

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної роботи освітнього ступеня «бакалавр»
за спеціальністю 121 «Інженерія програмного забезпечення»
(освітня програма «Інженерія програмного забезпечення»)

на тему:

«Програма для компіляції програмного коду»

Виконав студент групи ІПЗ-20-1
ШЕКЕТА Олександр Олексійович

Керівник роботи:
ДУНЄВ Сергій Сергійович

Рецензент:
МОРОЗОВ Андрій Васильович

Житомир – 2024

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

«ЗАТВЕРДЖУЮ»

Зав. кафедри інженерії
програмного забезпечення

Тетяна Вакалюк

«26» лютого 2024 р.

ЗАВДАННЯ
на кваліфікаційну роботу

Здобувач вищої освіти: **ШЕКЕТА Олександр Олексійович**

Керівник роботи: **ДУНЄВ Сергій Сергійович**

Тема роботи: **«Програма для компіляції програмного коду»**,
затверджена Наказом закладу вищої освіти від **«23» лютого 2024 р., №74/с**

Вихідні дані для роботи: **програма для компіляції програмного коду**

Консультанти з бакалаврської кваліфікаційної роботи із зазначенням розділів, що їх стосуються:

Розділ	Консультант	Завдання видав	Завдання прийняв
1	Дунєв С.С.	02.03.2024р	02.03.2024р
2	Дунєв С.С.	04.04.2024р	04.04.2024р
3	Дунєв С.С.	02.05.2024р	02.05.2024р

РЕФЕРАТ

Випускна кваліфікаційна робота бакалавра представлена у вигляді розробки компілятора для власної C-подібної програмної мови, яка інтегрує декілька сучасних функцій і частково семантично сумісна з мовою C. Пояснювальна записка містить детальний опис архітектури та алгоритмів компілятора, а також ключових особливостей мови програмування.

Основною метою роботи є створення надійного та ефективного компілятора, який приймає текстовий документ з кодом і видає виконуваний файл. Розглядаються основні етапи компіляції, включаючи лексичний аналіз, парсинг, семантичний аналіз та генерацію коду.

В роботі наведено порівняльний аналіз з існуючими компіляторами, обґрунтування вибору алгоритмів та підходів, що забезпечують високу швидкість обробки і точність виконання коду. Також включено розділ, присвячений спеціальним функціям мови, які розширюють можливості стандартного C.

Пояснювальна записка до дипломної роботи містить 100 сторінок, 66 ілюстрацій, 4 додатків, та 25 літературних джерел.

Цей дипломний проєкт спрямований на забезпечення глибокого розуміння механізмів компіляції та внесок у розвиток програмних інструментів, що підтримують розробку і виконання програмного коду на сучасному рівні технологій.

КЛЮЧОВІ СЛОВА: ТОКЕН, ЛЕКСИКА, АНАЛІЗАТОР, ПАРСЕР, КОМПІЛЯЦІЯ, СИНТАКСИС, ГЕНЕРАЦІЯ, СЕМАНТИКА, AST

					ІПЗ.КР.Б-121-24-ПЗ			
Змн.	Арк.	№ докум.	Підпис	Дата				
Розроб.		Шекета О.О.			Програма для компіляції програмного коду	Літ.	Арк.	Аркушів
Керівник		Дунєв С.С.					3	100
						ФІКТ Гр. ІПЗ-20-1		
Рецензент		Морозов А.В.						
Зав. каф.		Вакалюк Т.А.						

ABSTRACT

The final bachelor's thesis is presented in the form of the development of a compiler for a proprietary C-like programming language that integrates several modern functions and is partially semantically compatible with C. The explanatory note contains a detailed description of the compiler's architecture and algorithms, as well as the key features of the programming language.

The main goal of the work is to create a reliable and efficient compiler that accepts a text document with code and produces an executable file. The main stages of compilation are considered, including lexical analysis, parsing, semantic analysis and code generation.

The work provides a comparative analysis with existing compilers, justification for the choice of algorithms and approaches that ensure high processing speed and accuracy of code execution. There is also a section dedicated to special language features that extend the capabilities of standard C.

The explanatory note to the thesis contains 100 pages, 66 illustrations, 4 appendices, and 25 references.

This thesis project is aimed at providing a deep understanding of compilation mechanisms and contributing to the development of software tools that support the development and execution of program code at the current level of technology.

KEYWORDS: TOKEN, LEXER, ANALYZER, PARSER, COMPILATION, SYNTAX, GENERATION, SEMANTICS, AST

					ИПЗ.КР.Б-121-24-ИЗ	Арк.
						4

ЗМІСТ

РЕФЕРАТ	3
ABSTRACT	4
ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ.....	6
ВСТУП	7
РОЗДІЛ 1. АНАЛІЗ НАПРЯМКІВ ВИКОРИСТАННЯ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ.....	9
1.1 Постановка задачі.....	9
1.2 Аналіз аналогів програмного продукту	10
1.3 Вибір архітектури системи.....	12
1.4 Обґрунтування вибору інструментальних засобів та вимоги до апаратного забезпечення	13
Висновки до першого розділу	15
РОЗДІЛ 2. ПРОЄКТУВАННЯ ДОДАТКУ ДЛЯ КОМПІЛЯЦІЇ ПРОГРАМНОГО КОДУ	16
2.1 Визначення варіантів використання та об'єктно-орієнтованої структури системи	16
2.2 Проєктування та реалізація алгоритмів роботи системи	38
2.3 Реалізація функціоналу компілятора	39
Висновки до другого розділу	63
РОЗДІЛ 3. ІНТЕРФЕЙС ТА ПОРЯДОК РОБОТИ З КОМПІЛЯТОРОМ	64
3.1 Опис інтерфейсу користувача.....	64
3.2 Порядок роботи системи	67
3.3 Налаштування роботи системи.....	68
3.4 Тестування системи.....	73
Висновки до третього розділу	74
ВИСНОВКИ	76
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	78
ДОДАТКИ	81

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

AST – Abstract Syntax Tree;

ООП – об'єктно орієнтоване програмування;

ОС – операційна система;

ПЗ – програмне забезпечення;

Рис – рисунок.

					ІПЗ.КР.Б-121-24-ПЗ	Арк.
						6

ВСТУП

Актуальність теми полягає у постійному зростанні потреби у розробці нових технологій для компіляції програмного коду, особливо з урахуванням збільшення складності та розміру програмних систем. Сучасні технології надають нові можливості для оптимізації процесів розробки, адаптації та виконання програм, що забезпечує підвищення ефективності та надійності програмного забезпечення. Створення компілятора для власної мови програмування, що підтримує сучасні функції та є частково сумісною з С, відповідає актуальним вимогам ринку та сприяє розвитку мов програмування, здатних адаптуватися до специфічних потреб розробників.

Метою випускної кваліфікаційної роботи бакалавра є розробка компілятора для С-подібної мови програмування, яка включає реалізацію алгоритмів компіляції, аналізу, а також інтеграцію сучасних функцій, що забезпечують високу продуктивність і точність виконання програм.

Завдання розробки компілятора є наступними:

1. Аналіз існуючих програмних рішень — дослідження характеристик різних мов програмування та особливості їх компіляції для ідентифікації корисних властивостей, які можна інтегрувати у власний проєкт.
2. Проектування архітектури компілятора — розробка структури компілятора, яка дозволяє ефективно перетворювати вхідний код у вихідний виконуваний файл.
3. Реалізація лексичного аналізатора, парсера та генератора — створення модулів для аналізу та перетворення вхідного коду в проміжне представлення, а також подальше перетворення цього представлення у вихідний виконуваний файл.
4. Тестування компілятора — проведення тестів для перевірки точності та правильності роботи процесу компіляції.

Об'єктом дослідження є процес компіляції програмного коду, який включає різні етапи обробки від вхідного коду до виконуваного файлу. Особлива увага приділяється алгоритмам аналізу та синтезу коду в рамках компілятора.

Предметом дослідження є алгоритми та структура компілятора, який розробляється для власної мови програмування. Досліджуються методи лексичного аналізу, парсингу, а також техніки аналізу коду, які можуть бути застосовані для підвищення надійності виконання програм.

РОЗДІЛ 1. АНАЛІЗ НАПРЯМКІВ ВИКОРИСТАННЯ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

1.1 Постановка задачі

Основною метою розробки компілятора для власної С-подібної мови програмування є надання користувачам ефективного інструменту для перетворення написаного коду на виконуваний файл. Даний компілятор забезпечить підтримку сучасних програмних функцій і оптимізацію для підвищення продуктивності та безпеки виконання програм. Створення такого компілятора дозволить користувачам ефективно використовувати нову мову для розробки програмного забезпечення, що вимагає високої продуктивності та надійності.

Для реалізації поставленого завдання, основними етапами роботи є:

1. Аналіз вимог та збір вхідних даних:
 - вивчення існуючих рішень і визначення недоліків та переваг;
 - визначення ключових функцій, які повинен підтримувати компілятор, наприклад як обробка помилок.
2. Вибір технологій та розробка архітектури:
 - вибір програмних мов та інструментів для розробки компілятора;
 - розробка модульної архітектури компілятора, що дозволяє легко додавати нові можливості та оптимізації;
 - планування процесу компіляції, включаючи етапи аналізу та синтезу.
3. Розробка та тестування компілятора:
 - програмування основних компонентів компілятора, включаючи лексичний аналізатор, парсер та генератор коду;
 - розробка інтерфейсу командного рядка для взаємодії з користувачем.

Ці етапи дозволять створити повнофункціональний компілятор, який відповідатиме вимогам сучасних розробників програмного забезпечення та сприятиме розвитку нової програмної мови.

1.2 Аналіз аналогів програмного продукту

Аналізуючи існуючі мови програмування, особливо такі популярні мови як Rust і C, можна ідентифікувати ключові області для покращення у власній мові програмування. Цей аналіз дозволить вибудувати стратегію розвитку, враховуючи сильні та слабкі сторони кожної з мов.

Rust (рис. 1.1) — сучасна мова програмування, створена Mozilla у 2010 році, яка зосереджується на безпеці та паралельному виконанні коду. Вона використовує строгі механізми управління пам'яттю для забезпечення безпеки та ефективності, ідеально підходячи для розробки високонадійних систем.

```
use std::io;

fn main() {
    println!("Enter your name:");

    let mut name = String::new();

    io::stdin()
        .read_line(&mut name)
        .expect("Failed to read line");

    println!("Hello, {}", name.trim());
}
```

Рис. 1.1 – Приклад коду Rust

Переваги:

- висока безпека пам'яті завдяки строгій системі власності (ownership) та borrow-checker;

- наявність простору імен, що дозволяє уникнення конфліктів імен та поліпшення організації коду великих проектів;
- модернізація системи типів для забезпечення більш точного управління даними.

Недоліки:

- заплутаний синтаксис може створювати бар'єри для новачків;
- необхідність постійної корекції коду для сумісності з borrow-checker може сповільнювати розробку;
- слабка підтримка різних архітектур процесорів ускладнює використання мови для вбудованих систем.

C (рис. 1.2) – мова програмування, розроблена Деннісом Рітчі у 1972 році, широко використовується для системного програмування завдяки своїй здатності до маніпулювання системними ресурсами та близькості до машинного коду. Це основа для багатьох сучасних мов і платформ.

```
#include <stdio.h>

int main() {
    char name[100];

    printf("Enter your name: ");
    fgets(name, sizeof(name), stdin);

    printf("Hello, %s", name);

    return 0;
}
```

Рис. 1.2 – Приклад коду C

Переваги:

- висока продуктивність та контроль над системними ресурсами;
- широка підтримка на різноманітному обладнанні та операційних системах;

- прямий доступ до пам'яті через покажчики, що забезпечує гнучкість у програмуванні.

Недоліки:

- відсутність вбудованих механізмів безпеки пам'яті підвищує ризик помилок, які можуть призвести до серйозних вразливостей;
- не існує простору імен, що ускладнює управління великими проектами;
- відсутність перевантаження функцій та автоматичного розпізнавання типів зменшує гнучкість мови.

Беручи до уваги всі аспекти, ми можемо ретельно проаналізувати та включити найкращі функції та можливості Rust та C у нашу власну мову програмування. Такий інтегрований підхід гарантує, що наш продукт не лише задовольняє різноманітні потреби розробників, але й використовує сильні сторони цих визнаних мов програмування.

Завдяки комплексному підходу, наша мова гарантує розробникам доступ до потужного та зручного інструменту, що надаватиме широкий спектр функціональностей для безпечного та ефективного програмування, максимально скорочуючи час розробки.

1.3 Вибір архітектури системи

Основна архітектура компілятора, що обирається для розробки, заснована на моделі розділення процесу на три основних фази: лексичний аналізатор (Lexer), синтаксичний аналізатор (Parser) та генератор коду (Generator). Цей підхід дозволяє чітко структурувати роботу компілятора, оптимізувати кожен з етапів обробки вхідного коду та забезпечити високу модульність і гнучкість системи.

Lexer відповідає за аналіз вхідного тексту програми та перетворення його у потік токенів, що представляють собою атомарні елементи мови програмування. Такий підхід дозволяє ефективно ізолювати решту

компілятора від безпосереднього тексту програми, спрощуючи аналіз і обробку даних на наступних етапах.

Parser бере потік токенів від **Lexer** і перетворює їх у абстрактне синтаксичне дерево (AST), що відображає структурну ієрархію програми згідно з граматикою мови. Застосування AST як проміжної структури даних дає можливість детально аналізувати і модифікувати програму без потреби змінювати вхідний код.

Generator завершує процес компіляції, перетворюючи AST на цільовий код. Цей компонент також відповідає за оптимізацію вихідного коду і забезпечення його відповідності до технічних вимог та ресурсів обчислювальної системи, на якій буде виконуватися програма.

Компонентний підхід у архітектурі **Lexer** і **Parser** сприяє не тільки модульності і еластичності розробки, але й полегшує інтеграцію тестування і подальше масштабування компілятора. Важливо також відзначити, що при розробці не використовувались зовнішні бібліотеки, що надає розробникам повний контроль над всіма процесами в системі і зменшує залежність від сторонніх розробок.

Таким чином, вибір такої архітектури забезпечує високу ефективність компіляції та гнучкість у розширенні функціоналу, що є критично важливим для адаптації компілятора до специфічних вимог різних проектів.

1.4 Обґрунтування вибору інструментальних засобів та вимоги до апаратного забезпечення

При виборі інструментальних засобів для розробки компілятора, особливу увагу було приділено мові програмування C#. Цей вибір підкріплюється низкою значущих переваг, що роблять C# особливо придатною для цього проекту:

- **безпека типів:** C# є сильно типізованою мовою, що забезпечує жорсткий контроль за використанням типів даних. Це зменшує

ймовірність помилок, які можуть виникнути внаслідок неправильного використання типів, наприклад, коли програміст намагається виконати операції, несумісні з даними типами;

- **управління пам'яттю:** автоматичне управління пам'яттю через механізм збирання сміття зменшує ризик витоків пам'яті та інших проблем, пов'язаних з управлінням ресурсами. Програмістам не потрібно безпосередньо займатися звільненням пам'яті, що може бути джерелом помилок та витоків пам'яті в інших мовах, які вимагають вручну управління пам'яттю;
- **об'єктно-орієнтованість:** підтримка об'єктно-орієнтованих парадигм сприяє створенню модульної та легко розширюваної архітектури. Це особливо важливо для проектів, що потребують гнучкості у розробці та можливості легкого впровадження змін або доповнень до існуючого коду;
- **інтеграція з .NET Framework:** .NET надає багатий набір бібліотек і API для реалізації різних функцій від мережеских з'єднань до обробки даних. Ці можливості спрощують розробку та забезпечують потужні інструменти для реалізації складних завдань;
- **крос-платформеність:** завдяки .NET Core, програми на C# можуть виконуватися на різних платформах, включаючи Windows, macOS і Linux. Це забезпечує більшу гнучкість та доступність програм, розроблених на C#, дозволяючи їх розгортання в різноманітних середовищах.

З урахуванням цих переваг, C# стає оптимальним вибором для розробки компілятора, який повинен бути надійним, ефективним і гнучким у технічному виконанні. Також, використання цієї мови дозволяє залишити апаратні вимоги до системи на помірному рівні, оскільки C# та .NET Framework ефективно оптимізовані для виконання на сучасних обчислювальних системах.

Висновки до першого розділу

У першому розділі цієї роботи було визначено основні напрямки та цілі розробки компілятора для власної C-подібної мови програмування. Основною метою є створення ефективного інструменту, який дозволяє перетворювати вихідний код на виконуваний файл, підтримуючи сучасні програмні функції та забезпечуючи високу продуктивність та безпеку виконання програм. Ретельно плановані етапи аналізу, вибору технологій, розробки архітектури та тестування є критичними для досягнення цих цілей.

Аналіз аналогів показав, що, враховуючи сильні та слабкі сторони існуючих мов програмування, таких як Rust і C, нова мова може запропонувати значні переваги, інтегруючи найкращі практики і функціональні можливості для задоволення потреб сучасних розробників. Вибір архітектури системи, заснованої на компонентах та фазах обробки, спрощує розробку та дозволяє забезпечити високу гнучкість та масштабованість продукту.

Застосування мови C# у комбінації з .NET Framework для розробки компілятора виправдано завдяки її потужним можливостям управління пам'яттю, об'єктно-орієнтованістю, та крос-платформеності, що забезпечує широку підтримку та легке розгортання на різних платформах.

Таким чином, підготовка та планування, проведені у цьому розділі, закладають міцний фундамент для подальшої розробки компілятора, що стане надійним інструментом для програмістів, сприяючи розвитку нової мови програмування, яка здатна відповідати вимогам сучасних технологічних та програмних викликів.

РОЗДІЛ 2. ПРОЄКТУВАННЯ ДОДАТКУ ДЛЯ КОМПІЛЯЦІЇ ПРОГРАМНОГО КОДУ

2.1 Визначення варіантів використання та об'єктно-орієнтованої структури системи

У цьому розділі описуються основні варіанти використання компілятора, який розробляється для власної С-подібної мови програмування. Важливим аспектом є визначення функціональних та нефункціональних вимог до системи, а також оцінка часових характеристик реакції системи на різні запити користувача.

Функціональні вимоги:

- компіляція коду: система має здатність перетворювати текст програми, написаний на специфічній С-подібній мові, у виконуваний файл;
- відображення помилок: у разі виявлення помилок у коді під час компіляції, система повинна чітко вказувати місцезнаходження помилки, виводити відповідний фрагмент коду і підкреслювати точне місце помилки.

Нефункціональні вимоги:

- легкість використання: інтерфейс компілятора повинен бути інтуїтивно зрозумілим, що дозволяє користувачам легко виконувати компіляцію коду;
- надійність: система має стабільно функціонувати під час обробки коректного та некоректного коду;
- швидкість обробки: компіляція коду має відбуватися з мінімальними затримками.

Часові характеристики реакції системи:

Час реакції системи на вхідні дії користувача має бути оптимізований, забезпечуючи швидке відображення результатів компіляції чи помилок у коді.

Варіанти використання:

- компіляція програми: користувач подає на вхід текст програми, який компілятор перетворює у виконуваний файл;
- обробка помилок: у разі помилок у коді, система аналізує код, визначає рядок з помилкою, відображає помилковий код та підкреслює проблемне місце.

Користувач:

Основними користувачами системи є програмісти, які використовують цей компілятор для розробки та тестування програм на власній C-подібній мові. Вони потребують зручного інструменту для перевірки правильності та ефективності свого коду.

Ці визначення і описи лягають в основу подальшої деталізації та розробки компілятора, забезпечуючи зручний і ефективний інструмент для програмування.

Об'єктно-орієнтована структура:

У цьому розділі описується об'єктно-орієнтована структура компілятора для власної C-подібної мови програмування. Структура включає кілька ключових класів, кожен з яких відіграє певну роль у процесі компіляції.

Клас «Source» – використовується для збереження вмісту файлу коду, що компілюється, та загальної інформації про файл. Цей клас допомагає отримати інформацією, необхідною для аналізу та обробки коду:

- поле Path: шлях до файлу;
- поле Code: текст програми, який містить повний вміст файлу, який буде оброблятися компілятором.

Клас «Metadata» – забезпечує контекст для токенів та вузлів синтаксичного дерева, вказуючи, де саме в коді вони розташовані:

- поле Source: посилання на об'єкт Source, що дозволяє отримати доступ до вихідного файлу та його вмісту;
- поля Path та Code: властивості, які витягують дані з поля Source для швидкого доступу, які зменшують кількість коду, потрібну для отримання їх значень;
- поля Start та End: позиції, що визначають початок та кінець відповідного фрагменту коду в рамках файлу.

Клас «Stack» – використовується для управління позиціями під час аналізу, зокрема, дозволяє "повертатися" на попередні позиції, якщо це необхідно:

- поле stack: масив, що містить стек збережених станів;
- поле index: поточний стан (індекс) стеку;
- метод CopyRight(): скопіювати теперішню позицію на рівень вище та перейти до неї;
- метод CopyLeft(): скопіювати теперішню позицію на рівень нижче та перейти до неї;
- метод ShiftLeft(): переходить на рівень нижче без збереження поточної позиції.

Клас «Vocabulary» – виконує функції словника для зберігання та керування символьними токенами:

- метод ContainsKey(key): перевіряє, чи міститься даний символ у словнику;
- метод Count(): повертає кількість символів у словнику;
- метод Update(): оновлює інформацію про найдовші та найкоротші символи у словнику;
- метод ToDictionary(): повертає стандартний словник типу Dictionary з C#;

- метод Add(key, value): додає символ та відповідний тип токена до словника;
- метод Remove(key): видаляє символ зі словника.

Токени — це фундаментальні одиниці, з яких складається програмний код у процесі компіляції. Вони представляють собою різні елементи коду, такі як ідентифікатори, літерали, оператори тощо (рис. 2.1). В системі використовуються наступні класи токенів:

Клас «Token» – базовий клас для всіх типів токенів:

- поле Type: тип токена, що дозволяє ідентифікувати функціональну роль токена в коді;
- поле Metadata: об'єкт класу Metadata, який містить інформацію про розташування токена в коді.

Клас «SimpleToken» – використовується для представлення простих токенів, таких як окремі оператори або ключові символи;

Клас «LiteralToken» – базовий клас для всіх літералів, містить універсальне поле «Value» для значення. Від цього класу походять:

- CharToken: представляє символьний літерал;
- EmbeddedToken: використовується для вставки прямого C-коду перед його компіляцією C-компілятором;
- FloatToken: представляє літерали чисел з плаваючою точкою;
- IntegerToken: представляє літерали цілих чисел;
- StringToken: представляє рядкові літерали;
- SymbolToken: представляє токени, які відносяться до символів;
- WhiteSpaceToken: представляє пробіли та інші види "порожніх" символів у коді.

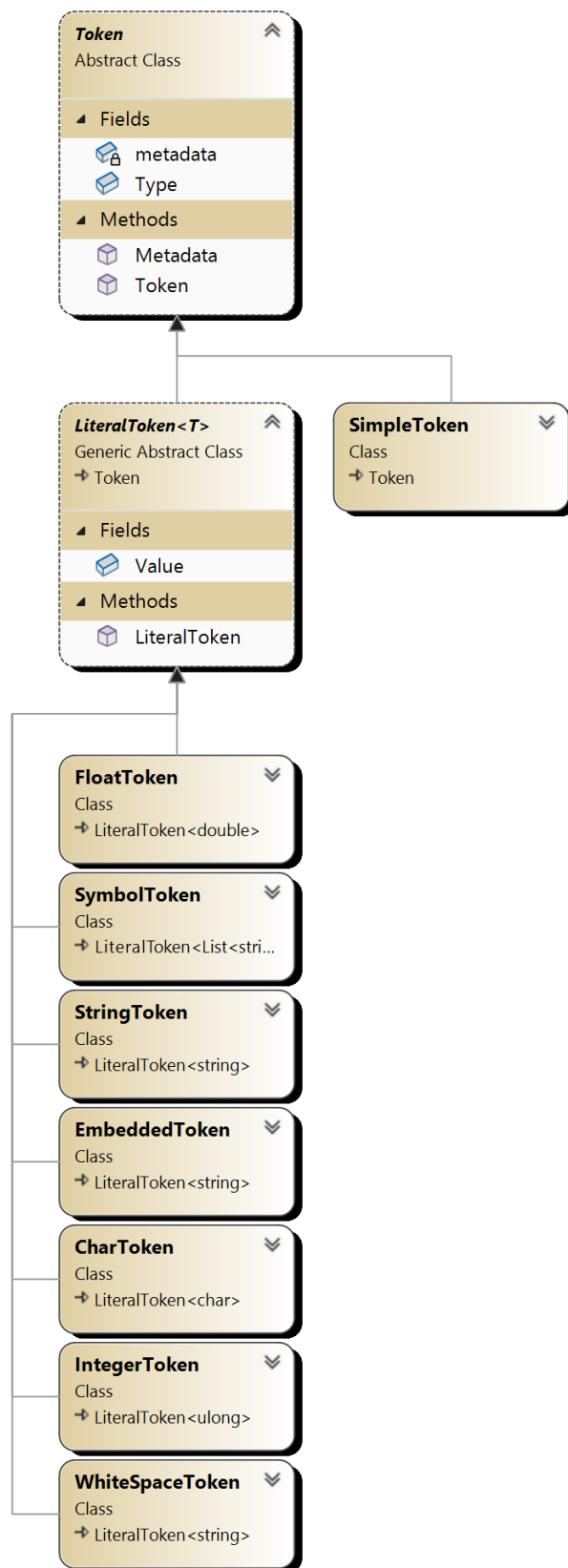


Рис. 2.1 – Діаграма класів токенів

Класи «Handlers» використовуються для обробки різноманітних завдань у системі компіляції, дозволяючи гнучко маніпулювати контекстом

та результатами операцій. Ці класи формують основу для реалізації патерну «композиція»:

Інтерфейс «IHandler» – базовий інтерфейс для всіх handler-ів, який визначає загальну структуру та поведінку:

- метод Execute(ctx, arg): виконує задану логіку і повертає об'єкт або NULL. Приймає як аргумент контекст і опціональний аргумент arg.

Клас «Callback» – виконує роль листа у патерні композиції, де він може містити обгортки, що модифікують чи контролюють виконання основної логіки:

- метод Execute(ctx, arg): запускає обгортки і основну логіку;
- метод Main(ctx, arg): абстрактний метод, який визначає основну логіку обробки;
- метод invoke(): викликає Main безпосередньо;
- метод Wrapper(wrapper): додає обгортку, яка може модифікувати поведінку перед чи після основної логіки;
- поля start і end: визначають початкові та кінцеві обгортки в ланцюжку.

Клас «Dispatcher» – відповідає за виконання ролі композита у патерні композиції, зберігаючи та управляючи множиною об'єктів «IHandler»:

- метод Add(handler): додає новий handler до списку;
- метод Remove(handler): видаляє handler зі списку;
- метод Contains(handler): перевіряє наявність handler у списку.

Класи «Wrappers» (рис. 2.2) в системі використовуються для додавання додаткової логіки обробки перед чи після виконання основних handler-ів. Вони можуть змінювати поведінку або результати обробки, а також забезпечувати додаткову перевірку умов:

- інтерфейс «IWrapper» – базовий інтерфейс для всіх обгортки;
- інтерфейс «IWithLowerWrapper» – інтерфейс для обгортки, що дозволяє взаємодіяти не просто з хендлером, а з іншою обгорткою, формуючи ланцюжок обробки;

- клас «Wrapper» – базовий клас, що реалізує функціонал обгортки, здатний інтегрувати як просту, так і ланцюгову обробку;
- клас «InvokeWrapper» – мінімалістична обгортка, яка просто передає виклик наступному об'єкту в ланцюжку без будь-яких змін;
- клас «ReversibleWrapper» – обгортка, яка додає можливість зберігати стан перед виконанням хендлера та повертатися до попереднього стану, якщо результат обробки є NULL. Це може бути корисно для відкату змін у випадку помилок або невдалих спроб виконання.

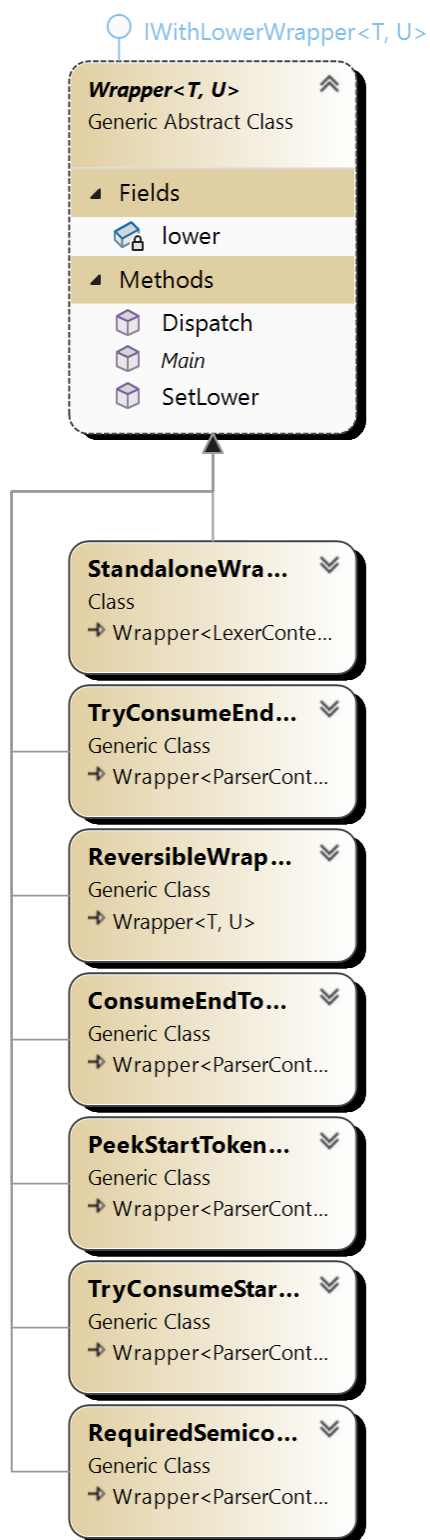


Рис. 2.2 – Діаграма класів обгортки

Класи «Context» забезпечують необхідну інформацію про поточний стан обробки коду, дозволяючи компонентам, таким як «Lexer» і «Parser», ефективно виконувати свої задачі на основі поточного контексту:

Інтерфейс «IReversibleContext» – інтерфейс для контекстів, які дозволяють збереження та відновлення свого стану:

- метод Backup(): зберігає поточний стан контексту;
- метод Commit(): зафіксує зміни, визначаючи поточний стан як основний;
- метод Restore(): відновлює останній збережений стан, скасовуючи нефіксовані зміни.

Інтерфейс «IScannerContext» – контекст, спеціалізований для сканування та аналізу вхідного файлу коду:

- метод TryPeek(expected): перевіряє, чи дорівнює поточний символ або токен очікуваному значенню без переміщення курсору;
- метод TryConsume(expected): перевіряє поточний символ або токен і переміщує курсор вперед, якщо він відповідає очікуваному;
- метод TryConsumeMultiple(expected): перевіряє послідовність символів або токенів на відповідність вказаному масиву значень;
- метод TryConsumeRequired(expected): аналог TryConsume, з обов’язковою умовою, що не було досягнуто кінця файлу;
- метод Consume(): зчитує поточний символ або токен і переміщує курсор вперед;
- метод Peek(): повертає поточний символ або токен без переміщення курсору;
- метод Compare(a, b): порівнює два значення;
- метод Read(): зчитує дані на поточній позиції;
- метод EndOfFile(): перевіряє, чи досягнуто кінця файлу;
- метод Fallback(): визначає поведінку, коли досягнуто кінця файлу.

Клас «Scanner» (рис. 2.3) – базовий клас для сканеру, який реалізує методи IScannerContext та IReversibleContext. Цей клас слугує основою для створення конкретних сканерів, які здійснюють лексичний аналіз інпуту програми. Він дозволяє зчитувати дані, визначати поточну позицію у файлі, і

реверсувати до попереднього стану у випадку помилок або коли необхідно повторно проаналізувати ділянку коду.

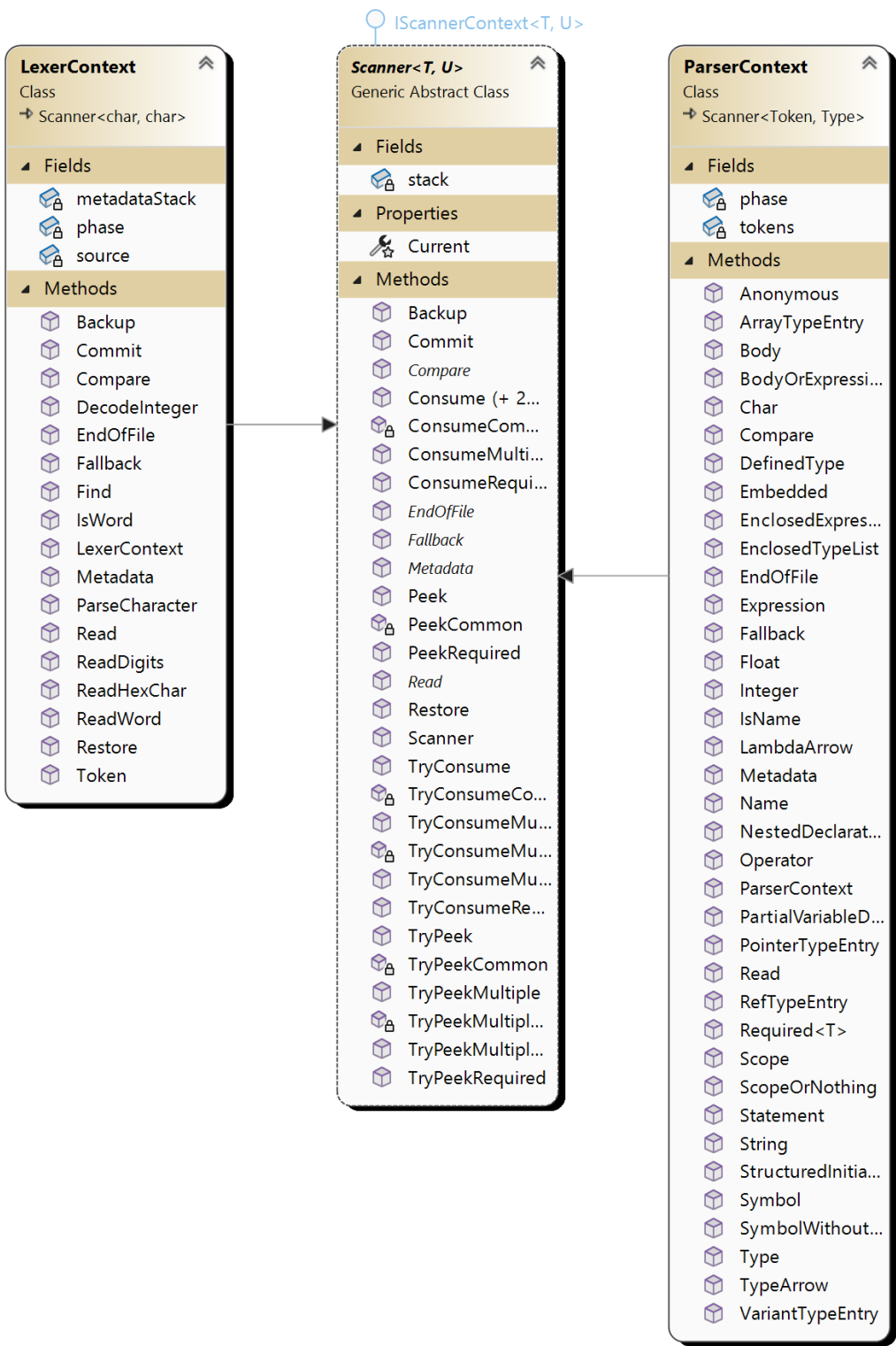


Рис. 2.3 – Діаграма класів контекстів

Класи «Exceptions» відіграють ключову роль у обробці помилок, що виникають під час компіляції. Ці класи винятків допомагають ідентифікувати

та керувати специфічними проблемами, що вимагають особливої уваги або втручання:

- `CompileException`: базовий клас для усіх помилок, пов'язаних з компіляцією. Може містити текст помилки та опціонально метадані про місце її виникнення;
- `EmptyConditionException`: виняток, що виникає, коли умова твердження залишається порожньою;
- `IncompatibleOperandsException`: виникає, коли операнди у виразі є несумісними для заданої операції;
- `InternalException`: використовується для вказівки на внутрішні помилки компілятора, не пов'язані з кодом користувача;
- `InvalidDereferenceOperandException`: помилка виникає, коли спроба дереференції здійснюється над об'єктом, який не є покажчиком;
- `InvalidIndexedOperandException`: виникає, коли спроба індексації здійснюється над типом, який не є масивом або покажчиком;
- `InvalidIndexTypeException`: виняток для випадків, коли тип індексу не є сумісним з вимогами індексації;
- `MissingBodyException`: виняток для ситуацій, коли твердження очікує тіло, але воно відсутнє;
- `NonArithmeticTypeException`: виникає, коли арифметичні операції виконуються над неарифметичним типом;
- `NonAssignableException`: стосується спроби присвоєння до об'єкту, який не може бути призначений;
- `UnknownIdentifierException`: виняток для випадків, коли в програмі використовується невідомий ідентифікатор;
- `UnknownOperandException`: виникає, коли в операції використовується невідомий операнд;
- `UnknownStatementException`: помилка, пов'язана з використанням невідомого твердження у коді.

Класи «Nodes» виконують ключову роль у створенні абстрактного синтаксичного дерева (AST), яке є основою для подальшого аналізу та трансформації програмного коду. Кожен вузол у дереві відображає конкретні елементи програми, такі як оператори, літерали чи твердження (рис. 2.4):

Клас «Node» – базовий клас для всіх вузлів у абстрактному синтаксичному дереві:

- поле parent: батьківський вузол, який використовується для зв'язку вузлів у дереві;
- поле Metadata: метадані, що містять інформацію про розташування вузла у вхідному коді;
- метод Update(x): оновлює метадані вузла на основі метаданих іншого вузла або токена;
- метод Child(node): встановлює даний вузол як дитячий, оновлюючи метадані відповідно;
- метод OptionalChild(node): те ж, що Child, але дозволяє NULL-значення для вузлів.

Клас «NodeCollection» – колекція вузлів певного типу, що дозволяє зберігати та управляти групами вузлів:

- поле nodes: список дочірніх вузлів;
- метод Add(node): додає вузол до колекції;
- метод Insert(index, node): вставляє вузол на задану позицію у списку.

Класи «SingleChildNode», «DoubleChildNode», «TripleChildNode», «OptionalChildNode» – ці класи використовуються для спрощення структури AST, де вузли можуть мати різну кількість дочірніх вузлів:

- SingleChildNode: вузол з одним дочірнім вузлом;
- DoubleChildNode: вузол з двома дочірніми вузлами;
- TripleChildNode: вузол з трема дочірніми вузлами;
- OptionalChildNode: вузол з одним необов'язковим дочірнім вузлом.

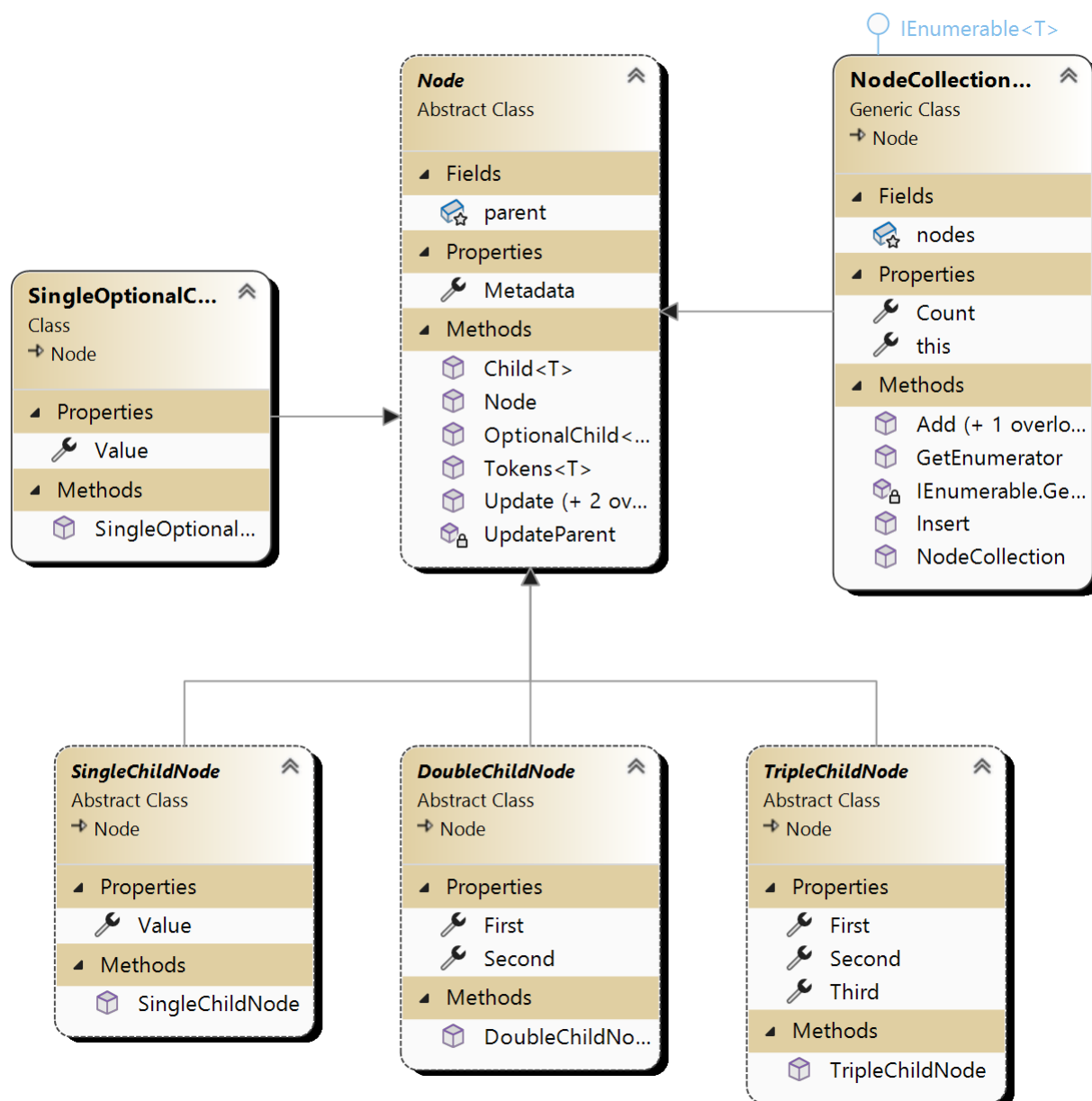


Рис. 2.4 – Діаграма класів вузлів

Класи терміналів в системі компіляції представляють собою базові елементи коду, які використовуються для побудови абстрактного синтаксичного дерева. Вони забезпечують деталізацію та специфікацію вихідного коду, дозволяючи точно відобразити структуру програми у внутрішньому представленні.

Літерали відповідають за представлення простих значень, які вводяться безпосередньо у коді.

Підкласи:

- CharNode: представляє символні літерали;

- EmbeddedNode: використовується для вбудованого C-коду;
- FloatNode: представляє числа з плаваючою точкою;
- IntegerNode: відображає цілі числа;
- StringNode: для рядкових літералів;
- SymbolNode: використовується для ідентифікаторів та символів у коді.

Фактори включають складніші конструкції, які мають власні правила синтаксису та поведінки:

- BorrowNode: використовується для обозначення операндів позичання, які вимагають посилань на змінні;
- NameNode: схожий на SymbolNode, але представляє локальні імена без префіксних просторів імен;
- ParenthesisNode: використовується для дужок, що огортають вирази;
- ArrayInitializationNode: для ініціалізації масивів, включає список значень.

Класи «Types» використовуються для визначення та управління типами даних у програмі, що дозволяє точно відслідковувати та управляти типами змінних та параметрів:

- «TypeNode»: базовий клас для всіх типів даних;
- «AutoConstTypeNode»: спеціальний «неявний» тип даних, де єдине, що про нього відомо, це те, що він є константою;
- «DefinedTypeNode»: містить окремі дані про тип у вигляді списку об'єктів типу «TypeEntryNode». Кількість цих об'єктів допомагає визначити вимір типу: якщо наявно два об'єкта, то тип одновимірний; якщо три, то тривимірний; якщо тільки один, то тип не має вимірності.

Класи «Entries» – використовуються для управління властивостями типів даних на рівні їх визначення:

Клас «TypeEntryNode»: Базовий клас для визначення властивостей підтипів:

- поле IsConstant: визначає, чи є вимір константою.

Класи «Dimensions» – окремі «прості» покажчикові види підтипу – масиви «ArrayTypeEntryNode», і самі покажчики «DimensionTypeEntryNode».

Класи «References» – підтипи «розумних» покажчиків, які мають особливі правила під час компіляції:

- «ArcTypeEntry»: покажчик з автоматичним підрахунком посилань, не був реалізований за час виконання роботи;
- «BrwTypeEntry»: тип, який має особливі обмеження щодо значень, що живуть поза даної області дії коду.

Класи «Variants» – тільки був реалізований функціонал PrimitiveTypeEntryNode – звичайний підтип, який лише вказує на символ, який використовується для під-типу.

Оператори у вашій системі компіляції розділені на кілька категорій в залежності від їх синтаксичних та семантичних ролей. Кожен клас операторів реалізує певну логіку виконання та обробки в рамках абстрактного синтаксичного дерева: унарні, бінарні та тернарні.

Бінарні оператори включають арифметичні, логічні та бітові операції, які взаємодіють з двома операндами. Вони забезпечують основну взаємодію між змінними та константами у програмах:

- арифметичні оператори:
 - AdditionNode;
 - SubtractionNode;
 - MultiplicationNode;
 - DivisionNode;
 - ModuloNode;
- присвоєння з операцією:
 - AdditionAssignmentNode;

- SubtractionAssignmentNode;
- MultiplicationAssignmentNode;
- DivisionAssignmentNode;
- ModuloAssignmentNode;
- бітові оператори:
 - BitwiseAndNode;
 - BitwiseOrNode;
 - BitwiseXorNode;
 - BitwiseNotNode;
 - BitwiseShiftLeftNode;
 - BitwiseShiftRightNode;
- оператори присвоєння для бітових операцій:
 - BitwiseAndAssignmentNode;
 - BitwiseOrAssignmentNode;
 - BitwiseXorAssignmentNode;
 - BitwiseNotAssignmentNode;
 - BitwiseShiftLeftAssignmentNode;
 - BitwiseShiftRightAssignmentNode;
- логічні оператори:
 - AndNode;
 - OrNode;
- оператори порівняння:
 - EqualsNode;
 - NotEqualsNode;
 - GreaterNode;
 - LessNode;
 - GreaterEqualsNode;
 - LessEqualsNode.

Тернарні оператори реалізують умовну логіку, яка дозволяє вибирати між двома значеннями на основі третього умовного виразу. Наявний тільки один оператор: ConditionalNode.

Унарні оператори працюють з одним операндом, модифікуючи його стан або значення. Це може бути зміна знаку, інкрементація, декрементація, або отримання адреси.

Префіксні оператори:

- PrefixIncrementNode;
- PrefixDecrementNode;
- NegateNode;
- NotNode;
- AddressNode;
- DereferenceNode;
- TypecastNode.

Постфіксні оператори, такі як індексація масиву чи виклик функції, використовуються для звернення або виконання після виразу:

- IndexNode;
- InvokeNode;
- PostfixIncrementNode;
- PostfixDecrementNode.

Твердження є основними блоками логіки в програмах, що визначають контроль потоку, декларації змінних, функцій та інші операції.

Основні твердження – реалізують різні структурні конструкції мови програмування, що дозволяє виконувати різні операції у програмах.

- DoNode: цикл do-while, виконує блок коду доки умова істинна;
- ForNode: цикл for, виконує ініціалізацію, умову, та ітерацію;
- IfNode: умовний оператор if, виконує блок коду, якщо умова істинна;
- ReturnNode: оператор return, повертає значення з функції;

- ScopeNode: визначає область видимості для змінних і логіки;
- UseNode: використовується для включення зовнішніх модулів або просторів імен;
- WhileNode: цикл while, виконує блок коду, поки умова істинна.

Декларації змінних – ці класи використовуються для опису змінних у програмі:

- VariableNode: базовий абстрактний клас для всіх змінних;
- StaticVariableNode: статична змінна, зберігає своє значення між викликами (не реалізовано);
- RuntimeVariableNode: змінна часу виконання, її значення може змінюватись.

Декларації функцій – ці класи визначають функції у програмі:

- FunctionNode: базовий абстрактний клас для всіх функцій;
- OperatorFunctionNode: функція, що перевизначає оператори (не реалізовано);
- NamedFunctionNode: іменована функція, основний тип функції в багатьох програмах.

Параметри Функцій – управляють параметрами, які передаються до функцій.

- FunctionParameterNode: базовий абстрактний клас для параметрів функцій;
- FunctionValueParameterNode: стандартний параметр;
- FunctionVariadicParameterNode: варіативний параметр (не реалізовано).

Сам компілятор розбитий на три етапи (фази): лексер, парсер та генератор.

Лексичний аналізатор виконує критичну роль у процесі компіляції, перетворюючи вхідний текст програми на серію токенів, які є основними будівельними блоками для синтаксичного аналізатора.

Клас «LexerContext»: контекст лексера управляє поточним станом аналізу, включаючи вихідний файл, позицію читання та стек метаданих. Він використовує методи для порівняння символів, визначення кінця файлу, читання поточних символів, та управління збереженням станів під час сканування;

Клас «LexerPhase»: це основна складова, яка організовує різні колбеки для розпізнавання та обробки токенів. Вона включає методи для ініціалізації вокабуляра і встановлення колбеків для обробки специфічних токенів:

- метод IsWord(allowDigits): перевіряє, чи є поточний символ частиною слова;
- метод ReadWord(): читає слово з поточної позиції;
- метод ReadDigits(): читає послідовність цифр з поточної позиції;
- метод DecodeInteger(digits, prefix): декодує цілі числа з визначеним форматом (наприклад, шістнадцяткові, вісімкові);
- метод Find(vocabulary): знаходить токени, використовуючи визначені в вокабулярі шаблони;
- метод ParseCharacter(): обробляє ескейп-послідовності в символах;
- метод ReadHexChar(): читає шістнадцятковий символ.

Список обробників:

- WhiteSpaceCallback: обробка пробільних символів;
- SingleLineCommentCallback та MultiLineCommentCallback: обробка однорядкових та багаторядкових коментарів;
- EmbeddedCallback, FloatCallback, StringCallback, CharCallback: обробка вбудованих кодів, чисел з плаваючою точкою, рядків та символів;
- HexCallback, OctalCallback, BinaryCallback: обробка чисел у різних системах числення;
- NumberCallback: обробка десяткових чисел;

- VocabularySymbolCallback і VocabularyTermCallback: обробка специфічних символів та ключових слів заздалегідь визначених у вокабулярі.

LexerPhase використовує набір обробників для ідентифікації та створення токенів відповідно до вхідного коду, що включає стандартні оператори, ідентифікатори, ключові слова, літерали тощо. Кожен колбек спеціалізований на конкретному типі токена або структурі мови.

Парсер є другою критичною стадією у процесі компіляції, де він аналізує та перевіряє синтаксис токенів, отриманих від лексера, та будує абстрактне синтаксичне дерево (AST). AST подальше використовується для семантичного аналізу та генерації вихідного коду.

Клас «ParserPhase»: організує різні обробники для розпізнавання конструкцій мови, включаючи оператори, твердження, типи даних, та інші синтаксичні елементи.

Клас «ParserContext»: управління поточним станом аналізу, включаючи список токенів та поточну позицію читання.

Методи класу ParserContext:

- Char(): зчитування токена типу CharToken;
- Float(): зчитування токена типу FloatToken;
- Integer(): зчитування токена типу IntegerToken;
- String(): зчитування токена типу CharToken;
- Symbol(): зчитування токена типу CharToken;
- SymbolWithoutGeneric(): теж саме, що Symbol(), але зі забороненим використанням шаблонних символів;
- Embedded(): зчитування токена типу EmbeddedToken;
- IsName(): перевірка, чи може даний символ бути локальним ідентифікатором;
- Name(): зчитування символу, і повернення його як NameNode;
- Statement(): зчитування твердження;

- Expression(): зчитування виразу;
- Type(): зчитування будь-якого типу;
- DefinedType(): зчитування явного типу;
- RefTypeEntry(): зчитування підтипу RefTypeEntryNode;
- PointerTypeEntry(): зчитування підтипу PointerTypeEntryNode;
- ArrayTypeEntry(): зчитування підтипу ArrayTypeEntryNode;
- VariantTypeEntry(): зчитування підтипу VariantTypeEntryNode;
- Required(node): повернення значення “node”, за умовою, що воно не довірнює NULL;
- Body(): зчитування обов’язкового твердження;
- BodyOrExpression(): зчитування обов’язкового твердження або виразу;
- PartialVariableDeclaration(): зчитування твердження декларації змінної;
- ScopeOrNothing(): зчитування або дужок, або нічого (крапки з комою);
- TypeArrow(): повернення токенів, які використовуються для позначення типу функції (->);
- EnclosedExpression(): зчитування виразу, який знаходиться в круглих дужках;
- Scope(): зчитування блоку коду;
- Operator(): зчитування токенів оператора;
- EnclosedTypeList(): зчитування списку типів, розділених між комою.

Генератор є третьою та останньою фазою у процесі компіляції, де аналізується абстрактне синтаксичне дерево (AST), перевіряється його семантика та будується вихідний С-код для генерації виконуваного файлу. Генератор виконує трансляцію з внутрішнього представлення мови в код, що може бути зкомпільований С-компілятором.

Клас «GeneratorContext»: керує поточним станом генерації, зберігає контекст для змінних, функцій, та областей видимості:

- метод FunctionRegister: реєстрація функцій у глобальному контексті та обробка перевантаження функцій;
- метод VariableRegister: реєстрація змінних з визначенням типів та ініціалізація змінних;
- метод EncodeType і EncodeTypedName: допоміжні функції для кодування типів та імен змінних у форматі C.

Процес генерації коду:

- перетворення функцій: код C-функцій генерується на основі AST, де кожна функція має свій унікальний ідентифікатор для підтримки перевантаження;
- перетворення типів даних: типи даних, визначені у користувацькій програмі, транслуються в еквівалентні типи C.
- реєстрація та використання змінних: змінні реєструються з їхніми типами, і генерується код для їх ініціалізації та використання;
- обробка структурних блоків: код для умовних операторів, циклів та інших структурних елементів перетворюється у відповідні конструкції мови C.

Кожен блок коду має свій GeneratorContext, що дозволяє зберігати області видимості змінних і функцій.

В генераторі реалізована підтримка вкладених та глобальних областей видимості, забезпечуючи правильну ізоляцію та доступність імен у відповідності до контексту їх використання.

Ця фаза критично важлива для перевірки семантичної коректності програми та її подальшої підготовки до компіляції на мові нижчого рівня, забезпечуючи, що вихідний код відповідає всім вимогам та оптимізований для виконання.

2.2 Проектування та реалізація алгоритмів роботи системи

Компілятор, який розроблено, є інтуїтивно зрозумілим у використанні та максимально ефективним для базових операцій компіляції. Алгоритм його роботи дозволяє користувачам легко проводити компіляцію вихідних кодів, з відповідними перевітками та управлінням помилками. Процес використання компілятора візуально зображено (рис. 2.5) та описано детально в наступних пунктах.

Початок використання:

1. Написання коду: користувач пише код, який потребує компіляції.
2. Компіляція коду: після завершення написання коду, користувач ініціює процес компіляції.

Обробка помилок:

1. Виявлення помилок: якщо в процесі компіляції з'являються помилки, система автоматично виявляє їх та повідомляє користувачу.
2. Виправлення помилок: користувач аналізує помилки та робить необхідні виправлення у коді.
3. Повторна компіляція: після виправлення помилок код знову подається на компіляцію.

Завершення роботи:

1. Збереження виконаного файлу: Якщо код не містить помилок, компілятор генерує виконуваний файл.
2. Кінець процесу: Після збереження файлу процес компіляції завершується.

Цей алгоритм спрощує взаємодію з компілятором, роблячи його доступним для користувачів з будь-яким рівнем знань у програмуванні.

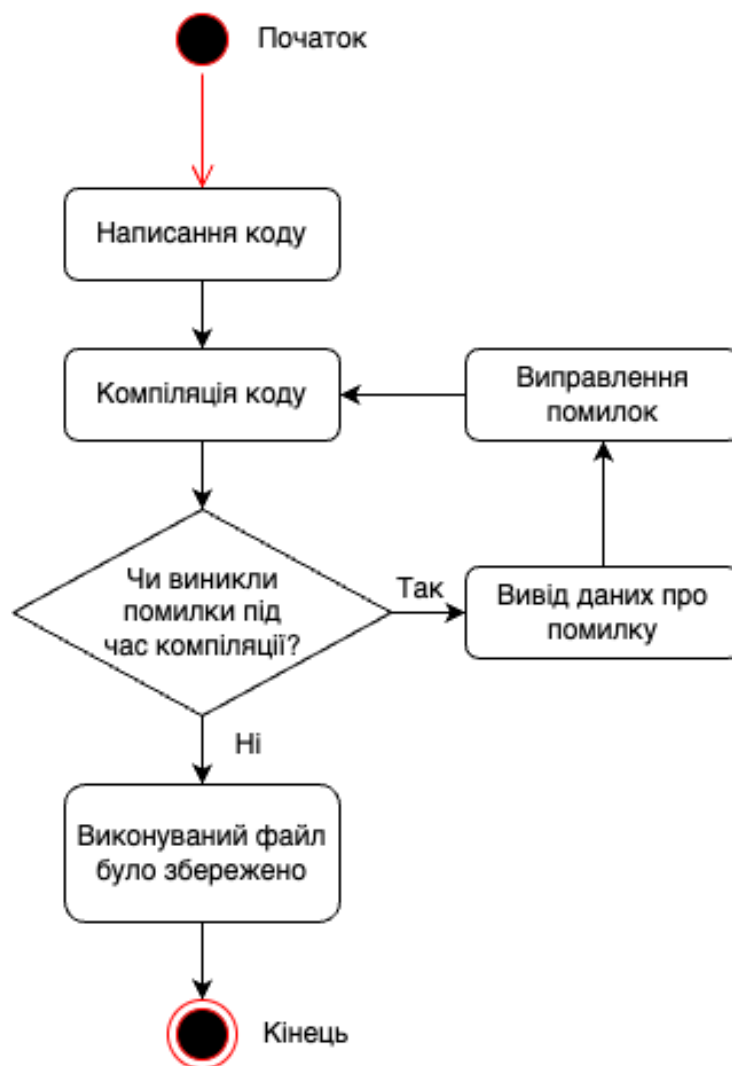


Рис. 2.5 – Діаграма активності

2.3 Реалізація функціоналу компілятора

Почнемо з фази «лексер».

Лексичний аналізатор, реалізований у класі `LexerPhase`, відіграє ключову роль у перетворенні вхідного коду на серію токенів, які подальше використовуються парсером для створення абстрактного синтаксичного дерева.

Ініціалізація лексичного аналізатора: у конструкторі класу `LexerPhase` ініціюється налаштування колбеків через метод `SetupCallbacks()`. Це дозволяє динамічно конфігурувати обробку різних типів токенів (рис. 2.6).


```

public LexerPhase()
{
    Callbacks = SetupCallbacks();
}

```

Рис. 2.6 – Конструктор класу LexerPhase

Цей метод встановлює колбеки для обробки різних категорій токенів, включаючи пробіли, коментарі, різні числові формати, символи та ключові слова. Використання диспатчера LexerDispatcher дозволяє ефективно управляти різними обробниками токенів (рис. 2.7).

```

private static IHandler<LexerContext, Token> SetupCallbacks()
{
    return new LexerDispatcher
    {
        new WhiteSpaceCallback(),
        new SingleLineCommentCallback(),
        new MultiLineCommentCallback(),
        new EmbeddedCallback(),
        new FloatCallback(),
        new StringCallback(),
        new CharCallback(),
        new HexCallback(),
        new OctalCallback(),
        new BinaryCallback(),
        new NumberCallback(),
        new VocabularySymbolCallback(SetupSymbols()),
        new VocabularyTermCallback(SetupTerms()),
        new SymbolCallback()
    };
}

```

Рис. 2.7 – Код методу SetupCallbacks

Визначення термінів: метод SetupTerms() створює словник, який використовується для розпізнавання та класифікації ключових слів у вхідному коді, кожне з яких має відповідний тип токена (рис. 2.8).

```
private static Vocabulary SetupTerms()
{
    return new Vocabulary()
    {
        {"typedef", Type.TYPDEF},
        {"template", Type.TEMPLATE},
        {"class", Type.CLASS},
        {"public", Type.PUBLIC},
        {"protected", Type.PROTECTED},
        {"private", Type.PRIVATE}
    }
}
```

Рис. 2.8 – Обрывок коду з методу SetupTerms

Аналогічним чином працюють символи (рис. 2.9):

```
private static Vocabulary SetupSymbols()
{
    return new Vocabulary()
    {
        {"!", Type.EXCLAMATION},
        {"%", Type.PERCENTAGE},
        {"&", Type.AMPERSAND},
        {"*", Type.ASTERISK},
        {"+", Type.PLUS}
    }
}
```

Рис. 2.9 – Обрывок коду з методу SetupSymbols

Лексичний аналіз вхідного коду: метод Tokenize виконує реальний процес лексичного аналізу. Він ітеративно читає символи з вхідного джерела до досягнення кінця файлу, при цьому кожен токен, що не є пробілом, додається до списку токенів. Також в кінці додається токен, який позначає кінець файлу (рис. 2.10).

```

public List<Token> Tokenize(Source source)
{
    var ctx = new LexerContext(this, source);
    var tokens = new List<Token>();

    while (!ctx.EndOfFile())
    {
        var token = ctx.Token();

        if (token.Type != Type.WHITESPACE)
        {
            tokens.Add(token);
        }
    }

    tokens.Add(new SimpleToken(Type.EOF, ctx.Metadata()));

    return tokens;
}

```

Рис. 2.10 – Код методу Tokenize

IsWord (рис. 2.11) визначає, чи є поточний символ, що розглядається сканером, частиною слова. Словом можуть бути літери, цифри та символи підкреслення ('_'). Цифри вважаються допустимими символами слова лише за умови, що allowDigits встановлено в true.

```

public bool IsWord(bool allowDigits = true)
{
    var c = Peek();

    if (!allowDigits && char.IsDigit(c))
    {
        return false;
    }

    return c == '_' || char.IsAsciiLetterOrDigit(c);
}

```

Рис. 2.11 – Код методу IsWord

ReadWord (рис. 2.12) зчитує послідовність символів, які утворюють "слово" за визначенням методу IsWord. Цей метод читає символи до тих пір, поки вони відповідають критеріям слова, що дозволяє ефективно зчитувати ідентифікатори та ключові слова.

```

public string ReadWord()
{
    if (!IsWord(false))
    {
        // read the rest so the error's metadata will wrap the entire "word"
        while (char.IsAsciiLetterOrDigit(Peek()))
        {
            Consume();
        }

        throw new CompileException("Unable to decode the token", Metadata());
    }

    var sb = new StringBuilder();

    while (IsWord())
    {
        sb.Append(Consume());
    }

    return sb.ToString();
}

```

Рис. 2.12 – Код методу ReadWord

Метод ReadDigits (рис. 2.13) зчитує послідовність цифр з поточної позиції у вхідному коді, що є корисним для зчитування числових літералів.

```

public string ReadDigits()
{
    var sb = new StringBuilder();

    while (char.IsDigit(Peek()))
    {
        sb.Append(Consume());
    }

    return sb.ToString();
}

```

Рис. 2.13 – Код методу ReadDigits

ParseCharacter (рис. 2.14) обробляє зчитування екранованих символів та спеціальних символів як '\n', '\t', тощо. Цей метод є важливим для правильної обробки символьних літералів.

```

public char ParseCharacter()
{
    var c = Consume();

    if (c != '\\')
    {
        return c;
    }

    c = Consume();

    switch (c)
    {
        case 'n':
            c = '\n';
            break;
        case 't':
            c = '\t';
            break;
        case 'r':
            c = '\r';
            break;
        case '0':
            c = '\0';
            break;
        case 'x':
            c = ReadHexChar();
            break;
        default:
            break;
    }

    return c;
}

```

Рис. 2.14 – Код методу ParseCharacter

Метод Find (рис. 2.15) використовується для ідентифікації та видобування токенів на основі заданого словника (Vocabulary). Цей метод виконує сканування можливих кандидатів з максимально можливою довжиною до мінімальної, що дозволяє точно розпізнавати найбільш специфічні токени перед загальними.

```

public Token? Find(Vocabulary vocabulary)
{
    // Read the candidates from longest possible length to lowest
    Backup();

    var candidates = new string[vocabulary.Longest - vocabulary.Shortest + 1];
    string text = "";

    for (int i = 0; i <= vocabulary.Longest - vocabulary.Shortest; i++)
    {
        text += Consume();
        candidates[i] = text;
    }

    Restore();

    // Check each candidate in the list
    for (int i = candidates.Length - 1; i >= 0; i--)
    {
        text = candidates[i];

        if (vocabulary.ContainsKey(text))
        {
            // Move the cursor to the end of the token
            for (int j = 0; j < text.Length; j++)
            {
                Consume();
            }

            return new SimpleToken(vocabulary[text], Metadata());
        }
    }

    return null;
}

```

Рис. 2.15 – Код методу Find

Метод DecodeInteger (рис. 2.16) призначений для розпізнавання інтегральних літералів із специфічними префіксами (наприклад, для шістнадцяткових, восьмеричних, та двійкових значень). Він перевіряє наявність префікса, якщо він вказаний, і продовжує зчитування числа за допомогою дозволених символів.

```

public Token? DecodeInteger(string allowed, char? prefix = null)
{
    if (Consume() != '0')
    {
        return null;
    }

    if (prefix != null && Consume() != prefix)
    {
        return null;
    }

    if (!char.IsAsciiLetterOrDigit(char.ToLower(Peek())))
    {
        return null;
    }

    var sb = new StringBuilder();

    while (allowed.Contains(char.ToLower(Peek())))
    {
        sb.Append(Consume());
    }

    if (sb.Length == 0)
    {
        return null;
    }

    return new IntegerToken(
        Convert.ToUInt64(sb.ToString(), allowed.Length),
        Metadata()
    );
}

```

Рис. 2.16 – Код методу DecodeInteger

Опис деяких обробників:

BinaryCallback – відповідає за розпізнавання двійкових чисел (рис. 2.17). Метод Main викликає DecodeInteger, передаючи рядок "01", що вказує на допустимі символи для двійкових чисел, та символ 'b', який позначає префікс двійкового числа.

```

protected override Token? Main(LexerContext ctx, object? arg)
{
    return ctx.DecodeInteger("01", 'b');
}

```

Рис. 2.17 – Код обробника BinaryCallback

CharCallback – розпізнає символні літерали, обрамлені одинарними лапками (рис. 2.18). Метод спершу перевіряє наявність початкової одинарної лапки, розбирає символ та очікує закриваючу лапку.

```
protected override Token? Main(LexerContext ctx, object? arg)
{
    if (!ctx.TryConsume('\'))
    {
        return null;
    }

    var result = ctx.ParseCharacter();
    ctx.Consume('\');

    return new CharToken(result, ctx.Metadata());
}
```

Рис. 2.18 – Код обробника CharCallback

EmbeddedCallback – призначений для вбудованих блоків коду, огорнутих у <<< >>> (рис. 2.19). Після виявлення початкових символів, метод збирає вміст до закриваючих символів.

```
protected override Token? Main(LexerContext ctx, object? arg)
{
    if (!ctx.TryConsumeMultiple('<', '<', '<'))
    {
        return null;
    }

    var sb = new StringBuilder();

    while (!ctx.TryConsumeMultipleRequired('>', '>', '>'))
    {
        sb.Append(ctx.Consume());
    }

    return new EmbeddedToken(sb.ToString(), ctx.Metadata());
}
```

Рис. 2.19 – Код обробника EmbeddedCallback

FloatCallback – розпізнає дійсні числа з плаваючою точкою (рис. 2.20). Цей метод обробляє ліву та праву частини дійсного числа, перевіряє наявність символу 'f' на кінці, та повертає відповідний токен. Дозволяє дійсним числам мати порожнє ціле число або з лівого, або з правого боку: в таких випадках їх значення автоматично буде вважатись “0”.

```

protected override Token? Main(LexerContext ctx, object? arg)
{
    // Read the left side.
    var left = ctx.ReadDigits();

    // Read the optional right side.
    var right = "";
    if (ctx.TryConsume('.'))
    {
        right = ctx.ReadDigits();
    }

    // If both left and right are empty or the last character isn't f.
    if ((left.Length == 0 && right.Length == 0) || !ctx.TryConsume('f'))
    {
        return null;
    }

    // It's safe now.
    if (left.Length == 0)
    {
        left = "0";
    }
    if (right.Length == 0)
    {
        right = "0";
    }

    var digits = left + "." + right;
    return new FloatToken(
        double.Parse(digits, CultureInfo.InvariantCulture), ctx.Metadata()
    );
}

```

Рис. 2.20 – Код обробника FloatCallback

Тепер перейдемо до фази «**парсер**».

Всередині ParserPhase, так же само, як і з лексером, ми спочатку ініціюємо обробники (рис. 2.21).

```

public ParserPhase()
{
    // initialization
    StructuredInitialization = new StructuredInitializationCallback();

    Expression = SetupTerminals();

    // nary
    Expression = SetupUnary(Expression);
    Expression = SetupBinaryTernary(Expression);

    // scope
    Scope = new ScopeCallback();

    // type sub handlers
    RefTypeEntry = SetupRefTypes();
    PointerTypeEntry = new PointerTypeEntryCallback();
    ArrayTypeEntry = new ArrayTypeEntryCallback();
    VariantTypeEntry = SetupVariantTypes();

    // type
    DefinedType = SetupDefinedType();
    Type = SetupType();

    // partial variable declaration (only will be used in for loop)
    PartialVariableDeclaration = new PartialVariableCallback();

    // anonymous
    Anonymous = SetupAnonymous();

    // sigh
    Declaration = SetupDeclarations();

    // statements
    Statement = SetupStatements();
}

```

Рис. 2.21 – Конструктор класу ParserPhase

Метод Precedence використовується для встановлення пріоритету операцій у виразах. Він приймає обробник і список обробників з пріоритетами, визначаючи порядок їх застосування.

Потім створюється новий диспетчер (рис. 2.22), який управляє різними обробниками, які працюватимуть за принципом пріоритету.

```

private static IParserHandler<Node> Precedence(IParserHandler<Node> handler, List<IOperatorWithPrecedence<No
{
    var dispatcher = new ParserDispatcher<Node>();
}

```

Рис. 2.22 – Обривок з коду методу Precedence

Кожен обробник зі списку callbacks реєструється у диспетчері, а також налаштовується на співпрацю з цим диспетчером для обробки токенів на одному рівні пріоритету (рис. 2.23).

```
foreach (var callback in callbacks)
{
    callback.SetSame(dispatcher);
    dispatcher.Add(callback);
}
```

Рис. 2.23 – Реєстрування обробників

Метод повертає складений обробник, який інтегрує базовий обробник handler з усіма обробниками в dispatcher, дозволяючи забезпечити правильне виконання операцій з урахуванням пріоритету (рис. 2.24). Цей метод використовується, наприклад, для ініціалізації бінарних і тернарних операторів (рис. 2.25).

```
return new StackedPrecedenceCallback(handler, dispatcher);
```

Рис. 2.24 – Повернення складеного обробника

```
private static IParserHandler<Node> SetupBinaryTernary(IParserHandler<Node> unary)
{
    var result = unary;

    // multiplicative
    result = Precedence(
        result,
        new()
        {
            new MultiplicationCallback(result),
            new DivisionCallback(result),
            new ModuloCallback(result),
        }
    );

    // arithmetic
    result = Precedence(
        result,
        new()
        {
            new AdditionCallback(result),
            new SubtractionCallback(result),
        }
    );

    // bitwise shift
```

Рис. 2.25 – Обривок з коду методу SetupBinaryTernary

Префіксні операції включають оператори, які ставляться перед своїми операндами, такі як ++x або --x. Реалізація включає методи для стекових та нормальних префіксних обробників (рис. 2.26).

```
private static IParserHandler<Node> PrefixCommon(
    IParserHandler<Node> handler,
    List<IOperatorWithPrecedence<Node>> callbacks, bool isStacked)
{
    var dispatcher = new ParserDispatcher<Node>();

    foreach (var callback in callbacks)
    {
        if (isStacked)
        {
            callback.SetSame(dispatcher);
        }

        dispatcher.Add(callback);
    }

    dispatcher.Add(handler);

    return dispatcher;
}
```

Рис. 2.26 – Код методу PrefixCommon

Цей базовий метод реєструє обробники префіксних операторів, які або можуть, або не можуть бути складеними (тобто не можуть бути повторюваними безпосередньо після себе без операндів між ними).

Постфіксні операції включають оператори, які ставляться після своїх операндів, наприклад, x++ або x--.

PostfixSingle – створення нового обробника для постфіксних операцій, що не мають бути повторюваними безпосередньо після себе (рис. 2.27).

```
private static IParserHandler<Node> PostfixSingle(
    IParserHandler<Node> handler,
    List<IParserHandler<Node>> callbacks)
{
    return new SinglePrecedenceCallback(handler, PostfixDispatcher(callbacks));
}
```

Рис. 2.27 – Код методу PostfixSingle

PostfixStacked – налаштування обробників для повторюваних постфіксних операторів, дозволяючи повторення тієї ж операції безпосередньо після себе, наприклад, виклик функцій або доступ до елементів масиву (рис. 2.28).

```
private static IParserHandler<Node> PostfixStacked(
    IParserHandler<Node> handler,
    List<IParserHandler<Node>> callbacks)
{
    return new StackedPrecedenceCallback(handler, PostfixDispatcher(callbacks));
}
```

Рис. 2.28 – Код методу PostfixStacked

Тепер, розглянемо обробники.

Обробник «IfCallback»:

1. Його конструктор встановлює обгортку для передперевірки токена IF, що використовується для визначення чи поточний токен у потоці парсинга є початком інструкції if (рис. 2.29).

```
public IfCallback()
{
    Wrapper(new PeekStartTokenWrapper<IfNode>(Tokens.Type.IF));
}
```

Рис. 2.29 – Конструктор класу IfCallback

2. Метод Parse обробляє умову if, опціональний блок else і elif. Він читає умову в середині круглих дужок, обов'язковий блок виконання для if і вибіркового блоку для else або elif (рис. 2.30).

```
protected IfNode Parse(ParserContext ctx)
{
    var token = ctx.Consume();

    var condition = ctx.Required(ctx.EnclosedExpression());
    var onTrue = ctx.Body();
    Node? onFalse = null;

    if (ctx.TryPeek(Tokens.Type.ELIF))
    {
        onFalse = Parse(ctx);
    }
    else if (ctx.TryConsume(Tokens.Type.ELSE))
    {
        onFalse = ctx.Body();
    }

    return Node.Tokens(new IfNode(condition, onTrue, onFalse), token);
}
```

Рис. 2.30 – Код методу Parse

Обробник «PartialVariableCallback»:

1. Його конструктор ініціює обгортку, яка перевіряє наявність токенів VAR або STATIC на початку оголошення змінної (рис. 2.31).

```
public PartialVariableCallback()
{
    Wrapper(new PeekStartTokenWrapper<VariableNode>(Tokens.Type.VAR, Tokens.Type.STATIC));
}
```

Рис. 2.31 – Конструктор класу PartialVariableCallback

2. Метод Main розпарсовує оголошення змінних, яке може бути з типом чи без, а також з початковим значенням або без нього (рис. 2.32).

```
protected override VariableNode? Main(ParserContext ctx, object? arg)
{
    var token = ctx.Consume();
    var name = ctx.Required(ctx.Name());

    TypeNode? type = null;

    if (ctx.TryConsume(Tokens.Type.COLON))
    {
        type = ctx.Required(ctx.Type());
    }

    // optional value
    Node? definition = null;
    if (ctx.TryConsume(Tokens.Type.EQUAL))
    {
        definition = ctx.Required(ctx.Expression());
    }

    // all done
    if (token.Type == Tokens.Type.VAR)
    {
        return new RuntimeVariableNode(name, type, definition);
    }
    else
    {
        return new StaticVariableNode(name, type, definition);
    }
}
```

Рис. 2.32 – Код обробника PartialVariableCallback

Обробник «BinaryCallback»: цей обробник виконує парсинг бінарної операції, зчитуючи лівий операнд, оператор і правий операнд. Використовується для визначення типу та виконання операції (рис. 2.33).

```
protected override DoubleChildNode? Main(ParserContext ctx, object? arg)
{
    var value = (Node?)arg ?? Lower?.Execute(ctx);

    if (value == null || !ctx.TryConsumeMultiple(ExpectedMultipleTypes))
    {
        return null;
    }

    var right = Dispatch(ctx);

    if (right == null)
    {
        return null;
    }

    return Create(value, right);
}
```

Рис. 2.33 – Код обробника BinaryCallback

Обробник «DefinedTypeCallback»: цей обробник обробляє визначені типи, включаючи посилання, покажчики, масиви і варіанти. Він збирає всі ці елементи в колекцію, яка формує структуру визначеного типу (рис. 2.34).

```
protected override DefinedTypeNode? Main(ParserContext ctx, object? arg)
{
    // Entries
    var entries = new NodeCollection<TypeEntryNode>
    {
        // Arc/Brw
        ctx.RefTypeEntry()
    };

    // Pointers
    while (entries.Add(ctx.PointerTypeEntry()));

    // Expect the variant here
    VariantTypeEntryNode? variant = ctx.VariantTypeEntry();

    if (variant == null)
    {
        return null;
    }

    // The arrays
    while (true)
    {
        var entry = ctx.ArrayTypeEntry();

        if (entry == null)
        {
            break;
        }

        entry.IsConstant = variant.IsConstant;
        entries.Add(entry);
    }

    // The variant is always at the end
    entries.Add(variant);

    return new DefinedTypeNode(entries);
}
```

Рис. 2.34 – Код обробника DefinedTypeCallback

Тепер розглянемо останню фазу «генератор».

Генератор, на відміну від лексера та парсера, не використовує композицію. Через це, він має дуже просту структуру та архітектуру.

Розглянемо принципи роботи ключових методів.

Метод «FunctionRegister»:

1. Перевірка на глобальний контекст: якщо функція не оголошена в глобальному контексті, викидається виняток. Це запобігає визначенню функцій у непідтримуваних областях (рис. 2.35).

```
protected string FunctionRegister(FunctionNode function)
{
    if (Parent != null)
    {
        throw new CompileException(
            "Functions can only be declared globally",
            function.Metadata
        );
    }
}
```

Рис. 2.35 – Початок функції FunctionRegister

2. Перевірка на підтримку перевантажень операторів: перевіряється, чи є функція іменованою. Якщо ні, то генерується виняток, оскільки перевантаження операторів не підтримується (рис. 2.36).

```
if (function is not NamedFunctionNode namedFunction)
{
    throw new CompileException(
        "Operator overloads are currently unsupported",
        function.Metadata
    );
}
```

Рис. 2.36 – Перевірка всередині FunctionRegister

3. Реєстрація функції: функція додається в словник, якщо її ще не існує. Також перевіряється, що типи повернення співпадають у всіх перевантаженнях функції (рис. 2.37).

```

if (!Functions.ContainsKey(namedFunction.Name.Value))
{
    Functions[namedFunction.Name.Value] = new(namedFunction.ReturnType);
}

if (!CompareTypes(namedFunction.ReturnType, Functions[namedFunction.Name.Value].ReturnType))
{
    throw new CompileException(
        "Overloaded function has a different type",
        namedFunction.Metadata
    );
}

var index = Functions[namedFunction.Name.Value].Variants.Count;

```

Рис. 2.37 – Код реєстрації функції

4. Реєстрація перевантаження: викликається метод `FunctionRegisterOverload` для перевірки унікальності сигнатур перевантаження (рис. 2.38).

```

if (!FunctionRegisterOverload(Functions[namedFunction.Name.Value], namedFunction.Parameters))
{
    throw new CompileException(
        "Function overload with the same signature already exists",
        function.Metadata
    );
}

```

Рис. 2.38 – Код реєстрації перевантаження функції

5. Створення локального контексту для тіла функції: ініціалізує новий контекст генератора для подальшої обробки тіла функції (рис. 2.39).

```

var ctx = new GeneratorContext(Packages, CurrentNamespace, this, function.ReturnType, GlobalDepth + 1);

```

Рис. 2.39 – Створення контексту

6. Генерація коду C: генерується вихідний код на C для функції, включаючи оголошення та визначення параметрів (рис. 2.40).

```

// C code
var sb = new StringBuilder();

sb.Append($"{EncodeType(function.ReturnType)} _func");

if (CurrentNamespace.Count > 0)
{
    sb.Append($"_{string.Join('_', CurrentNamespace)}");
}

sb.Append($"_{namedFunction.Name.Value}_{index}");

for (int i = 0; i < namedFunction.Parameters.Count; i++)
{
    sb.Append(EncodeTypedName(namedFunction.Parameters[i].Name, namedFunction.Parameters[i].Type));

    if (i < namedFunction.Parameters.Count - 1)
    {
        sb.Append(", ");
    }
}

sb.AppendLine("");

// inner statement(s)
if (function.Body != null)
{
    sb.AppendLine(ctx.Statement(function.Body));
}

return sb.ToString();

```

Рис. 2.40 – Код генерації С-коду

Метод «VariableRegister»:

1. Перевірка на редефініцію змінної: якщо змінна вже існує у контексті, викидається виняток, щоб запобігти її повторному оголошенню (рис. 2.41).

```

protected string VariableRegister(VariableNode variable)
{
    VariableRegistration registration;

    if (Variables.ContainsKey(variable.Name.Value))
    {
        throw new CompileException("Variable redefinition", variable.Metadata);
    }
}

```

Рис. 2.41 – Код перевірки змінної

2. Обробка змінної без явного типу (рис. 2.42): якщо тип змінної не вказано, змінна повинна мати значення, з якого можна визначити її тип.

```
// no type at all
if (variable.Type == null)
{
    if (variable.Value == null)
    {
        throw new CompileException(
            "Variable with no type must have a value",
            variable.Metadata
        );
    }

    //Variables[variable.Name.Value] = DetermineType(variable.Value);
    registration = new(DetermineType(variable.Value));
}
```

Рис. 2.42 – Код для змінної без явного типу

3. Обробка авто-константної змінної: якщо тип змінної вказано як авто-константний, перевіряється наявність значення. Якщо тип першої частини є масивом, то всі наступні типи також вважаються константами (рис. 2.43).

```

// auto const
else if (variable.Type is AutoConstTypeNode)
{
    if (variable.Value == null)
    {
        throw new CompileException(
            "Auto constant variable must have a value",
            variable.Metadata
        );
    }

    var newType = DetermineType(variable.Value);
    newType.Entries[0].IsConstant = true;

    // If the first type is an array, everything that follows must
    // be constant, too
    if (newType.Entries[0] is ArrayTypeEntryNode)
    {
        foreach (var entry in newType.Entries)
        {
            entry.IsConstant = true;
        }
    }

    //Variables[variable.Name.Value] = newType;
    registration = new(newType);
}

```

Рис. 2.43 – Код для автоматичних констант

4. Обробка змінної з вказаним типом: перевіряється сумісність типу змінної та типу її значення, якщо значення задано (рис. 2.44).

```

// defined type
else if (variable.Type is DefinedTypeNode definedType)
{
    if (variable.Value != null && !TypesCompatible(definedType, DetermineType(variable.Value)))
    {
        throw new CompileException(
            "Variable's value has invalid type",
            variable.Value.Metadata
        );
    }

    registration = new(definedType);
}

```

Рис. 2.44 – Код для змінних з типом

5. Генерація коду C: реєструє змінну у контексті та формує рядок коду для її оголошення у вихідному файлі. Якщо змінна має початкове значення, воно також включається у код (рис. 2.45).

```
// C code
var sb = new StringBuilder();

sb.Append(EncodeTypedName(variable.Name, registration.Type));

if (variable.Value != null)
{
    sb.Append($" = {Expression(variable.Value)}");
}

sb.Append(";");

return sb.ToString();
```

Рис. 2.45 – Код для генерації коду змінної в C

Сам код для компіляції вихідного C-коду здійснюється в головному файлі наступним чином:

1. Перевірка аргументів: цей блок перевіряє, чи користувач вказав достатню кількість аргументів. Якщо ні, виводиться інструкція з використання та програма завершується з кодом помилки 1 (рис. 2.46).

```
// verify the arguments
if (args.Length < 2)
{
    Console.WriteLine("Usage: cvc [input] [output] (--dump)");
    return 1;
}
```

Рис. 2.46 – Код перевірки аргументів

2. Отримання шляху до кореневої папки: витягується шлях до директорії вхідного файлу. Якщо це не вдається, виводиться повідомлення про помилку, і програма завершується з кодом 1 (рис. 2.47).

```
// get the root path
var rootPath = Path.GetDirectoryName(args[0]);

if (rootPath == null)
{
    Console.WriteLine($"Invalid root file: {args[0]}");
    return 1;
}
```

Рис. 2.47 – Код отримання кореневої папки

3. Парсинг файлів: ініціалізується словник для зберігання контекстів різних пакетів. Викликається функція Parse, яка обробляє вхідний файл та усі його залежності (рис. 2.48).

```
// parse every file
var packagesCtx = new Dictionary<string, GeneratorContext>();

try
{
    var code = Parse(Path.GetFileName(args[0]), rootPath, packagesCtx, true);
}
```

Рис. 2.48 – Код парсингу

4. Виведення згенерованого коду: якщо вказано аргумент --dump, виводиться згенерований код на консоль (рис. 2.49).

```
if (args.Length > 2 && args[2] == "--dump")
{
    Console.WriteLine("Generated code:");
    Console.WriteLine(code);
}
```

Рис. 2.49 – Необов’язковий код виведення

5. Запис до тимчасового файлу та компіляція: згенерований код записується у тимчасовий файл, який потім компілюється за допомогою компілятора gsc. Налаштовується процес для виконання компіляції, приховуючи вікно та перенаправляючи стандартний вивід та помилки (рис. 2.50).

```

// write to the temporary file
var fileName = Path.GetTempFileName();
File.WriteAllText(fileName, code);

// compile using GCC
var info = new ProcessStartInfo()
{
    FileName = "gcc",
    Arguments = $"-x c {fileName} -o {args[1]} -w -O2",
    RedirectStandardOutput = true,
    RedirectStandardError = true,
    UseShellExecute = false,
    CreateNoWindow = true
};

// Start the process
using var process = Process.Start(info);

if (process == null)
{
    Console.WriteLine("Unable to launch the C compiler");
    return 1;
}

process.WaitForExit(); // Wait for the process to complete

```

Рис. 2.50 – Код для компіляції С-коду

6. Читання та обробка результатів компіляції: програма очікує завершення процесу компіляції, читає вивід та помилки. Якщо виникли помилки компіляції, вони виводяться на консоль (рис. 2.51).

```

// Read the outputs
string output = process.StandardOutput.ReadToEnd();
string errors = process.StandardError.ReadToEnd();

// Check for errors
if (!string.IsNullOrEmpty(errors))
{
    Console.WriteLine("The C compiler was unable to compile the generated code:");
    Console.WriteLine(errors);
}

```

Рис. 2.51 – Обробка результатів С-компілятора

7. Обробка винятків: обробляються винятки, які можуть виникнути під час роботи програми. Для винятків, пов'язаних із компіляцією, використовується функція `PrintError` для детального відображення інформації про помилки (рис. 2.52).

```

catch (CompileException e)
{
    PrintError(e.Metadata, e.Message);
}

catch (Exception e)
{
    Console.WriteLine($"Exception: {e.Message}");
}

```

Рис. 2.52 – Код обробки помилок

Висновки до другого розділу

В процесі розробки компілятора для спеціалізованої мови програмування було здійснено аналіз усіх аспектів, які включають зчитування та обробку коду, його синтаксичний аналіз, та генерацію вихідного коду на мові C. Архітектура системи передбачає використання окремих фаз, кожна з яких відповідає за свою частину процесу компіляції, починаючи від лексичного аналізу до генерації коду.

Окрім розробки компілятора, було враховано можливість залучення зовнішніх пакетів, що дозволяє розширювати функціональність і забезпечувати модульність системи. Вся структура проекту підтримує чітку організацію коду і масштабованість, дозволяючи додавати нові модулі або модифікувати існуючі без втручання в основний код.

За допомогою системи викидання виключень та їх обробки, компілятор здатен ідентифікувати та відображати помилки, які можуть виникати під час компіляції коду, надаючи користувачу детальну інформацію про проблемні ділянки у коді.

На завершення, весь процес проектування та розробки компілятора орієнтований на забезпечення ефективної та надійної підтримки програмування, що включає ретельну перевірку коду, його оптимізацію та підготовку до виконання у вигляді виконуваного файлу, готового до роботи на цільовій платформі.

РОЗДІЛ 3. ІНТЕРФЕЙС ТА ПОРЯДОК РОБОТИ З КОМПІЛЯТОРОМ

3.1 Опис інтерфейсу користувача

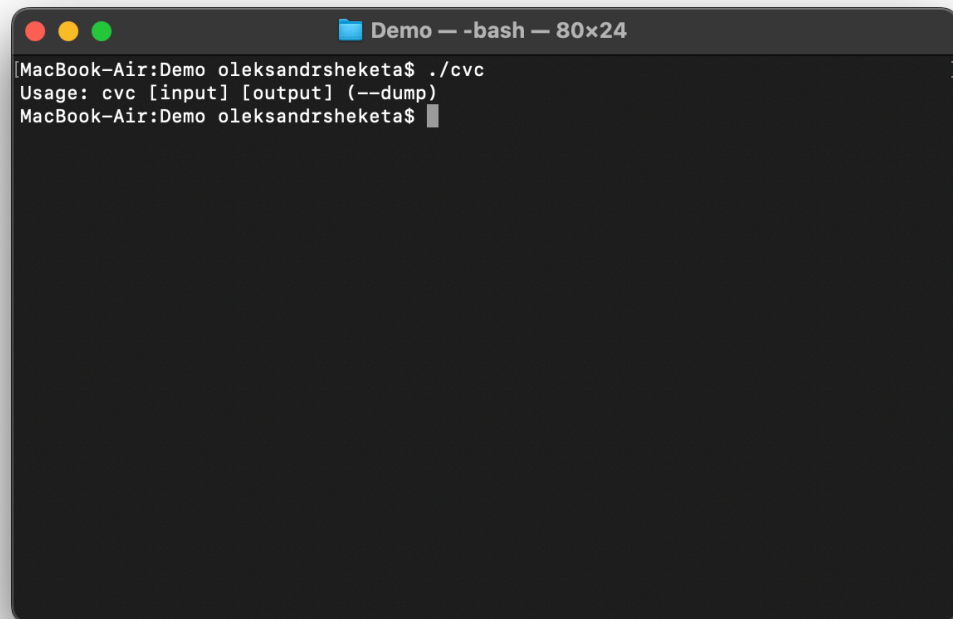
Інтерфейс користувача нашої системи — це важливий елемент, що забезпечує взаємодію між користувачем та системою через консольний додаток. Цей додаток дозволяє вводити команди у командному рядку, що особливо зручно для розробників, які цінують швидкий доступ до функцій компілятора без графічного інтерфейсу. Користувачам доступні наступні основні команди:

1. Вказівка шляху до вхідного файлу: користувач вказує шлях до файлу для компіляції як перший параметр у командному рядку.
2. Вказівка шляху для збереження результату: місце зберігання скомпільованого файлу визначається останнім аргументом командного рядка.
3. Опція `--dump`: вивід згенерованого C коду в консоль для перевірки перед компіляцією.

Інтерфейс максимізовано спрощує процес компіляції, надаючи ефективне управління через командний рядок. Це дозволяє інтегрувати компілятор у складні системи або використовувати його в автоматизованих скриптах.

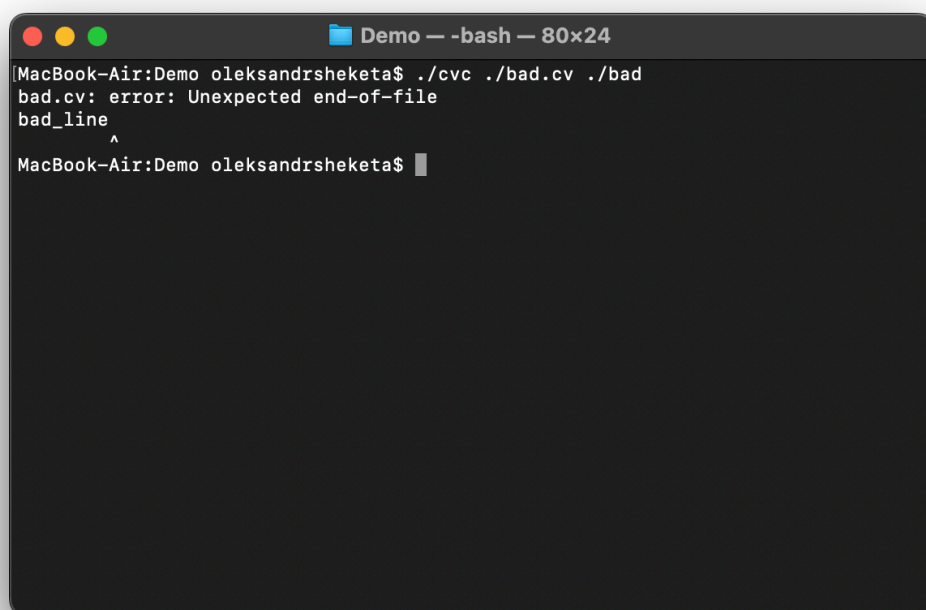
Додаткові випадки використання:

- якщо компілятор запущений без аргументів, користувачу буде виведено інструкцію на екрані (рис. 3.1);
- при виникненні помилки під час компіляції, на екран виводиться ім'я файлу та місце помилки у коді (рис. 3.2);
- якщо компіляція пройшла без помилок, процес завершується успішно (рис. 3.3), а на диску з'являється виконуваний файл (рис. 3.4).



```
MacBook-Air:Demo oleksandrsheketa$ ./cvc
Usage: cvc [input] [output] (--dump)
MacBook-Air:Demo oleksandrsheketa$
```

Рис. 3.1 – Запуск компілятора без аргументів



```
MacBook-Air:Demo oleksandrsheketa$ ./cvc ./bad.cv ./bad
bad.cv: error: Unexpected end-of-file
bad_line
      ^
MacBook-Air:Demo oleksandrsheketa$
```

Рис. 3.2 – Помилка під час компіляції

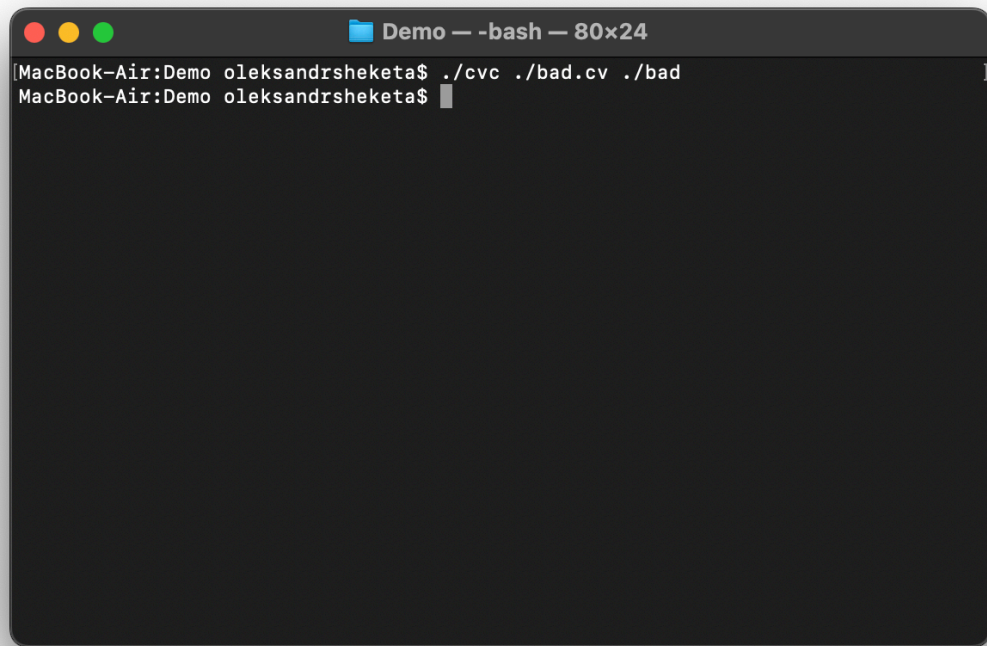


Рис. 3.3 – Успішна компіляція після виправлення помилок

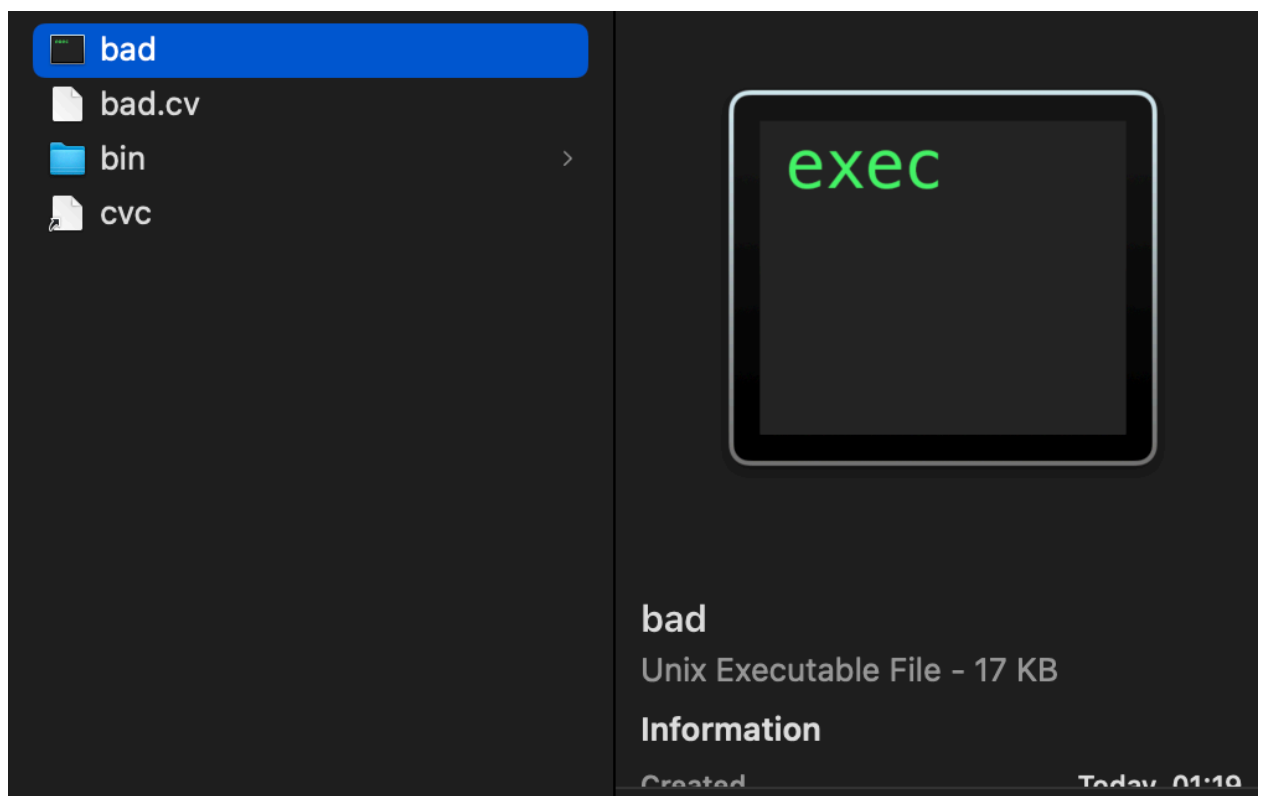


Рис. 3.4 – Вихідний виконуваний файл

3.2 Порядок роботи системи

Процес роботи системи починається з ініціації користувачем компіляції через консольний інтерфейс. Після отримання команд від користувача, система розпочинає послідовність дій, кожна з яких має важливе значення для досягнення кінцевого результату:

1. Читання коду: Система зчитує вихідний файл з кодом. Цей крок включає в себе перевірку на наявність файлу за вказаним шляхом та його доступність для читання. Якщо файл не знайдено або відсутні права на його читання, система повідомляє про помилку.
2. Лексичний аналіз: Зчитаний текст програми обробляється лексером, який розбиває текст на лексеми—мінімальні значущі елементи мови програмування. Це включає в себе ідентифікацію ключових слів, ідентифікаторів, літералів, операторів тощо.
3. Синтаксичний аналіз: Лексеми, отримані на попередньому кроці, передаються парсеру, який аналізує їх у контексті граматичних правил мови програмування для побудови абстрактного синтаксичного дерева (AST). AST представляє структурну ієрархію програми, що дозволяє на наступному етапі ефективно генерувати код.
4. Генерація коду: На основі AST система генерує вихідний код на мові C. Цей код містить усі необхідні конструкції для виконання програми, і його структура оптимізована для подальшої компіляції.
5. Компіляція: Згенерований код на мові C компілюється у виконуваний файл. Цей крок включає в себе перевірку на наявність синтаксичних та семантичних помилок у згенерованому коді, а також його оптимізацію для підвищення продуктивності виконання.
6. Виконання: Завершальний крок дозволяє користувачу запустити згенерований виконуваний файл. Це дає можливість перевірити, чи

програма виконує задані користувачем функції і чи є результат відповідним очікуванням.

Такий детальний підхід у порядку роботи системи забезпечує високу точність компіляції та ефективність процесу, знижуючи ризик виникнення помилок і збільшуючи довіру користувачів до системи.

3.3 Налаштування роботи системи

Для коректної роботи системи компіляції нашого компілятора, користувачі мають спочатку встановити компілятор GCC на свій комп'ютер і додати його до системного шляху (PATH), щоб можна було викликати GCC з будь-якого місця у командному рядку. Нижче наведено кроки для налаштування системи під операційні системи Windows та macOS.

Для користувачів Windows:

1. Завантаження та інсталяція:

- а) Відвідайте офіційну сторінку MinGW (Minimalist GNU for Windows) (рис. 3.5).

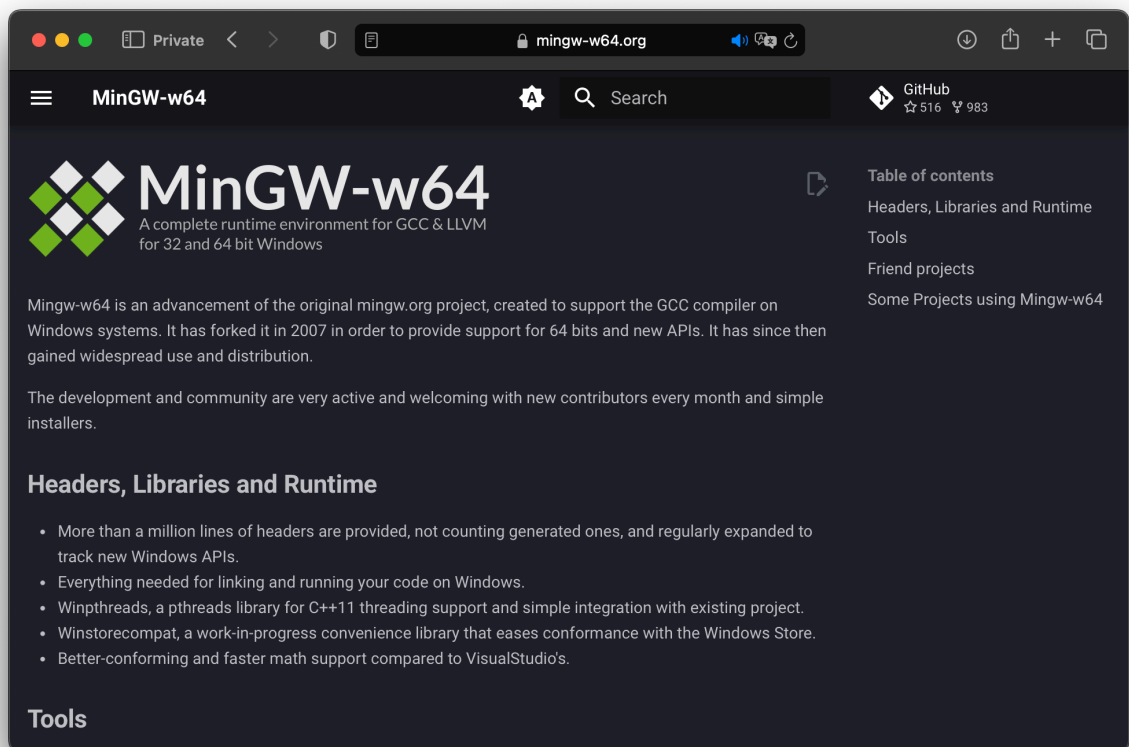


Рис. 3.5 – Офіційний сайт MinGW

- б) Завантажте і встановіть повний пакет, який включає GCC (рис. 3.6).

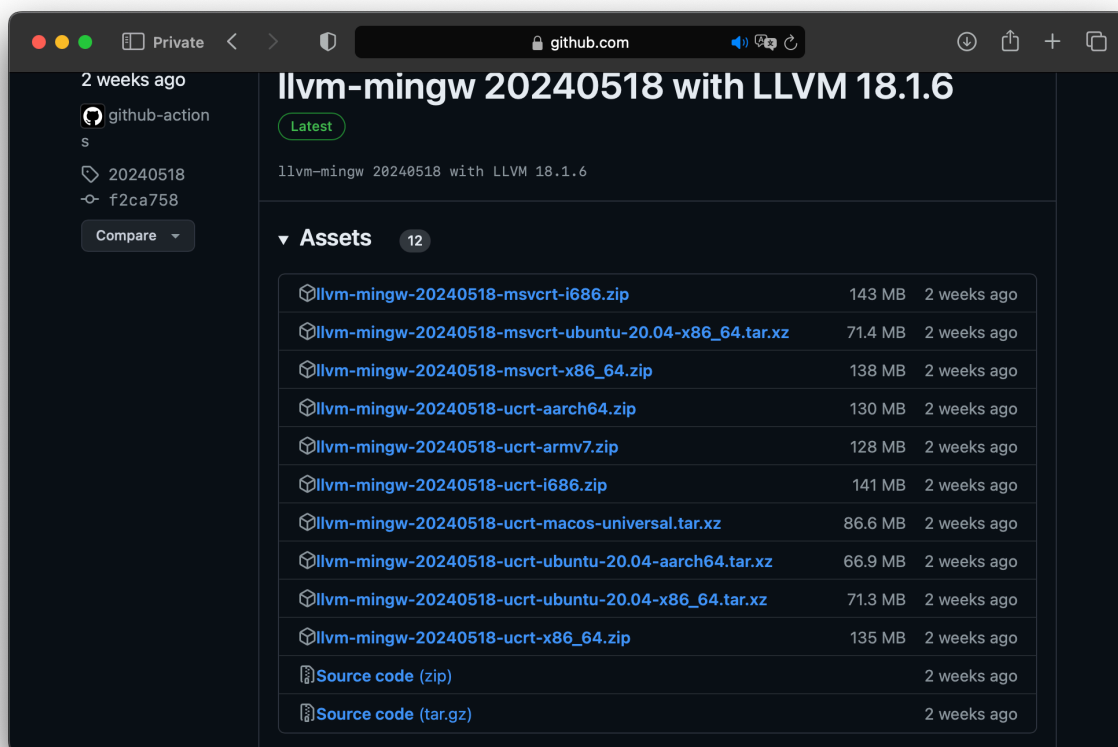


Рис. 3.6 – Сторінка для завантаження MinGW

2. Додавання до PATH:

- Відкрийте 'Системні властивості' (Правий клік на 'Цей комп'ютер' -> 'Властивості' -> 'Додаткові параметри системи') (рис. 3.7).

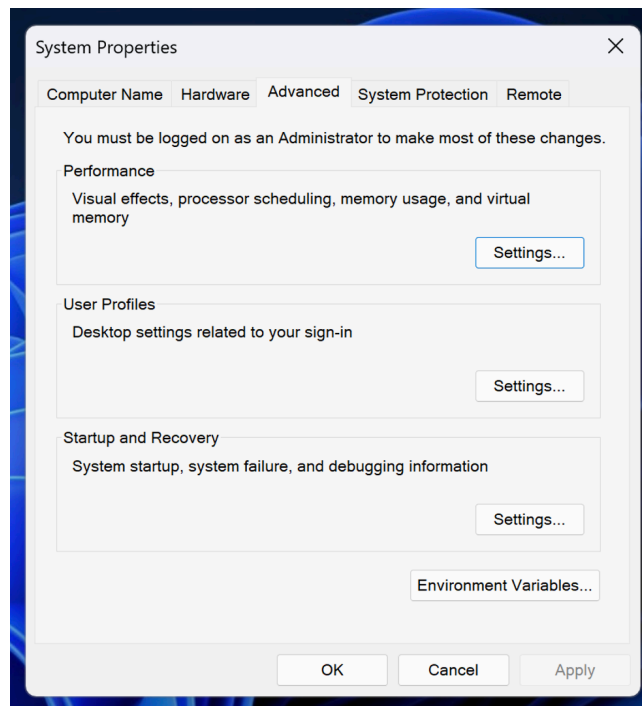


Рис. 3.7 – Вікно «Додаткові параметри системи»

б) Натисніть 'Змінити...' у розділі 'Змінні середовища' (рис. 3.8).

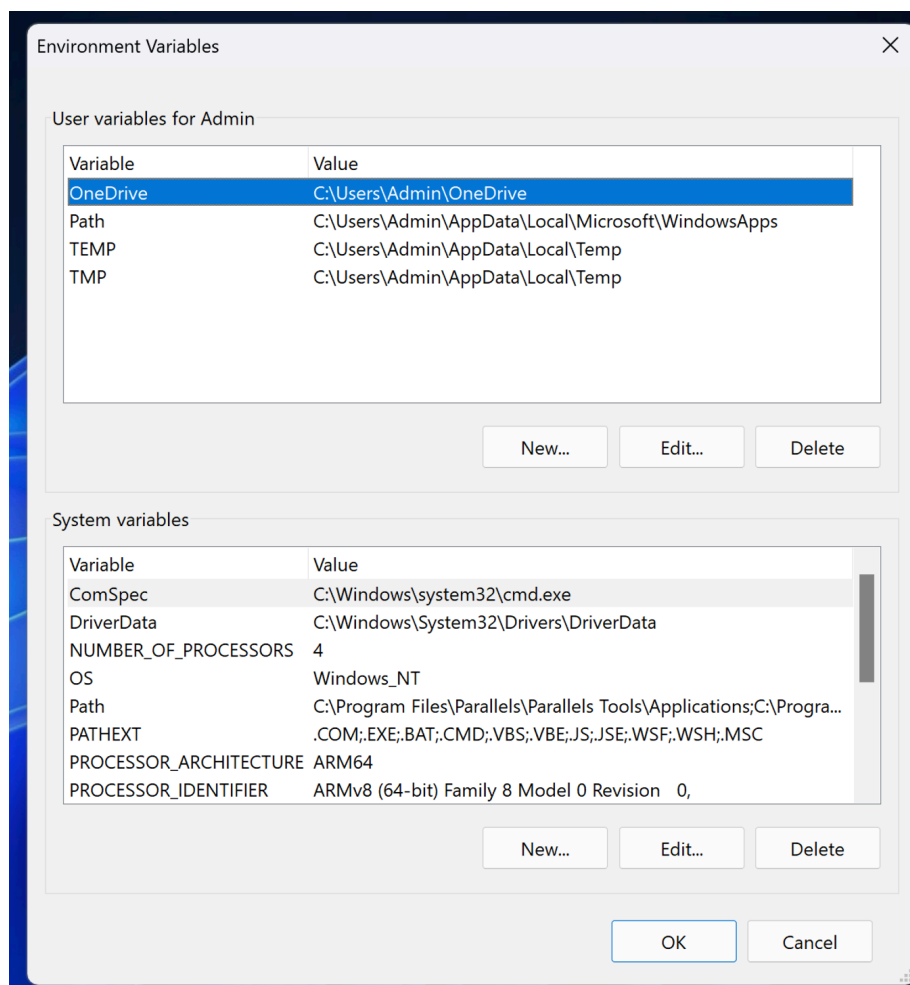


Рис. 3.8 – Вікно «Змінні середовища»

- с) У списку системних змінних знайдіть і виберіть 'Path', потім натисніть 'Редагувати...' (рис. 3.9).

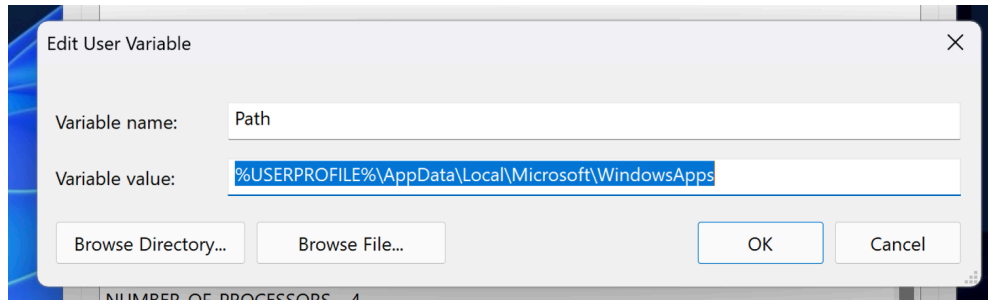


Рис. 3.9 – Редагування PATH

- д) Додайте шлях до каталогу bin вашої інсталяції MinGW або TDM-GCC.

Для користувачів macOS:

1. Використання Homebrew:

- а) Відкрийте термінал і встановіть Homebrew (рис. 3.10).

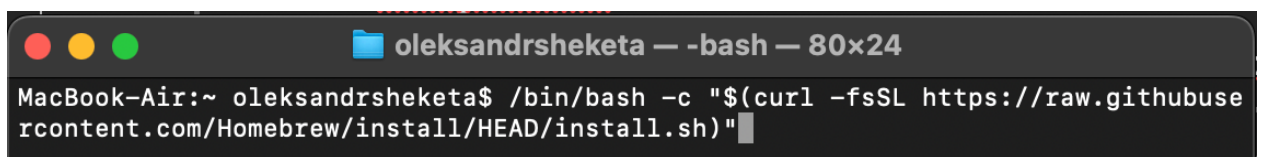


Рис. 3.10 – Встановлення Homebrew

- б) Після інсталяції Homebrew, встановіть GCC (рис. 3.11):

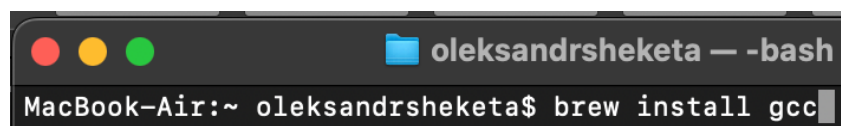


Рис. 3.11 – Встановлення GCC

2. Перевірка інсталяції:

- а) У терміналі введіть `gcc --version` для перевірки, чи було GCC успішно встановлено і правильно налаштовано в вашому PATH (рис. 3.12).

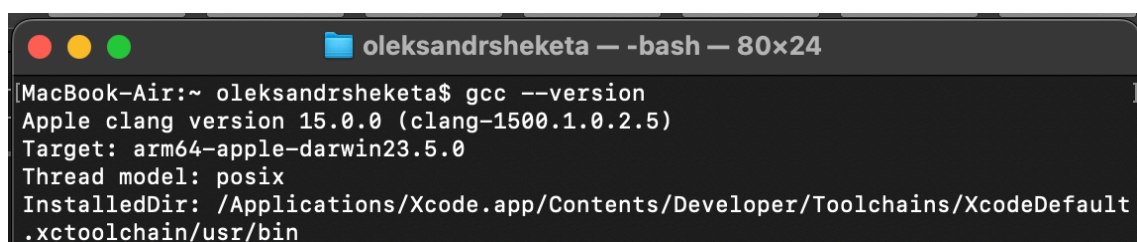


Рис. 3.12 – Перевірка роботи GCC

Ці налаштування забезпечують, що компілятор зможе використовувати GCC для компіляції згенерованих C-файлів.

3.4 Тестування системи

Тестування компілятора є критично важливим етапом розробки, оскільки воно гарантує, що всі компоненти системи працюють належним чином і що кінцевий продукт задовольняє всі вимоги та очікування користувачів. Важливість цього процесу полягає в забезпеченні стабільності, ефективності та надійності розробленого програмного забезпечення.

Наступні ключові аспекти тестування охоплюють широкий спектр перевірок:

Тестування синтаксису: Основою цього етапу є перевірка здатності системи коректно ідентифікувати та обробляти синтаксичні конструкції мови програмування. Тестування включає створення різноманітних тестових файлів, що містять специфічні для мови конструкції, такі як цикли, умовні вирази, функції. Завдяки цьому перевіряється здатність системи аналізувати та виконувати синтаксичний розбір цих структур, а також точність відповідності виробленого аналізу до вимог синтаксису.

Тестування генерації коду: На цьому етапі проводиться перевірка точності коду, згенерованого з абстрактного синтаксичного дерева (AST). Тестування включає аналіз відповідності згенерованого коду стандартам мови програмування C, наявності синтаксичних помилок, а також оцінку ефективності та оптимізації згенерованого коду. Це забезпечує переконання в тому, що програма не тільки коректно виконує свої функції, але й робить це максимально ефективно.

Виконання програм: Запуск тестових програм використовується для оцінки їх функціональності та відповідності заданим критеріям. Цей процес включає перевірку наявності всіх необхідних функцій та відсутності помилок або відхилень у роботі програми. Виконання програм дозволяє оцінити

реальну поведінку програми в контрольованих умовах, забезпечуючи повне розуміння її діяльності в реальному світі.

Застосування цього всебічного підходу до тестування забезпечує не лише високу якість кінцевого продукту, але й виявлення потенційних проблем на ранніх етапах розробки. Це істотно зменшує ризики та витрати, пов'язані з подальшою експлуатацією системи, забезпечуючи ефективне виправлення будь-яких недоліків до моменту реального застосування програми. Такий підхід є фундаментальним для забезпечення надійності, стабільності та коректної роботи компілятора в умовах його фактичного використання.

Висновки до третього розділу

В цьому розділі було детально описано реалізовану систему компілятора для власної програмної мови, починаючи від інтерфейсу користувача та закінчуючи процесом тестування. Інтерфейс користувача, зосереджений на консольному вводі, забезпечує необхідні параметри для взаємодії з системою, що дозволяє користувачам легко керувати процесом компіляції. Було продемонстровано порядок роботи системи, що включає лексичний аналіз, синтаксичний аналіз, генерацію коду та його компіляцію. Такий підхід сприяє кращому розумінню взаємодії компонентів системи і підкреслює її модульність.

Налаштування системи було мінімізовано до вибору параметрів запуску через командний рядок, що спрощує її використання та унеможливорює помилки, пов'язані з конфігурацією. Тестування системи вручну, яке було описано, включає ряд сценаріїв, що забезпечують перевірку всіх аспектів системи, від лексичного аналізу до виконання згенерованих програм. Це підтверджує коректність роботи компілятора і забезпечує високий рівень довіри до результатів його роботи.

Завдяки цьому детальному аналізу, можна зрозуміти структуру системи, її ключові можливості та взаємодії між її компонентами. Такий підхід не тільки висвітлює функціональність системи для кінцевих користувачів, але й виокремлює аспекти, які можуть бути поліпшені в майбутніх версіях.

ВИСНОВКИ

В рамках виконання даної кваліфікаційної роботи бакалавра було розроблено компілятор для нової C-подібної програмної мови, яка інтегрує сучасні функції та забезпечує часткову семантичну сумісність із мовою C. Основною метою проекту було створення надійного та ефективного інструменту для компіляції текстових документів з кодом виконуваних файлів. Протягом роботи було здійснено аналіз існуючих рішень, визначення ключових вимог до компілятора, проектування його архітектури, вибір підходящих технологій і методів розробки, а також реалізація і тестування створеного продукту.

Результатом роботи став компілятор, здатний ефективно перетворювати програмний код у виконуваних файли, забезпечуючи при цьому високу швидкість обробки і точність виконання. Окрему увагу приділено механізмам обробки помилок, що дозволяють користувачам легко ідентифікувати та виправляти помилки в програмному коді. Додатково, компілятор підтримує ряд спеціальних функцій, які розширюють можливості стандартного C, що робить нову мову більш гнучкою та адаптивною до сучасних вимог програмування.

Значна увага в процесі розробки була приділена інтуїтивно зрозумілому інтерфейсу, що забезпечує легкість у використанні компілятора, надійність в роботі з програмним кодом та ефективність обробки даних. Такий підхід сприяє більшій продуктивності розробників і зменшенню часу на виправлення помилок.

Аналіз сучасних мов програмування, таких як GCC для мови C та Rust Compiler, дозволив виявити ключові тренди та напрямки для покращення. Це, у свою чергу, надало можливість інтегрувати в компілятор передові технології обробки та оптимізації коду, що в значній мірі збільшує його конкурентоспроможність на ринку програмних продуктів.

Завершення цього проекту не тільки забезпечило важливий внесок у розвиток інструментів програмування, але й сприяло подальшому професійному зростанню автора як кваліфікованого інженера програмного забезпечення. Результати роботи мають значний потенціал для використання в академічних та комерційних цілях, а також можуть бути використані для подальших досліджень і розвитку в області компіляторних технологій.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. List<T> Class [Електронний ресурс] – Microsoft – Режим доступу до ресурсу: <https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1?view=net-5.0>
2. Composite [Електронний ресурс] – Refactoring.Guru – Режим доступу до ресурсу: <https://refactoring.guru/design-patterns/composite>
3. References and Borrowing [Електронний ресурс] – Rust Programming Language – Режим доступу до ресурсу: <https://doc.rust-lang.org/1.8.0/book/references-and-borrowing.html>
4. Lifetimes [Електронний ресурс] – Rust Programming Language – Режим доступу до ресурсу: <https://doc.rust-lang.org/1.8.0/book/lifetimes.html>
5. Target-typed new expressions [Електронний ресурс] – Microsoft – Режим доступу до ресурсу: <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/proposals/csharp-9.0/target-typed-new>
6. Null coalescing assignment [Електронний ресурс] – Microsoft – Режим доступу до ресурсу: <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/proposals/csharp-8.0/null-coalescing-assignment>
7. Lexical Analysis [Електронний ресурс] – Stanford University – Режим доступу до ресурсу: <https://web.stanford.edu/class/cs143/lectures/lecture03.pdf>
8. Error Handling, Syntax-Directed Translation, Recursive Descent Parsing [Електронний ресурс] – Stanford University – Режим доступу до ресурсу: <https://web.stanford.edu/class/cs143/lectures/lecture06.pdf>
9. Semantic Analysis in Compiler Design [Електронний ресурс] – GeeksforGeeks – Режим доступу до ресурсу: <https://www.geeksforgeeks.org/semantic-analysis-in-compiler-design/>
10. Recursive-Descent Parsing [Електронний ресурс] – University of Rochester – Режим доступу до ресурсу:

https://www.cs.rochester.edu/u/nelson/courses/csc_173/grammars/parsing.html

- 11.const (C++) [Электронный ресурс] – Microsoft – Режим доступа до ресурсу: <https://learn.microsoft.com/ru-ru/cpp/cpp/const-cpp?view=msvc-170>
- 12.C Operator Precedence [Электронный ресурс] – cppreference – Режим доступа до ресурсу: https://en.cppreference.com/w/c/language/operator_precedence
- 13.Options Controlling the Kind of Output [Электронный ресурс] – GCC – Режим доступа до ресурсу: <https://gcc.gnu.org/onlinedocs/gcc/Overall-Options.html>
- 14.C# language documentation [Электронный ресурс] – Microsoft – Режим доступа до ресурсу: <https://learn.microsoft.com/en-us/dotnet/csharp/>
- 15.Constraints on type parameters (C# Programming Guide) [Электронный ресурс] – Microsoft – Режим доступа до ресурсу: <https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/generics/constraints-on-type-parameters>
- 16.where (generic type constraint) [Электронный ресурс] – Microsoft – Режим доступа до ресурсу: <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/where-generic-type-constraint>
- 17.Implicit conversions [Электронный ресурс] – cppreference – Режим доступа до ресурсу: <https://en.cppreference.com/w/c/language/conversion>
- 18.operator overloading [Электронный ресурс] – cppreference – Режим доступа до ресурсу: <https://en.cppreference.com/w/cpp/language/operators>
- 19.Arc [Электронный ресурс] – Rust – Режим доступа до ресурсу: <https://doc.rust-lang.org/rust-by-example/std/arc.html>
- 20.On source-to-source compilers [Электронный ресурс] – ResearchGate – Режим доступа до ресурсу:

https://www.researchgate.net/publication/303348270_On_source-to-source_compilers

- 21.C# Dev Kit for Visual Studio Code [Электронный ресурс] – Microsoft – Режим доступа до ресурсу: <https://learn.microsoft.com/en-us/visualstudio/subscriptions/vs-c-sharp-dev-kit>
- 22.Covariance and contravariance in generics [Электронный ресурс] – Microsoft – Режим доступа до ресурсу: <https://learn.microsoft.com/en-us/dotnet/standard/generics/covariance-and-contravariance>
- 23.Phases of a Compiler [Электронный ресурс] – GeeksforGeeks – Режим доступа до ресурсу: <https://www.geeksforgeeks.org/phases-of-a-compiler/>
- 24.Structure of a compiler [Электронный ресурс] – Computer Science - Western University – Режим доступа до ресурсу: <https://www.csd.uwo.ca/~mmorenom/CS447/Lectures/Introduction.html/no-de10.html>
- 25.Debug C# code and inspect data [Электронный ресурс] – Microsoft – Режим доступа до ресурсу: <https://learn.microsoft.com/en-us/visualstudio/get-started/csharp/tutorial-debugger?view=vs-2022>

ДОДАТКИ

					ІПЗ.КР.Б-121-24-ПЗ	Арк.
						81

Реалізація класу Dispatcher.

Лістинг коду:

```
using System.Collections;

namespace Cover.Handlers
{
    public class Dispatcher<T, U> : IHandler<T, U>, IEnumerable<IHandler<T, U>>
    {
        private readonly List<IHandler<T, U>> handlersList;
        private readonly HashSet<IHandler<T, U>> handlersSet;
        public int Count => handlersList.Count;
        public IHandler<T, U> this[int index]
        {
            get => handlersList[index];
            set => handlersList[index] = value;
        }

        public Dispatcher()
        {
            handlersList = new();
            handlersSet = new();
        }

        virtual public bool Add(IHandler<T, U> handler)
        {
            if (handlersSet.Add(handler))
            {
                handlersList.Add(handler);
                return true;
            }

            return false;
        }

        virtual public bool Remove(IHandler<T, U> handler)
        {
            if (handlersSet.Remove(handler))
            {
                handlersList.Remove(handler);
                return true;
            }

            return false;
        }

        virtual public IHandler<T, U> Add<W>() where W : IHandler<T, U>, new()
        {
            var handler = new W();

            handlersList.Add(handler);
        }
    }
}
```

					ІПЗ.КР.Б-121-24-ІЗ	Арк.
						82

```

        handlersSet.Add(handler);

        return handler;
    }

    virtual public bool Contains(IHandler<T, U> handler)
    {
        return handlersSet.Contains(handler);
    }

    virtual public W? Find<W>() where W : IHandler<T, U>
    {
        var list = FindAll<W>();

        if (list.Count > 0)
        {
            return list[0];
        }

        return default;
    }

    virtual public bool Remove<W>() where W : IHandler<T, U>
    {
        var list = FindAll<W>();

        if (list.Count > 0)
        {
            handlersSet.Remove(list[0]);
            handlersList.Remove(list[0]);

            return true;
        }

        return false;
    }

    virtual public List<W> FindAll<W>() where W : IHandler<T, U>
    {
        return handlersList.OfType<W>().ToList();
    }

    virtual public bool RemoveAll<W>() where W : IHandler<T, U>
    {
        var list = FindAll<W>();

        if (list.Count == 0)
        {
            return false;
        }

        foreach (var handler in list)
        {

```

```

        handlersSet.Remove(handler);
        handlersList.Remove(handler);
    }

    return true;
}

virtual public void Insert(int index, IHandler<T, U> handler)
{
    if (!handlersSet.Contains(handler))
    {
        handlersList.Insert(index, handler);
        handlersSet.Add(handler);
    }
}

virtual public void Insert<W>(int index) where W : IHandler<T, U>, new()
{
    var handler = new W();

    handlersList.Insert(index, handler);
    handlersSet.Add(handler);
}

public U? Execute(T ctx, object? arg = default)
{
    U? result = default;

    foreach (var handler in this)
    {
        result = handler.Execute(ctx, arg);

        if (result != null)
        {
            break;
        }
    }

    return result;
}

public IEnumerator<IHandler<T, U>> GetEnumerator()
{
    return handlersList.GetEnumerator();
}

IEnumerator IEnumerable.GetEnumerator()
{
    return GetEnumerator();
}
}
}

```

Додаток Б

Реалізація класу Scanner.

Лістинг коду:

```
using System.Text;
using Cover.Exceptions;

namespace Cover.Contexts
{
    public abstract class Scanner<T, U> : IScannerContext<T, U>
    {
        private readonly Stack stack;
        protected int Current
        {
            get => stack.Current;
            set => stack.Current = value;
        }

        public Scanner()
        {
            stack = new();
        }

        virtual public void Backup()
        {
            stack.CopyRight();
        }

        virtual public void Commit()
        {
            stack.CopyLeft();
        }

        virtual public void Restore()
        {
            stack.ShiftLeft();
        }

        private T ConsumeCommon(bool allowEndOfFile)
        {
            if (EndOfFile())
            {
                if (!allowEndOfFile)
                {
                    throw new CompileException("Unexpected end-of-file", Metadata());
                }

                return Fallback();
            }

            var result = Read();
            Current++;

            return result;
        }
    }
}
```

					ІПЗ.КР.Б-121-24-ІЗ	Арк.
						85

```

    }

    private T PeekCommon(bool allowEndOfFile)
    {
        Backup();
        var result = ConsumeCommon(allowEndOfFile);
        Restore();
        return result;
    }

    private bool TryPeekCommon(bool allowEndOfFile, U expected)
    {
        return Compare(PeekCommon(allowEndOfFile), expected);
    }

    private bool TryConsumeCommon(bool allowEndOfFile, U expected)
    {
        if (TryPeekCommon(allowEndOfFile, expected))
        {
            ConsumeCommon(allowEndOfFile);
            return true;
        }

        return false;
    }

    private bool TryConsumeMultipleCommon(bool allowEndOfFile, params U[]
expected)
    {
        Backup();
        foreach (var item in expected)
        {
            if (!TryConsumeCommon(allowEndOfFile, item))
            {
                Restore();
                return false;
            }
        }

        Commit();
        return true;
    }

    private bool TryPeekMultipleCommon(bool allowEndOfFile, params U[] expected)
    {
        Backup();
        foreach (var item in expected)
        {
            if (!TryConsumeCommon(allowEndOfFile, item))
            {
                Restore();
                return false;
            }
        }
    }

```

```

    }

    Restore();
    return true;
}

public bool TryPeek(U expected)
{
    return TryPeekCommon(true, expected);
}

public bool TryConsume(U expected)
{
    return TryConsumeCommon(true, expected);
}

public bool TryConsumeMultiple(params U[] expected)
{
    return TryConsumeMultipleCommon(true, expected);
}

public bool TryPeekMultiple(params U[] expected)
{
    return TryPeekMultipleCommon(true, expected);
}

public bool TryPeekMultipleRequired(params U[] expected)
{
    return TryPeekMultipleCommon(false, expected);
}

public bool TryConsumeRequired(U expected)
{
    return TryConsumeCommon(false, expected);
}

public bool TryPeekRequired(U expected)
{
    return TryPeekCommon(false, expected);
}

public bool TryConsumeMultipleRequired(params U[] expected)
{
    return TryConsumeMultipleCommon(false, expected);
}

public void ConsumeMultiple(params U[] expected)
{
    var status = true;
    Backup();

    foreach (var item in expected)
    {

```

```

        if (!TryConsume(item))
        {
            Restore();
            status = false;
            break;
        }
    }

    if (status)
    {
        Commit();
        return;
    }

    var sb = new StringBuilder();
    foreach (var item in expected)
    {
        sb.Append(item?.ToString());
    }

    throw new CompileException(
        string.Format("Expected something else (" + sb.ToString() + ")"),
        Metadata()
    );
}

public T Consume()
{
    return ConsumeCommon(true);
}

public T Peek()
{
    return PeekCommon(true);
}

public T ConsumeRequired()
{
    return ConsumeCommon(false);
}

public T PeekRequired()
{
    return PeekCommon(false);
}

public W Consume<W>(U expected) where W : T
{
    if (!TryPeekCommon(false, expected))
    {
        throw new CompileException(
            string.Format("Expected something else (" + expected?.ToString() +
"
```

"))),

					ИПЗ.КР.Б-121-24-ИЗ	Арк.
						88

```

        Metadata()
    );
}

var result = ConsumeCommon(false) ??
    throw new InternalException("Consume returned NULL. This is bad.");

try
{
    return (W)result;
}
catch (InvalidCastException)
{
    throw new InternalException("Typecast inside consume failed.");
}
}

public T Consume(U expected)
{
    return Consume<T>(expected);
}

public abstract T Read();

public abstract bool Compare(T a, U b);

public abstract bool EndOfFile();

public abstract T Fallback();

public abstract Metadata Metadata();
}
}

```

Додаток В

Реалізація класу ParserPhase.

Лістинг коду:

```

using Cover.Nodes;
using Cover.Tokens;
using Cover.Nodes.Statements;
using Cover.Parser.Handlers;
using Cover.Parser.Handlers.Operators;
using Cover.Parser.Handlers.Statements.Declarations.Modifiers;
using Cover.Parser.Handlers.Terminals.Factors;
using Cover.Parser.Handlers.Statements;
using Cover.Parser.Handlers.Statements.Declarations.Variable;
using Cover.Parser.Handlers.Statements.Declarations.Anonymous;
using Cover.Parser.Handlers.Statements.Flows;
using Cover.Parser.Handlers.Terminals.Literals;
using Cover.Parser.Handlers.Statements.Declarations.Structured;
using Cover.Parser.Handlers.Statements.Declarations.Aggregate;
using Cover.Parser.Handlers.Statements.Declarations;

```

					ІПЗ.КР.Б-121-24-ІЗ	Арк.
						89

```

using Cover.Parser.Handlers.Operators.Unary.Prefix;
using Cover.Parser.Handlers.Operators.Unary.Postfix;
using Cover.Parser.Handlers.Operators.Binary;
using Cover.Parser.Handlers.Operators.Ternary;
using Cover.Parser.Handlers.Operators.Precedence;

using AsyncCallback =
Cover.Parser.Handlers.Statements.Declarations.Modifiers.AsyncCallback;
using Cover.Nodes.Types;
using Cover.Parser.Handlers.Types.Defined;
using Cover.Parser.Handlers.Types;
using Cover.Nodes.Types.Defined.Entries.References;
using Cover.Nodes.Types.Defined.Entries.Dimensions;
using Cover.Nodes.Types.Defined.Entries.Variants;
using Cover.Parser.Handlers.Types.Defined.Entries.Dimensions;
using Cover.Parser.Handlers.Types.Defined.Entries.Variants;
using Cover.Parser.Handlers.Types.Defined.Entries.Variants.Symboled;
using Cover.Parser.Handlers.Types.Defined.Entries.References;
using Cover.Nodes.Types.Defined;
using Cover.Parser.Handlers.Terminals.Factors.Initialization;
using Cover.Nodes.Terminals.Factors.Initialization.Structured;

namespace Cover.Parser
{
    public class ParserPhase
    {
        readonly public IParserHandler<Node> Statement;
        readonly public IParserHandler<Node> Expression;
        readonly public IParserHandler<Node> PartialVariableDeclaration;
        readonly public IParserHandler<Node> Anonymous;
        readonly public IParserHandler<ScopeNode> Scope;

        readonly public IParserHandler<TypeNode> Type;
        readonly public IParserHandler<DefinedTypeNode> DefinedType;

        readonly public IParserHandler<Node> Declaration;
        readonly public IParserHandler<StructuredInitializationNode>
StructuredInitialization;

        readonly public IParserHandler<RefTypeEntryNode> RefTypeEntry;
        readonly public IParserHandler<PointerTypeEntryNode> PointerTypeEntry;
        readonly public IParserHandler<ArrayTypeEntryNode> ArrayTypeEntry;
        readonly public IParserHandler<VariantTypeEntryNode> VariantTypeEntry;

        public ParserPhase()
        {
            // initialization
            StructuredInitialization = new StructuredInitializationCallback();

            Expression = SetupTerminals();

            // nary
            Expression = SetupUnary(Expression);
        }
    }
}

```

					ИПЗ.КР.Б-121-24-ПЗ	Арк.
						90

```

        Expression = SetupBinaryTernary(Expression);

        // scope
        Scope = new ScopeCallback();

        // type sub handlers
        RefTypeEntry = SetupRefTypes();
        PointerTypeEntry = new PointerTypeEntryCallback();
        ArrayTypeEntry = new ArrayTypeEntryCallback();
        VariantTypeEntry = SetupVariantTypes();

        // type
        DefinedType = SetupDefinedType();
        Type = SetupType();

        // partial variable declaration (only will be used in for loop)
        PartialVariableDeclaration = new PartialVariableCallback();

        // anonymous
        Anonymous = SetupAnonymous();

        // sigh
        Declaration = SetupDeclarations();

        // statements
        Statement = SetupStatements();
    }

    private static IParserHandler<VariantTypeEntryNode> SetupVariantTypes()
    {
        return new ParserDispatcher<VariantTypeEntryNode>()
        {
            new FunctionTypeEntryCallback(),
            new BitFieldTypeEntryCallback(),
            new PrimitiveTypeEntryCallback()
        };
    }

    private static IParserHandler<RefTypeEntryNode> SetupRefTypes()
    {
        return new ParserDispatcher<RefTypeEntryNode>()
        {
            new ArcTypeEntryCallback(),
            new BrwTypeEntryCallback()
        };
    }

    private static List<ModifierCallback> DefaultModifiers()
    {
        return new()
        {
            new AbstractCallback(),
            new InlineCallback(),

```

```

        new InternalCallback(),
        new OverrideCallback(),
        new PrivateCallback(),
        new ProtectedCallback(),
        new PublicCallback(),
        new StaticCallback(),
        new VirtualCallback(),
    };
}

private static IParserHandler<Node> SetupAnonymous()
{
    return new ParserDispatcher<Node>()
    {
        new AnonymousUnionCallback(),
        new AnonymousStructCallback(),
        new AnonymousFillerCallback()
    };
}

private static IParserHandler<Node> SetupStatements()
{
    return new ParserDispatcher<Node>()
    {
        new EmbeddedCallback(),
        new HoldCallback(),
        new ScopeCallback(),
        new TypedefCallback(),
        new WhileCallback(),
        new DoCallback(),
        new IfCallback(),
        new ForCallback(),
        new ReturnCallback(),
        new BreakCallback(),
        new ContinueCallback(),
        new ForEachCallback(),
        new SwitchCallback(),
        new TryCallback(),
        new ThrowCallback(),
        new UseCallback(),
    };
}

private IParserHandler<Node> SetupTerminals()
{
    return new ParserDispatcher<Node>()
    {
        SetupFactors(),
        SetupLiterals(),
    };
}

private IParserHandler<Node> SetupFactors()

```

```

{
    return new ParserDispatcher<Node>()
    {
        new AsyncCallback(
            new ParserDispatcher<Node>()
            {
                new LambdaCallback()
            }
        ),

        new LambdaCallback(),
        new ParenthesisCallback(),
        new NewCallback(),
        new UniformCallback(),
        new BorrowCallback(),
        new ArrayInitializationCallback(),
        StructuredInitialization,
    };
}

private static IParserHandler<Node> SetupLiterals()
{
    return new ParserDispatcher<Node>()
    {
        new CharCallback(),
        new FloatCallback(),
        new IntegerCallback(),
        new StringCallback(),
        new SymbolCallback()
    };
}

private static IParserHandler<Node> SetupDeclarations()
{
    // declarations
    var declarations = new ParserDispatcher<Node>()
    {
        WithModifiers(
            new()
            {
                new AsyncCallback(
                    new FunctionCallback()
                ),
                new FunctionCallback(),
                new ClassCallback(),
                new InterfaceCallback(),
                new StructCallback(),
                new EnumCallback(),
                new FlagCallback(),
                new UnionCallback(),
            },

            DefaultModifiers()
        )
    };
}

```

```

        ),

        WithModifiers(
            new()
            {
                new VariableCallback(),
            },
            new()
            {
                new PublicCallback(),
                new InternalCallback(),
            }
        )
    };

    declarations.Add(new TemplateCallback(declarations));

    return declarations;
}

private static ParserDispatcher<Node> WithModifiers(List<IParserHandler<Node>>
callbacks, List<ModifierCallback> modifiers)
{
    var dispatcher = new ParserDispatcher<Node>();

    foreach (var modifier in modifiers)
    {
        dispatcher.Add(modifier);
        modifier.Nested = dispatcher;
    }

    foreach (var callback in callbacks)
    {
        dispatcher.Add(callback);
    }

    return dispatcher;
}

private static IParserHandler<DefinedTypeNode> SetupDefinedType()
{
    return new DefinedTypeCallback();
}

private IParserHandler<TypeNode> SetupType()
{
    return new ParserDispatcher<TypeNode>
    {
        DefinedType,
        new AutoConstTypeCallback(),
    };
}

```

```

private static IParserHandler<Node> SetupBinaryTernary(IParserHandler<Node>
unary)
{
    var result = unary;

    // multiplicative
    result = Precedence(
        result,
        new()
        {
            new MultiplicationCallback(result),
            new DivisionCallback(result),
            new ModuloCallback(result),
        }
    );

    // arithmetic
    result = Precedence(
        result,
        new()
        {
            new AdditionCallback(result),
            new SubtractionCallback(result),
        }
    );

    // bitwise shift
    result = Precedence(
        result,
        new()
        {
            new BitwiseShiftLeftCallback(result),
            new BitwiseShiftRightCallback(result),
        }
    );

    // comparison (greater/less)
    result = Precedence(
        result,
        new()
        {
            new LessCallback(result),
            new LessEqualsCallback(result),
            new GreaterCallback(result),
            new GreaterEqualsCallback(result),
        }
    );

    // comparison (equals)
    result = Precedence(
        result,
        new()

```

					ИПЗ.КР.Б-121-24-ПЗ	Арк.
						95

```

        {
            new EqualsCallback(result),
            new NotEqualsCallback(result),
        }
    );

    // bitwise and
    result = Precedence(
        result,
        new()
        {
            new BitwiseAndCallback(result),
        }
    );

    // bitwise xor
    result = Precedence(
        result,
        new()
        {
            new BitwiseXorCallback(result),
        }
    );

    // bitwise or
    result = Precedence(
        result,
        new()
        {
            new BitwiseOrCallback(result),
        }
    );

    // and
    result = Precedence(
        result,
        new()
        {
            new AndCallback(result),
        }
    );

    // or
    result = Precedence(
        result,
        new()
        {
            new OrCallback(result),
        }
    );

    // ternary conditional
    result = Precedence(

```

```

        result,
        new()
    {
        new ConditionalCallback(result),
    }
    );

    // assignment
    result = Precedence(
        result,
        new()
    {
        new AdditionAssignmentCallback(result),
        new AssignmentCallback(result),
        new BitwiseAndAssignmentCallback(result),
        new BitwiseNotAssignmentCallback(result),
        new BitwiseOrAssignmentCallback(result),
        new BitwiseShiftLeftAssignmentCallback(result),
        new BitwiseShiftRightAssignmentCallback(result),
        new BitwiseXorAssignmentCallback(result),
        new DivisionAssignmentCallback(result),
        new ModuloAssignmentCallback(result),
        new MultiplicationAssignmentCallback(result),
        new SubtractionAssignmentCallback(result),
        new NotAssignmentCallback(result)
    }
    );

    return result;
}

private static IParserHandler<Node> Precedence(IParserHandler<Node> handler,
List<IOperatorWithPrecedence<Node>> callbacks)
{
    var dispatcher = new ParserDispatcher<Node>();

    foreach (var callback in callbacks)
    {
        callback.SetSame(dispatcher);
        dispatcher.Add(callback);
    }

    return new StackedPrecedenceCallback(handler, dispatcher);
}

private static IParserHandler<Node> PrefixSingle(
    IParserHandler<Node> handler,
    List<IOperatorWithPrecedence<Node>> callbacks)
{
    return PrefixCommon(handler, callbacks, false);
}

private static IParserHandler<Node> PrefixStackable(

```

```

        IParserHandler<Node> handler,
        List<IOperatorWithPrecedence<Node>> callbacks)
    {
        return PrefixCommon(handler, callbacks, true);
    }

    private static IParserHandler<Node> PrefixCommon(
        IParserHandler<Node> handler,
        List<IOperatorWithPrecedence<Node>> callbacks, bool isStacked)
    {
        var dispatcher = new ParserDispatcher<Node>();

        foreach (var callback in callbacks)
        {
            if (isStacked)
            {
                callback.SetSame(dispatcher);
            }

            dispatcher.Add(callback);
        }

        dispatcher.Add(handler);

        return dispatcher;
    }

    private static ParserDispatcher<Node>
    PostfixDispatcher(List<IParserHandler<Node>> callbacks)
    {
        var dispatcher = new ParserDispatcher<Node>();

        foreach (var callback in callbacks)
        {
            dispatcher.Add(callback);
        }

        return dispatcher;
    }

    private static IParserHandler<Node> PostfixSingle(
        IParserHandler<Node> handler,
        List<IParserHandler<Node>> callbacks)
    {
        return new SinglePrecedenceCallback(handler,
        PostfixDispatcher(callbacks));
    }

    private static IParserHandler<Node> PostfixStacked(
        IParserHandler<Node> handler,
        List<IParserHandler<Node>> callbacks)
    {

```

```

        return new StackedPrecedenceCallback(handler,
PostfixDispatcher(callbacks));
    }

    private static IParserHandler<Node> SetupUnary(IParserHandler<Node> terminal)
    {
        var result = terminal;
        // postfix

        // stackable postfix
        result = PostfixStacked(
            result,
            new()
            {
                new IndexCallback(),
                new MemberCallback(),
                new DereferenceMemberCallback(),
                new InvokeCallback()
            }
        );

        // increment | decrement
        result = PostfixSingle(
            result,
            new()
            {
                new PostfixIncrementCallback(),
                new PostfixDecrementCallback()
            }
        );

        // prefix
        // await
        result = PrefixSingle(
            result,
            new()
            {
                new AwaitCallback(result),
            }
        );

        // typecasts
        result = PrefixStackable(
            result,
            new()
            {
                new TypecastCallback(result),
            }
        );

        // address | dereference
        result = PrefixStackable(
            result,

```

```

        new()
        {
            new AddressCallback(result),
            new DereferenceCallback(result),
        }
    );

    // increment | decrement | negate
    result = PrefixSingle(
        result,
        new()
        {
            new PrefixIncrementCallback(result),
            new PrefixDecrementCallback(result),
            new NegateCallback(result),
            new NotCallback(result),
        }
    );

    return result;
}

public ProgramNode Parse(List<Token> tokens)
{
    var ctx = new ParserContext(this, tokens);
    var result = new ProgramNode();

    while (!ctx.EndOfFile())
    {
        result.Add(ctx.Required(ctx.Statement()));
    }

    return result;
}
}
}

```