реалізувати на всіх платформах у будь-яких ситуаціях, особливо з інтерпретованими мовами, які використовують динаміну компіляцію.

Розрізняють низькорівневу та високорівневу оптимізацію. Низькорівнева оптимізація перетворює програму на рівні елементарних команд, наприклад, інструкцій процесора конкретної архітектури. Високорівнева оптимізація здійснюється на рівні структурних елементів програми, таких як модулі, функції, розгалуження, цикли.

Методи, використовувані для оптимізації, можуть бути класифіковані за сферою застосування: вони можуть впливати як на окремий оператор, так і на цілу програму. Локальні методи (зачіпають невелику частину програми) простіше реалізувати, ніж глобальні (застосовувані до всієї програми), але при цьому глобальні методи часто виявляються більш вигідними. Розглянемо найбільш вживані типи оптимізацій:

1. Оптимізація вічка – це тип оптимізації, який досліджує кілька суміжних інструкцій, щоб побачити, чи можна їх замінити однією інструкцією чи коротшою послідовністю інструкцій. Наприклад, множення значення на 2 може бути більш ефективним шляхом зсуву значення вліво або шляхом додавання значення до самого себе.

2. Локальна оптимізація – тип оптимізації, який розглядає лише інформацію, локальну для базового блоку. Оскільки базові блоки не мають потоку керування, ці оптимізації потребують дуже малого аналізу, заощаджуючи час і зменшуючи вимоги до зберігання, але це також означає, що інформація не зберігається при переходах.

3. Глобальна оптимізація –це тип оптимізації, який діє на цілі функції. Це дає їм більше інформації для роботи, але часто робить необхідними дорогі обчислення. У найгіршому випадку необхідно робити припущення, коли відбуваються виклики функцій або доступ до глобальних змінних, оскільки інформації про них мало. Вони також називаються «внутрішньопроцедурними методами».

4. Оптимізація циклу – це тип оптимізації, який діє на оператори, які утворюють цикл, такі як цикл for, наприклад рух коду, інваріантний до циклу. Оптимізація циклу може мати значний вплив, оскільки багато програм проводять великий відсоток свого часу всередині циклів.

5. Міжпроцедурна оптимізація – це тип оптимізації, який аналізує весь вихідний код програми. Більша кількість вилученої інформації означає, що оптимізації можуть бути більш ефективними, ніж тоді, коли вони мають доступ

лише до локальної інформації, тобто в межах однієї функції. Такий тип оптимізації також може дозволити впроваджувати нові методи.

6. Оптимізація машинного коду – це тип оптимізації, який аналізує зображення виконуваного завдання програми після пов’язування всього виконуваного машинного коду. Деякі методи, які можна застосувати в більш обмеженому обсязі, наприклад макростиснення, яке економить простір за рахунок згортання загальних послідовностей інструкцій, є більш ефективними, коли для аналізу доступне все зображення виконуваного завдання.

Важливою концепцією оптимізації програми є ідея, що оптимізація зазвичай має певну ціну. Одним із прикладів цього є те, що, коли фрагмент коду оптимізовано для більш швидкої роботи, збільшення швидкості може стати причиною читабельності коду, використання пам’яті, гнучкості програми чи ряду інших витрат. Це означає, що оптимізація програми має бути цілеспрямованим процесом, з наміром зробити один аспект програми ефективнішим, водночас готовий пожертвувати ефективністю інших аспектів.

На різних етапах розробки програми можна виконувати різні види оптимізації програми. Під час проектування можна зробити широку оптимізацію, переконавшись, що програма працює ефективно. Під час роботи з фактичним вихідним кодом оптимізація може включати забезпечення відсутності сторонніх команд, повторюваних викликів або погано написаних функцій. Під час компіляції багато оптимізацій автоматично виконується компілятором, і програміст може керуватися використанням різних перемикачів або директив компілятора.

Автоматична оптимізація, як це відбувається з компілятором або програмою оптимізації виділення, часто може включати прийоми, які є занадто складними, щоб бути практичними для програмістів. Це може включати переміщення інструкцій у програмі, щоб вони виконувались не в порядку, написаному спочатку, але більш ефективним для процесора способом. Це також може включати навмисне зміщення ресурсів, таких як блоки пам’яті, щоб отримати до них швидший доступ. Більшість програм оптимізації відбувається автоматично на рівні компілятора.

Розглянемо конкретні методи оптимізації, які в першу чергу, призначені для роботи з циклами. До числа таких відносять:

1. Індукційний аналіз змінних – якщо змінна в циклі є простою лінійною функцією змінної індексу, наприклад $j := 4 * i + 1$, її можна відповідним чином оновлювати щоразу, коли змінна циклу змінюється. Це збільшує ефективність коду, а також може дозволити визначенням змінної індексу стати мертвим кодом.

2. Поділ циклу на частини – метод намагається розбити цикл на кілька циклів в одному діапазоні індексів, де кожен з них займає лише частину циклу.

3. Об'єднання циклів – метод, який зменшує накладні витрати циклу. Якщо два цикли виконують однакову кількість ітерацій та не посилаються один на одного, то їх можна об’єднати.

4. Інверсія циклу – метод, який змінює стандартний цикл while на цикл do-while, обгорнутий умовним if, зменшуючи кількість ітерацій на дві, для випадків коли цикл виконується.

5. Розщеплення циклу – метод, який намагається спростити цикл або усунути залежності, розбиваючи його на кілька циклів, які мають однакові тіла, але повторюють різні суміжні частини діапазону індексів.

При оптимізації компіляторів також використовується оптимізація потоку даних, яка заснована на аналізі потоку даних. Оптимізація аналізі потоку даних, насамперед залежить від того, як певні властивості даних поширюються керуючими ребрами в графі потоку керування. Деякі з них включають:

1. Видалення загального підвиразу – оптимізація потоку даних, яка шукає екземпляри однакових виразів і аналізує можливість заміни їх однією змінною, що містить обчислене значення.

2. Постійне згортання і розмноження – оптимізація потоку даних, яка замінює вирази, що складаються з констант їх кінцевим значенням під час компіляції, а не обчислення під час виконання програми.

3. Ліквідація мертвих змінних – оптимізація потоку даних, яка видаляє присвоєння змінних, які згодом не зчитуються, або тому що життя змінної закінчується, або через наступне присвоєння, яке перезапише перше значення.

4. Класифікація псевдонімів та аналіз покажчиків – за наявності покажчиків важко зробити будь-яку оптимізацію, оскільки потенційно будь-яка змінна може бути змінена під час призначення місця в пам'яті.

Також при оптимізації компіляторів використовують SSA-форму і оптимізації, заснованій на ній. SSA – це форма подання графа потоку даних, у якому кожна змінна присвоює значення тільки один раз.

Ці оптимізації призначені для виконання після перетворення програми в спеціальну форму під назвою Static Single Assignment, в якій кожній змінній призначається лише одне місце. Хоча деякі функціонують без SSA, вони найбільш ефективні з SSA. Багато оптимізацій, перерахованих в інших розділах, також мають переваги без особливих змін, таких як розподіл реєстрів.

До SSA-оптимізації відносять такі методи:

1. Глобальна нумерація значень – усуває надмірність, будуючи графік значень програми, а потім визначаючи, які значення обчислюються за допомогою еквівалентних виразів. Метод здатний ідентифікувати деяку надлишковість, яку не може виключити загальний підвираз, і навпаки.

2. Розріджене поширення умовної константи – метод одночасно видаляє деякі види мертвого коду та поширює константи по всій програмі. Більше того, він може знайти більше постійних значень і таким чином, більше можливостей для вдосконалення, ніж окремо застосовуючи видалення мертвого коду та постійне поширення в будь-якому порядку або в будь-якій кількості повторів.

У теорії компілятора видалення мертвого коду – це оптимізація компілятора для видалення коду, яка не впливає на результати програми. Видалення такого коду має кілька переваг: воно зменшує розмір програми, що є важливим фактором у деяких контекстах, і дозволяє запущеній програмі уникнути виконання невідповідних операцій, що скорочує час її виконання. Він також може забезпечити подальшу оптимізацію за рахунок спрощення структури програми. Мертвий код включає код, який ніколи не може бути виконаний (недоступний код), і код, який лише впливаємертві змінні (записуються, але ніколи не читаються), тобто не мають відношення до програми.

Вилучення з програми непотрібного коду здатне пришвидшити час виконання програми за рахунок зменшення кількості виконуваних в ній операцій і зменшити розмір програми або проміжного представлення. Розглянемо приклад коду на мові програмування C на рисунку 1.3. В даному прикладі операція додавання var2 = var1 + 3 є мертвим кодом, так як змінна var2 не використовується в подальших обчисленнях і є мертвою, починаючи з цієї точки і закінчуючи кінцем процедури. Після видалення цієї інструкції отримаємо мертву змінну (рисунок 1.4).

```
1 int func(void)
2 {
3     int var1 = 24;
4     int var2;
5     var2  = var1 + 3; /*Мертвий код */
6     return var1 << 2;
7 }
```

Рисунок 1.3 – Приклад мертвого коду

```
1 int func(void)
2 {
3     int var1 = 24;
4     int var2; /*Мертва змінна*/
5     return var1 << 2;
6 }
```

Рисунок 1.4 – Приклад мертвої змінної

Однак на практиці також часто розділи коду представляють мертвий або недоступний код лише за певних умов, які можуть бути невідомими під час компіляції або складання. Такі умови можуть накладатися різними середовищами виконання (наприклад, різними версіями операційної системи або різними наборами і комбінаціями драйверів або служб, завантажених у конкретному цільовому середовищі), що може вимагати різних наборів особливих випадків у коді, але при в той же час стає умовно мертвим кодом для інших випадків.

Крім того, програмне забезпечення можна налаштувати, щоб включати або виключати певні функції залежно від уподобань користувача, роблячи невикористані частини коду непотрібними в конкретному сценарії.

Хоча модульне програмне забезпечення може бути розроблено для динамічного завантаження бібліотек лише на вимогу, у більшості випадків неможливо завантажити лише відповідні підпрограми з певної бібліотеки, і навіть якщо це буде підтримуватися, програма може все ще включають розділи коду, які можна вважати мертвим кодом у даному сценарії, але вже не можна виключити під час компіляції.

На найнижчому рівні оптимізації компіляторів є написання коду, використовуючи мову асемблер, яка призначена для певної апаратної платформи і може виробляти найбільш ефективний і компактний код, якщо програміст користується повним набором машинних інструкцій.

Багато операційних систем, що використовуються у вбудованих системах, традиційно написані на асемблері з цієї причини. Коли ефективність і розмір менш важливі, великі частини можуть бути написані на мові високого рівня. Тобто основна низькорівнена оптимізація – це і є написання коду на асемблері (мова програмування максимально наближена до машинного коду).

Більшість компіляторів написаних на мові асемблері є двохпрохідними, тому що для отримання правильного об'єктного коду вони здійснюють два перегляди вихідної програми.

Під час першого проходу асемблер створює таблицю символів та збирає всі імена, визначені у програмі; під час другого проходу він транслює програму, використовуючи інформацію, зібрану при першому проході.

У загальному випадку символічні імена можуть бути визначені у будь-якому місці програми, оскільки у будь-якому випадку асемблер переглядає всю програму. Проте програмісту буває зручно розміщувати всі визначення в початку програми. На рисунку 1.5 зображено принцип роботи двохпрохідного асемблера.

Перший прохід. Створення таблиці символів

Програма			Таблиця символів	
123	LOAD	X1+7	X1	60
124	JUMP	THRU	THRU	
125 GR:	ADD	#1	GR	125
126	DIVIDE	#10	ST	127
127 ST:	STORE	TEMP	TEMP	62

Другий прохід. Використання таблиці символів

Програма			Програма з числовими адресами	
123	LOAD	X1+7	LOAD	67
124	JUMP	THRU	JUMP	327
125 GR:	ADD	#1	ADD	#1
126	DIVIDE	#10	DIVIDE	#10
127 ST:	STORE	TEMP	STORE	62

Рисунок 1.5 – Двохпрохідний асемблер

Окрім двохпрохідних асемблерів часто використовується однопрохідний асемблер, який працює швидше, тому що повинен переглядати програму 1 раз. Однак у однопрохідних асемблерах виникає проблема посилань вперед; часто такі посилання обробляє завантажувач.

У загальному випадку однопрохідні асемблери надають менше можливостей, ніж двохпрохідні. Крім того, вони вносять додаткові обмеження на способи завдання адрес, використання імен, розподіл пам'яті або вимагають складнішого завантажувача, який виконує частину функцій асемблера у процесі завантаження програми в пам'ять.

1.3 Основні характеристики інтегрованих середовищ мови програмування С

С – це мова комп’ютерного програмування загального призначення. За конструкцією, функції С чітко відображають можливості цільових ЦП. Він знайшов тривале застосування в операційних системах, драйверах пристроїв, стеках протоколів, хоча все менше для прикладного програмного забезпечення, і є

поширеним у комп'ютерних архітектурах, які варіюються від найбільших суперкомп'ютерів до найменших мікроконтролерів і вбудованих систем.

C є обов'язковою процедурною мовою, що підтримує структуроване програмування, область видимості лексичних змінних і рекурсію, зі статичною системою типів.

Він був розроблений для компіляції та забезпечення низькорівневого доступу до пам'яті та мовних конструкцій, які ефективно відображаються з машинними інструкціями, і все це з мінімальною підтримкою під час виконання.

Незважаючи на свої низькорівневі можливості, мова була розроблена для заохочення кросплатформного програмування.

Сучасні компілятори часто не є окремими, автономними інструментами, а є частиною інтегрованого середовища розробки (IDE), яку іноді називають середовищами програмування.

Крім надання інструменту компіляції, сучасне середовище IDE пропонує інструменти для мовно-орієнтованого редагування, налагодження, визначення робочих профілів програми, управління конфігурацією.

Існує багато інтегрованих середовищ, які мають свої плюси і мінуси, але для мови програмування C найкращим прикладом IDE є Microsoft Visual Studio, яка пропонує такі основні групи операцій:

1) редагування з засобами вирізання, вставки, скасування операції тощо;

2) пошук із засобами заміни тексту та локалізації функцій в процесі налагодження;

3) перегляд різних вікон, що містять засоби діагностики та іншу інформацію, пов'язану з поточним проектом (точки переривання програми, зміст регістрів, розташування змінних, використання класів);

4) керування проектом, включаючи запуск нових проектів, компіляцію і зв'язування різних компонентів проекту;

5) налагодження з можливістю запуску програми в режимі покрокового виконання, установки точок переривання, відстеження значень виразів, перегляду таблиць символів;

6) засоби виконання, пов’язані з IDE.

Існують також інші IDE, такі як Project Rider, Eclipse, Visual Studio Code, MonoDevelop, Code::Blocks, Sharp Developer, але Microsoft Visual Studio є найкращим варіантом, оскільки має найширший функціонал з вище перерахованих, має підтримку технологій UWP і WPF та платформи .NET.

При проектуванні компілятора важливим фактором є вимоги. Зазвичай керуються такими загальними вимогами:

- коректність;
- ефективна компіляція;
- сумісність:
- зручність використання:
- ефективна компіляція;
- мінімальний розмір компілятора;
- мінімальна довжина цільового коду;
- створення ефективного цільового коду;
- портабельність;
- практичність.

1.4 Постановка задачі. Розробка та вдосконалення методів і засобів створення спеціалізованого двохпрохідного компілятора.

Поява мов високого рівня істотно спростило процес програмування, хоча і не звело його до загально-користувацького рівня. Спочатку таких мов були одиниці, потім десятки, зараз, напевно, їх нараховується більш сотні. Процесу цьому не видно кінця. Проте як і раніше переважають комп'ютери традиційної архітектури, що вміють розуміти тільки машинні команди, тому питання про створення компіляторів продовжує бути актуальним.

Дослідження, проведені в ході аналізу предметної області, наводять до висновків про актуальність розробки компіляторів та необхідність їх оптимізації.

Наявні на теперішній час компілятори занадто мало виділяють уваги відносно оптимізації компіляції коду.

Об’єктом дослідження є спеціалізований двохпрохідний компілятор.

Предметом дослідження є методи та засоби створення спеціалізованого двохпрохідного компілятора.

Метою дослідження є розробка, оптимізація та модифікація спеціалізованого двохпрохідного компілятора. Заради досягнення мети потрібно успішно виконати наступні завдання:

– аналіз існуючих засобів створення та методів розробки компіляторів;
– аналіз існуючих методів оптимізації компіляторів;
– проектування програмного продукту на основі дослідження;
– програмну реалізацію системи;
– тести програмної реалізації системи;
– аналіз результатів.

Розроблюваний програмний продукт можна представити як дієвий та ефективний засіб для компіляції та оптимізації коду.

1.5 Висновки

Технологія компілятора виникла з необхідності суворо визначеного перетворення вихідної програми високого рівня в цільову програму низького рівня для цифрового комп'ютера. Також при процесі перетворення мови програмування високого рівня в машинну мову, який відомий як компіляція, можлива додаткова оптимізація на різних етапах роботи компілятора, яка покращить ефективність та швидкість роботи вихідного коду. За результатами досліджень визначено доцільність створення компіляторів для оптимізації компіляції коду.

В даному розділі проаналізовано предметну область. Описано структуру компілятора та принципи його роботи. Описано етапи та фази етапів процесу компіляції. Розглянуто переваги та недоліки однопрохідних та багатопрохідних компіляторів. Розглянуто різницю між низькорівневою та високорівневою

оптимізацією. Проаналізовано існуючі методи оптимізації компіляції, їх переваги та недоліки. Також розглянуто двохпрохідний асемблер як приклад низькорівневої оптимізації. Розглянуто мову програмування С та методи інтегрованих середовищ. Проаналізувавши дану інформацію, обрано Microsoft Visual Studio як середовище розробки даного проекту.

За допомогою проведених досліджень вивели вимоги і завдання для подальшого вдосконалення програмного продукту. В результаті аналізу проблеми оптимізації була сформована задача на оптимізацію та покращення процесу компіляції.

2 АНАЛІЗ АЛГОРИТМІВ СКЛАДОВИХ КОМПІЛЯТОРА ТА ЙОГО ОПТИМІЗАЦІЇ

2.1 Загальні алгоритми фази аналізу компілятора

2.1.1 Алгоритм лексичного аналізу

Перш ніж почати проектування та програмування удосконаленого компілятора, проаналізуємо алгоритми компілятора, а саме алгоритми на яких базуються етапи роботи компілятора. Початковим етапом роботи компілятора є фаза аналізу вхідного коду, яка складається з лексичного, синтаксичного та семантичного аналізів. Розглянемо алгоритми роботи кожного з них.

Лексичний аналіз - це перший етап роботи компілятора. Він бере модифікований вихідний код від мовних препроцесорів, які записуються у вигляді речень. Лексичний аналізатор розбиває ці синтаксиси на серію токенів, видаляючи будь-які пробіли або коментарі у вихідному коді. Якщо лексичний аналізатор виявляє токен недійсним, він генерує помилку. Лексичний аналізатор тісно співпрацює з аналізатором синтаксису. Він зчитує потоки символів з вихідного коду, перевіряє наявність дійсних токенів і передає дані аналізатору синтаксису, коли той їх викликає. Токенами у мові програмування вважаються ключові слова, константи, ідентифікатори, рядки, числа, оператори та знаки пунктуації.

Наприклад, у мові C рядок оголошення змінної

int value = 100;

містить токени:

int (keyword), value (identifier), = (operator), 100 (constant) і ; (symbol).

Проаналізуємо, як теорія мови використовує такі терміни:

1. Алфавіти. Будь-який кінцевий набір символів $\{0,1\}$ — це набір двійкових алфавітів, $\{0,1,2,3,4,5,6,7,8,9, A, B, C, D, E, F\}$ — набір шістнадцяткових алфавітів, $\{az, AZ\}$ — набір алфавітів англійської мови.

2. Рядки. Будь-яка кінцева послідовність алфавітів (символів) називається рядком. Довжина рядка – це загальна кількість зустрічей алфавітів, наприклад, довжина рядка tutorialspoint дорівнює 14 і позначається $|tutorialspoint| = 14$.

Рядок без алфавітів, тобто рядок нульової довжини відомий як порожній рядок і позначається ε (епсилон).

3. Спеціальні символи - типова мова високого рівня містить символи, що описані в таблиці 2.1.

Таблиця 2.1 – Опис спеціальних символів

<table>
<tr><td>Арифметичні символи</td><td>Додавання(+), Віднімання(-), По модулю(%), Множення(*), Ділення(/)</td></tr>
<tr><td>Розділові знаки</td><td>Кома(,), Крапка з комою(;), Крапка(.), Стрілка(→)</td></tr>
<tr><td>Присвоєння</td><td>=</td></tr>
<tr><td>Спеціальне присвоєння</td><td>+=, /=, *=, -=</td></tr>
<tr><td>Порівняння</td><td>==, !=, <, <=, >, >=</td></tr>
<tr><td>Препроцесор</td><td>#</td></tr>
<tr><td>Специфікатор розташування</td><td>&</td></tr>
<tr><td>Логічні оператори</td><td>&, &&, |, ||, !</td></tr>
<tr><td>Оператори зміни</td><td>>>, >>>, <<, <<<</td></tr>
</table>

4. Мова - мова розглядається як скінченний набір рядків над деяким скінченним набором алфавітів. Комп'ютерні мови розглядаються як скінченні множини, і над ними можна виконувати математично задані операції. Скінченні мови можна описати за допомогою регулярних виразів.

5. Регулярні вирази - лексичному аналізатору потрібно сканувати та ідентифікувати лише кінцевий набір дійсних рядків/токенів/лексем, які належать даній мові. Він шукає шаблон, визначений правилами мови.

6. Операції:

– Об'єднання двох мов L і M:

$$L \cup M = \{s \mid s \text{ знаходиться в } L \text{ або } s \text{ знаходиться в } M\}. \quad (2.1)$$

– Конкатенація (додавання) двох мов L і M:

$$LM = \{st \mid s \text{ знаходиться в } L, \text{а } t \text{ знаходиться в } M\}. \quad (2.2)$$

– Замикання мови L:

$$L* = \text{ нуль або більше зустрічей мови } L. \quad (2.3)$$

7. Позначення. Якщо r і s є регулярними виразами, що позначають мови $L(r)$ і $L(s)$, то:

– Об'єднання: $(r)|(s)$ – це регулярний вираз, що позначає $L(r) \cup L(s)$;

– Додавання: $(r)(s)$ є регулярним виразом, що позначає $L(r)L(s)$;

– Замикання: $(r)*$ є регулярним виразом, що позначає $(L(r))*$;

– (r) — регулярний вираз, що позначає $L(r)$.

8. Пріоритетність та асоціативність:

– $*$, $(.)$ і $|$ є лівими асоціативними;

– $*$ має найвищий пріоритет;

– Конкатенація $(.)$ має другий пріоритет;

– $|$ має найнижчий пріоритет з усіх.

9. Представлення дійсних токенів мови в регулярному виразі - Якщо x є регулярним виразом, то:

– $x*$ означає нуль або більше випадків x, тобто він може генерувати $\{ e, x, xx, xxx, xxxx, \dots \}$;

– $x+$ означає одне або кілька зустрічей x, тобто він може генерувати $\{ x, xx, xxx, xxxx \dots \}$ або $xx*$;

– x? означає щонайбільше одне поява x, тобто він може генерувати або {x}, або {e};

– [az] – це всі нижні алфавіти англійської мови;

– [AZ] — це всі прописні алфавіти англійської мови;

– $[0-9]$ – це всі натуральні цифри, які використовуються в математиці.

10. Представлення появи символів за допомогою регулярних виразів

– буква $= [a - z]$ або $[A - Z]$;

– цифра $= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$ або $[0-9]$;

– знак $= [+ \mid -]$.

11. Представлення мовних лексем за допомогою регулярних виразів

– Десятковий знак = (знак) ? (цифра) $+$;

– Ідентифікатор = (літера)(літера | цифра)$*$.

Єдина проблема, яка залишається у лексичного аналізатора, полягає в тому, як перевірити правильність регулярного виразу, що використовується для визначення шаблонів ключових слів мови. Добре прийнятним рішенням є використання детермінованих (скінченних) кінцевих автоматів для перевірки.

Скінченні автомати — це кінцевий автомат, який приймає рядок символів як вхідні дані і відповідно змінює свій стан. Скінченні автомати є розпізнавачем регулярних виразів. Коли рядок регулярного виразу подається в кінцеві автомати, він змінює свій стан для кожного літералу. Якщо вхідний рядок успішно оброблений і автомат досягає свого остаточного стану, тобто щойно поданий рядок вважається дійсним токеном мови.

Згідно із загальною класифікацією, дані наступні визначення.

Детермінований скінченний автомат або детермінований скінченний автомат акцептор – це скінченний автомат, який приймає скінченний рядок символів.

$$(\Sigma, S, s_0, \delta, F), \tag{2.4}$$

де Σ – вхідна абетка (скінченний, не порожній набір символів);

S – скінченний, не порожній набір станів;

s_0 – початковий стан, елемент з S;

δ – функція переходу: $\delta : S \times \Sigma \to S$ (в недетермінованих скінченних автоматах це буде $\delta : S \times \Sigma \to P(S)$, тобто δ повертає набір станів);

F – набір кінцевих станів, (можливо порожня) підмножина S.

Для обох детермінованих і недетермінованих СА, зручно дозволити δ бути неповною функцією, тобто $\delta(q, x)$ не має бути визначеною для кожної комбінації $q \in S$ i $x \in \Sigma$. Якщо СА M перебуває в стані q, наступний символ x і $\delta(q, x)$ не визначена, тоді M може повідомити про помилку (тобто відхилити ввід).

Скінчений перетворювач – це абстракції, що використовується для опису шляху зміни стану об'єкта в залежності від поточного стану та інформації отриманої ззовні.

$$(\Sigma, \Gamma, S, s_0, \delta, \omega), \qquad (2.5)$$

де Σ – вхідна абетка (скінченний, не порожній набір символів);

Γ – вихідна абетка (скінченний, не порожній набір символів);

S – скінченний, не порожній набір станів;

s_0 – початковий стан, елемент з S (в недетермінованих скінченних автоматах, s_0 це набір початкових станів);

δ – функція переходу: $\delta : S \times \Sigma \rightarrow S$;

ω – функція виходу.

Якщо функція виходу є функцією стану і вхідної абетки ($\omega : S \times \Sigma \rightarrow \Gamma$) таке визначення відповідає моделі Мілі (СА, що використовує вхідні дії та стани, тобто, вихід базується на вході і стані), і може бути виконана як автомат Мілі. Якщо функція виходу залежить тільки від стану ($\omega : S \rightarrow \Gamma$) тоді таке визначення відповідає моделі Мура (СА використовує тільки вхідні дії, тобто, вихід базується тільки на стані), і може бути виконана як автомат Мура. Скінченний автомат без функції виходу відомий як напівавтомат або як модель станів і переходів.

Відповідність, яка відображає вхідні ланцюжки a автомата M у вихідні ланцюжки w називають автоматним відображенням, а також автоматним оператором M. Якщо результат застосування цього оператора до ланцюжка a – вихідний ланцюжок w, то це позначають $M(a) = w$. Кількість символів у

ланцюжку a, як завжди, називають довжиною ланцюжка a та позначають $|a|$ чи $l(a)$.

Автоматне відображення має дві властивості:

1. Ланцюжки a та $w = M(a)$ мають однакову довжину: $|a| = |w|$ (збереження довжини).

2. Якщо $a = a1a2$ і $M(a1a2) = w1w2$, де $|a1| = |w1|$, то $M(a1) = w1$, тобто образ відрізка довжиною l дорівнює відрізку образу з такою самою довжиною.

Друга властивість означає, що автоматні оператори – це оператори без випередження, тобто такі, котрі, обробляючи ланцюжок зліва направо, і-та буква вихідного ланцюжка залежить тільки від перших і букв вхідного ланцюжка.

Коли лексичний аналізатор зчитує вихідний код, він сканує код по буквах і коли зустрічається пробіл, символ оператора або спеціальні символи, він вирішує, що слово завершене. Наприклад:

int int value;

Під час сканування обох лексем до int лексичний аналізатор не може визначити, чи це ключове слово int чи ініціали значення ідентифікатора int.

Правило найдовшого збігу стверджує, що перевірена лексема має визначатися на основі найдовшого збігу серед усіх доступних лексем.

Лексичний аналізатор також дотримується пріоритету правил, коли зарезервованому слову, наприклад, ключовому слову мови, надається пріоритет перед введенням користувача. Тобто, якщо лексичний аналізатор знаходить лексему, яка збігається з будь-яким наявним зарезервованим словом, він повинен генерувати помилку.

2.1.2 Ланцюжковий алгоритм вирішення колізій хеш-таблиць

Таблиця символів є важливою структурою даних, створеною та підтримуваною компілятором для відстеження семантики змінних, тобто вона зберігає інформацію про область і зв'язну інформацію про імена, інформацію про екземпляри різних сутностей, такі як імена змінних і функцій, класи, предмети тощо.

Таблиця символів використовується на різних етапах компілятора, а саме:

1. Лексичний аналіз – створює нові записи таблиці в таблиці, наприклад записи про токени.
2. Аналіз синтаксису – додає в таблицю інформацію про тип атрибута, область дії, розмір, адресу посилання, використання тощо.
3. Семантичний аналіз – використовує доступну інформацію в таблиці, щоб перевірити семантику, тобто переконатися, що вирази та присвоєння є семантично правильними (перевірка типу), і відповідно оновити її.
4. Проміжне генерування коду – посилається на таблицю символів, щоб знати, скільки і який тип часу виконання виділено, а таблиця допомагає додавати інформацію про тимчасові змінні.
5. Оптимізація коду – використовує інформацію в таблиці символів для машинозалежної оптимізації.
6. Генерація цільового коду – генерує код за допомогою адресної інформації ідентифікатора, представленого в таблиці.

Записи таблиці символів – кожен запис у таблиці символів пов’язаний з атрибутами, які підтримують компілятор на різних етапах.

Елементи, що зберігаються в таблиці символів:

– імена змінних і константи;
– назви процедур і функцій;
– літеральні константи та рядки;
– компілятор створив тимчасові елементи;
– етикетки вихідними мовами.

Операції таблиці символів – основні операції, визначені в таблиці символів описані в таблиці 2.2.

Таблиця 2.2 – Опис операцій таблиці символів

Функція	Функціонал цієї функції
allocate	Виділяє нову порожню таблицю символів
free	Видаляє всі записи та звільняє зберігання таблиці символів
lookup	Для пошуку імені та повернення вказівника на його запис
insert	Щоб вставити ім'я в таблицю символів і повернути покажчик на її запис
set_attribute	Щоб пов'язати атрибут із заданим записом
get_attribute	Щоб отримати атрибут, пов'язаний із заданим записом

Для реалізації таблиці символів оберемо таку типову структуру як хеш таблиця. Хеш-таблиця — це масив із діапазоном індексів: від 0 до розміру таблиці -1. Ці записи є вказівниками, що вказують на назви таблиці символів. Для пошуку імені використовуємо хеш-функцію, яка призведе до цілого числа від 0 до розміру таблиці – 1. Вставка та пошук можна зробити дуже швидко – $O(1)$. Перевагою є можливість швидкого пошуку, а недоліком є те, що хешування складно реалізувати.

Алгоритм пошуку, який використовує хешування, складається з двох частин. Перша частина - це обчислення хеш-функції, яка перетворює ключ пошуку в індекс масиву. Ідеальний випадок полягає в тому, що немає двох хешів ключів пошуку до одного індексу масиву, однак це не завжди так, і його неможливо виключити для невидимих даних. Отже, друга частина алгоритму — це дозвіл зіткнень.

Існує два основних варіанта хеш-таблиць: з ланцюжками і з відкритою адресацією.

Відкрита адресація – це метод вирішення колізій, у якому всі записи зберігаються в самому масиві сегментів, а хешування виконується за допомогою

зондування. Коли потрібно вставити новий запис, сегменти перевіряються, починаючи з хешованого слота і продовжуючи певну послідовність проб, доки не буде знайдено незайнятий слот. Під час пошуку запису сегменти скануються в тій самій послідовності, поки не буде знайдено цільовий запис або невикористаний слот масиву, що вказує на невдалий пошук.

Послідовності зондів включають:

1. Лінійне зондування, при якому інтервал між зондами фіксується.
2. Квадратичне зондування, при якому інтервал між зондами збільшується шляхом додавання послідовних результатів квадратичного полінома до значення, заданого вихідним обчисленням хешування.
3. Подвійне хешування, при якому інтервал між зондами обчислюється за допомогою вторинної хеш-функції.

Продуктивність відкритої адресації може бути повільнішою в порівнянні з окремим ланцюжком, оскільки послідовність проб збільшується, коли коефіцієнт навантаження α наближається до 1. Зондування призводить до нескінченного циклу, якщо коефіцієнт навантаження досягає 1, у випадку повністю заповненої таблиці. Середня вартість лінійного аналізу залежить від здатності хеш-функції рівномірно розподіляти елементи по таблиці, щоб уникнути кластеризації, оскільки формування кластерів призведе до збільшення часу пошуку.

В ланцюжковому методі розв’язання колізій процес включає створення зв’язаного списку з парою ключ-значення для кожного індексу масиву пошуку. Елементи, що зіткнулися, об’єднані в один зв’язаний список, який можна перейти, щоб отримати доступ до елемента за допомогою унікального ключа пошуку. Розв'язання зіткнень через ланцюжок зі зв'язаним списком є поширеним методом реалізації хеш-таблиць.

Якщо елемент можна порівняти чисельно чи лексично та вставлений у список, зберігаючи загальний порядок, це призводить до швидшого завершення невдалих пошуків.

Розв’язання колізій за допомогою ланцюжкового методу зображено на рисунку 2.1.

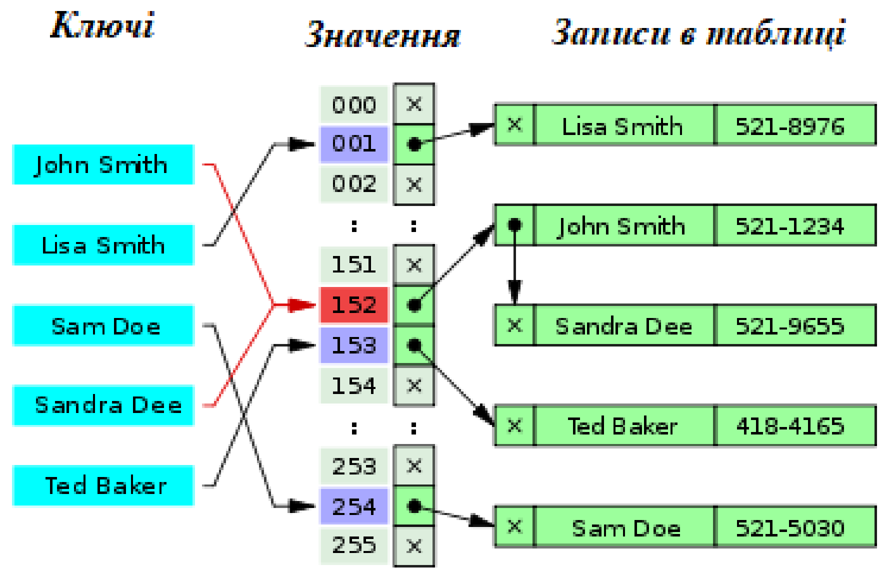


Рисунок 2.1 - Розв'язання колізій за допомогою ланцюжків

2.1.3 Алгоритм синтаксичного аналізу та алгоритм Ерлі

У результаті синтаксичного аналізу вихідний текст перетворюється на структуру даних, зазвичай — на дерево, яке відбиває синтаксичну структуру вхідний послідовності і добре підходить подальшої обробки.

Як правило, результатом синтаксичного аналізу є синтаксична будова речення, представлена або у вигляді дерева залежностей , або у вигляді дерева складових , або у вигляді деякого поєднання першого та другого способів уявлення. Використовуються такі типи алгоритмів семантичного аналізу:

1) низхідний парсер- продукції граматики розкриваються, починаючи зі стартового символу, до отримання необхідної послідовності токенів;
2) метод рекурсивного спуску;
3) LL-аналізатор; висхідний парсер – продукції відновлюються з правих частин, починаючи з токенів і закінчуючи стартовим символом;
4) LR-аналізатор;
5) GLR-парсер.

Розглянемо алгоритм Ерлі, який відноситься до типу GLR-парсерів. Алгоритм Ерлі – алгоритм синтаксичного аналізу пропозиції з контекстно-вільної

граматики, заснований на методі динамічного програмування. На відміну від алгоритму Кока-Янгера-Касамі, який вимагає приведення граматики до нормальної форми Хомського, алгоритм Ерлі привабливий тим, що не накладає обмежень на контекстно-вільну граматику, що використовується для аналізу. Крім того, Алгоритм Кока-Янгера-Касамі працює за принципом «знизу-вгору», тобто будує можливі дерева розбору пропозиції, починаючи з вершини. На відміну від нього Алгоритм Ерлі реалізує стратегію виведення «зліва-направо».

Алгоритм Ерлі за вхідним ланцюжком та граматикою породжує список розбору для даного вхідного ланцюжка в заданій граматиці.

Нехай вхідна граматика задається четвіркою G = (T,N,S,P).

Вхідний ланцюжок задається послідовністю терміналів w=a1,a2,…,an.

Списком розбору будемо називати послідовність списків ситуацій $I_0, I_1, \ldots I_n$.

Ситуацією будемо називати конструкцію вигляду [A→X_1,.., $X_k \cdot X_{k+1}$,…, X_m,i] (де k, i — довільні натуральні числа від 0 до m, A – метасимвол, який не належить ні N ні T), якщо A→ X_1,.., X_m – правило з P.

Список ситуацій I_j для слова w будемо будувати таким чином:

[A→α·β, i] (i≤j) належить I_j тоді і тільки тоді, якщо існують такі γ та δ, що S⇒* γAδ,γ⇒*$a_1 \ldots a_i$ та α⇒*a_i+1…a_j. Тобто певна частина слова w [1..j] може бути виведена використовуючи A→α·β. Зрозуміло, що w буде належати L(G), якщо [S→α·0] буде належати I_n.

Одержаний список розбору може слугувати базою для багатьох алгоритмів, зокрема побудови правого розбору ланцюжка.

Вхід алгоритму:

Граматика $G = (T, N, S, P)$

Вхідний ланцюжок $w = a1, a2, \ldots, an$

Вихід алгоритму:

Список розбору $I0, I1, \ldots\ In$.

Тепер розглянемо етапи роботи алгоритму Ерлі:

1) Спочатку ініціюємо список I_0.

2) Для всіх правил $S \rightarrow \alpha$, включити $[S \rightarrow \cdot\, \alpha, 0]$ в I_0. Поки в I_0 можна включати, виконуємо кроки 3-4.

3) Якщо $[B \rightarrow \gamma\, \cdot, 0]$ належить I_0, то включити в I_0 всі ситуації вигляду $[A \rightarrow \alpha B \cdot \beta, 0]$ (якщо вони ще не там) для всіх $[A \rightarrow \alpha \cdot B\beta, 0]$, що вже належать $I0$.

4) Якщо $A \rightarrow [\alpha \cdot B\beta, 0]$ належить I_0, то включити в I_0 ситуації вигляду $[B \rightarrow \cdot\, \gamma, 0]$ (якщо вона ще не там) для всіх правил вигляду $B \rightarrow \gamma$.

5) Нехай маємо побудовані списки $I_0, I_1, \ldots\ I_{j-1}$. Для кожної ситуації $[B \rightarrow \alpha \cdot aj\beta, i]$ з I_{j-1} включити до I_j ситуацію $[B \rightarrow \alpha aj \cdot B, i]$. Поки в I_j можна включати, виконуємо кроки 6-7.

6) Нехай $[A \rightarrow \alpha\, \cdot, i]$ належить до I_j. Шукати в I_i ситуації вигляду $[B \rightarrow \alpha \cdot A\beta, k]$. Для кожної з них включити до I_j ситуацію $[B \rightarrow \alpha A \cdot \beta, k]$.

7) Нехай $[A \rightarrow \alpha \cdot B\beta, i]$ належить I_j. Для кожного правила $B \rightarrow \gamma$ включити до I_j ситуацію $[B \rightarrow \cdot\, \gamma, j]$.

2.1.4 Алгоритм семантичного аналізу

Це третій етап роботи компілятора, на якому семантика, що використовується в програмі, перевіряється для забезпечення коректності граматики коду.

Семантичний аналіз включає набір процедур, які у відповідний час викликаються синтаксичним аналізатором, як того вимагає граматика. Ця семантика чітка і узгоджена з тим, як передбачається використовувати типи даних і структури контролю.

Семантичний аналізатор буде використовувати інформацію, збережену в синтаксичному дереві та таблиці символів, щоб перевірити семантичну узгодженість вихідної програми відповідно до визначення мови.

Приклад семантичного аналізу:

float a = 12.5;

float b = x * 4;

У цьому випадку ціле число 4 буде приведено до 4.0, щоб могло статися множення.

Семантика мови надає значення мовним конструкціям, таким як її лексеми та синтаксична структура. Семантика може бути як статичною, так і динамічною. Статична семантика – це семантика програми, яка не змінюється, вона оголошується перед їх оголошенням і перевіряється під час компіляції.

Динамічна семантика це семантика, яка живе, рухається та існує під час виконання програми. Вони визначають значення різних блоків вихідної програми, таких як оператори та вирази.

Типи оголошених змінних відносно семантичного аналізу. Вони поділяються на 3 основні категорії: базові типи, складні типи та складені (комплексні) типи.

1. Базові типи - це примітивні типи, які надаються безпосередньо апаратними засобами, такими як int, float, double, char, bool.

2. Складні типи - це типи, побудовані як агрегації базових типів і простих складених типів. До них належать масиви, записи, структури, об’єднання, покажчики, класи.

3. Комплексні типи - це абстрактні типи даних, мова високого рівня може підтримувати або не підтримувати ці типи, якщо вони не підтримуються, програміст може вибрати потрібний тип. До них належать списки, черги, дерева, купи, таблиці, стеки.

Типи зберігаються в таблиці символів перед будь-якою перевіркою типу. При обробці оператора заголовка програми ідентифікатору програми присвоюється тип програми та поточний покажчик області встановлений так, щоб вказувати на основну програму. Якщо мова допускає визначені користувачем типи даних, установка цих типів даних повинна відбутися в записах таблиці символів для оголошених ідентифікаторів. Ці ідентифікатори потім додаються до абстрактного синтаксису.

Дії, що виконуються під час семантичного аналізу:

1) перевірка типу – це гарантує, що типи даних використовуються узгоджено відповідно до їхніх визначень;

2) перевірка етикетки – це гарантує, що посилання на етикетки використовуються в програмі;

3) контроль потоку – це забезпечує правильне використання керуючих структур, наприклад, гарантує, що в циклі оператор break виконується лише всередині циклу;

4) роздільна здатність – це передбачає визначення сфери дії імені;

5) перевірка, пов'язана з масивом – це визначає, чи всі посилання на масив у програмі знаходяться в межах оголошеного діапазону.

Атрибут – це властивість, значення якої присвоюється граматичному символу. Функції обчислення атрибутів, також відомі як семантичні функції, є функціями, пов'язаними з виробленням граматики і використовуються для обчислення значень атрибута. Функції предикатів – це функції, які визначають синтаксис і статичні семантичні правила певної граматики. Типи атрибутів:

1) тип – це пов'язує об'єкти даних із дозволеним набором значень;

2) розташування – може бути змінено підпрограмою керування пам'яттю операційної системи;

3) значення – це результат операції присвоєння;

4) ім'я – їх можна змінити під час виклику та повернення підпрограми;

5) компонент – об'єкти даних, що складаються з інших об'єктів даних. Ця прив'язка представлена вказівником і згодом змінюється.

Синтезовані атрибути отримують значення зі значень атрибутів їхніх дочірніх вузлів. Вони визначаються семантичним правилом, пов'язаним із виробництвом у вузлі таким чином, що виробництво має нетермінал як голову. Синтезованим атрибутом називається таке S в $S \rightarrow ABC$, якщо він приймає значення зі свого дочірнього вузла (A, B, C).

$$E \rightarrow E + T \{ E.value = E.value + T.value \}, \quad (2.6)$$

де E – батьківський вузол;

T – дочірній вузол.

Успадковані атрибути – це атрибути, що беруть значення від своїх батьків та/або братів і сестер. Вони визначаються семантичним правилом, пов’язаним з виробництвом у батьківському, таким чином, що виробництво має нетермінал у своєму тілі. Вони корисні, коли структура дерева аналізу не відповідає абстрактному синтаксичному дереву вихідної програми. Вони не можуть бути оцінені шляхом попереднього замовлення обходу дерева аналізу, оскільки вони залежать як від лівих, так і від правих братів і сестер.

$$S \rightarrow ABC, \tag{2.7}$$

де S, A, B, C – атрибути;

A може отримати свої значення з S, B і C;

B може отримати свої значення з S, A і C;

C може отримати свої значення з A, B і S.

Граматика атрибутів - це окремий випадок безконтекстної граматики, коли до одного або кількох нетерміналів додається додаткова інформація, щоб надати контекстно-залежну інформацію. Також можемо визначити його як SDD без побічних ефектів. Це середовище, що забезпечує семантику безконтекстної граматики і допомагає у специфікації синтаксису та семантики мови програмування. Якщо розглядати як дерево розбору, воно може передавати інформацію між вузлами дерева.

$$E \rightarrow E + T \{ E.value = E.value + T.value \}, \tag{2.8}$$

де E, T – нетермінальні значення.

Права сторона містить семантичні правила, які визначають, як інтерпретувати граматику. Нетермінальні значення E і T додаються, а їх результат копіюється до нетермінального E.

Потрібно побудувати граматику атрибутів, яка анотує число значенням, яке воно представляє. Спочатку пов'язуємо атрибути з граматичними символами, що описано в таблиці 2.3.

Таблиця 2.3 – Зв'язування атрибутів з граматичними символами

Символ	Атрибути
number	val
sign	neg
list	pos, val
bit	pos, val

Граматика атрибутів описана в таблиці 2.4 складатиметься з наступних функцій:

1. Кожен символ X матиме набір атрибутів $A(X)$.
2. $A(X)$ може бути:
   - Зовнішні атрибути, отримані поза граматикою, зокрема таблиця символів;
   - Синтезовані атрибути передані вгору по дереву розбору;
   - Успадковані атрибути, що передаються по дереву аналізу;
3. Кожен продукт граматики матиме набір семантичних функцій і функцій предикатів.

Таблиця 2.4 – Граматика атрибутів

Граматика	Функції атрибутів
$number \rightarrow sign\ list$	$list.pos = 0$ $if\ sign.neg$: $number.val = -list.val$ $else$: $number.val = -list.val$
$sign \rightarrow +$	$sign.neg = false$
$sign \rightarrow -$	$sign.neg = true$
$list \rightarrow bit$	$bit.pos = list.pos$ $list.val = pos.val$
$list_0 \rightarrow list_1.bit$	$list_1.pos = list_0.pos + 1$ $bit.pos = list_0.pos$ $list_0.val = list_1.val + bit.val$
$bit \rightarrow 0$	$bit.val = 0$
$bit \rightarrow 1$	$bit.val = 2^{bit.pos}$

2.2 Алгоритм генерації цільового коду

Генерацію коду можна розглядати як завершальний етап компіляції. Завдяки генерації проміжного коду до кінцевого коду можна застосувати процес оптимізації, але це можна розглядати як частину самого етапу генерації коду. Код, згенерований компілятором, є об'єктним кодом якоїсь мови програмування нижнього рівня, наприклад, мови асемблера. Бачимо, що вихідний код, написаний мовою вищого рівня, перетворюється на мову нижчого рівня, що призводить до об’єктного коду нижнього рівня, який повинен мати такі мінімальні властивості:

– він повинен містити точне значення вихідного коду;

– він має бути ефективним з точки зору використання ЦП та управління пам’яттю.

Далі проміжний код перетворюється в цільовий об’єктний код. На додаток до основного перетворення з проміжного представлення в лінійну послідовність машинних інструкцій, типовий генератор коду намагається якимось чином оптимізувати згенерований код.

Завдання, які зазвичай є частиною етапу "генерації коду" складного компілятора, включають:

– вибір інструкцій – які інструкції використовувати;

– планування інструкцій – у якому порядку розміщувати ці інструкції. Планування – це оптимізація швидкості, яка може мати критичний вплив на конвеєрні машини;

– розподіл регістрів – розподіл змінних у регістрах процесора;

– генерування даних налагодження, якщо потрібно, щоб код можна було налагодити.

Вибір інструкцій, як правило, виконується шляхом обходу пост-ордера по абстрактному синтаксичному дереву, узгоджуючи конкретні конфігурації дерева з шаблонами. Наприклад, дерево $W:= ADD(X, MUL(Y, Z))$ може бути перетворено в лінійну послідовність інструкцій шляхом рекурсивного генерування послідовностей для $t1:= X$i; $t2:= MUL(Y, Z)$, а потім видання інструкції $ADD\ W, t1, t2$.

У компіляторі, який використовує проміжну мову, може бути два етапи вибору інструкцій – один для перетворення дерева синтаксичного аналізу в проміжний код, а другий етап набагато пізніше для перетворення проміжного коду в інструкції з набору інструкцій цільової машини. Ця друга фаза не вимагає обходу дерева; це можна зробити лінійно і, як правило, передбачає просту заміну операцій на проміжній мові їх відповідними кодами операцій.

Основні блоки складаються з послідовності інструкцій з трьома адресами. Генератор коду приймає цю послідовність інструкцій як вхідні дані. getReg генератор коду використовує функцію getReg для визначення стану доступних регістрів і розташування значень імен. getReg працює наступним чином:

– Якщо змінна Y вже є в регістрі R, вона використовує цей регістр.

– Інакше, якщо доступний деякий регістр R, він використовує цей регістр.

– В іншому випадку, якщо обидва наведені вище варіанти неможливі, він вибирає регістр, який вимагає мінімальної кількості інструкцій завантаження та збереження.

Для інструкції $x = y\, OP\, z$ генератор коду може виконувати наступні дії. Припустимо, що L — це місце (бажано, регістр), де має бути збережений вихід у $OP\ z$:

1) Викликати функцію getReg, щоб визначити розташування L.

2) Визначте поточне розташування y, звернувшись до дескриптора адреси y. Якщо y на даний момент не знаходиться в регістрі L, згенеруйте таку інструкцію, щоб скопіювати значення y в $L. MOV\ y', L$; де y' представляє скопійоване значення y.

3) Визначте поточне розташування z, використовуючи той самий метод, що використовувався на кроці 2 для y, і створіть таку інструкцію: $OP\ z', L$; де z' являє собою скопійоване значення z.

4) Тепер L містить значення $y\ OP\ z$, яке призначене для призначення x. Отже, якщо L є регістром, оновіть його дескриптор, щоб вказати, що він містить значення x. Оновіть дескриптор x, щоб вказати, що він зберігається в місці L.

5) Якщо y і z більше не використовуються, їх можна повернути системі.

6) Інші конструкції коду, такі як цикли та умовні оператори, перетворюються на мову асемблера за допомогою загальної збірки.

2.3 Загальна оптимізація компілятора

2.3.1 Оптимізація процесу компіляції

Проаналізувавши алгоритми загальних етапів роботи компілятора, можна зрозуміти, що процес компіляції коду, в плані використання пам’яті комп’ютера та затраченого часу, є досить трудомістким процесом. Тому виникає потреба в оптимізації процесу компіляції коду. В загальному, метою оптимізації є отримання оптимальної системи, істинно оптимальна система в процесі оптимізації досягається далеко не завжди. Оптимізована система зазвичай є оптимальною тільки для однієї задачі або групи користувачів: десь може бути важливіше зменшення часу, необхідного програмі для виконання роботи, навіть ціною споживання більшого обсягу пам'яті; в додатках, де важливіше пам'ять, можуть вибиратися більш повільні алгоритми з меншими запитами до пам'яті.

Більш того, часто не існує універсального рішення, яке працює добре у всіх випадках, тому інженери використовують компромісні рішення для оптимізації лише ключових параметрів. До того ж, зусилля, необхідні для досягнення повністю оптимальної програми, яку неможливо далі поліпшити, практично завжди перевищують вигоду, яка може бути від цього отримана, тому, як правило, процес оптимізації завершується до того, як досягається повна оптимальність. На щастя, в більшості випадків навіть при цьому досягаються помітні поліпшення.

Тому, оберемо та проаналізуємо декілька загальних алгоритмів оптимізації компіляції та один модифікований алгоритм який буде спрямований на оптимізацію компіляції при роботі з циклами, оскільки цикли споживають досить великий об’єм пам’яті при великій кількості ітерацій, і їх оптимізація значно вплине на час виконання всього процесу компіляції.

2.3.2 Загальні алгоритми оптимізації компіляторів

Усунення зайвих інструкцій. На рівні вихідного коду користувач може зробити оптимізацію, що описана в таблиці 2.5.

Таблиця 2.5. – Усунення зайвих інструкцій

int add_ten (int x)
{
int y, z;
y = 10;
z = x + y;
return z;
}
	int add_ten (int x)
	{
	int y;
	y = 10;
	y = x + y;
	return y;
	}
		int add_ten (int x)
		{
		int y = 10;
		return x + y;
		}
			int add_ten (int x)
			{
			return x + 10;
			}

На рівні компіляції компілятор шукає інструкції, зайві за своєю природою. Багаторазове завантаження та зберігання інструкцій може мати однакове значення, навіть якщо деякі з них видалено. Наприклад: $MOV\ x, R0$; $MOV\ R0, R1$.

Можемо видалити першу інструкцію та переписати речення так: $MOV\ x, R1$.

Недосяжний код – це частина програмного коду, до якої ніколи не звертаються через програмні конструкції. Можливо, програмісти випадково написали фрагмент коду, до якого неможливо дістатися. Наприклад:

```
void add_ten (int x)
{
        return x + 10;
        printf("value of x is %d", x);
}
```

У цьому сегменті коду оператор printf ніколи не буде виконано, оскільки програмне керування повертається назад, перш ніж воно зможе виконати, отже printf можна видалити.

Потік оптимізації управління. У коді є випадки, коли програмне керування перескакує вперед і назад, не виконуючи жодного значного завдання. Ці стрибки можна прибрати. Розглянемо наступний фрагмент коду:

```
MOV R1, R2
GOTO L1
L1 :  GOTO L2
```

L2 : INC R1

У цьому коді мітку $L1$ можна видалити, коли вона передає керування $L2$. Таким чином, замість переходу до $L1$, а потім до $L2$, елемент керування може безпосередньо досягти $L2$, як показано нижче:

MOV R1, R2

GOTO L2

L2 : INC R1

Алгебраїчне спрощення виразу. Бувають випадки, коли алгебраїчні вирази можна зробити простими. Наприклад, вираз $a = a + 0$ можна замінити самим a, а вираз $a = a + 1$ можна просто замінити на $INC\ a$.

Зниження сили. Є операції, які забирають більше часу та місця. Їх «силу» можна зменшити, замінивши їх іншими операціями, які займають менше часу та місця, але дають той самий результат. Наприклад, $x * 2$ можна замінити на x<< 1, що включає лише один зсув вліво. Хоча вихід $a * a$ і a^2 однаковий, другий є набагато ефективнішим у реалізації.

2.4 Модифікований алгоритм методу “loop fusion”

Для покращення оптимізації нашого спеціалізованого компілятора удосконалимо метод «loop fusion», де він є методом об'єднання двох тіл циклів в одне та модифікуємо його за допомогою техніки розкрутки циклу. Цей метод безпечний, коли кожне визначення та кожне використання в результуючому циклі мають таке ж значення, що й у вихідних циклах.

У кожному випадку трансформований код потребує короткого циклу прологу, який відокремлює достатню кількість ітерацій, щоб гарантувати, що розгорнутий цикл обробляє ціле кратне чотирьох ітерацій. Якщо всі відповідні межі циклу відомі під час компіляції, компілятор може визначити, чи потрібен пролог.

Розгортання внутрішнього циклу та розгортання зовнішнього циклу, дають різні результати для цього конкретного гнізда циклу. Розгортання внутрішнього

циклу створює код, який виконує набагато менше послідовностей тестування та розгалуження, ніж вихідний код. На відміну від цього, розгортання зовнішнього циклу з наступним злиттям внутрішніх циклів не тільки зменшує кількість послідовностей тестування та розгалуження, але також створює повторне використання $y(i)$ і послідовний доступ до x і m. Доступ до m є послідовним, оскільки компілятор зберігає масиви в порядку великих стовпців.

Розгортання циклу має як прямий, так і непрямий вплив на код, який компілятор може створити для даного циклу. Кінцева продуктивність циклу залежить від усіх ефектів, прямих і непрямих.

З точки зору прямих переваг, розгортання має зменшити кількість операцій, необхідних для завершення циклу. Зміни контрольного потоку зменшують загальну кількість послідовностей тестування та розгалуження. Розгортання може створити повторне використання в тілі циклу, зменшуючи трафік пам'яті. Нарешті, якщо цикл містить циклічний ланцюжок операцій копіювання, розгортання може видалити копії.

Однак, як небезпека, розгортання збільшує розмір програми, як у її формі, так і в остаточному вигляді виконуваного коду. Зростання ir збільшує час компіляції. Зростання виконуваного коду має незначний ефект, доки цикл не переповнить кеш інструкцій — тоді деградація, ймовірно, переважає будь-які прямі переваги.

Компілятор також може розгорнути для непрямих ефектів, які можуть вплинути на продуктивність. Ключовим побічним ефектом розгортання є збільшення кількості операцій всередині тіла циклу. Інші оптимізації можуть використовувати цю зміну кількома способами:

1. Збільшення кількості незалежних операцій у тілі циклу може призвести до кращого розкладу інструкцій. З більшою кількістю операцій планувальник має більше шансів утримувати кілька функціональних блоків зайнятими і приховати затримку довготривалих операцій, таких як розгалуження та доступ до пам'яті.

2. Розгортання може перемістити послідовні звернення до пам'яті в одну ітерацію циклу, де компілятор може запланувати їх разом. Це може покращити локальність або дозволити використання багатослівних операцій.

3. Розгортання може виявити надмірності між ітераціями, які важче виявити в оригінальному коді. У розгорнутому циклі локальна нумерація значень знайде та усуває ці надмірності.

4. Розгорнутий цикл може оптимізуватися інакше, ніж вихідний цикл. Наприклад, збільшення кількості разів, коли змінна зустрічається всередині циклу, може змінити вагові коефіцієнти, що використовуються для вибору коду розливу в розповсюджувачі регістрів. Зміна шаблону розливу реєстрів може радикально вплинути на швидкість кінцевого коду циклу.

5. Розгорнуте тіло циклу може мати більший попит на регістри, ніж початкове тіло циклу. Якщо підвищений попит на регістри спричиняє додатковий витік регістрів, то отриманий трафік пам'яті може перевищити потенційні переваги розгортання.

Ці непрямі взаємодії набагато важче охарактеризувати та зрозуміти, ніж прямі ефекти. Вони можуть суттєво підвищити продуктивність. Вони також можуть призвести до погіршення продуктивності . Труднощі прогнозування таких непрямих ефектів спонукали деяких дослідників виступати за адаптивний підхід до вибору факторів розгортання. У таких системах компілятор пробує кілька факторів розгортання та вимірює продуктивність отриманого коду.

Модифікуємо алгоритм "loop fusion" за допомогою техніка розкрутки циклу яка полягає в тому, що за одну ітерацію обробляються не один, а кілька елементів. Ми як би «розкручуємо» цикл, збільшуючи довжину його тіла та зменшуючи кількість вітків. Таким чином виконуємо менше ітерацій, за одну ітерацію виконуємо більше корисної роботи, а накладні витрати становлять уже не таку велику долю. Розглянемо приклад застосування цієї техніки на функції що складає цілі числа:

Function combine_plus(vec *v, int *dest)

$for\ (i\ =\ 0;\ i\ <\ limit;\ i{+}{=}\ 2)$

$acc\ =\ acc\ +\ v{-}{>}\ data[i]\ +\ v{-}{>}\ data[i+1];$

$if\ (i\ <\ len)\ acc\ {+}{=}\ v{-}{>}\ data[i];$

$*\ dest\ =\ acc;$

Тепер програма додає до функції два елемента для ітерації. Зверніть увагу, що останній елемент може бути не оброблений у циклі, тому його потрібно додати до функції.

Псевдокод алгоритму оптимізуючого методу «loop fusion»:

Input: queue[] before topological sorting

Output: queue[] after topological sorting

Algorithm:

tmp_loop ←loop

Function TOPOLOGICAL_SORT(QUEUE[])

$g \leftarrow ac_g$

$index \leftarrow 0$

$i \leftarrow 1$

while $(i < g \rightarrow vertex_count_num + 1)$

if $(!\,vertex[i] \rightarrow Get_In_Edge())$

$queue[index++] \leftarrow i$

else

return

for $(i = 0; i < head; i++)$

$j \leftarrow queue[i]$

$e \leftarrow g \rightarrow Get_out_Edge(j)$

while$(e)\{$

$e1 \leftarrow e$

$e \leftarrow g \rightarrow Get_Next_Out_Edge(e)$

$sink \leftarrow g \rightarrow Get_{Sink(e1)}$

$g \rightarrow Delete_Edge(e1)$

if $(!\,vertex[i] \rightarrow Get_In_Edge())$

$queue[index++] \leftarrow sink$

return $queue[]$

2.5 Висновки

Хоча на сьогодні існує досить багато компіляторів, доступних для різних мов завжди найкращі з них виділяються. Тому їх конкурентоспроможність заснована на довговічності, оптимізації, швидкості та перевірці коду, помилок та синтаксису є досить актуальною. Через це можемо чітко зрозуміти, що компілятор є важливим стовпом для мов програмування, а його швидкість та ефективність роботи є ключовими факторами.

В даному розділі описано загальну технологію роботи компіляторів. В процесі аналізу описано основні функції алгоритмів різних етапів роботи компілятора. Переглянуто загальні характеристики цих алгоритмів та переваги і недоліки деяких з них відносно алгоритмів того ж типу. В підрозділах описано складові елементи алгоритмів кожного етапу роботи компілятора, визначено їх структуру та принципи роботи. Також розглянуто та описано загальні алгоритми оптимізації компіляторів, оскільки оптимізація компілятора впливає на одні з основних критеріїв оцінки компілятора, як ефективність, швидкість роботи та виявлення помилок при збірці.

В даному розділі описано методи та алгоритми основних етапів роботи розроблювального спеціалізованого двохпрохідного компілятора. Описано та визначено модифікований метод оптимізації «loop fusion» як основу другого проходу компілятора та його оптимізації. Модифікований алгоритм оптимізації є необхідним для організації оптимального та ефективного програмного продукту, тому в подальшому він буде застосований під час програмної реалізації продукту для поставленої задачі.

3 МОДЕЛЮВАННЯ СИСТЕМИ ДЛЯ РОЗРОБКИ КОМПІЛЯТОРА

3.1 Визначення архітектури компілятора

Проаналізувавши загальну інформацію про компілятори, знаємо що будь-який компілятор складається з двох фаз: фази аналізу та фази синтезу (рисунок 3.1). Та кожна з цих фаз ділиться на етапи, кожен з яких бере вхідні дані з попереднього етапу і подає свої результати на наступний етап роботи компілятора.

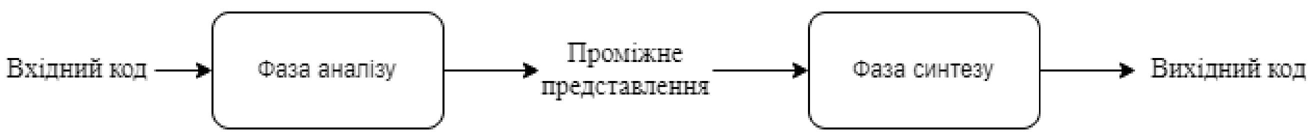


Рисунок 3.1 – Загальна структура компілятора

У реалізації даного компілятора міститься 7 етапів роботи компілятора, а саме: лексичний аналіз, синтаксичний аналіз, семантичний аналіз, генерація проміжного коду, машинно-незалежна оптимізація коду, генерація кінцевого коду та машинно-залежна оптимізація коду.

Всі ці етапи вже були детально розглянуті, але потрібно наголосити що, до машинно-незалежної оптимізації коду відносяться загальні алгоритми оптимізації коду, а до машинно-залежної оптимізації коду відноситься модифікований алгоритм “loop fusion” для оптимізації етапу генерації кінцевого коду.

Вищерозглянуті етапи роботи компілятора та їх взаємодія зображено на рисунку 3.2.

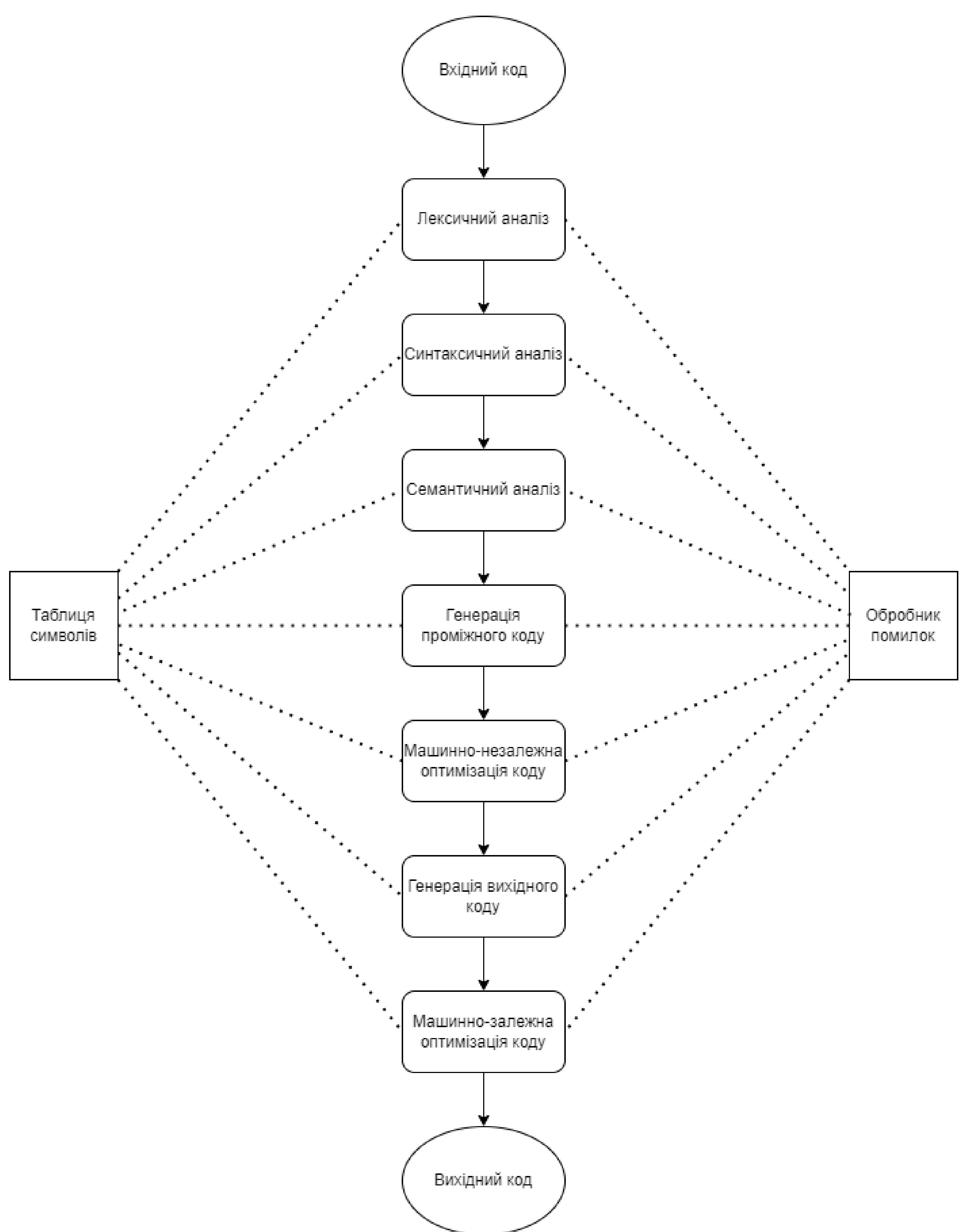


Рисунок 3.2 — Архітектура розроблюваного компілятора

3.2 Проектування складових компілятора

Першим етапом проектування компілятора буде проектування лексичного аналізатора. Лексичний аналізатор – це програма, яка перетворює вхідний рядок символів, що є текстом вхідної програми, у рядок токенів. Ця програма також має виявляти лексичні помилки.

До основних функцій лексичного аналізатора відносяться:

- знаходження лексичних помилок;
- створення хеш-таблиць;
- виділення коментарів;
- знаходження та згортання лексем.

Принцип роботи лексичного аналізатора полягає в тому, що вхідний символ вхідного тексту аналізується і за результатами перевірок визначається тип поточної лексеми. Як тільки з’являється символ, який не може містити поточний токен, це означає кінець цього токена і початок нового. Згенерований токен додається до хеш-таблиці. Схематично роботу лексичного аналізатора показано на рисунку 3.3.

Однією з функцій лексичного аналізатора є виділення коментарів. Однак при реалізації цього компілятора коментарі будуть видалені, оскільки для вирішення поставленої задачі по розробці проекту вони не мають значення. Виявлення лексичних помилок є ще однією важливою функцією лексичного аналізатора. Цей тип помилки включає:

- недопустимий символ;
- неправильна лексема;
- неочікуване закінчення файлу.

Другим етапом проектування даного компілятора буде проектування хеш-таблиці з ланцюжковим методом вирішення колізій. Для цього створимо блок-схему даного алгоритму (рисунок 3.4). Також розробимо блок схеми функцію пошуку ідентифікатора в хеш таблиці (рисунок 3.5) та функцію додавання нових ідентифікаторів до розроблюваної хеш-таблиці (рисунок 3.6).

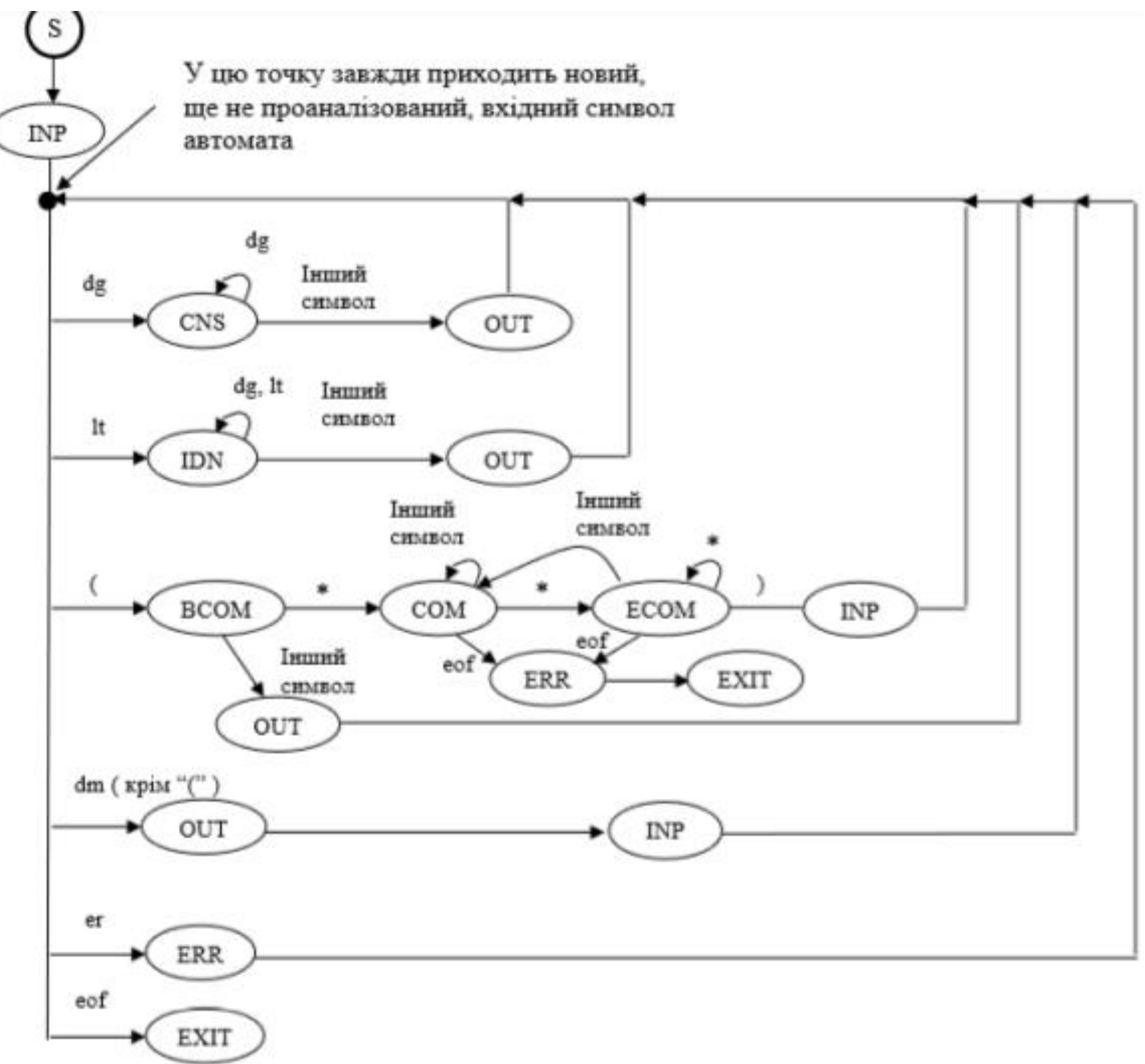


Рисунок .3.3 – Принцип роботи лексичного аналізатора

Третім етапом проектування даного компілятора буде розробка синтаксичного аналізатора. Синтаксичний аналізатор – це програма, яка перетворює вхідну послідовність токенів в абстрактне синтаксичне дерево. Для прикладу розглянемо дерево, що зображено на рисунку 3.8.

Дерево також можна зберігати як рядок, що описано у функції:

```
(fundecl unsigned gcd
  (params (param unsigned x) (param unsigned y))
  (block  (while (> x 0) (block (vardecl unsigned temp y) (= x (% y x)) (= y temp)))
    (return y)))
```

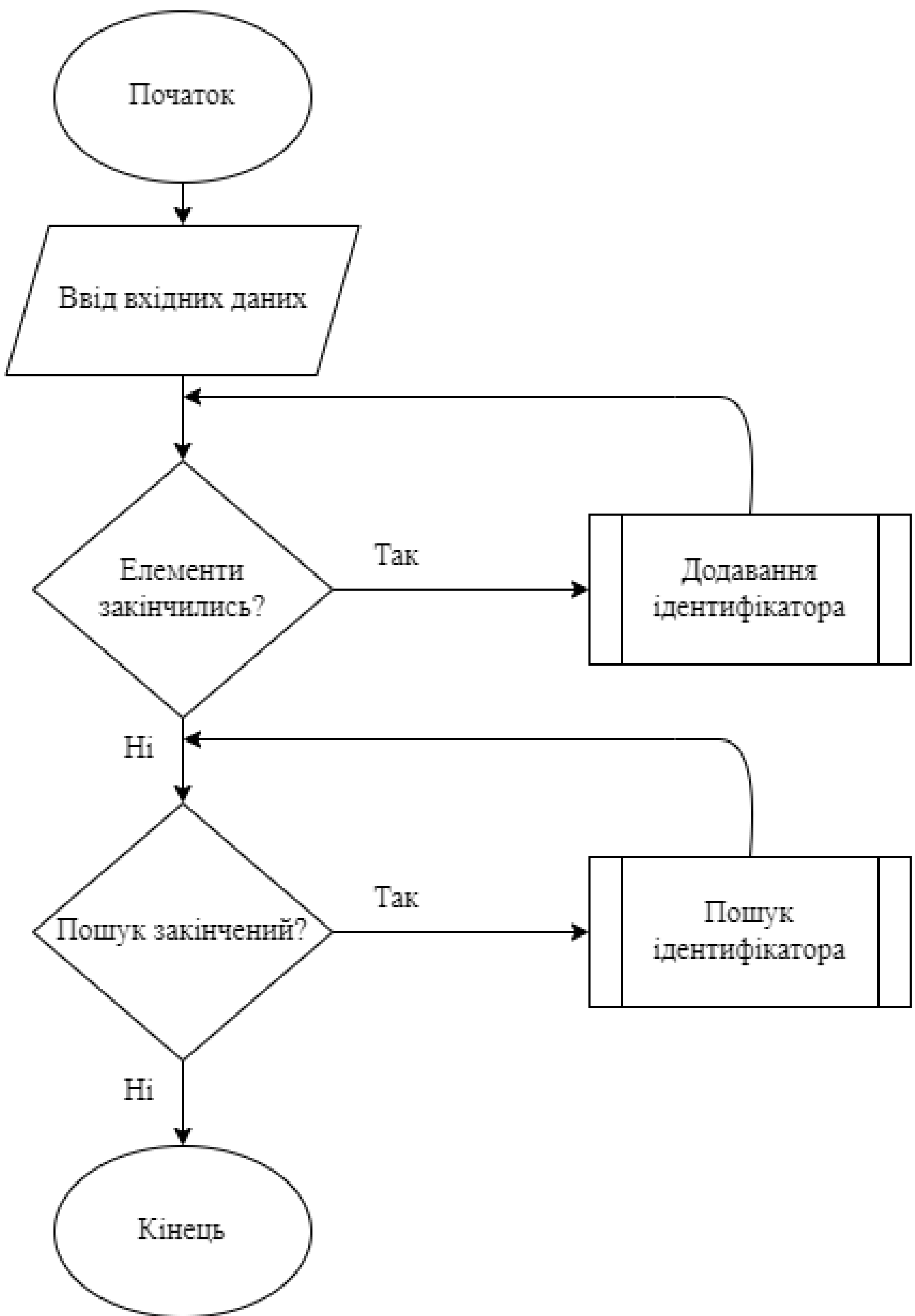


Рисунок 3.5 – Блок схема хеш-таблиці

Технічно кожен вузол у AST зберігається як об’єкт з іменованими полями, багато з яких самі є вузлами дерева. Зауважте, що на цьому етапі компіляції дерево, безумовно, є просто деревом. Немає циклів.

При створенні синтаксичного аналізатора потрібно звернути увагу на складність граматики (наприклад, чи є граматика LL чи LR), а також чи існують якісь правила зняття неоднозначностей, які, можливо, доведеться зламати. Деякі мови насправді вимагають трохи семантичного аналізу, щоб розібрати.

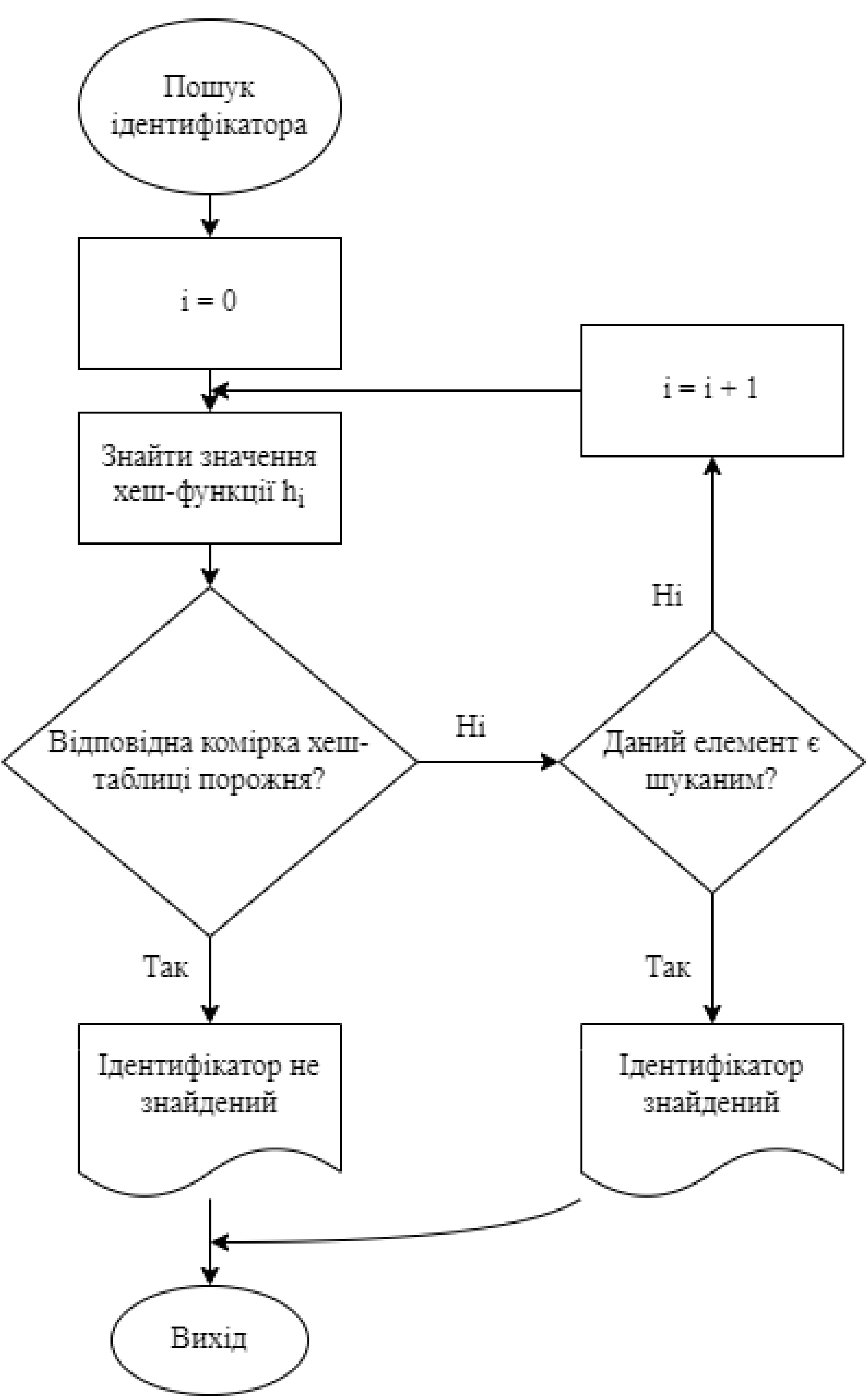


Рисунок 3.6 – Блок схема функції пошуку ідентифікатора

Схематичний опис принципу роботи синтаксичного аналізу зображено на рисунку 3.9.

Помилки, які можуть виникати під час аналізу, і називаються синтаксичними помилками, включають такі речі, які вказані нижче:

– j = 4 * (6 − x;
– i = /5;
– 42 = x * 3.

Немає необхідності фактично відокремлювати сканування (лексичний аналіз/токенізацію) від синтаксичного аналізу/генерування дерева.

Системи, засновані на PEG, насправді не мають сканера: вони виконують синтаксичний аналіз у прогнозному вигляді, змішавши разом лексичні та синтаксичні правила.

Четвертим етапом проектування буде розробка семантичного аналізу.

Під час семантичного аналізу потрібно перевірити правила законності, і при цьому зв'язуємо частини синтаксичного дерева (розв'язуючи посилання на ідентифікатори, вставляючи операції приведення для неявних примусів тощо), щоб сформувати семантичний граф.

Набори правил законності різний для кожної мови. Приклади правил законності, які ви можете побачити в мові програмування C, включають:

– кілька декларацій змінної в межах області видимості;
– посилання на змінну перед її оголошенням;
– посилання на ідентифікатор, який не має декларації;
– порушення правил доступу (публічних, приватних, захищених, ...);
– занадто багато аргументів у виклику методу;
– недостатньо аргументів у виклику методу;
– невідповідність типів (їх багато).

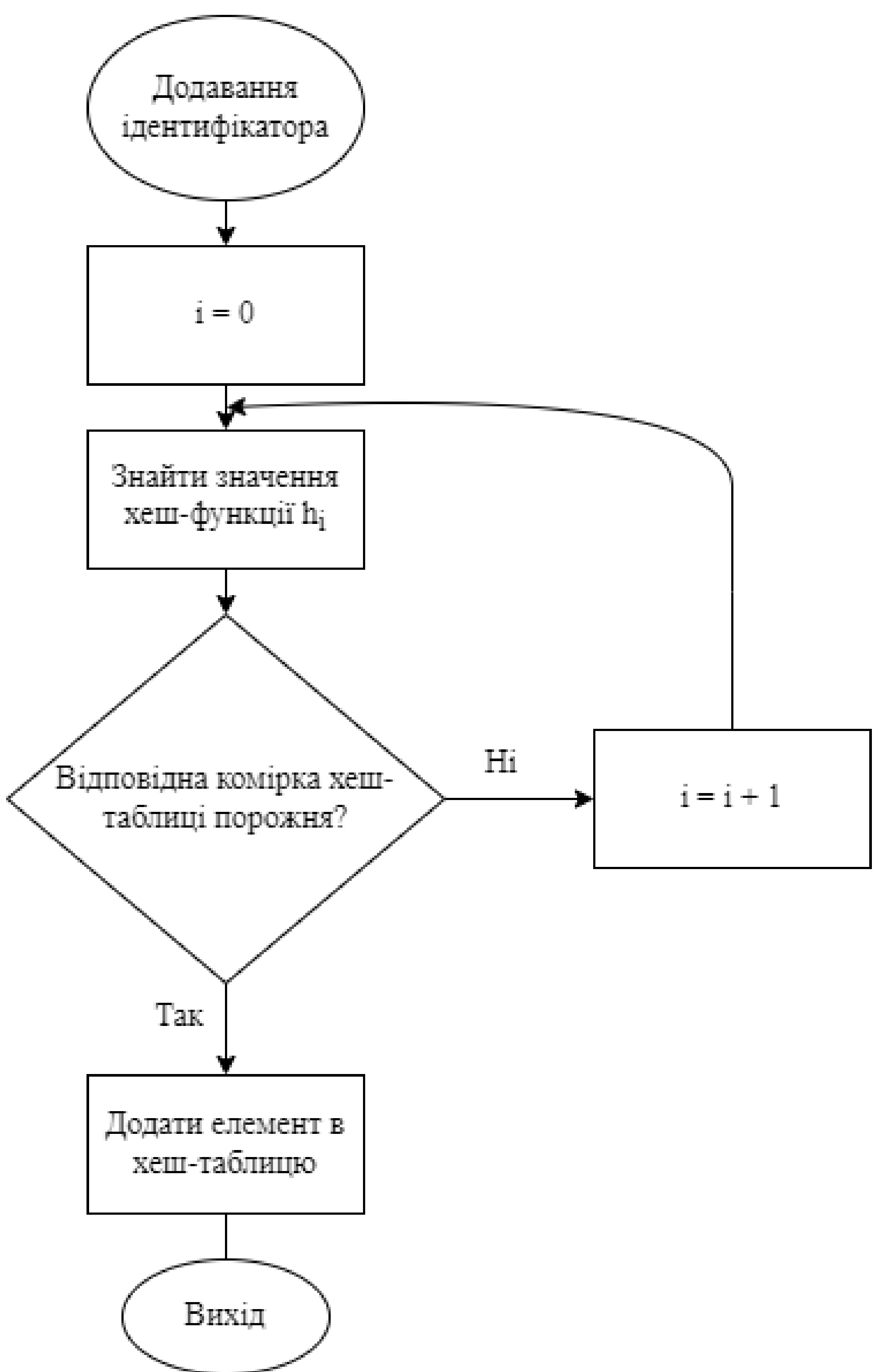


Рисунок 3.7 – Блок схема функції додання нового ідентифікатора

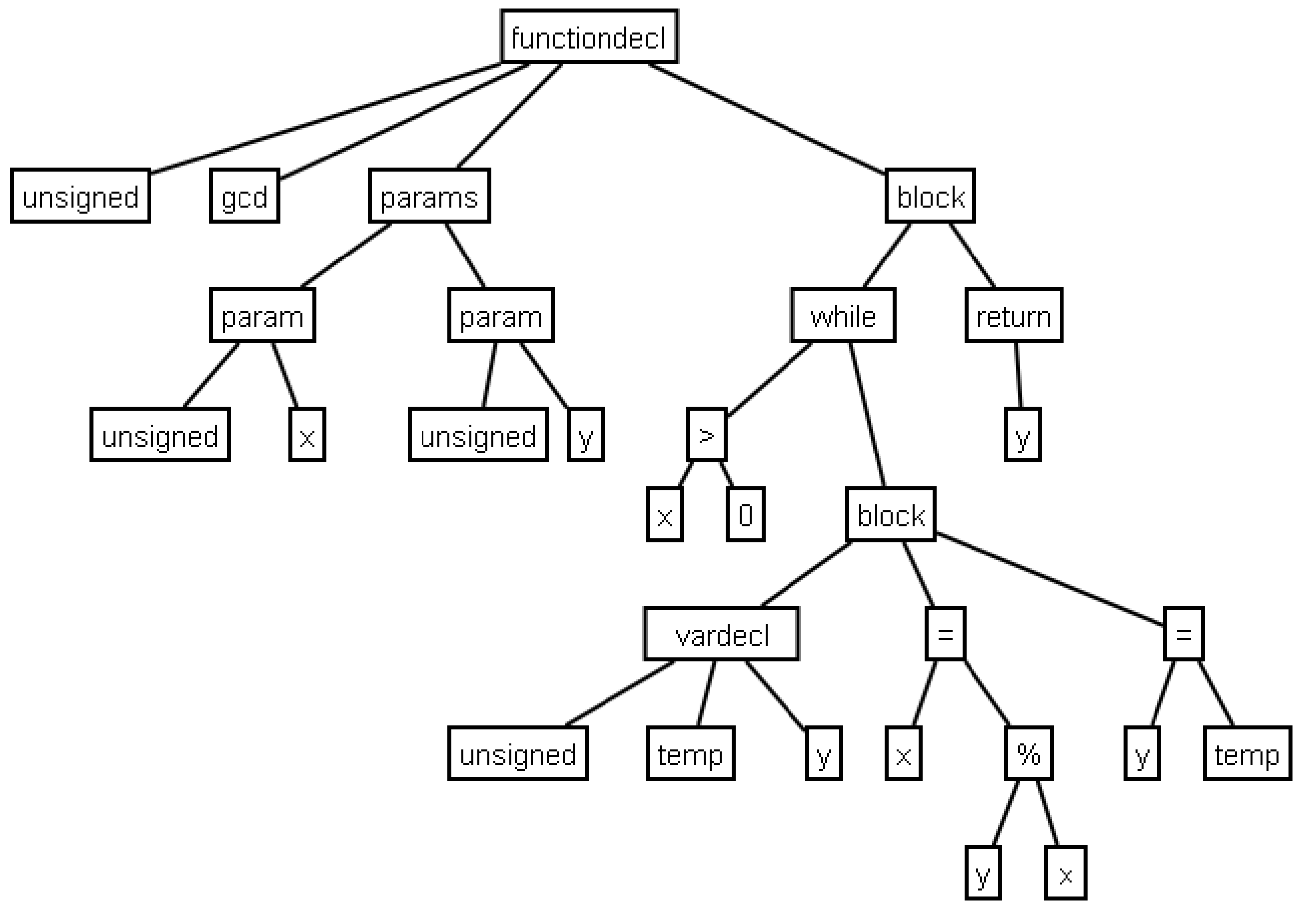


Рисунок 3.8 – Абстрактне синтаксичне дерево

Помилки, що виникають на цьому етапі, називаються статичними семантичними помилками.

П’ятим етапом проектування буде проектування генерації коду. Генератор коду створює потоковий графік, що складається з кортежів, згрупованих у основні блоки зображено на рисунку 3.11.

3.3 Проектування модифікованого методу оптимізації “loop fusion”

Перед початком проектування модифікованого методу оптимізації “loop fusion” розглянемо які є вимоги до його застосування.

Для того, щоб два цикли, були з'єднані, вони повинні задовольняти наступним умовам:

1) обидва цикли повинні бути суміжними;
2) обидва цикли повинні повторюватись однакову кількість разів;

3) обидва цикли повинні бути еквівалентними;

4) між циклами не може бути жодних недійсних залежностей.

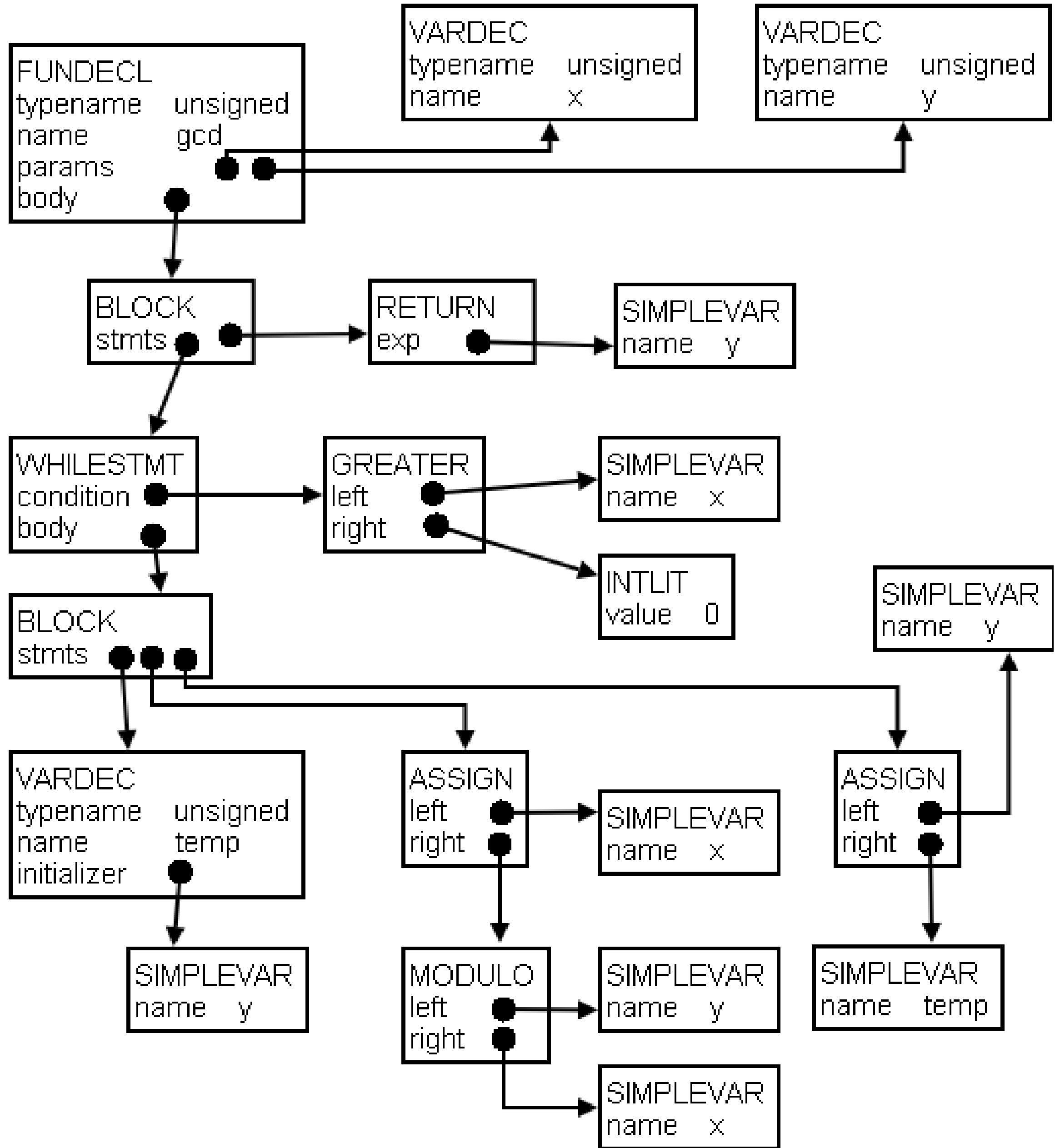


Рисунок 3.9 – Принцип роботи синтаксичного аналізатора

Опишемо як виглядає модифікований алгоритм оптимізації “loop fusion”:

1) для кожного рівня коду, від крайнього до внутрішнього алгоритм збирає цикли, які є кандидатами на з’єднання циклів;

2) алгоритм сортує список кандидатів у еквівалентні набори;

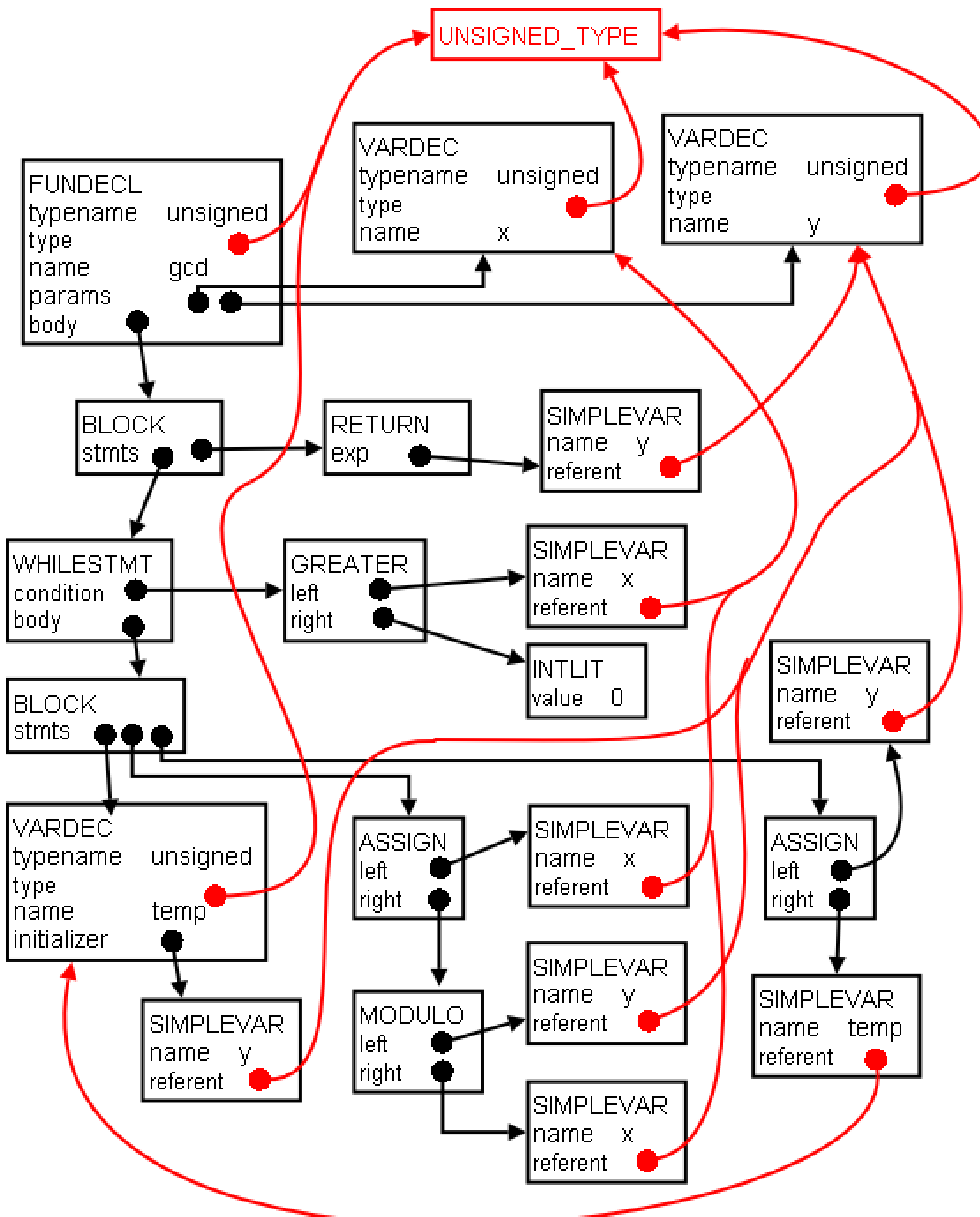


Рисунок 3.10 – Схематика принципу роботи семантичного аналізу

3) перевірка умови, що цикли мають однакову кількості ітерацій, якщо ні, алгоритм пропускає дану пару циклів;

4) перевірка умови, що цикли суміжні, якщо ні, алгоритм пропускає дану пару циклів;

5) перевірка умови, що цикли мають дійсні залежності, якщо ні, алгоритм пропускає дану пару циклів;

6) об’єднання пари циклів;

7) застосування модифікації “розкрутки циклу”;

8) оновлює список кандидатів на об’єднання, та якщо ще є кандидати, алгоритм повертається на крок 3.

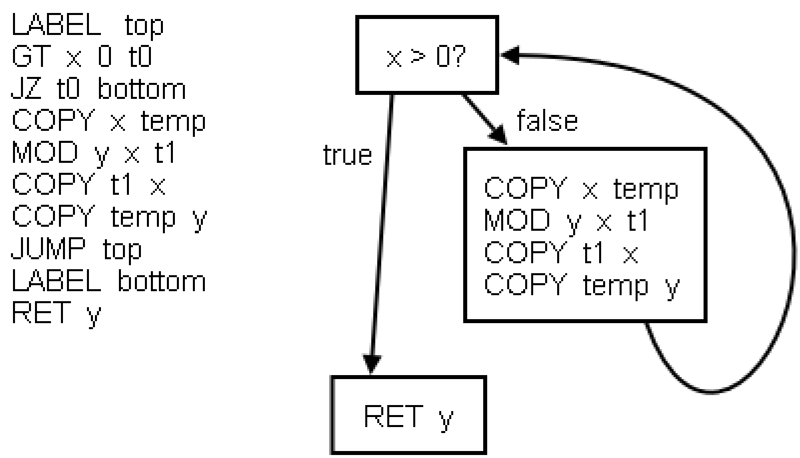


Рисунок 3.11 – Принцип роботи генерації коду

Блок-схема модифікованого алгоритму оптимізації «loop fusion» зображена на рисунку 3.12.

Прохід злиття циклу виконується на різних типах входів. Мова програмування C містить програми з усіма видами циклів, як одиночно вкладені так і багато вкладені цикли, упорядковані цикли, for, while і do-while тощо.

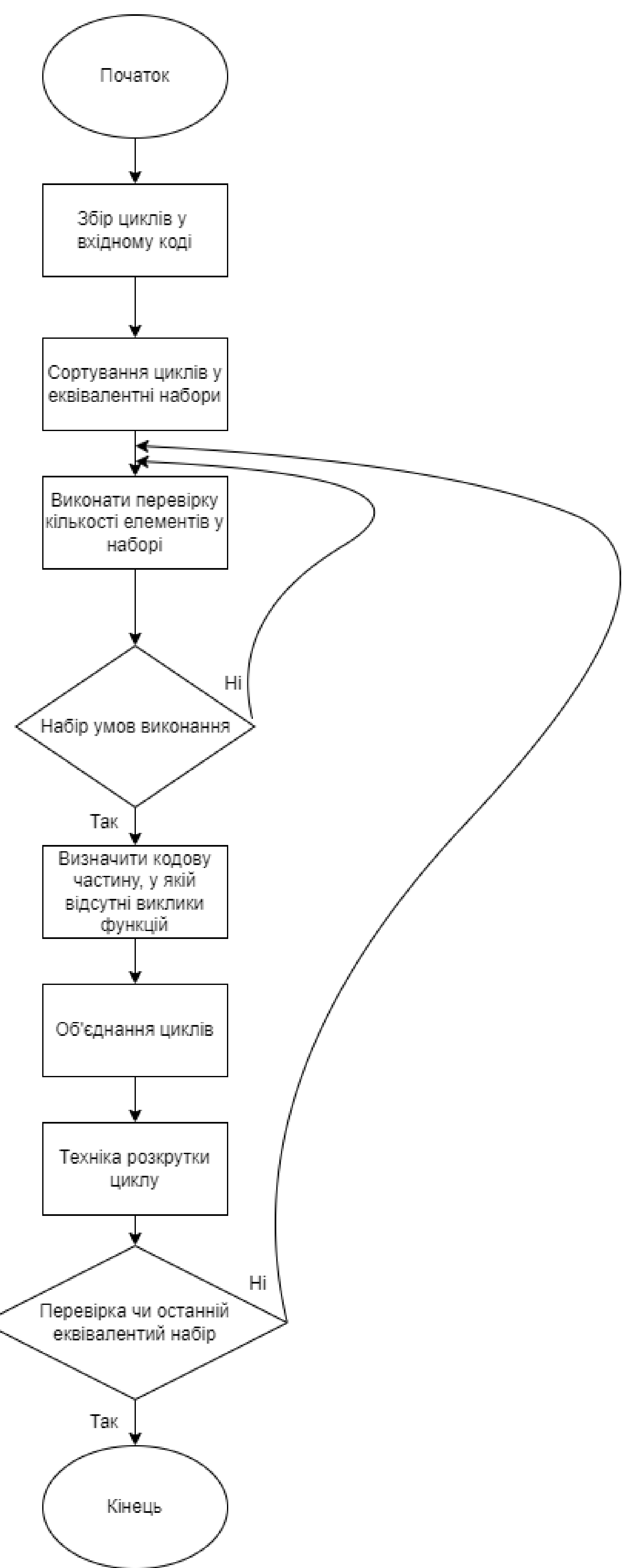


Рисунок 3.12 – Блок-схема модифікованого алгоритму «loop fusion»

3.4 Висновки

На сьогодні існує безліч різних компіляторів для всіх мов програмування. Компілятори постійно вдосконалюються, підходи, що використовуються ними, покращуються. Але питання оптимізації компіляторів і донині є актуальним. Найбільш поширеними цілями оптимізації є скорочення часу виконання програми, підвищення продуктивності, компактифікація програмного коду, економія пам'яті, мінімізація енерговитрат, зменшення кількості операцій вводу-виводу.

В даному розділі розглянуто загальну архітектуру компіляторів. Проналізувавши дану інформацію, визначено архітектуру розроблюваного компілятора, де схематично зображено всі етапи роботи компілятора та взаємодії між ними. Також в даному розділі описано та проектовано лексичний аналізатор, хеш-таблицю з ланцюжковим методом вирішення колізій, синтексичний аналізатор, семантичний аналізатор та генерацію коду.

Далі описано та проектовано модифікований метод оптимізації «loop fusion». Описано вимоги до його застосування, а саме суміжність, кількість ітерацій, еквівалентність та відсутність недійсних залежностей. Також схематично описано принцип роботи даного алгоритму, та зображено його на блок-схемі.

4 РОЗРОБКА ПРОГРАМНО-ТЕХНІЧНОГО ЗАСОБУ

4.1 Програмно-технічна реалізація компілятора

4.1.1 Виконавчий файл

Весь програмний код буде розбито на файли відповідно їх функціоналу. Перш за все створимо основний файл “main.c” що буде містити функцію main та буде виконуваним файлом. Для початку підключимо хедери інших файлів цього ж проекту. Хедери – це заголовні файли, які мають розширення .h. Метою заголовних файлів є зручне зберігання набору оголошень об'єктів для їхнього подальшого використання в інших програмах. Наприклад був підключений хедер файлу “parser.c”:

```
#include "parser.h"
```

Далі встановимо дві директиви, для зручності написання коду, які будуть відповідати за розмір та ім’я вхідного файлу який буде компілюватися нашим компілятором. Директива #define визначає ідентифікатор та послідовність символів, якою буде замінюватись цей ідентифікатор при його виявленні в тексті програми. Ідентифікатор також називається іменем макросу, а процес заміщення називається підстановкою макросу.

```
#define MIN_ARGS 2
#define SOURCE_ARG 1
```

Головна функція main буде мати два параметри, перший з яких зчитує розмір файлу, а другий ім’я файлу який буде компілюватися.

```
int main(int argc, char **argv)
```

Першим елементом у головному файлі буде перевірка на наявність файлу який буде компілюватися. В разі його відсутності буде видавати помилку.

```
fatal_error("No file provided! Usage: compiler <source>");
```

Далі йде функція яка буде відкривати доступ до змісту файлу, що потрібно щоб його скомпілювати.

```
open_file(argv[SOURCE_ARG]);
```

Далі йде набір функцій які і будуть компілювати вхідний файл до етапу генерація проміжного коду, в даному випадку до рівня асемблера. Окремо кожну з них розглянемо пізніше.

```
byte *code = compile(parse(lex()));
```

Потім використовуємо функцію закриття файлу для оптимізації часу загального виконання програми.

```
close_file();
```

Перед виконанням коду асемблера та початку етапу генерації вихідного коду, виведемо асемблер код, який попередньо був скомпільований в змінну code.

```
print_asm(code);
```

Тепер виконаємо асемблер код та згенеруємо вихідний код.

```
run(code);
```

Останнім елементом головного файлу буде функція завершення роботи програми, яка буде повертати значення 0 в разі успішного завершення роботи.

```
return EXIT_SUCCESS;
```

4.1.2 Таблиця символів

Перш ніж почнемо розробку лексичного та синтаксичного аналізаторів, створимо таблицю символів як структуру даних у файлі “sym.c”. Підключимо дві стандартні зовнішні бібліотеки, перша з яких відповідає за виділення пам'яті, контроль процесу виконання програми та перетворенням типів, а друга містить функції для роботи з нуль-термінованими рядками і різними функціями роботи з пам'яттю.

```
#include <stdlib.h>
#include <string.h>
```

Створимо таблицю символів за допомогою оголошення глобальної змінної symbol_table типу Table.

```
struct Table* symbol_table[MAX_SYMBOL_TABLE_SIZE];
```

Таблиця символів матиме 4 функції:

1) Отримання ідентифікатора символа:
   struct Table *get_sym(int id)
2) Додавання нового символу до таблиці символів:
   int add_sym(char *name)
3) Отримання значення символу за його ідентифікатором:
   void set_sym(int id, int val)
4) Отримання розміру таблиці символів

int get_table_size()

4.1.3 Лексичний аналізатор

Далі створимо зміст файлу “lex.c”, який і буде відповідати за етап лексичного аналізу. Перш за все підключимо дві стандартні зовнішні бібліотеки, перша з яких відповідає за виділення пам'яті, контроль процесу виконання програми та перетворенням типів, а друга бібліотека відповідає за оголошення функцій для класифікації символів.

```
#include <stdlib.h>
#include <ctype.h>
```

Оскільки, головною метою лексичного розбору є перетворення послідовності символів в послідовність токенів, додамо нову глобальну змінну tok типу struct Token, яка і буде послідовністю токенів.

```
struct Token tok;
```

Далі створимо функцію токенізації яка утворює позначки з вихідного потоку та впорядковує їх за відповідними типами. Всього буде 3 типи: ідентифікатор, число та знаки. Для зручності спочатку перевіряємо чи вхідні символи є знаками як виключення, оскільки їх завжди найменше. Як приклад токенізації знаку “+”:

```
case '+': tok.type = OP1; tok.attr = ADD_TYPE; break;
```

Наступним йде перевірка чи вхідний символ є ідентифікатором за допомогою функції isalpha(), яка перевіряє чи вхідний символ є буквою алфавіту.

```
if (isalpha(ch)) {
```

Після проходження перевірки додаємо його до послідовногості токенів та зберігаємо його та його тип.

```
tok.type = ID;
tok.attr = add_sym(id_name);
```

Наступною йде перевірка чи вхідний символ є числом за допомогою функції isdigit(), яка перевіряє чи вхідний символ є символом десяткової цифри.

```
else if (isdigit(ch)) {
```

Після проходження перевірки додаємо його до послідовногості токенів та зберігаємо його та його тип.

```
tok.type = NUM;
tok.attr = val;
```

Останнія умовою є те, якщо вхідний символ не пройшов жодну з 3 перевірок, викликається функція помилки.

```
fatal_error("Lexer: Unexpected symbol");
```

4.1.4 Синтаксичний аналізатор

Наступним кроком буде створення файлу “parser.c”, що буде відповідати за синтаксичний аналіз. Синтаксичний аналізатор буде працювати в два етапи: на першому ідентифікуються осмислені токени, на другому створюється дерево розбору.

Для початку підключимо дві стандартні зовнішні бібліотеки, перша з яких містить визначення макросів, констант та оголошення функцій і типів, призначених для виконання операцій введення і виведення, а друга відповідає за виділення пам'яті, контроль процесу виконання програми та перетворенням типів.

```
#include <stdio.h>
#include <stdlib.h>
```

Першим етапом є ідентифікація вже створеної послідовності токенів лексичним аналізатором, яка перевіряє істинність послідовності токенів відносно таблиці символів.

if (tok.type == type1 && lookahead().type == type2) {

accept(type1); accept(type2);

В разі помилки виконується функція:

if (!accept(type)) { fatal_error("Parser: Syntax error");

Другий етап є створенням дерева розбору. Дерево розбору реалізуємо за допомогою зв'язних списків типу Node. Реалізація зв'язаного списку в С здійснюється за допомогою покажчиків. Пов'язаний список складається з багатьох вузлів, які пов'язані між собою. Кожен вузол в основному розділений на дві частини, одна частина містить дані, а інша частина підключена до іншого вузла. Схематика роботи звязних списків зображено на рисунку 4.1.

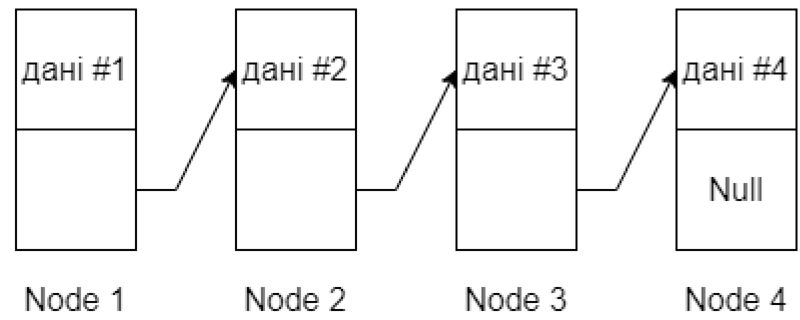


Рисунок 4.1 – Схематика зв'язних списків

Для створення дерева розбору оголосимо глобальну змінну типу Node.

struct Node* node = safe_malloc(sizeof(struct Node));

Для запису символа з послідовності токенів до дерева розбору зчитуються символ та його тип, та перевіряється відповідно типу даних дерева, після чого символ та його тип записуються до дерева.

```
if (accept(ID)) {
    node->type = VAR_TYPE;
    node->val = tok_attr;
```

В разі помилки виконується функція:

fatal_error("Parser: Unexpected factor");

4.1.5 Програмна реалізація модифікованого методу «loop fusion»

Метод “loop fusion” – це оптимізація компілятора та перетворення циклу, яке замінює декілька циклів одним. Реалізація оптимізації за допомогою цього методу представлена у вигляді перевантажених математичних функцій, які будуть заміщувати стандартні математичні функції мови С, при виклику їх у вхідному коді компільованої програми.

Підключимо набір зовнішніх бібліотек:

1. #include <math.h> – надає прототипи функцій, розроблених для виконання простих математичних операцій.
2. #include <assert.h> – цей макрос реалізує висловлювання, котре може використовуватись для отримання припущень зроблених програмою.
3. #include <memory> – визначає загальні утиліти для керування динамічною пам'яттю.
4. #include <iostream.h> – використовується для організації введення-виведення в мові програмування С.

Після підключення всіх необхідних бібліотек необхідно розробити внутрішній конструктор, який створює неініціалізований масив який буде мати параметри довжини масиву та його змісту.

```
Array(size_t n) : length(n), data(new float[n]) { }
```

Далі створимо функцію типу public для створення масиву в діапазоні цілих чисел, де верхня межа є ексклюзивною.

```
static Array Range(size_t start, size_t end)
```

Також створимо функції що будуть відповідати за основні операції з масивами, наприклад визначення довжини масиву та повернення значення елементу масиву за його індексом.

```
size_t size() const { return length; }
float& operator[](size_t i) { return data[i]; }
```

Далі оголошуємо перевантажений оператор додавання як функцію вільного друга, де синтаксис програми визначає operator+ як вільну функцію, яка є другом цього класу, незважаючи на те, що вона з'являється як оголошення функції-члена.

```
friend Array operator+(const Array& a, float b) {
    Array c(a.size());
    for (size_t i = 0; i < a.size(); ++i) {
        c[i] = a[i] + b; }
    return c; }
```

Аналогічно, можемо визначити перевантаження для всіх математичних функцій в яких використовуються цикли, але як приклад розглянемо ще перевантаження для функції sin().

```
friend Array sin(const Array& a) {
    Array b(a.size());
    for (size_t i = 0; i < a.size(); ++i) {
        b[i] = std::sin(a[i]); }
    return b; }
```

Також додамо функцію розкрутки циклу яка полягає в тому, що за одну ітерацію обробляються не один, а кілька елементів.

```
void combine_plus(vec *v, int *dest)
```

4.2 Обробка та аналіз тестових наборів даних

В якості тестових наборів даних було взято 3 програми, в яких зосереджена робота з циклами, а саме з циклами, які підпорядковуються умовам злиття для можливості застосування модифікованого методу оптимізації "loop fusion".

Перед тим, як застосовувати ці тестові дані, була проведена попередня обробка. Вона полягає у тому, щоб ці дані пройшли валідацію та перевірку на цілісність та коректність, тобто всі тестові програми не мають помилок та готові до компіляції.

Першим тестовим набором даних буде виступати програма “testing_program1”, яка обчислює мінімальний час отримання посилок від відділення пошти до адресата.

Проведемо компіляцію даної програми за допомогою розробленого компілятора і визначимо час компіляції програми. Для цього потрібно в параметрах головної функції main вказати шлях до програми яку збираємося компілювати, що зображено на рисунку 4.2.

```
16  {
17      argv = "D:\testing_program1.cpp";
18      if (argc < MIN_ARGS) {
```

Рисунок 4.2 – Вказання шляху до компільованої програми

Далі після дебагінгу можемо запустити програму, і побачити виведення та збереження у окремий файл проміжного представлення компільованої програми (асемблер код), що зображено на рисунку 4.3 та в терміналі Microsoft Visual Studio можемо подивитися на час виконання компіляції даної тестової програми, а саме “testing_program1” (рисунок 4.4).

```
512         cmp     DWORD PTR [rbp-4], 1
513         jne     .L554
514         cmp     DWORD PTR [rbp-8], 65535
515         jne     .L554
516         mov     edi, OFFSET FLAT:_ZStL8__ioinit
517         call    std::ios_base::Init::Init() [complete object constructor]
518         mov     edx, OFFSET FLAT:__dso_handle
519         mov     esi, OFFSET FLAT:_ZStL8__ioinit
520         mov     edi, OFFSET FLAT:_ZNSt8ios_base4InitD1Ev
521         call    __cxa_atexit
522 .L554:
523         nop
524         leave
525         ret
526 _GLOBAL__sub_I_main:
527         push    rbp
528         mov     rbp, rsp
529         mov     esi, 65535
530         mov     edi, 1
531         call    __static_initialization_and_destruction_0(int, int)
532         pop     rbp
533         ret
```

Рисунок 4.3 - Вивід проміжного представлення (асемблер коду) компільованої програми

```
'compiler_2.exe' (Win32): Loaded 'C:\Windows\System32\KernelBase.dll'.
'compiler_2.exe' (Win32): Loaded 'C:\Windows\System32\vcruntime140d.dll'.
'compiler_2.exe' (Win32): Loaded 'C:\Windows\System32\ucrtbased.dll'.
'compiler_2.exe' (Win32): Loaded 'C:\Windows\System32\ucrtbased.dll'.
'compiler_2.exe' (Win32): Unloaded 'C:\Windows\System32\ucrtbased.dll'
'compiler_2.exe' (Win32): Loaded 'C:\Windows\System32\kernel.appcore.dll'.
'compiler_2.exe' (Win32): Loaded 'C:\Windows\System32\msvcrt.dll'.
'compiler_2.exe' (Win32): Loaded 'C:\Windows\System32\rpcrt4.dll'.
The thread 0x1ae0 has exited with code 1 (0x1).
The thread 0x2608 has exited with code 1 (0x1).
The thread 0x2f80 has exited with code 1 (0x1).
The program '[12968] compiler_2.exe' has exited with code 1 (0x1).
```

Рисунок 4.4 – Компіляція першої тестової програми

З рисунку 4.4 бачимо, що час компіляції даної тестової програми займає 1296 ms та не виникло жодних помилок під час компіляції створеним компілятором.

Далі проведемо компіляцію наступної тестової програми за допомогою розроблювального компілятора. Другим тестовим набором даних буде програма “testing_program2”, яка є програмною реалізацією алгоритму Дейкстри. Також спочатку вводимо шлях до тестової програми та проводимо дебагінг. Після можемо запустити компілятор та переглянути час виконання компіляції даної тестової програми, що зображено на рисунку 4.5.

```
'compiler_2.exe' (Win32): Loaded 'C:\Windows\System32\ntdll.dll'.
'compiler_2.exe' (Win32): Loaded 'C:\Windows\System32\kernel32.dll'.
'compiler_2.exe' (Win32): Loaded 'C:\Windows\System32\KernelBase.dll'.
'compiler_2.exe' (Win32): Loaded 'C:\Windows\System32\vcruntime140d.dll'.
'compiler_2.exe' (Win32): Loaded 'C:\Windows\System32\ucrtbased.dll'.
'compiler_2.exe' (Win32): Loaded 'C:\Windows\System32\kernel.appcore.dll'.
'compiler_2.exe' (Win32): Loaded 'C:\Windows\System32\msvcrt.dll'.
'compiler_2.exe' (Win32): Loaded 'C:\Windows\System32\rpcrt4.dll'.
The thread 0x173c has exited with code 1 (0x1).
The program '[11672] compiler_2.exe' has exited with code 1 (0x1).
```

Рисунок 4.5 – Компіляція другої тестової програми

З рисунку 4.5 бачимо, що час компіляції даної тестової програми займає 1167 ms та не виникло жодних помилок під час компіляції створеним компілятором.

І третім тестовим набором даних буде програма “testing_program3”, яка є реалізацією вирішення задачі перекачки палива з різних баків для встановлення

одного рівня палива у всіх баках. Також спочатку вводимо шлях до тестової програми та проводимо дебагінг. Після можемо запустити компілятор та переглянути час виконання компіляції даної тестової програми, що зображено на рисунку 4.6.

```
'compiler_2.exe' (Win32): Loaded 'C:\Windows\System32\ntdll.dll'.
'compiler_2.exe' (Win32): Loaded 'C:\Windows\System32\kernel32.dll'.
'compiler_2.exe' (Win32): Loaded 'C:\Windows\System32\KernelBase.dll'.
'compiler_2.exe' (Win32): Loaded 'C:\Windows\System32\vcruntime140d.dll'.
'compiler_2.exe' (Win32): Loaded 'C:\Windows\System32\ucrtbased.dll'.
'compiler_2.exe' (Win32): Loaded 'C:\Windows\System32\kernel.appcore.dll'.
'compiler_2.exe' (Win32): Loaded 'C:\Windows\System32\msvcrt.dll'.
'compiler_2.exe' (Win32): Loaded 'C:\Windows\System32\rpcrt4.dll'.
The thread 0x2e5c has exited with code 1 (0x1).
The thread 0x2e6c has exited with code 1 (0x1).
The program '[10524] compiler_2.exe' has exited with code 1 (0x1).
```

Рисунок 4.6 – Компіляція третьої тестової програми

З рисунку 4.6 бачимо, що час компіляції даної тестової програми займає 1052 ms та не виникло жодних помилок під час компіляції створеним компілятором.

4.3 Оцінка ефективності розроблюваної системи

Для оцінки ефективності розробленого спеціалізованого двопрохідного компілятора проведемо компіляцію тестових наборів даних за допомогою існуючого компілятора «x86-64 gcc 4.7.1»та розробленого в даній роботі і порівняємо результати час компіляції тестових програм на обох компіляторах.

GNU Compiler Collection (зазвичай використовують скорочення GCC) — набір компіляторів для різних мов програмування, розроблений у рамках проекту GNU. GCC є вільним програмним забезпеченням, поширюється у тому числі фондом вільного програмного забезпечення (FSF) на умовах GNU GPL та GNU LGPL та є ключовим компонентом GNU toolchain. Він використовується як стандартний компілятор для вільних UNIX-подібних операційних систем.

Спочатку названий GNU C Compiler, підтримував лише мову C. Пізніше GCC був розширений для компіляції вихідних кодів мовами програмування, як C++, Objective-C, Java, Фортран, Ada, Go, GAS і D.

Для використання компілятора «x86-64 gcc 4.7.1» скористуємося веб-додатком Compiler Explorer і в полі вибору компіляторів оберемо потрібний нам (рисунок 4.7).

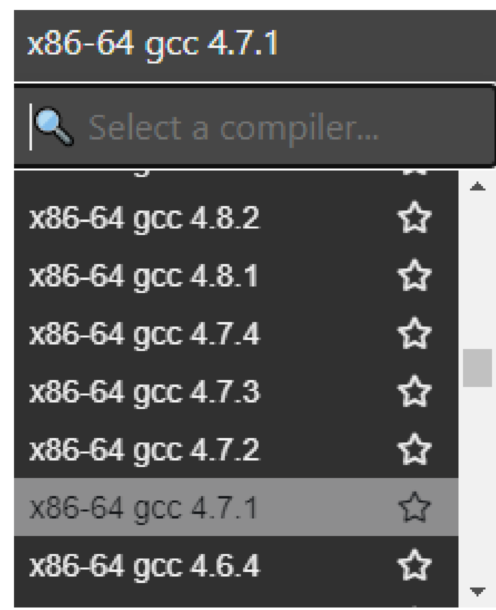


Рисунок 4.7 – Вибір компілятора

Далі проведемо компіляцію першої тестової програми та переглянемо час компіляції програми (рисунок 4.8).

Output (0/5) x86-64 gcc 4.7.1 - 1759ms (839702B) ~45689 lines filtered

Рисунок 4.8 – Компіляція першої тестової програми

Аналогічно проведемо компіляцію другої тестової програми (рисунок 4.9) та третьої тестової програми (рисунок 4.10).

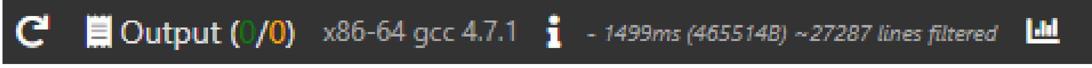


Рисунок 4.9 – Компіляція другої тестової програми

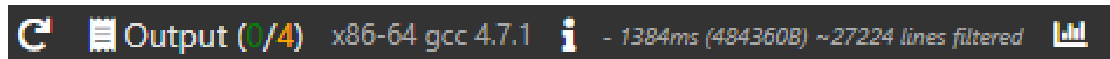


Рисунок 4.10 – Компіляція третьої тестової програми

Отримані дані, які були отримані під час тестування, виведемо їх у графічний формат вигляду та створимо діаграму (рисунок 4.11) для зручного візуального перегляду часу компіляції тестових програм на обох компіляторах.

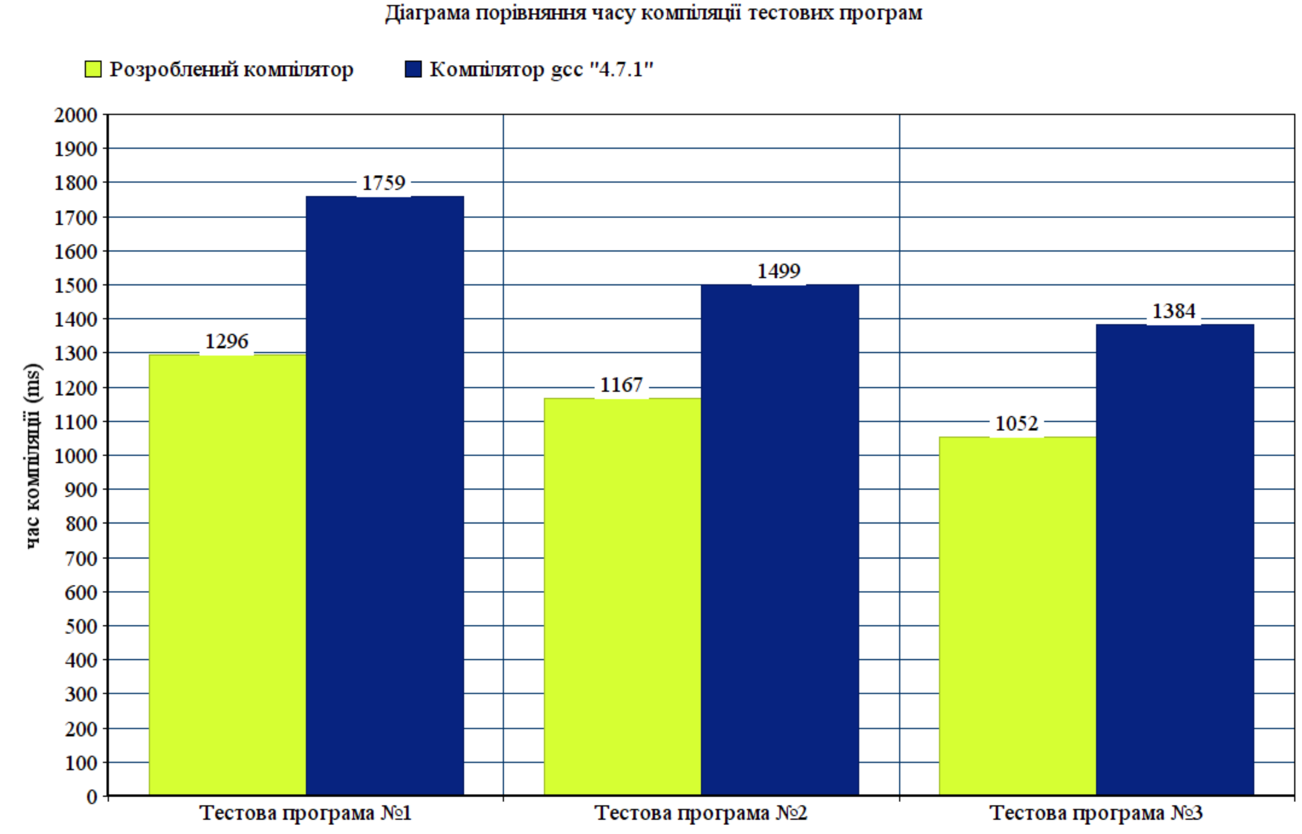


Рисунок 4.11 – Діаграма порівняння часу компіляції програм

Як бачимо з діаграми, час компіляції тестових програм на розробленому спеціалізованому двопрохідному компілятору з модифікованим методом оптимізації «loop fusion» є меншим за час компіляції тих ж тестових програм компілятором gcc 4.7.1. Різниця в часі компіляції сягає від 300ms до 450ms що є значним покращенням часу виконання компіляції програм.

Час компіляції першої тестової програми став швидшим на 26%, час компіляції другої тестової програми став швидшим на 22% і час компіляції третьої тестової програми став швидшим на 24%.

4.4 Висновки

В даному розділі програмно реалізовано розроблюваний програмний продукт. Було розподілено код на файли згідно їх функціоналу, наприклад створені файли лексичного аналізатора, синтаксичного аналізатора та описано взаємодію між ними. Описано ключові елементи розробки компілятора. Також описано функціонал створених функцій та технологій їх використання. В даному розділі програмно реалізовано модифікований метод “loop fusion”, де описано як та за допомогою яких бібліотек його було реалізовано. Описано як впливає реалізований метод на оптимізацію компільованого коду.

Далі проведено обробку та аналіз тестових даних, де було обрано 3 тестових програми в яких зосереджена робота з циклами, які задовольняють умови використання оптимізуючого методу «loop fusion». Перед застосуванням цих тестових наборів даних, проведено попередню обробку, яка полягає у тому, щоб ці дані пройшли валідацію та перевірку на цілісність та коректність, тобто всі тестові програми немають помилок та готові до компіляції. Далі проведено компіляцію даних тестових програм та було визначено час їх компіляції.

В даному розділі проведено оцінку ефективності розробленого компілятора. Для цього було прокомпільовано всі тестові набори даних за допомогою компілятора «gcc 4.7.1» та визначено час компіляції цих тестових програм за допомогою розробленого компілятора. Проаналізовано результати тестування та визначено ефективність та дієвість модифікованого методу оптимізації «loop fusion».

ВИСНОВКИ

У даній кваліфікаційній роботі магістра за результатами теоретичних та практичних досліджень розроблено методи та засоби створення спеціалізованого двохпрохідного компілятора.

У першому розділі досліджено предметну область, описано структуру компілятора та принципи його роботи. В даному розділі розглянуто етапи та фази цих етапів процесу компіляції. Розглянуто переваги та недоліки однопрохідних та багатопрохідних компіляторів. Проаналізовано уже існуючі методи оптимізації компіляції. В результаті аналізу проблеми низької швидкодії компіляторів була сформована задача на оптимізацію та покращення процесу компіляції. За допомогою проведених досліджень вивели вимоги і завдання для подальшого вдосконалення програмного продукту.

У другому розділі описано загальну технологію роботи компіляторів. В процесі аналізу розглянуто основні функції алгоритмів різних етапів роботи компілятора. Переглянуто загальні характеристики цих алгоритмів та переваги і недоліки деяких з них відносно алгоритмів того ж типу. В підрозділах описано складові елементи алгоритмів кожного етапу роботи компілятора, визначено їх структуру та принципи роботи. Також розглянуто та описано загальні алгоритми оптимізації компіляторів, оскільки оптимізація компілятора впливає на одні з основних критеріїв оцінки компілятора, як ефективність, швидкість роботи та виявлення помилок при збірці. В розділі описано методи та алгоритми основних етапів роботи розроблювального спеціалізованого двохпрохідного компілятора. Описано та визначено модифікований метод оптимізації «loop fusion» як основу другого проходу компілятора та його оптимізації. Модифікований алгоритм оптимізації є необхідним для організації оптимального та ефективного програмного продукту, тому в подальшому від буде застосований під час програмної реалізації продукту для поставленої задачі.

У третьому розділі розглянуто загальну архітектуру компіляторів. Визначено архітектуру розроблюваного компілятора, де схематично зображено всі етапи

роботи компілятора та взаємодії між ними. Також в даному розділі описано та проектовано лексичний аналізатор, хеш-таблицю з ланцюжковим методом вирішення колізій, синтаксичний аналізатор, семантичний аналізатор та генерацію коду. Далі описано та проектовано модифікований метод оптимізації «loop fusion». Описано вимоги до його застосування, а саме суміжність, кількість ітерацій, еквівалентність та відсутність недійсних залежностей. Також схематично описано принцип роботи даного алгоритму, та зображено його на блок-схемі.

У четвертому розділі програмно реалізовано розроблюваний програмний продукт. Було розподілено код на файли згідно їх функціоналу та поетапно описано ключові елементи розробки компілятора. Також описано функціонал створених функцій та технологій їх використання. В даному розділі програмно реалізовано модифікований метод “loop fusion”, де описано як та за допомогою яких бібліотек було його реалізовано. Описано як впливає реалізований метод на оптимізацію компільованого коду. Далі проведено обробку та аналіз тестових даних. Перед застосуванням цих тестових наборів даних, було проведено попередню обробку. Далі проведено компіляцію даних тестових програм та було визначено час їх компіляції. Також проведено оцінку ефективності розробленого компілятора. Проаналізовано результати тестування та визначено ефективність та дієвість модифікованого методу оптимізації «loop fusion».

Наукова новизна отриманих результатів:

– удосконалено метод оптимізації «loop fusion» спеціалізованого двохпрохідного компілятора на основі використання техніки розкрутки циклу;

– набула подальшого розвитку інформаційна технологія створення двохпрохідних компіляторів.

Під час виконання досліджень було опубліковано статтю на тему «Щодо актуальності задачі визначення оптимальних характеристик компілятора» в збірнику наукових праць за матеріалами XIII Всеукраїнської науково-практичної конференції «Актуальні проблеми комп’ютерних наук АПКН-2021». Хмельницький -2021. С. 139-142.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1. Кривоносов Ю.А., Боровик О.В.. Щодо актуальності задачі визначення оптимальних характеристик компілятора. Актуальні проблеми комп'ютерних наук. *Збірник наукових праць за матеріалами XIII всеукраїнської науково-практичної конференції «Актуальні проблеми комп'ютерних наук АПКН-2021»*. Хмельницький, 2021. – С. 139-142.

2. A. Brauckmann, A. Goens, J. Castrillon. ComPy-Learn: A toolbox for exploring machine learning representations for compilers. *Forum for Specification and Design Languages (FDL)*. 2020. pp. 1-4, doi: 10.1109/FDL50818.2020.9232946.

3. M. A. Aguilar*et.* Work-in-progress: multi-grained performance estimation for MPSoC compilers. *International Conference on Compilers, Architectures and Synthesis For Embedded Systems (CASES)*. 2017. pp. 1-2. doi: 10.1145/3125501.3125521.

4. Q. Huang. AutoPhase: Compiler Phase-Ordering for HLS with Deep Reinforcement Learning. *IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2019. pp. 308-308, doi: 10.1109/FCCM.2019.00049.

5. R. Tian, L. Guo, J. Li, B. Ren, G. Kestor. A High Performance Sparse Tensor Algebra Compiler in MLIR. *IEEE/ACM 7th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*. 2021. pp. 27-38, doi: 10.1109/LLVMHPC54804.2021.00009.

6. J. Chen, N. Xu, P. Chen, H. Zhang. Efficient Compiler Autotuning via Bayesian Optimization. *IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 2021. pp. 1198-1209. doi: 10.1109/ICSE43902.2021.00110.

7. L. Simon, D. Chisnall, R. Anderson. What You Get is What You C: Controlling Side Effects in Mainstream C Compilers. *IEEE European Symposium on Security and Privacy (EuroS&P)*. 2018. pp. 1-15. doi: 10.1109/EuroSP.2018.00009.

8. C. H. Jong, N. Medina, N. Fakhriyah, C. Hidayat, S. Hamali. Using Goal Programming Method for Optimization of Production Planning. *International*

Conference on Information Management and Technology. 2018. pp. 155-159. doi: 10.1109/ICIMTech.2018.8528156.

9. S. Kalayci, S. Arslan. A dynamic programming based optimization approach for appointment scheduling in banking. *International Conference on Computer Science and Engineering (UBMK)*. 2017. pp. 625-629. doi: 10.1109/UBMK.2017.8093482.

10. V. S. Borra, K. Debnath. Comparison Between the Dynamic Programming and Particle Swarm Optimization for Solving Unit Commitment Problems. *IEEE Jordan International Joint Conference on Electrical Engineering and Information Technology (JEEIT)*. 2019. pp. 395-400. doi: 10.1109/JEEIT.2019.8717481.

11. Chuanbin Zheng, Jingchun Feng, Qianqian Lu, Ke Zhang, Ranran Chen, Song Xue. Programme time optimization model for large-scale construction programme in China. *3rd International Conference on Control, Automation and Robotics (ICCAR)*. 2017. pp. 330-333. doi: 10.1109/ICCAR.2017.7942713.

12. Program optimization. URL: https://nwikihr.cyou/wiki/program_optimization (дата звернення 05.02.2022)

13. S. Fan, X. Yao, S. Cao, B. Zhao. An Optimization Method Based on Adaptive Dynamic Programming for Cleaning Photovoltaic Panels. *39th Chinese Control Conference (CCC)*. 2020. pp. 1565-1568. doi: 10.23919/CCC50068.2020.9189437.

14. F. Ma, Y. Xu, P. Xu. A nonlinear programming based universal optimization model of TDOA passive location. *12th International Conference on Intelligent Systems and Knowledge Engineering (ISKE)*. 2017. pp. 1-3, doi: 10.1109/ISKE.2017.8258784.

15. Z. Yimin, S. Guojun, Y. Xiaoguang. Cloud service selection optimization method based on parallel discrete particle swarm optimization. *Chinese Control And Decision Conference (CCDC)*. 2018. pp. 2103-2107. doi: 10.1109/CCDC.2018.8407473.

16. Z. Li, L. Jia, C. Liu. An Effective Solution to Nonlinear Bilevel Programming Problems Using Improved Particle Swarm Optimization Algorithm. *13th International Conference on Computational Intelligence and Security (CIS)*. 2017. pp. 16-19, doi: 10.1109/CIS.2017.00012.

17. Методи оптимізації програми. URL: http://elartu.tntu.edu.ua/bitstream/lib/31186/2/MNTK_2019v2_Kukuruza_A_O-Program_optimization_methods_53-54.pdf (дата звернення 12.02.2022)

18. Загальна схема роботи компіляторів. URL: https://studfile.net/preview/5465784/ (дата звернення 20.02.2022)

19. Chao Luo, Kan Long, Bo Wang. Research and design of module and symbol table in ASN.1 compiler. *International Conference on Computer Application and System Modeling (ICCASM)* 2010. pp. 111-114, doi: 10.1109/ICCASM.2010.5619243.

20. J. L. Sierra. Some misconceptions frequently detected in students of compiler construction courses. *International Symposium on Computers in Education (SIIE)*. 2019. pp. 1-6. doi: 10.1109/SIIE48397.2019.8970122.

21. S. Lee, H. Lee, S. Ryu. Broadening Horizons of Multilingual Static Analysis: Semantic Summary Extraction from C Code for JNI Program Analysis. *35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2020. pp. 127-137.

22. V. Zhukovskyy, D. Dmitriev, N. Zhukovska, A. Safonyk, A. Sydor. VHDL Compiler with Natural Parallel Comands Execution. *IEEE EUROCON 2021 - 19th International Conference on Smart Technologies*. 2021. pp. 331-337. doi: 10.1109/EUROCON52738.2021.9535606.

23. S. Yang, Z. Shi, G. Zhang, M. Li, Y. Ma, L. Sun. Understand Code Style: Efficient CNN-Based Compiler Optimization Recognition System. *IEEE International* Conference on Communications (ICC). 2019. pp. 1-6. doi: 10.1109/ICC.2019.8761073.

24. O. Rodriguez-Prieto, A. Mycroft, F. Ortin. An Efficient and Scalable Platform for Java Source Code Analysis Using Overlaid Graph Representations. *IEEE Access*. 2020. vol. 8. pp. 72239-72260. doi: 10.1109/ACCESS.2020.2987631

25. J. L. Sierra. Some misconceptions frequently detected in students of compiler construction courses. *International Symposium on Computers in Education (SIIE)*. 2019. pp. 1-6. doi: 10.1109/SIIE48397.2019.8970122.

26. S. H. H. Ding, B. C. M. Fung, P. Charland. Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and

Compiler Optimization. *IEEE Symposium on Security and Privacy (SP)*. 2019. pp. 472-489. doi: 10.1109/SP.2019.00003.

27. Symbol table in compiler. URL: https://www.geeksforgeeks.org/symbol-table-compiler/?ref=lbp (дата звернення 01.03.2022)

28. Semantic analysis in compiler design. URL: https://iq.opengenus.org/semantic-analysis-in-compiler-design/ (дата звернення 07.03.2022)

29. A. Kumar Das. A Linting tool without a compiler in it. *IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)*. 2021. pp. 1-4. doi: 10.1109/CONECCT52877.2021.9622676.

30. A. Brauckmann, A. Goens, J. Castrillon. ComPy-Learn: A toolbox for exploring machine learning representations for compilers. *Forum for Specification and Design Languages (FDL)*. 2020. pp. 1-4. doi: 10.1109/FDL50818.2020.9232946.

31. S. -M. Liu, L. Tang, N. -C. Huang, D. -Y. Tsai, M. -X. Yang, K. -C. Wu. Fault-Tolerance Mechanism Analysis on NVDLA-Based Design Using Open Neural Network Compiler and Quantization Calibrator. *International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*. 2020. pp. 1-3. doi: 10.1109/VLSI-DAT49148.2020.9196335.

32. M. Tatsuoka, M. Kaneko. Wire congestion aware high level synthesis flow with source code compiler. *International Conference on IC Design & Technology (ICICDT*). 2018. pp. 101-104. doi: 10.1109/ICICDT.2018.8399766.

33. L. Martirosyan. Easy to use evaluation of quality characteristics for a hierarchy of RTL compilers. *IEEE East-West Design & Test Symposium (EWDTS)*. 2017. pp. 1-4. doi: 10.1109/EWDTS.2017.8110072.

34. S. Gayathri, T. C. Taranath. RTL synthesis of case study using design compiler. *International Conference on Electrical, Electronics, Communication, Computer, and Optimization Techniques (ICEECCOT)*. 2017. pp. 1-7. doi: 10.1109/ICEECCOT.2017.8284603.

35. Compiler design – lexical analysis. URL: https://www.tutorialspoint.com/compiler_design/compiler_design_lexical_analysis.htm (дата звернення 10.03.2022)

36. Compiler design – code generation URL: https://www.tutorialspoint.com/compiler_design/compiler_design_code_generation.htm (дата звернення 15.03.2022)

37. B. Abci, J. A. Hage, M. E. Badaoui El Najjar, V. Cocquempot. Multi-Robot Autonomous Navigation System Using Informational Fault Tolerant Multi-Sensor Fusion with Robust Closed Loop Sliding Mode Control. *21st International Conference on Information Fusion (FUSION)*. 2018. pp. 1-5. doi: 10.23919/ICIF.2018.8455787.

38. H. Jo, H. Cha. Sequence Control Verification of a Central Solenoid Converter for Nuclear Fusion Reactors by Using a Hardware-in-the-Loop. *IEEE Transactions on Industrial Electronics*. 2017. vol. 64. No. 9. pp. 6864-6873. doi: 10.1109/TIE.2017.2686305.

39. João M.P.Cardoso, José Gabriel F.Coutinho, Pedro C.Diniz. Embedded Computing for High Performance. *Efficient Mapping of Computations Using Customization, Code Transformations and Compilation*. 2017. No. 5. P. 137-183.

40. V. Ilic, M. Marijan, A. Mehmed, M. Antlanger. Development of Sensor Fusion Based ADAS Modules in Virtual Environments. *Zooming Innovation in Consumer Technologies Conference (ZINC)*. 2018. pp. 88-91. doi: 10.1109/ZINC.2018.8448849.

41. Q. Li, X. Chen, J. Chen. A Method to Improve the GCC Series of Phenology Cameras Based on Histogram Features Using Multiple Linear Regression. *IEEE International Geoscience and Remote Sensing Symposium*. 2019. pp. 6606-6609. doi: 10.1109/IGARSS.2019.8898529.

42. J. Vankeirsbilck, J. Van Waes, H. Hallez, J. Boydens. Automated Regression Testing of a GCC Toolchain used on Embedded CPU Programs. *IEEE XXVIII International Scientific Conference Electronics (ET)*. 2019. pp. 1-4. doi: 10.1109/ET.2019.8878623.

43. J. Rohde, C. Hochberger. AutoBoxing: Improving GCC Passes to Optimize HW/SW Multi-Versioning of Kernels for HLS. *International Conference on Field-Programmable Technology (ICFPT)*. 2019. pp. 319-322. doi: 10.1109/ICFPT47387.2019.00057.

44. M. Al-Muhanadi, H. Al-Fadhel, A. Al-Jalahma. How can the Accounting Profession Contribute to the Reduction of CO2 Emissions in the GCC Region?. *Second International Sustainability and Resilience Conference: Technology and Innovation in* Building Designs. 2020. pp. 1-4. doi: 10.1109/IEEECONF51154.2020.9319946.

45. C. Rao, C. Wang, Z. Hu, X. Xiao, M. Goh. Gray Uncertain Linguistic Multiattribute Group Decision Making Method Based on GCC-HCD. *IEEE Transactions* on Computational Social Systems. 2022. pp. 10-15. doi: 10.1109/TCSS.2022.3166526.

46. S. Kim, B. On, S. Im, S. Kim. Performance comparison of FFT-based and GCC-PHAT time delay estimation schemes for target azimuth angle estimation in a passive SONAR array. *IEEE Underwater Technology (UT)*. 2017. pp. 1-4. doi: 10.1109/UT.2017.7890280.

47. H. AlQaidoom, A. Shah. Digital Literacy and the Attitude of Educators Towards MOOC Platform in GCC Countries. *IEEE International Conference on Innovative Research and Development (ICIRD)*. 2019. pp. 1-6. doi: 10.1109/ICIRD47319.2019.9074637.

48. A. M. Desoky, E. A. H. Elamir, G. A. Mousa. A GCC Evidence on the Effect of Board and Audit Committee Features on CSRD: the Case of Employee and Product Information. *International Conference on Decision Aid Sciences and Application (DASA)*. 2020. pp. 264-269. doi: 10.1109/DASA51403.2020.9317081.

49. A. Mehmood, M. U. Akram, A. Tariq. Vertebra localization and centroid detection from cervical radiographs. *International Conference on Communication, Computing and Digital Systems (C-CODE)*. 2017. pp. 287-292. doi: 10.1109/C-CODE.2017.7918944.

50. A. Ballman and D. Svoboda. Avoiding Insecure C++ —How to Avoid Common C++ Security Vulnerabilities. *IEEE Cybersecurity Development (SecDev)*. 2016. pp. 65-65. doi: 10.1109/SecDev.2016.022.

ДОДАТОК А

(обов’язковий)

ЛІСТИНГ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ СПЕЦІАЛЬНОГО ДВОХПРОХІДНОГО КОМПІЛЯТОРА

Модуль «Лексичний аналізатор».

```
#include <stdlib.h>
#include <ctype.h>

#include "lex.h"
#include "input.h"
#include "defs.h"
#include "func.h"
#include "sym.h"
#include "ast.h"

bool look_done = false;
struct Token look_tok;

struct Token lex()
{
    int ch = 0;
    struct Token tok;

    // If lookaheaded
    if (look_done) {
        look_done = false;

        return look_tok;
    }

    eat:
    switch (ch = fgetc(get_file())) {
        case ' ': case '\n': goto eat;
        case EOF: tok.type = EOP; break;
        case '+': tok.type = OP1; tok.attr = ADD_TYPE; break;
        case '-': tok.type = OP1; tok.attr = SUB_TYPE; break;
        case '*': tok.type = OP2; tok.attr = MUL_TYPE; break;
        case '/': tok.type = OP2; tok.attr = DIV_TYPE; break;
        case '(': tok.type = LBR; break;
        case ')': tok.type = RBR; break;
        case '=': tok.type = EQ; break;
        case ';': tok.type = SEM; break;
        default:
            // ID
            if (isalpha(ch)) {
                char *id_name = safe_malloc(MAX_LEN);
                int len = 0;
                id_name[len++] = ch;
                do {
                    if (MAX_LEN == len)
                        fatal_error("Lexer: Variable name is too long");
                    id_name[len++] = (ch = fgetc(get_file()));
                } while (isalpha(ch) || isdigit(ch));
                id_name[len - 1] = '\0';
```

```
            tok.type = ID;
            tok.attr = add_sym(id_name);

        // NUM
        } else if (isdigit(ch)) {
            int val = ch - '0';
            while (isdigit(ch = fgetc(get_file())))
                val = val * 10 + (ch - '0');

            tok.type = NUM;
            tok.attr = val;

        // Error
        } else {
            fatal_error("Lexer: Unexpected symbol");
        }

        ungetc(ch, get_file());
    }
  return tok;
}

struct Token lookahead()
{
  look_tok = lex();
  look_done = true;

  return look_tok;
}
```

Модуль «Вхідних даних».

```
#include "input.h"
#include "func.h"

FILE *file;

void open_file(const char *filename)
{
	file = fopen(filename, "rb");
	if (NULL == file) {
		fatal_error("Input: File doesn't exist!");
	}
}

FILE *get_file()
{
	return file;
}

void close_file()
{
	fclose(file);
}
```

Модуль «Генерації проміжного коду».

```
#include <stdio.h>
```

```
#include "asm.h"
#include "sym.h"
#include "codes.h"
#include "gen.h"

void print_asm(byte *code)
{
   byte cur_byte;
   int i = 0;

   while ((cur_byte = code[i++]) != RET) {
      switch (cur_byte) {
         case PUSH: printf("PUSH %i\n", code[i++]); break;
         case READ: cur_byte = code[i++]; printf("READ %s\n",get_sym(cur_byte)->name); break;
         case WRITE: cur_byte = code[i++]; printf("WRITE %s\n", get_sym(cur_byte)->name); break;
         case ADD: printf("ADD POP, POP\n"); break;
         case SUB: printf("SUB POP, POP\n"); break;
         case MUL: printf("MUL POP, POP\n"); break;
         case DIV: printf("DIV POP, POP\n"); break;
      }
   }
   printf("RET\n");
}
```

Модуль «Створення абстрактного дерева».

```
#include <stdlib.h>

#include "ast.h"
#include "func.h"

struct Node *make_node(int type, struct Node *op1, struct Node *op2, int val)
{
   struct Node *node = safe_malloc(sizeof(struct Node));
   node->type = type;
   node->val = val;
   node->op1 = op1;
   node->op2 = op2;

   return node;
}
```

Модуль «Проміжних функцій».

```
#include <stdlib.h>
#include <stdio.h>

#include "func.h"

void *safe_malloc(size_t size)
{
   void *const mem = malloc(size);
   if (size != 0 && !mem)
      fatal_error("Out of memory");
   return mem;
}

void fatal_error(char *msg)
{
```

```
    fprintf(stderr, "%s\n", msg);
    exit(EXIT_FAILURE);
}
```

Модуль «Генерації коду».

```
#include "gen.h"
#include "sym.h"
#include "ast.h"
#include "parser.h"
#include "codes.h"

byte obj[MAX_PROGRAM_SIZE];
byte *cur = obj;

void gen(int code)
{
    *cur++ = code;
}

byte *compile(struct Node *node)
{
    if (NULL == node)
        return NULL;

    switch (node->type) {
        case SEQ_TYPE: COMPILE_BOTH; break;
        case SET_TYPE: compile(node->op2); gen(WRITE); gen(node->op1->val); break;
        case VAR_TYPE: gen(READ); gen(node->val); break;
        case NUM_TYPE: gen(PUSH); gen(node->val); break;
        case ADD_TYPE: COMPILE_BOTH; gen(ADD); break;
        case SUB_TYPE: COMPILE_BOTH; gen(SUB); break;
        case MUL_TYPE: COMPILE_BOTH; gen(MUL); break;
        case DIV_TYPE: COMPILE_BOTH; gen(DIV); break;
        case RET_TYPE: COMPILE_BOTH; gen(RET); break;
    }

    return obj;
}
```

Модуль «Виконавчий файл».

```
#include <stdlib.h>

#include "defs.h"
#include "input.h"
#include "lex.h"

#include "gen.h"
#include "vm.h"
#include "asm.h"
#include "func.h"

#define MIN_ARGS 2
#define SOURCE_ARG 1

int main(int argc, char **argv)
{
    argv = "D:\testing_program1.cpp";
    if (argc < MIN_ARGS) {
```

```
        fatal_error("No file provided! Usage: compiler <source>");
    }

    open_file(argv[SOURCE_ARG]);

    byte *code = compile(parse(lex()));

    close_file();

    printf("Generated ASM:\n");
    print_asm(code);

    printf("Execution result:\n");
    run(code);

    return EXIT_SUCCESS;
}
```

Модуль «Парсер».

```
#include <stdio.h>
#include <stdlib.h>

#include "parser.h"
#include "defs.h"
#include "func.h"
#include "sym.h"
#include "lex.h"
#include "ast.h"

struct Token tok;

static bool accept(int type);
static bool accept_two(int type1, int type2);
static void expect(int type);
static struct Node *expr();
static struct Node *factor();
static struct Node *term();

static bool accept(int type)
{
    if (tok.type == type) {
        tok = lex();
        return true;
    } else {
        return false;
    }
}

static bool accept_two(int type1, int type2)
{
    if (tok.type == type1 && lookahead().type == type2) {
        accept(type1);
        accept(type2);
        return true;
    } else {
        return false;
    }
}
```

```
static void expect(int type)
{
    if (!accept(type)) {
        fatal_error("Parser: Syntax error");
    }
}

static struct Node *factor()
{
    struct Node* node = safe_malloc(sizeof(struct Node));
    node->op1 = NULL;
    node->op2 = NULL;

    int tok_attr = tok.attr;
    if (accept(ID)) {
        node->type = VAR_TYPE;
        node->val = tok_attr;
    } else if (accept(NUM)) {
        node->type = NUM_TYPE;
        node->val = tok_attr;
    } else if (accept(LBR)) {
        free(node);
        node = expr();
        accept(RBR);
    } else {
        fatal_error("Parser: Unexpected factor");
    }

    return node;
}

static struct Node *term()
{
    struct Node* node;
    node = factor();

    int tok_attr = tok.attr;
    while (accept(OP2)) {
        node = make_node(tok_attr, node, factor(), 0);

        tok_attr = tok.attr;
    }

    return node;
}

static struct Node *expr()
{
    struct Node* node = NULL;

    int tok_attr = tok.attr;
    if (accept_two(ID, EQ)) {
        node = safe_malloc(sizeof(struct Node));
        node->type = SET_TYPE;
        node->op1 = make_node(VAR_TYPE, 0, 0, tok_attr);
        node->op2 = expr();
    } else {
        node = term();

        tok_attr = tok.attr;
        while (accept(OP1)) {
            node = make_node(tok_attr, node, term(), 0);
```

```
            tok_attr = tok.attr;
        }
    }

    return node;
}

struct Node *produce()
{
    struct Node* node = safe_malloc(sizeof(struct Node));
    node->op1 = expr();
    node->op2 = NULL;

    expect(SEM);

    if (tok.type != EOP) {
        node->type = SEQ_TYPE;
        node->op2 = produce();
    } else {
        node->type = RET_TYPE;
    }

    return node;
}

struct Node *parse(struct Token start_tok)
{
    tok = start_tok;

    return produce();
}
```

Модуль «Хеш-таблиці».

```
#include <stdlib.h>
#include <string.h>

#include "sym.h"
#include "func.h"

struct Table* symbol_table[MAX_SYMBOL_TABLE_SIZE];
int table_size = 0;

struct Table *get_sym(int id)
{
    return symbol_table[id];
}

int add_sym(char *name)
{
    int i;
    for (i = 0; i < table_size; i++) {
        if (strcmp(symbol_table[i]->name, name) == 0)
            return symbol_table[i]->id;
    }

    struct Table *item = safe_malloc(sizeof(struct Table));
    item->id = table_size;
    item->name = name;
```

```
    symbol_table[table_size] = item;

    return table_size++;
}

void set_sym(int id, int val)
{
    symbol_table[id]->val = val;
}

int get_table_size()
{
    return table_size;
}
```

ДОДАТОК Б

(обов'язковий)

ПУБЛІКАЦІЯ

УДК 004.03

Кривоносов Ю. А., Боровик О. В.

Хмельницький національний університет

ЩОДО АКТУАЛЬНОСТІ ЗАДАЧІ ВИЗНАЧЕННЯ ОПТИМАЛЬНИХ ХАРАКТЕРИСТИК КОМПІЛЯТОРА

Проведено аналіз етапів роботи компілятора, фаз етапів, безпосередньо процесу компіляції, фізичного складу компілятора. На основі цього зроблено висновок про те, що питання вибору концепції розробки компілятора конкретної мови програмування, основних методів, моделей та інструментів реалізації потребують окремого дослідження.

The analysis of stages of work of the compiler, phases of stages, directly process of compilation, physical structure of the compiler is carried out. Based on this, it is concluded that the choice of the concept of compiler development of a particular programming language, basic methods, models and implementation tools require a separate study.

Велике різноманіття високорівневих програмних мов та фреймворків вимагає створення відповідних програмних засобів компіляції та інтерпретації коду. Архітектура цих засобів безпосередньо впливає на швидкість та безпомилковість їх роботи. Саме цим пояснюється значна увага до вказаного питання науковців і практиків [1-9].

Незважаючи на значну увагу, яка приділена питанням побудови компіляторів, а також на значну кількість компіляторів, які застосовуються на практиці, на сьогодні ще не до кінця вивчені питання встановлення їх оптимальних характеристик.

Тому метою даної роботи є огляд питань, пов'язаних з актуальністю задачі визначення оптимальних характеристик компілятора.

Задача компілятора звичайно розглядається на двох етапах [9].

1. Етап аналізу, на якому аналізується вихідний текст.
2. Етап синтезу, на якому генерується машинно-орієнтоване подання.

Процес компіляції являє собою перетворення однієї мови на іншу, перехід від вихідного коду до цільового коду, що виконується на машині, можливо, після деяких перетворень. Процес компіляції також включає третю мову – мову реалізації, під якою розуміють мову написання компілятора. Нею може бути та ж мова, що і вихідний чи цільовий код, але це необов'язково. Тобто входом компілятора служить ланцюжок символів на мові програмування L1. Вихід компілятора (об'єктна програма) також є ланцюжком символів, але належить іншій

мові L2, наприклад, мови деякого комп'ютера. При цьому сам компілятор написаний на мові L3, можливо, відрізняється від перших двох. Таким чином, можна говорити про компілятор як про відображення множини L1 в множину L2.

Етап аналізу розділяють на три окремі фази: лексичний аналіз; синтаксичний аналіз; семантичний аналіз.

Лексичний аналіз – це відносно проста фаза, у якій формуються символи (чи лексеми) мови. Задачею фази лексичного аналізу чи лексичного аналізатора є перехід від послідовності знаків до символів мови, з якими надалі будуть працювати синтаксична і семантична фази.

У процесі синтаксичного аналізу визначається загальна структура програми, що включає розуміння порядку проходження символів у програмі. Це означає, що синтаксичний аналізатор повинен мати інформацію про контекст, у якому він працює. Результатом роботи синтаксичного аналізатора є подання програми в деревоподібній формі, що називають синтаксичним деревом. Фаза синтаксичного аналізу є ключовою на етапі аналізу. Вона безпосередньо взаємодіє з лексичною фазою, а результати її роботи надалі будуть використовуватися семантичною фазою.

Синтаксичний аналізатор зчитує символи в програмі зліва направо. У процесі зчитування він повинен уміти визначати, чи є послідовність вже прочитаних символів початком програми. Деякі властивості мов програмування не можуть бути перевірені простим скануванням зліва направо без створення таблиць довільного розміру. Перевірка таких властивостей мов програмування (що називають статичною семантикою) виконується в семантичній фазі аналізу.

Етап синтезу процесу компіляції складається з наступних основних фаз: генерація машинно-незалежного коду; оптимізація машинно-незалежного коду; розподіл пам'яті. Генерація машинного коду; оптимізація машинного коду.

В окремих випадках деякі з цих фаз можуть бути відсутніми. Наприклад, якщо компілятор безпосередньо компілює в машинний код, перші дві фази можуть пропускатися. Оптимізація коду може відбуватися на рівні машинонезалежного коду, на рівні машинного коду, на обох рівнях чи на жодному.

Потреба в оптимізації генерованого коду може бути різною. Якщо потрібен ефективний код, компілятор зобов'язаний забезпечити значну оптимізацію. У той же час в багатьох середовищах швидкість роботи програмного забезпечення не є критичним параметром, отже, необхідна лише незначна оптимізація. Деякі типи оптимізації реалізувати просто, і тому їх часто включають у компілятори, тоді як інші форми оптимізації, особливо глобальні (на відміну від локальних), трудомісткі і вимагають значних витрат часу при компіляції, а тому застосовуються рідко. У фазі розподілу пам'яті кожна стала та змінна в програмі отримують зарезервоване місце в пам'яті для збереження свого значення.

Якщо логічно компілятор складається з етапів і фаз, фізично він складений із проходів. Компілятор здійснює прохід щоразу при зчитуванні вихідного коду чи його подання. Ранні компілятори були багатопрохідними через недостатній обсяг пам'яті машин того часу. Сучасні компілятори є переважно однопрохідними, тобто повний процес компіляції цілком виконується при однократному зчитуванні коду. У цьому випадку різні описані фази будуть виконуватися паралельно (що, як правило, є найбільш зручним), що усуває необхідність складного зв'язку між різними проходами

Загальна структура компілятора (рисунок 1) багато в чому залежить від його фазової структури і структури синтаксичного аналізатора, а структура синтаксичного аналізатора відбиває властивості вихідної мови. Звичайно при проектуванні компілятора керуються такими вимогами: ефективна компіляція; мінімальний розмір компілятора; мінімальна довжина цільового коду; створення ефективного цільового коду; портабельність; простота використання; практичність.

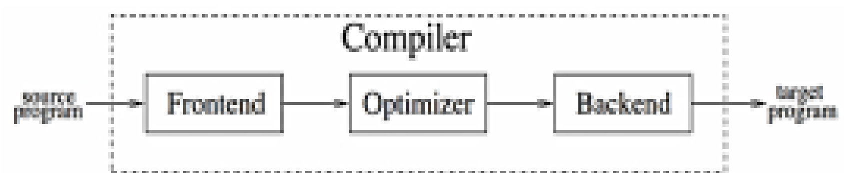


Рисунок 1 – Основні складові компілятора

Одночасно задовольнити всім цим вимогам неможливо, тому деяким з них доводиться віддавати перевагу. У навчальних середовищах, наприклад, ефективність компіляції й гарні засоби діагностики можуть бути більш важливими, ніж створення ефективного цільового коду, тоді як для вбудованих систем першочергове значення має розмір і ефективність цільового коду. Багато компіляторів дозволяють користувачу самому визначати режим роботи компілятора – ступінь оптимізації, виконання перевірок часу виконання і т.д.

Існує величезна кількість різних мов програмування, починаючи з таких традиційних мов програмування як Fortran і Pascal і закінчуючи сучасними об'єктно-орієнтованими мовами такими, як C# і Java. Практично кожна мова програмування має якісь особливості з точки зору творця транслятора.

Проведений аналіз етапів роботи компілятора, фаз етапів, безпосередньо процесу компіляції, фізичного складу компілятора дозволяє зробити висновок про те, що питання дослідження концепції розробки компілятора мови програмування, основні методи, моделі, інструменти реалізації тощо потребують окремого вивчення. Адже особливості конкретних мов програмування вказують на те, що одна і та ж структура компілятора для них не є раціональною.

Таким чином, першим етапом подальших досліджень має бути пошук відповідей на питання щодо визначення раціональної структури компілятора в залежності від мови програмування L1 і мови L3, що застосовується для написання самого компілятора, а наступним, в залежності від отриманих результатів вирішення першого етапу, визначення методу і засобів створення спеціалізованого компілятора.

Перелік посилань

1. Ахо А., Сети Р., Ульман Д. Компиляторы: принципы, технологии и инструменты.: Пер. с англ. – М.: Издательский дом «Вильямс», 2001. - 768 с.
2. Ахо А., Сети Р., Лам М., Ульман Д. Компиляторы: принципы, технологии и инструментарий.: Пер. с англ. - М.: ООО «И. Д. Вильямс», 2008. - 1184 с.
3. Вирт Н. Построение компиляторов / Пер. с англ. Борисов Е. В., Чернышов Л. Н. – М.: ДМК Пресс, 2010. – 192 с.
4. Гордеев А. В. Молчанов А. Ю. Системное программное обеспечение. СПб.: Питер, 2001. – 736 с.
5. Aycock, John (2003-06-01). A Brief History of Just-in-time. ACM Comput. Surv. **35** (2). c. 97–113. ISSN 0360-0300. doi:10.1145/857076.857077.
6. List of languages that compile to JS jashkenas/coffeescript. GitHub.
7. Semantic patching with Coccinelle [LWN.net]. lwn.net.
8. Java theory and practice: Dynamic compilation and performance measurement. www.ibm.com (en). 2004-12-21.
9. Церінгер Б.К. Дослідження методів розробки компіляторів для мов програмування. Атестаційна робота. - Харків: ХНУРЕ, 2019. – 64 с.

ДОДАТОК В
ПРЕЗЕНТАЦІЯ

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
Кафедра комп'ютерної інженерії та інформаційних систем

Метод та засоби створення спеціалізованого двохпрохідного компілятора

Виконав: студент 2 курсу КІ2м-20-1 Кривоносов Ю.А.
Науковий керівник: д. т. н., проф. Боровик О.В.

Хмельницький - 2022

Мета і задачі дослідження

- Об'єктом дослідження є спеціалізований двохпрохідний компілятор.
- Предметом дослідження є методи та засоби створення спеціалізованого двохпрохідного компілятора.
- Метою роботи є удосконалення методів та засобів розробки компілятора мови програмування, основних методів, моделей та інструментів реалізації. В роботі проведений аналіз предметної області, досліджені основні методи, моделі, інструменти реалізації, концепція розробки компілятора мови програмування.

Мета і задачі дослідження

Заради досягнення мети потрібно успішно виконати наступні завдання:

- провести аналіз існуючих засобів створення та методів розробки компіляторів;
- провести аналіз існуючих методів оптимізації компіляторів;
- розробити проектування програмного продукту на основі дослідження;
- створити програмну реалізацію системи;
- провести тести програмної реалізації системи;
- провести аналіз результатів.

Наукова новизна отриманих результатів

На основі проведених досліджень розроблена архітектура і компоненти спеціалізованого двохпрохідного компілятора.

Практична значимість отриманих результатів полягає у розробленому компіляторі, який дозволяє оптимізувати процес компіляції.

Наукова новизна отриманих результатів:

- удосконалено метод оптимізації «loop fusion» спеціалізованого двохпрохідного компілятора на основі використання техніки розкрутки циклу;
- набула подальшого розвитку інформаційна технологія створення двохпрохідних компіляторів.

Метод «loop fusion»

Для покращення оптимізації нашого спеціалізованого компілятора удосконалимо метод «loop fusion», де він є методом об'єднання двох тіл циклів в одне. Цей метод безпечний, коли кожне визначення та кожне використання в результуючому циклі мають таке ж значення, що й у вихідних циклах.

Модифікація методу «loop fusion»

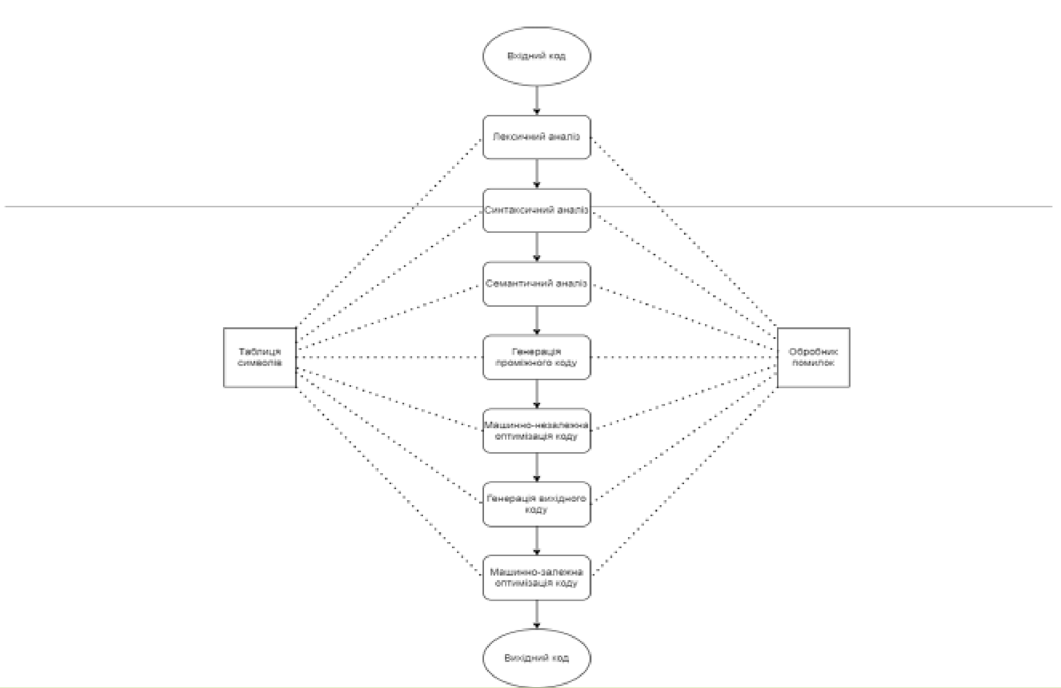
Модифікуємо алгоритм “loop fusion” за допомогою техніки розкрутки циклу яка полягає в тому, що за одну ітерацію обробляються не один, а кілька елементів. Ми як би «розкручуємо» цикл, збільшуючи довжину його тіла та зменшуючи кількість витків.

Таким чином ми виконуємо менше ітерацій, за одну ітерацію виконуємо більше корисної роботи, а накладні витрати становлять уже не таку велику долю.

Проектування компілятора

У реалізації даного компілятора міститься 7 етапів роботи компілятора, а саме:

- лексичний аналіз;
- синтаксичний аналіз,
- семантичний аналіз,
- генерація проміжного коду,
- машинно-незалежна оптимізація коду,
- генерація кінцевого коду
- машинно-залежна оптимізація коду (модифікований метод «loop fusion»).



Блок-схема розробленого компілятора

Вимоги до використання методу «loop fusion»

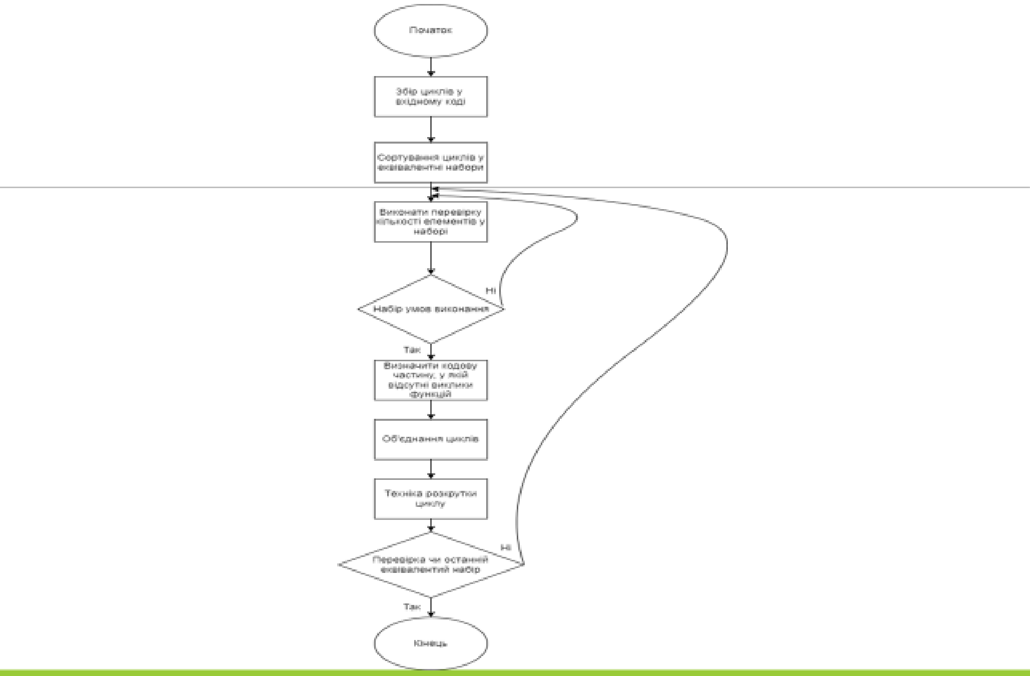
Перед початком проектування модифікованого методу оптимізації “loop fusion” розглянемо які є вимоги до його застосування.

Для того, щоб два цикли, були з'єднані, вони повинні задовольняти наступним умовам:

- обидва цикли повинні бути суміжними;
- обидва цикли повинні повторюватись однакову кількість разів;
- обидва цикли повинні бути еквівалентними;
- між циклами не може бути жодних недійсних залежностей.

Алгоритм роботи модифікованого методу «loop fusion»

1. для кожного рівня коду, від крайнього до внутрішнього алгоритм збирає цикли, які є кандидатами на з’єднання циклів;
2. алгоритм сортує список кандидатів у еквівалентні набори;
3. перевірка умови, що цикли мають однакову кількості ітерацій, якщо ні, алгоритм пропускає дану пару циклів;
4. перевірка умови, що цикли суміжні, якщо ні, алгоритм пропускає дану пару циклів;
5. перевірка умови, що цикли мають дійсні залежності, якщо ні, алгоритм пропускає дану пару циклів;
6. об’єднання пари циклів;
7. застосування модифікації “розкрутки циклу”;
8. оновлює список кандидатів на об’єднання, та якщо ще є кандидати, алгоритм повертається на крок 3.



Блок схема модифікованого алгоритму «loop fusion»

Програмна реалізація модифікованого методу «loop fusion»

Метод "loop fusion" – це оптимізація компілятора та перетворення циклу, яке замінює декілька циклів одним. Реалізація модифікації оптимізації за допомогою цього методу представлена у вигляді перевантажених математичних функцій, які будуть заміщувати стандартні математичні функції мови С, при виклику їх у вхідному коді компільованої програми.

Наприклад, оголошуємо перевантажений оператор додавання як функцію вільного друга, де синтаксис програми визначає operator+ як вільну функцію, яка є другом цього класу, незважаючи на те, що вона з'являється як оголошення функції-члена.

```
friend Array operator+(const Array& a, float b) {
    Array c(a.size());
    for (size_t i = 0; i < a.size(); ++i) {
      c[i] = a[i] + b; }
    return c; }
```

Тестування

В якості тестових наборів даних було взято 3 програми, в яких зосереджена робота з циклами, а саме з циклами, які підпорядковуються умовам злиття для можливості застосування модифікованого методу оптимізації “loop fusion”.

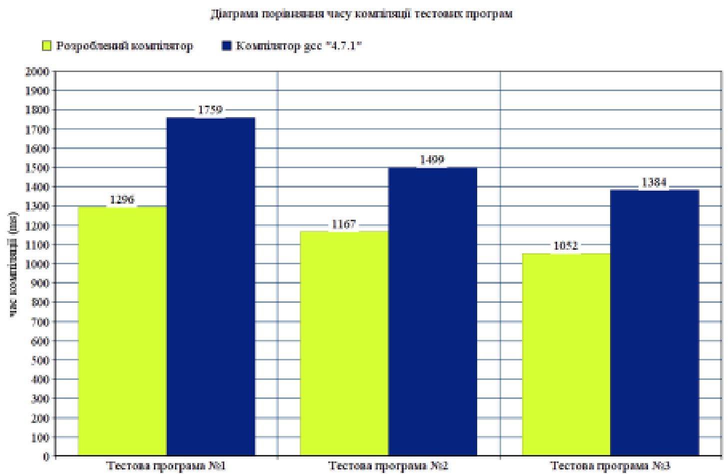
- Перша тестова програма є задачею яка вирішується за допомогою алгоритму пошуку мінімального часу.
- Друга тестова програма є програмною реалізацією алгоритму Дейкстри.
- Третя тестова програма є задачею в якій використовується алгоритм побудови абстрактних дерев.

Оцінка ефективності

Для оцінки ефективності розробленого спеціалізованого двопрохідного компілятора проведемо компіляцію тестових наборів даних за допомогою існуючого компілятора «x86-64 gcc 4.7.1»та розробленого в даній роботі і порівняємо результати час компіляції тестових програм на обох компіляторах.

Отримані дані, які були отримані під час тестування, виведемо їх у графічний формат вигляду та створимо діаграму для зручного візуального перегляду часу компіляції тестових програм на обох компіляторах.

Оцінка ефективності



У відсотковому відношенні компіляція тестових програм розробленим компілятором є ефективнішою на 20-25% за компіляцію тестових програм за допомогою gcc 4.7.1.

Публікації за матеріалами дипломної роботи

Кривоносов Ю.А., Боровик О.В.. Щодо актуальності задачі визначення оптимальних характеристик компілятора. Актуальні проблеми комп'ютерних наук. *Збірник наукових праць за матеріалами XIII всеукраїнської науково-практичної конференції «Актуальні проблеми комп'ютерних наук АПКН-2021*». Хмельницький, 2021. – С. 139-142.

Висновки

У даній кваліфікаційній роботі магістра за результатами теоретичних та практичних досліджень розроблено методи та засоби створення спеціалізованого двохпрохідного компілятора.

- У першому розділі досліджено предметну область, описано структуру компілятора та принципи його роботи. В даному розділі розглянуто етапи та фази цих етапів процесу компіляції. Розглянуто переваги та недоліки однопрохідних та багатопрохідних компіляторів. Проаналізовано уже існуючі методи оптимізації компіляції. В результаті аналізу проблеми низької швидкодії компіляторів була сформована задача на оптимізацію та покращення процесу компіляції. За допомогою проведених досліджень вивели вимоги і завдання для подальшого вдосконалення програмного продукту.

Висновки

- У другому розділі описано загальну технологію роботи компіляторів. В процесі аналізу розглянуто основні функції алгоритмів різних етапів роботи компілятора. Переглянуто загальні характеристики цих алгоритмів та переваги і недоліки деяких з них відносно алгоритмів того ж типу. В підрозділах описано складові елементи алгоритмів кожного етапу роботи компілятора, визначено їх структуру та принципи роботи. Також розглянуто та описано загальні алгоритми оптимізації компіляторів, оскільки оптимізація компілятора впливає на одні з основних критеріїв оцінки компілятора, як ефективність, швидкість роботи та виявлення помилок при збірці. В розділі описано методи та алгоритми основних етапів роботи розроблювального спеціалізованого двохпрохідного компілятора. Описано та визначено модифікований метод оптимізації «loop fusion» як основу другого проходу компілятора та його оптимізації. Модифікований алгоритм оптимізації є необхідним для організації оптимального та ефективного програмного продукту, тому в подальшому він буде застосований під час програмної реалізації продукту для поставленої задачі.

Висновки

❖ У третьому розділі розглянуто загальну архітектуру компіляторів. Визначено архітектуру розроблюваного компілятора, де схематично зображено всі етапи роботи компілятора та взаємодії між ними. Також в даному розділі описано та проектовано лексичний аналізатор, хеш-таблицю з ланцюжковим методом вирішення колізій, синтаксичний аналізатор, семантичний аналізатор та генерацію коду. Далі описано та проектовано модифікований метод оптимізації «loop fusion». Описано вимоги до його застосування, а саме суміжність, кількість ітерацій, еквівалентність та відсутність недійсних залежностей. Також схематично описано принцип роботи даного алгоритму, та зображено його на блок-схемі.

Висновки

❖ У четвертому розділі програмно реалізовано розроблюваний програмний продукт. Було розподілено код на файли згідно їх функціоналу та поетапно описано ключові елементи розробки компілятора. Також описано функціонал створених функцій та технологій їх використання. В даному розділі програмно реалізовано модифікований метод "loop fusion", де описано як та за допомогою яких бібліотек було його реалізовано. Описано як впливає реалізований метод на оптимізацію компільованого коду. Далі проведено обробку та аналіз тестових даних. Перед застосуванням цих тестових наборів даних, було проведено попередню обробку. Далі проведено компіляцію даних тестових програм та було визначено час їх компіляції. Також проведено оцінку ефективності розробленого компілятора. Проаналізовано результати тестування та визначено ефективність та дієвість модифікованого методу оптимізації «loop fusion».

Ім'я користувача: Кафедра КІ

ID перевірки: 1011225470

Дата перевірки: 17.05.2022 19:44:32 EEST

Тип перевірки: Doc vs Internet + Library

Дата звіту: 17.05.2022 19:45:15 EEST

ID користувача: 100005591

Назва документа: Кривоносов_Удосконалення технології мульти-проксування для забезпечення конфіденц...

Кількість сторінок: 78 **Кількість слів:** 13627 **Кількість символів:** 101205 **Розмір файлу:** 2.31 MB ID файлу: 1011117252

26.6% Схожість

Найбільша схожість: 6.13% з Інтернет-джерелом (https://uk.wikipedia.org/wiki/%D0%9E%D0%BF%D1%82%D0%B8%D0%.

25.3% Джерела з Інтернету | 133 .... Сторінка 80

1.43% Джерела з Бібліотеки | 65 .... Сторінка 81

0% Цитат

Не знайдено жодних цитат

Не знайдено жодних посилань

0% Вилучень

Немає вилучених джерел

Модифікації

Виявлено модифікації тексту. Детальна інформація доступна в онлайн-звіті.

Замінені символи | 29

Ім'я користувача:
Кафедра КІ

ID перевірки:
1011239576

Дата перевірки:
18.05.2022 20:06:32 EEST

Тип перевірки:
Doc vs Internet + Library

Дата звіту:
18.05.2022 20:15:35 EEST

ID користувача:
100005591

Назва документа: Кривоносов_2_Метод та засоби створення спеціалізованого двохпрохідного компілятора

Кількість сторінок: 78 **Кількість слів:** 13567 **Кількість символів:** 100803 **Розмір файлу:** 2.32 MB **ID файлу:** 1011130741

13.2% Схожість

Найбільша схожість: 2.06% з Інтернет-джерелом (https://wikizero.com/uk/%D0%9A%D1%96%D0%BD%D1%86%D0%B5%...

12.2% Джерела з Інтернету | 130 .... Сторінка 80

1.06% Джерела з Бібліотеки | 93 .... Сторінка 81

0% Цитат

Не знайдено жодних цитат

Не знайдено жодних посилань

71.5% Вилучень

Деякі джерела вилучено автоматично (фільтри вилучення: кількість знайдених слів є меншою за **8 слів** та **0%**)

Немає вилучених Інтернет-джерел

71.5% Вилученого тексту з Бібліотеки | 1 .... Сторінка 81

Модифікації

Виявлено модифікації тексту. Детальна інформація доступна в онлайн-звіті.

Замінені символи | 28

Anti-Plagiarism v-15.257

Максимальное совпадение с одним документом 1.0%

Словари проверки: en_US, ru_RU, ua_UA. **Ошибок в документах: 7%**

<table>
<tr><td rowspan="3">ID: 103585
Название: Удосконалення технології мульти-проксування для забезпечення конфіденційності передачі інформації в мережі Інтернет
Добавлено в БД: 2022-05-17
Авторы: Кривоносов Ю.А.
Руководители: Боровик О.В.
Консультанты:
Опоненты:</td><td colspan="2">Документ</td><td colspan="2">Суммарное совпадение по Базе Данных</td></tr>
<tr><td>Символы</td><td>Лексемы</td><td>Символы</td><td>Лексемы</td></tr>
<tr><td>91215</td><td>803</td><td>2298 (3%)</td><td>20 (2%)</td></tr>
</table>

Источник плагиата

<table>
<tr><td rowspan="2">ID</td><td rowspan="2">Описание</td><td colspan="2">Наличие плагиата в документе</td></tr>
<tr><td>Символы</td><td>Лексемы</td></tr>
</table>

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

РЕЦЕНЗІЯ НА ДИПЛОМНУ РОБОТУ

Дипломник: __Кривоносов Юрій Андрійович______________________

Тема: __Метод і засоби створення спеціалізованого двохпрохідного компілятора

Спеціальність**:** 123 «Комп'ютерна інженерія»

Обсяг дипломної роботи:

Кількість листів креслень _–__; кількість сторінок записки _108__

1. Короткий зміст роботи та прийнятих рішень _У роботі запропоновано удосконалений метод оптимізації компіляторів

____

2. Висновок про відповідність роботи дипломному завданню _____________ Дипломна робота відповідає виданому завданню __________________________________

3. Характеристика виконання кожного розділу, ступінь використання останніх досягнень науки і техніки і передових методів роботи: _У першому розділі проведено аналіз предметної області та поставлення задачі. У другому розділі було проведено аналіз існуюих методів складових компілятора та його оптимізації. У третьому розділі розроблено архітектуру розроблюваного компілятора та удосконаленого методу. У четвертому розділі розроблено програмно-технічний засіб, проведено його тестування та оцінку його ефективності.

4. Позитивні сторони роботи: ___Запропонована система дозоляє спростити та удосконалити існуючий метод оптимізації компіляції програм.

5. Негативні сторони роботи: В роботі було розглянуто удосконалення лише одного методу, який оптимізує лише певну частину програмного коду.

____

6. Оцінка графічного оформлення та пояснювальної записки роботи: __–____

____

____

____

7. Відгук про роботу в цілому: __В загальному робота виконана на достатньому рівні.____

____

____

8. Інші зауваження: ___–____

____

____

____

9. Оцінка дипломної роботи:

Розглянувши позитивні і негативні сторони представленої дипломної роботи вважаю, що робота заслуговує оцінки “добре” 3.75 (С)__________________________________

Рецензент (прізвище, ім’я, по батькові, посада, місце роботи) Мартинюк В. В. доктор технічних наук, професор кафедри комп’ютерної інженерії та системного програмування Хмельницького національного університету______________________

“_18_”___ _05____2022р. ________________

Завідувачу кафедри КІІС
д-р.техн.наук, проф. Говорущенко Т. О.

Криволосов Юрій Анатолійович
ПІБ здобувача вищої освіти

ФІТ, 2 курсу, групи КІ2м-20-1

ЗАЯВА

З правилами чинного Положення «Про дотримання академічної доброчесності в Хмельницькому національному університеті» від 26.09.2020 (зі змінами від 26.11.2020), згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування заходів дисциплінарної та академічної відповідальності, ознайомлений (а). Про використання програмно-технічних засобів для перевірки кваліфікаційних робіт здобувачів вищої освіти на плагіатоповіщений (а) та надаю свою згоду на обробку та збереження університетом моєї роботи в інституційному репозитарії університету.

Також надаю університету право на передачу моєї роботи для обробки та збереження в базах даних програмно-технічних засобів (Unicheck та Anti-Plagiarism) та використання роботи для виявлення плагіату в інших роботах, які перевіряються програмно-технічними засобами та користувачами, що мають доступ до цих програмно-технічних засобів, виключно в обмежених цілях для виявлення плагіату в текстах робіт.

Робота для перевірки університетом надається в друкованому та електронному варіанті. Електронна версія моєї роботи збігається (ідентична) з друкованою.

дата

підпис

РІШЕННЯ ЕКСПЕРНОЇ КОМІСІЇ

КАФЕДРИ КОМП’ЮТЕРНОЇ ІНЖЕНЕРІЇ ТА ІНФОМАЦІЙНИХ СИСТЕМ

ПРО ДОПУСК КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

Підтверджуємо ознайомлення з результатом звіту подібності щодо роботи, генерованого системою виявлення текстових збігів/ідентичності/схожості:

Назва: Метод та засоби створення спеціалізованого двохпровідного компіляра

Автор: Кривоносов Юрій Андрійович

Спеціальність: 123 – Компютерна інженерія

Освітня програма: освітньо-наукова

Науковий керівник: Боровик О.В., д.т.н., професор

Після аналізу звіту подібності зроблено такий висновок:

№	Висновок	Позначка про відповідність
1	Запозичення, виявлені в роботі, є законними і не є плагіатом. Робота приймається до захисту.	відповідає
2	Виявлені запозичення не є плагіатом, розміщені в розділах, які не описують безпосередньо авторське дослідження, але кількість цитат перевищує обсяг, виправданий поставленою метою роботи Робота приймається до захисту, але має бути відкоригована. Відкоригований варіант має бути поданий на кафедру за 2 дні до захисту, разом із заявою щодо самостійності виконання письмової роботи та ідентичності друкованої та електронної версії роботи	
3	Виявлені запозичення не є плагіатом, але частково розміщені в розділах, які описують безпосередньо авторське дослідження, а кількість цитат перевищує обсяг, виправданий поставленою метою роботи. В зв’язку з цим мета роботи та поставлені завдання не були досягнені. Робота може бути допущена до захисту (наступного року) після того як буде відкоригована та допрацьована і успішно пройде повторну перевірку на академічний плагіат.	
4	Робота містить навмисні текстові спотворення, передбачувані спроби укриття запозичень або інші прояви академічного плагіату. Робота містить фабрикацію або фальсифікацію даних. Робота не допускається до захисту.	

Підтвердження:

Запозичення, виявлені в роботі, є законними і не є плагіатом, оскільки:

1) запозичення розміщені в розділах аналізу існуючих аналогів та прототипів, які не описують безпосередньо авторське дослідження і не стосуються результатів роботи;
2) окремі виявлені збіги є загальновживаними фразами або виразами, про що свідчить посилання системи на збіг з 10-20 джерелами на один фрагмент речення;
3) всі зафіксовані системою ознаки модифікації тексту відносяться до комбінування латинських символів зі україномовними скороченнями індексів в формулах, що не є модифікацією тексту.

Сумарний обсяг всіх запозичень, визначений системою виявлення збігів/ідентичності/схожості, складає 13.2% і адресується до 223 першоджерела, що, з урахуванням наведених обґрунтувань, відповідає характеру наукового дослідження і свідчить на користь кваліфікаційної роботи.

Керівник роботи _______________ О. В. Боровик

Гарант ОП _______________ О. С. Савенко

Завідувач кафедри КІСч _______________ Т. О. Говорущенко