НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

КАФЕДРА СИСТЕМНОГО ПРОГРАМУВАННЯ І
СПЕЦІАЛІЗОВАНИХ КОМП'ЮТЕРНИХ СИСТЕМ

«На правах рукопису»
УДК 004.4`4

«До захисту допущено»

Завідувач кафедри СПСКС

_____Віталій РОМАНКЕВИЧ
(підпис) (ім'я, прізвище)

"___"_____2021р.

Магістерська дисертація

на здобуття ступеня магістра

зі спеціальності 123 Комп'ютерна інженерія
освітньої програми «Системне програмування та спеціалізовані комп'ютерні системи»

на тему: «Способи трансляції мови Typescript у проміжну мову CIL платформи .NET»

Виконав : студент II курсу, групи КВ-93мн
Іваненко Антон Романович _____ _____
(прізвище, ім'я, по батькові) (підпис)

Науковий керівник доцент, доцент, к.т.н Марченко О.І. _____ _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Рецензент _____ _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант з нормоконтролю доцент, с.н.с.,к.т.н. Боярінова Ю.Є. _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2021 року

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет прикладної математики

Кафедра системного програмування і спеціалізованих комп'ютерних систем

Рівень вищої освіти – другий (магістерський)
за освітньо-науковою програмою
Спеціальність 123 Комп'ютерна інженерія (Спеціалізовані комп'ютерні системи)

ЗАТВЕРДЖУЮ
Завідувач кафедри СПСКС
_____ Віталій РОМАНКЕВИЧ
(підпис)
1 листопада 2019р.

ЗАВДАННЯ
на магістерську дисертацію студенту
Іваненку Антону Романовичу

1. Тема дисертації «Способи трансляції мови Typescript у проміжну мову CIL платформи .NET»,
науковий керівник дисертації доцент кафедри СП і СКС, к.т.н, Марченко О.І,
затверджені наказом по університету від « 26 » березня 2021 р. № 899 -С

2. Термін подання студентом дисертації 11 травня 2021 р.

3. Об'єкт дослідження: процес трансляції конструкцій мов програмування у проміжну мову CIL платформи .NET

4. Предмет дослідження: способи трансляції мови Typescript у проміжну мову CIL платформи .NET.

5. Перелік завдань, які потрібно розробити: аналіз існуючих способів трансляції мови Typescript у проміжну мову CIL платформи .NET; аналіз платформи будови платформи .NET; розробка способів трансляції змінних, типів даних, операцій, рядків та масивів, умовних конструкцій, циклічних конструкцій та функцій мови Typescript у проміжну мову CIL платформи .NET; створення тестового транслятора для підмножини мови Typescript; аналіз отриманих результатів.

6. Перелік ілюстративного матеріалу – презентація.

7. Перелік публікацій: тези до доповіді на XIII конференції молодих вчених ПМК-2020 року за темою «Аналіз способів трансляції мови Typescript у проміжну мову CIL платформи .NET»; стаття до наукового, фахового журналу «Комп'ютерно-інтегровані технології: освіта, наука, виробництво», випуск №42 за темою «Спосіб трансляції конкатенації рядкових виразів мови Typescript у проміжну мову CIL платформи .NET»;

8. Дата видачі завдання: 1 листопада 2019 року

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1.	Вивчення літератури за тематикою проекту	01.12.2019	
2.	Розроблення та узгодження технічного завдання	25.12.2019	
3.	Аналіз існуючих рішень	20.04.2020	
4.	Підготовка матеріалів першого розділу дисертації	20.05.2020	
5.	Підготовка матеріалів другого розділу дисертації	15.09.2020	
6.	Підготовка доповіді та тез на конференцію ПКМ-2020	10.11.2020	
7.	Дослідження способів трансляції та розробка тестового транслятора	20.02.2021	
8.	Підготовка статті до наукового фахового журналу	10.03.2021	
9.	Підготовка матеріалів третього розділу дисертації	20.03.2021	
10.	Підготовка матеріалів четвертого розділу дисертації	10.04.2021	
11.	Підготовка матеріалів п’ятого розділу дисертації	20.04.2021	
12.	Оформлення графічної частини дисертації	01.05.2021	
13.	Оформлення документації дипломного проекту	04.05.2021	
14.	Попередній розгляд магістерської дисертації на кафедрі	05.05.2021	

Студент _____ (підпис) _____ (ім‘я, прізвище)

Науковий керівник дисертації _____ (підпис) _____ (ім‘я, прізвище)

РЕФЕРАТ

Актуальність теми. У зв'язку з стрімким ростом популярності мови програмування Javascript все більше компаній і розробників починають використовувати її у своїх проектах. Але підтримка коду у великих програмах стає важкою через один з найбільших недоліків мови – динамічну типізацію. Для усунення цього недоліку компанія Microsoft у 2012 році запропонувала нову мову програмування під назвою Typescript. Розроблена мова є надмножиною JavaScript яка додає статичну типізацію і транслюється у Javascript відповідно. Завдяки набутої популярності мови програмування Javascript та появи її типізованої версії, виникла ідея поєднати Typescript і платформу .NET, а саме – створити транслятор мови програмування Typescript у проміжну мову CIL.

Об'єктом дослідження є процес трансляції конструкцій мов програмування у проміжну мову CIL платформи .NET.

Предметом дослідження є способи трансляції мови Typescript у проміжну мову CIL платформи .NET.

Мета роботи: прискорення процесу трансляції мови програмування Typescript у проміжну мову CIL платформи .NET; розробка більш швидких способів трансляції умовних конструкцій, циклів, функцій, рядків та масивів; порівняльний аналіз розроблених способів.

Наукова новизна полягає в наступному:

1. Вперше запропоновано спосіб безпосередньої трансляції умовних конструкцій мови Typescript у проміжну мову CIL платформи .NET, який дозволяє згенерувати код з більш високою швидкодією.

2. Вперше запропоновано спосіб безпосередньої трансляції циклічних конструкцій мови Typescript у проміжну мову CIL платформи .NET, який дозволяє згенерувати код з більш високою швидкодією.

3. Вперше запропоновано спосіб безпосередньої трансляції функцій рядків та масивів мови Typescript у проміжну мову CIL платформи .NET, який дозволяє згенерувати код з більш високою швидкодією.

4. Вперше виконано порівняльний аналіз розроблених способів із існуючим рішенням, який показав ефективність запропонованих способів.

Практична цінність отриманих в дисертації результатів полягає в тому, що запропоновані способи трансляції мови Typescript у проміжну мову CIL платформи .NET та розроблений тестовий транслятор підтверджують, що безпосередня трансляція можлива і ефективна. Ці результати є підставою для створення у майбутньому повноцінного транслятора мови Typescript у платформу .NET, що дозволить збільшити популярність обраних технологій, залучити більше розробників для використання платформи .NET у розробці серверної частини додатків, що використовують мову програмування Typescript чи Javascript для розробки клієнтської частини.

Апробація роботи. Основні положення і результати роботи були представлені та обговорювались на XIII науковій конференції магістрантів та аспірантів «Прикладна математика та комп'ютинг» ПМК-2020 (Київ, 18-20 листопада 2020 р.).

Публікації. Результати дисертації викладено в наукових працях, у тому числі:

- тези до доповіді на XIII конференції молодих вчених ПМК-2020 року за темою «Аналіз способів трансляції мови Typescript у проміжну мову CIL платформи .NET»;

- стаття до наукового, фахового журналу «Комп'ютерно-інтегровані технології: освіта, наука, виробництво» випуск №42 за темою «Спосіб трансляції конкатенації рядкових виразів мови Typescript у проміжну мову CIL платформи .NET»;

Структура та обсяг роботи. Магістерська дисертація складається з вступу, п'яти розділів та висновків.

У вступі подано загальну характеристику роботи, зроблено оцінку сучасного стану проблеми, обґрунтовано актуальність напрямку досліджень, сформульовано мету і задачі досліджень, показано наукову новизну отриманих

результатів і практичну цінність роботи, наведено відомості про апробацію результатів і їхнє впровадження.

У першому розділі розглянуто існуючі методи трансляції мови Javascript у проміжну мову CIL платформи .NET, а також проведений аналіз, який дає змогу визначити основні переваги та недоліки цих підходів.

У другому розділі розглянуто платформу .NET, її розвиток та архітектуру. Проаналізовано основні інструкції мови CIL та наведено список інструкцій у вигляді таблички тих, що будуть використані у запропонованих способах.

У третьому розділі описано запропоновані способи трансляції масивів, рядків, циклічних та умовних конструкцій, функцій, змінних та операцій над ними.

У четвертому розділі розглянуто створення тестового транслятора на основі запропонованих способів, а також його інтеграцію у утиліту dotnet.

У п'ятому розділі проаналізовано швидкодію згенерованого коду тестовим транслятором у порівнянні з результатом роботи транслятора JScript на прикладі певних задач.

У висновках стисло представлені результати проведеного дослідження.

Дисертація представлена на 82 аркушах, містить 70 ілюстрацій, 5 таблиць, 4 додатки та посилання на список використаних літературних джерел.

Ключові слова: транслятор, компілятор, генератор коду, Typescript, CIL, CLR, .NET

ABSTRACT

Actuality of theme. Due to the rapid growth of the Javascript programming language, more and more companies and developers are starting to use it in their projects. But code support in a large program becomes important because of one of the biggest drawbacks of the language - dynamic typing. To take advantage of this shortcoming, Microsoft in 2012 proposed a new programming language called Typescript. The developed language is a JavaScript superset that adds static typing and is passed according to Javascript. Due to the growing popularity of language programming programs Javascript and appeared its standard version, which created the idea of combining Typescript and .NET platform, namely - to create a translator of the programming language Typescript in CIL.

The object of research is a process of translation of constructions of programming languages into the CIL of .NET platform.

The subject of the research are ways to translate the Typescript language into the CIL of the .NET platform.

Purpose: speeding up the process of translating the Typescript programming language into the CIL of the .NET platform; development of faster ways of translation of conditional constructions, cycles, functions, strings and arrays; comparative analysis of the developed methods.

The scientific novelty is as follows:

1. For the first time, a method of direct translation of conditional constructions of the Typescript language into the intermediate CIL language of the .NET platform has been proposed, which allows to generate faster code.

2. For the first time, a method of directly translating cyclic constructs of the Typescript language into the intermediate CIL language of the .NET platform has been proposed, which allows to generate faster code.

3. For the first time, a method of direct translation of Typescript string and array functions into the intermediate CIL language of the .NET platform has been proposed, which allows to generate faster code.

4. For the first time a comparative analysis of the developed methods with the existing solution was performed, which showed the effectiveness of the proposed methods.

The practical value of the results obtained in the dissertation is that the proposed methods of translation of the Typescript language into the intermediate language CIL of the .NET platform and the developed test translator confirm that live translation is possible and effective. These results are the basis for the future creation of a full-fledged Typescript language translator in the .NET platform, which will increase the popularity of selected technologies, attract more developers to use the .NET platform in the development of server applications using Typescript or Javascript to develop the client part.

Approbation of work. The main provisions and results of the work were presented and discussed at the XIII scientific conference of undergraduates and graduate students "Applied Mathematics and Computing" PMK-2020 (Kyiv, November 18-20, 2020).

Publications. The results of the dissertation are presented in scientific works, including:

- abstracts for the report at the XIII Conference of Young Scientists PMK-2020 on "Analysis of Typescript translation methods into Common Intermediate Language of .NET platform";

- Article to the scientific, professional journal "Computer-integrated technologies: education, science, production" on the topic "Translation the concatenation of Typescript string expressions into Common Intermediate Language of .NET platform";

Structure and scope of work. The master's dissertation consists of an introduction, five sections and conclusions.

The introduction presents a general description of the work, assesses the current state of the problem, substantiates the relevance of research, formulates the purpose and objectives of research, shows the scientific novelty of the results and the practical value of the work, provides information on testing results and their implementation.

The second section presents the implementation of lexical, syntactic and semantic analysis of the selected subset of the TypeScript language. The methods of generating CIL instructions, which are implemented in the code generator, are investigated.

The third section describes the proposed methods of translation of arrays, strings, cyclic and conditional structures, functions, variables and operations on them.

The fourth section discusses the creation of a test translator based on the proposed methods, as well as its integration into the dotnet utility.

The fifth section analyzes the speed of the generated code by the test translator in comparison with the result of the JScript translator on the example of certain tasks.

The conclusions summarize the results of the study.

The dissertation is presented on 82 sheets, contains 70 illustrations, 5 tables, 4 attachments and references to the list of used literature sources.

Keywords: translator, compiler, code generator, Typescript, CIL, CLR, .NET.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ

API (Application Programming Interface) – прикладний програмний інтерфейс

CFG (Control flow graph) – граф потоку керування

CLI (Common Language Infrastructure) – специфікація загальномовної інфраструктури.

CLR (Common language runtime) – загальномовне виконуюче середовище

CPS (Common Project System) – загальна система проектів

DLL (Dynamic link-library) – динамічно приєднувана бібліотека

IDE (Integrated Development Environment) – інтегрована середа розробки

REPL (Read-eval-print loop) – просте інтерактивне середовище програмування

VES (Virtual Execution System) – віртуальна система виконання

БНФ (Бекуса-Наура форма) – спосіб запису правил контекстно-вільної граматики

Дерево розбору – внутрішня форма подання вхідної програми, що містить структурні одиниці мови, а також зв’язки між ними

ВСТУП

Сьогодні з кожним днем з’являється все більше технологічних засобів для розробки програмного забезпечення. Створюються нові мови програмування, спроектовані для спеціальних завдань, розроблюються віртуальні машини, для виконання інтерпретації чи just-in-time компіляції для відповідних мов, з’являються нові IDE та засоби профілювання.

До однієї з таких технологій і відноситься досить популярна платформа .NET. Свій початок вона бере у далекому 2000 році. Компанія Microsoft запропонувала програмну технологію для розробки десктопних програм та веб-застосунків. Це було продовження ідей та принципів, покладених в технологію Java. Головним недоліком цієї платформи була підтримка операційних систем сімейства Windows, але у 2016 році відбувся перший реліз крос-платформної версії з відкритим вихідним кодом під назвою .NET Core, що значно збільшило її популярність серед розробників веб-додатків.

Для створення веб-застосунків зазвичай ще потрібна мова програмування Javascript для виконання браузером певних сценарії, що відповідають за логіку клієнтської частини. Простота та швидкість розробки на цій мові програмування швидко зробили її однією з найбільш використовуваних у світі. Але через головний недолік – відсутність типізації, створення великих проектів стало дуже важким, бо налагодження програм э дуже складним процесом. Для усунення цього недоліку компанія Microsoft розробила у 2012 році мову Typescript, яка є надмножиною мови програмування Javascript, додає статичну типізацію та транслюється у Javascript відповідно.

Через популярність обох технологій – платформи .NET та мови програмування Typescript, виникла ідея їх поєднати, а саме – створити транслятор мови програмування Typescript у проміжну мову CIL платформи .NET.

У даній роботі представлено та досліджено способи трансляції умовних конструкцій, циклів, функцій, рядків та масивів, а також проведено порівняльний аналіз коду, отриманого за розробленими способами, з кодом, згенерованим існуючим аналогом – транслятором JScript.

1. АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ ТА ОБГРУНТУВАННЯ ДОСЛІДЖЕННЯ СПОСОБІВ ТРАНСЛЯЦІЇ МОВИ TYPESCRIPT У ПРОМІЖНУ МОВУ CIL ПЛАТФОРМИ .NET

1.1. Характеристика існуючих способів трансляції мови Typescript у проміжну мову CIL платформи .NET

Через відсутність існуючих прямих рішень компіляції мови програмування TypeScript безпосередньо у CIL було вирішено проаналізувати компілятори, розроблені для мови JavaScript, що виконують таку компіляцію. Серед таких трансляторів можна виділити два працездатних рішення Jurassic та JScript.NET.

1.1.1. Бібліотека Jurassic

Jurassic – це реалізація стандарту ECMAScript, мови JavaScript та її середовища виконання. Цей інтерпретатор розповсюджується як бібліотека, яка написана на мові програмування C#, що дозволяє вбудовувати його у власні програми.

До переваг цього інтерпретатора можна віднести можливість вбудовувати його у власні програми для використання скриптової мови у своїх додатках.

До недоліків можна віднести відсутність можливості компілювати програму у виконуваний файл чи бібліотеку.

Інтеграція з власною програмою відбувається наступним чином:

1) потрібно встановити відповідний NuGet пакет;
2) створити об’єкт ScriptEngine та виконати передану програму (рисунок 1);

```
var engine = new Jurassic.ScriptEngine();
engine.Execute("console.log('testing')");
```

Рисунок 1 – Приклад виконання програми, яка виводить результат у консоль

1.1.2. Транслятор JScript

JScript.NET – скриптова мова програмування компанії «Microsoft», що є реалізацією стандарту ECMAScript. Синтаксис JScript багато в чому аналогічний мові JavaScript, але додає типізацію, що робить його схожим на мову програмування TypeScript. Компілятор цієї мови дозволяє створювати виконувані файли та динамічні бібліотеки, що дає змогу писати повноцінні програми.

Для розглядання ефективності згенерованих CIL інструкцій було вирішено протестувати компіляцію циклічних конструкцій та масивів (рис 2).

```
var arrTest = new int[100000];
for(var i = 0; i < arrTest.Length; i++) {
    arrTest[i] = i;
}
```

Рисунок 2 – Цикл for на мові програмування Jscript

Проаналізувавши отриманий вихідний файл за допомогою IL декомпілятора JetBrains dotPeek було визначено, що генерується багато важких інструкцій (тобто таких, які виконуються за більшу кількість часу у порівнянні з іншими) та викликів додаткових функцій бібліотеки Microsoft.JScript, що видно з фрагменту коду встановлення значення елементу масиву за індексом (рисунок 3). Цю операцію можна виконати за меншу кількість інструкцій (рисунок 4), як, наприклад, було отримано при компіляції відповідної мовної конструкції на мові програмування C#.

```
/ [33 7 - 33 194]
L_004f: ldsfld      int32[] 'JScript 0'::arrTest
L_0054: ldsfld      object 'JScript 0'::i
L_0059: ldc.i4.s    9 // 0x09
L_005b: ldc.i4.0
L_005c: call        object [Microsoft.JScript]Microsoft.JScript.Convert::Coerce2(object, valuetype [mscorlib]System.TypeCode, bool)
L_0061: unbox       [mscorlib]System.Int32
L_0066: ldind.i4
L_0067: ldsfld      object 'JScript 0'::i
L_006c: ldc.i4.s    9 // 0x09
L_006e: ldc.i4.0
L_006f: call        object [Microsoft.JScript]Microsoft.JScript.Convert::Coerce2(object, valuetype [mscorlib]System.TypeCode, bool)
L_0074: unbox       [mscorlib]System.Int32
L_0079: ldind.i4
L_007a: stelem.i4

/ [34 7 - 34 27]
L_007b: ldsfld      object 'JScript 0'::i
L_0080: stloc.2     // i
```

Рисунок 3 – Фрагмент декомпільованої програми на мові програмування JScript (тіло циклу)

Графік порівняння швидкодії двох програм для масивів різної величини зображено на рис 5. З нього видно, що компіляцію відповідної конструкції мови JScript можна зробити ефективнішою.

```
// [18 7 - 18 30]
IL_003e: ldloc.1      // numArray
IL_003f: ldloc.2      // index
IL_0040: ldloc.2      // index
IL_0041: stelem.i4
IL_0042: nop
```

Рисунок 4 – Фрагмент IL інструкцій, які дозволяють встановити елемент масиву за індексом

Варто підмітити, що до недоліків транслятора JScript можна віднести відсутність підтримки платформ .NET Core та її продовженням - .NET 5, які є переписаними версіями платформи .NET Framework, що мають більшу швидкодію та підтримують різні операційні системи. Детальніше про це написано в наступному розділі.

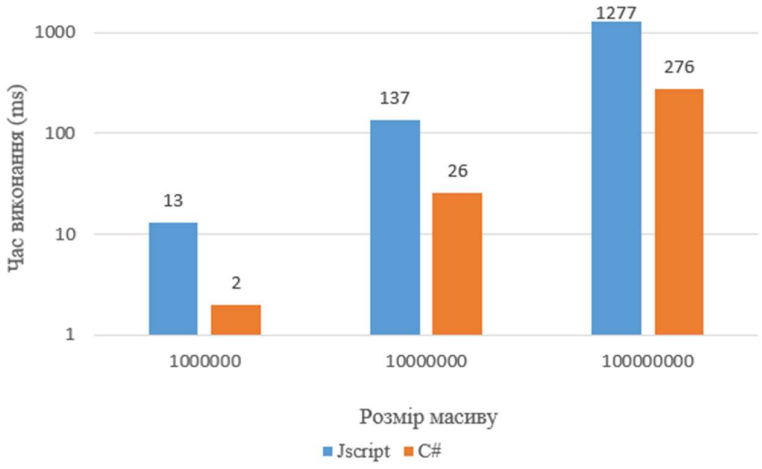

Рисунок 5 – Графік порівняння швидкодії Jscript та C#

1.2. Обґрунтування дослідження способів трансляції

Темою магістерської дисертації було обрано дослідження способів трансляції мови TypeScript у проміжну мову CIL платформи .NET, а саме:

- спосіб трансляції простих типів даних;
- спосіб трансляції конкатенації рядкових виразів;
- спосіб трансляції масивів;
- спосіб трансляції умовних конструкції;
- спосіб трансляції циклічних конструкцій;
- спосіб трансляції функцій;

Дане дослідження може бути використане, як фундамент для подальшої розробки повноцінного компілятора мови Typescript у проміжну мову CIL. Завдяки чому, можна популяризувати платформу .NET та мову програмування Typescript серед більшої кількості розробників, а також звести розробку веб-додатків, у яких використовується мова програмування Typescript чи Javascript у клієнтській частині та платформа .NET у серверній частині, до використання єдиної мови програмування.

1.3. Обґрунтування вибору середовища розробки

На сьогоднішній день платформа .NET підтримує декілька мов програмування, які можна використати для розробки тестового транслятора, а саме C#, Visual Basic та F#. Хоча Visual Basic все ще і підтримується, але не рекомендується починати нові проекти на цій мові, так як функціональні можливості для цієї мови не розширюються. Основна увага приділена розробці мов програмування C# та F#. Через більшу популярність мови C# (рисунок 6), було вирішено обрати саме її, бо компілятор Roslyn чи транслятор IronPython написані саме на ній.

Рисунок 6 – Графік порівняння популярності мов програмування C#, VisualBasic та F# серед розробників

Для генерації виконуваних файлів для середовища .NET було використано бібліотеку Mono.Cecil [12], яка надає зручний інтерфейс для запису інструкцій CIL. Дана бібліотека розроблена співробітником компанії Microsoft Джейбі Ейваном та активно підтримується спільнотою.

Для тестування компонентів тестового транслятора за допомогою модульних та інтеграційних тестів було обрано фреймворк XUnit.

Для тестування коду, згенерованого тестовим транслятора обрано програму dotPeek яка дозволяє аналізувати згенеровані CIL інструкції.

ВИСНОВКИ ДО РОЗДІЛУ 1

У цьому розділі було досліджено існуючі способи трансляції мови програмування Javascript у проміжну мову CIL платформи .NET через відсутність способів для мов Typescript.

У ході аналізу було визначено, що кожен з існуючих аналогів має певні недоліки.

До недоліків бібліотеки Jurassic відноситься відсутність можливості транслювати програму у виконуваний файл чи динамічну бібліотеку, що унеможливлює створення повноцінних проектів.

У свою чергу, транслятор JScript генерує не оптимальні IL інструкції, що підтверджується тестуванням у порівнянні з результатом роботи компілятора мови програмування C#. Крім того, даний транслятор не підтримує трансляцію у актуальні версії платформи .NET.

Як результат проведеного аналізу, можна зробити висновок, що обрана тема магістерської, а саме – дослідження способів трансляції мови TypeScript у проміжну мову CIL платформи .NET є актуальною.

Для дослідження було вирішено обрати такі засоби розробки, як: мова програмування C#, бібліотека Mono.Cecil, фреймоврк xUnit та програма dotPeek.

2. ПЛАТФОРМА .NET ТА ПРОМІЖНА МОВА CIL

2.1. Огляд платформи .NET

Представлена Microsoft у 2002 році платформа .NET – це величезна і одна з найбільш універсальних платформ для розробки мобільних, настільних, веб-додатків, сервісів, IoT та ігор. Вона охоплює безліч інструментів і технологій, а також декілька мов програмування, такі як C #, Visual C ++, VB.NET, та інші.

.NET пропонує безмежні можливості для створення надійного та сучасного програмного забезпечення. Ось приклади того, що можливо розробити на основі цієї платформи, назвемо лише декілька:

- Веб-сайти
- Програми на основі консолі
- Програми графічного інтерфейсу Windows
- Програми служб Windows
- Веб-сервіси
- Програми для мобільних пристроїв та КПК (для портативних пристроїв)

Варто відзначити, що платформа .NET довгий час розвивався головним чином як платформа для Windows під назвою .NET Framework. 2019 року компанія Microsoft випустила останню версію цієї гілки платформи – .NET Framework 4.8.

З 2014 року Microsoft почали розвивати альтернативну платформу - .NET Core, яка вже проектувалась для підтримки різних операційних систем і повинна була увібрати в себе всі можливості застарілої платформи .NET Framework і додати нову функціональність. Компанія протягом останніх п’яти років послідовно випустила ряд версій цієї платформи: .NET Core 1, .NET Core 2, .NET Core 3. Логічним розвитком .NET Core 3.0 платформа .NET 5. Тому слід розрізняти .NET Framework, який призначений переважно для Windows, і кросплатформний .NET 5. Порівняння цих платформ наведене у таблиці 1.

Таблиця 1 – Порівняння платформ .NET Framework та .NET 5

Критерій	**.NET Framework**	**.NET Core (.NET 5)**
Операційні системи	Windows	Windows, Linux, Mac
Технології	Підтримує Web Forms, Web API, Web Pages, and MVC	Підтримує Web API, Web Pages, MVC та Blazor
Швидкодія	Гірша	Краща
Спільнота	Відсутня	Відкритий код, спільнота може вносити зміни

З цього можна зробити висновок, що розробка транслятора у першу є більш перспективною чергу для платформ .NET Core та .NET 5. Але ми можемо спробувати додати підтримку обох версій – старої .NET Framework та нової .NET 5. Це припущення випливає з теорії, описаної в наступному підрозділі.

2.2. Архітектура платформи .NET

Платформа .NET є багаторівневою, модульною та ієрархічною. Кожен рівень цієї платформи – це певний шар абстракції. Самому верхньому рівню абстракції відповідають мови, які підтримує платформа .NET, а самому нижчому - загальномовне середовище виконання. Це важливо, оскільки загальномовне середовище виконання тісно співпрацює з операційною системою для управління .NET додатків. Платформа розділена на модулі, кожен з яких має свою чітку відповідальність. Архітектура платформи .NET показана на рисунку 7.

Common Type System (Загальна система типів) – частина .NET Framework, формальна специфікація, що визначає, як який-небудь тип (клас, інтерфейс, структура, вбудований тип даних) повинен бути визначений для його правильного виконання середовищем. NET [6]. Крім того, даний стандарт визначає, як визначення типів і спеціальні значення типів представлені в комп'ютерній пам'яті.

Метою розробки CTS було забезпечення можливості програмам, написаним на різних мовах програмування, легко обмінюватися інформацією. Як це прийнято в мовах програмування, тип може бути описаний як визначення набору допустимих значень (наприклад, «всі цілі від 0 до 10») і допустимих операцій над цими значеннями (наприклад, додавання і віднімання).

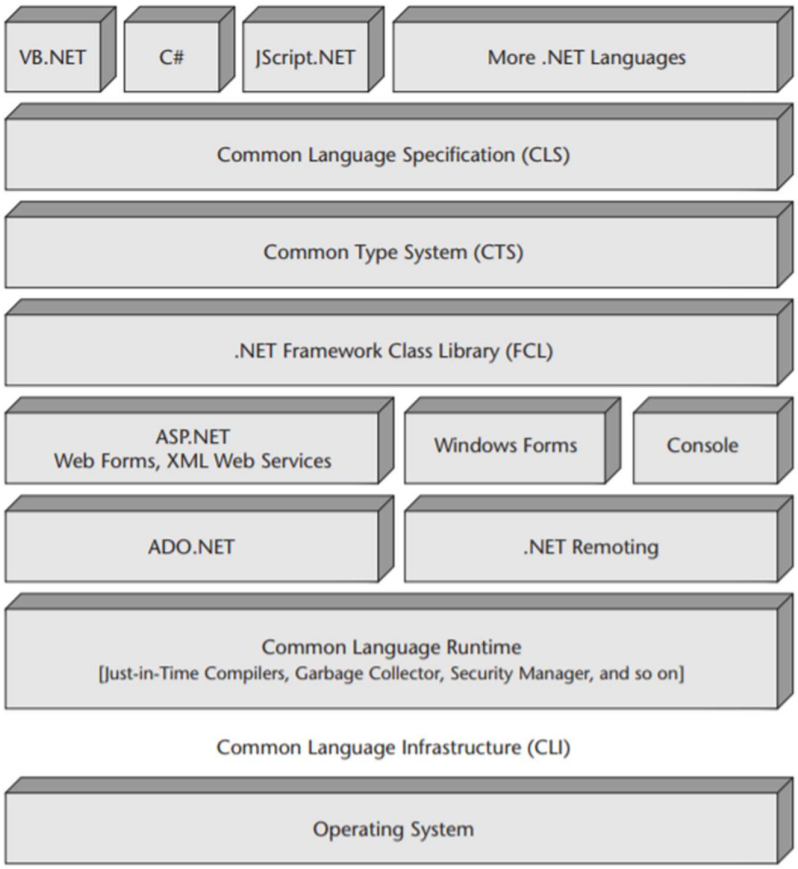

Рисунок 7 – Архітектура платформи .NET

.NET Framework включає реалізацію базових стандартних бібліотек CLI. Бібліотека класів .NET Framework (FCL) організована в ієрархію просторів імен. Більшість вбудованих інтерфейсів прикладного програмування (API) є частиною просторів імен System.* або Microsoft.*. Ці бібліотеки класів реалізують багато загальних функцій, таких як читання та запис файлів, графічний візуалізація, взаємодія з базами даних та маніпулювання документами XML. Бібліотеки класів доступні для всіх мов, сумісних з CLI. FCL реалізує бібліотеку базового класу CLI

(BCL) та інші бібліотеки класів - деякі визначаються CLI, а інші - специфічні для Microsoft.

BCL включає невелику підмножину всієї бібліотеки класів і є основним набором класів, які служать базовим API CLR. Для .NET Framework більшість класів, які вважаються частиною BCL, знаходяться в динамічних бібліотеках mscorlib.dll, System.dll та System.Core.dll. Класи BCL доступні в .NET Framework, а також його новій реалізації .NET 5.

Враховуючи інформацію, у яких динамічних бібліотеках кожна з версій платформ .NET зберігає базові класи, можна додати можливість вказувати залежні бібліотеки під час компіляції в залежності від цільової платформи. Так, наприклад, для платформи .NET Framework серед залежностей буде вказана mscorelib.dll, а для .NET 5 – не буде вказана. Дана функціональна можливість буде реалізована у генераторі коду тестового транслятора.

Для розробки транслятора мови Typescript розглянемо більш детально середовище CLR та проміжну мову CIL.

2.3. Проміжна мова CIL та віртуальна машина CLR

Common language runtime (CLR) є віртуальною машиною платформи .NET. Це середовище виконання, яке запускає коди та допомагає полегшити процес розробки, надаючи різні сервіси. В основному воно відповідає за управління виконанням програм .NET незалежно від будь-якої мови програмування .NET. Внутрішньо CLR реалізує VES (віртуальну систему виконання), яка визначена в реалізації CLI (Common Language Infrastructure) [6].

Код, який працює під час виконання CLR, називається керованим кодом. Іншими словами, ви можете сказати, що CLR забезпечує кероване середовище виконання програм .NET, покращуючи захист, включаючи міжмовну інтеграцію та багатий набір бібліотек класів, тощо. CLR присутній у кожній версії платформи .NET.

Проаналізуємо принцип роботи віртуальної машини CLR (рисунок 8).

Нехай, ми маємо програму на мові програмування C#, збережену у файлі (source code). Компілятор (Language Compiler) виконує компіляцію вихідного код у CIL або IL разом із її метаданими. Метадані включають усі типи, фактичну реалізацію кожної функції програми. CLR надає послуги та

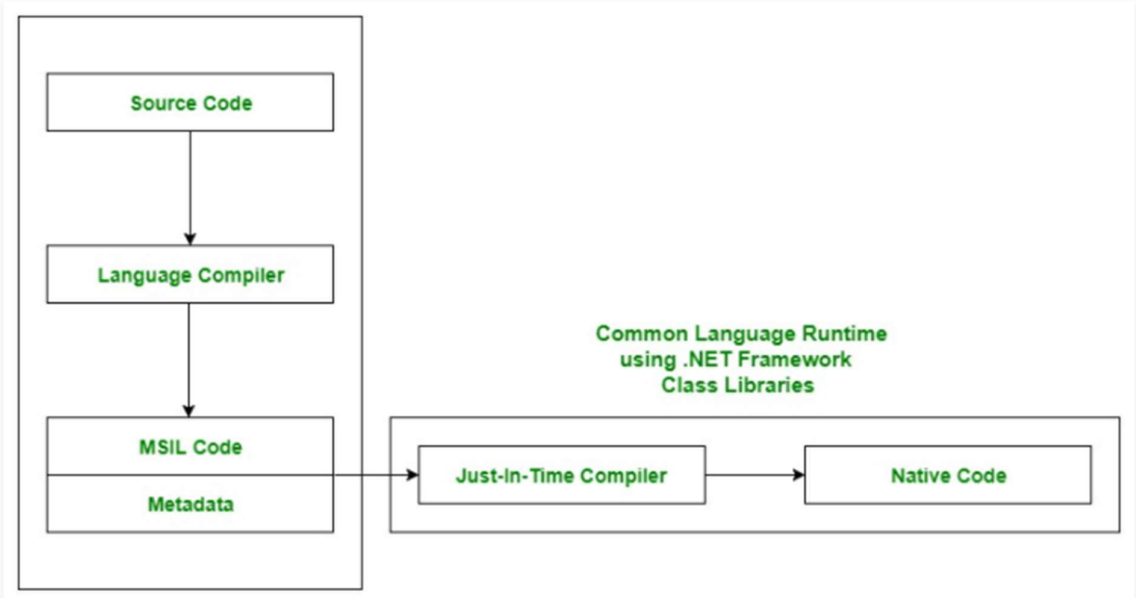

Рисунок 8 – Приклад роботи CLR

середовище виконання коду CIL. Внутрішньо CLR включає компілятор JIT (Just-In-Time), який перетворює код CIL в машинний код, який далі виконується процесором. CLR також використовує бібліотеки класів .NET Framework. Метадані надають інформацію про мову програмування, середовище, версію та бібліотеки класів до CLR, за допомогою якого віртуальна машина виконує код CIL. Оскільки CLR є загальним, то він дозволяє екземпляру класу, який написаний іншою мовою, викликати метод класу, який реалізовано іншою мовою.

Використовуючи метадані можливо буде проаналізувати базові бібліотеки та відповідно транслювати типи мови програмування Typescript у типи платформи .NET, що і буде використано у генераторі коду тестового транслятора.

JIT (just-in-time компілятор) відповідає за перетворення CIL (Common Intermediate Language) у машинний код.

Також CLR має збирач сміття (Garbage collector), який використовується для забезпечення функції автоматичного керування пам'яттю. Якби не було збирача сміття, програмістам довелося б писати код управління пам’яттю, що було б накладними витратами.

CIL є об'єктно-орієнтованою мовою, яка базується на понятті стека, що означає, що параметри команд та результати зберігаються в одному стеку, а не в декількох регістрах або інших місцях пам'яті, як у більшості мов програмування. Таким чином, код, який додає два числа мовою асемблера x86, де eax та edx визначають два різні регістри загального призначення: add eax, edx , буде реалізовану мовою CIL, як зображено на рисунку 9, де 0 відповідає за регістр eax, а 1 – edx.

```
ldloc.0    // push local variable 0 onto stack
ldloc.1    // push local variable 1 onto stack
add        // pop and add the top two stack items then push the result onto the stack
stloc.0    // pop and store the top stack item to local variable 0
```

Рисунок 9 – Приклад додавання двох чисел мовою CIL

В останньому прикладі значення двох регістрів, eax та edx, спочатку завантажуються в стек. Коли додаткову інструкцію викликають, операнди "вискакують" або отримуються, а результат зберігається у стеку. Потім отримане значення витискається зі стеку і записується в регістр eax.

CIL дозволяє створювати об'єкти, викликати методи та використовувати інші типи членів, наприклад поля.

Кожному методу потрібно (за деякими винятками) знаходитись у класі. Так само і статичний метод, зображений на рисунку 10. Метод Add не вимагає оголошення будь-якого екземпляра Foo, оскільки він оголошений як статичний. Виклик такого методу зображено на рисунку 11.

```
.class public Foo {
    .method public static int32 Add(int32, int32) cil managed {
        .maxstack 2
        ldarg.0 // load the first argument;
        ldarg.1 // load the second argument;
        add     // add them;
        ret     // return the result;
    }
}
```

Рисунок 10 – Статичний метод Add мовою CIL

```
ldc.i4.2
ldc.i4.3
call int32 Foo::Add(int32, int32)
stloc.0
```

Рисунок 11 – Виклик статичного методу Add мовою CIL

Клас екземпляра містить принаймні один конструктор та кілька членів екземпляра. На рисунку 12 зображено клас, який має набір методів, що представляють дії Car-об'єкта. Виклик конструктора класу та методу Move зображено на рисунку 13.

Так як у мові CIL всі методи повинні знаходитись у певному класі, то функцій мови Typescript відповідно будуть транслюватись у статичні методи загального класу.

```
.class public Car {
    .method public specialname rtspecialname instance void .ctor(int32, int32) cil managed {
        /* Constructor */
    }

    .method public void Move(int32) cil managed { /* Omitting implementation */ }
    .method public void TurnRight() cil managed { /* Omitting implementation */ }
    .method public void TurnLeft() cil managed { /* Omitting implementation */ }
    .method public void Brake() cil managed { /* Omitting implementation */ }
}
```

Рисунок 12 – Виклик статичного методу Add мовою CIL

```
ldc.i4.1
ldc.i4.4
newobj instance void Car::.ctor(int, int)

ldloc.0    // Load the object "myCar" on the stack
ldc.i4.3
call instance void Car::Move(int32)
```

Рисунок 13 – Виклик статичного методу Add мовою CIL

Мова CIL налічує більше 150 інструкцій [7, 14]. У таблиці 2 на наведено основні інструкції в алфавітному порядку, які будуть використані в наступному розділі у запропонованих способах трансляції мови програмування Typescript.

Таблиця 2 – Інструкції мови CIL

Опкод	**Інструкція**	**Опис**
0x58	add	Додайте два значення, повертаючи нове значення.
0x5F	and	Побітове значення AND двох інтегральних значень повертає інтегральне значення.
0x8C	box<тип>	Виділяє місце в кучі для типу значення та повертає посилання
0x38	br<мітка>	Перейти до мітки
0x39	brfalse<мітка>	Перейти до мітки, якщо false
0x3A	brtrue<мітка>	Перейти до мітки, якщо true
0x28	call <метод>	Викликати метод
0x6F	callvirt<метод>	Викликати метод екземпляру класу
0xFE 0x01	ceq	Записати у стек 1, якщо значення1 дорівнює значенню2, інакше - 0.
0xFE 0x02	cgt	Записати у стек 1, якщо значення1 більше значення2, інакше - 0.

Продовження таблиці 2

Опкод	Інструкція	Опис
0xFE 0x04	clt	Записати у стек 1, якщо значення1 менше значення2, інакше - 0.
0x5B	div	Ділить два значення, щоб повернути частку або результат із плаваючою комою.
0x25	dup	Продублювати значення на вершині стека.
0x5A	mul	Множить значення
0x65	neg	Бере негативне значення
0x8D	newarr<тип>	Створіть новий масив з елементами заданого типу.
0x73	newobj<ctor>	Створює неініціалізований об'єкт та викликає конструктор ctor.
0x00	nop	Нічого не робить
0x66	not	Побітове доповнення (логічне не).
0x60	or	Побітове АБО двох цілих значень повертає ціле число.
0x26	pop	Виштовхує значення зі стеку
0x2A	ret	Вийти з методу
0x9C	stelem.i1	Виставляє елемент масиву за індексом у значенням int8 зі стеку.
0x9E	stelem.i4	Виставляє елемент масиву за індексом у значенням int32 зі стеку.
0xA2	stelem.ref	Виставляє елемент масиву за індексом у значенням за посиланням зі стеку.
0xFE 0x0E	stloc<index>	Виштовхує значення зі стеку в локальну змінну за індексом

Продовження таблиці 2

Опкод	Інструкція	Опис
0x0A-0x0D	stloc.0-stloc.3	Часткові випадки попередньої інструкції для значень індексу 0-3
0x80	stsfld<поле>	Виставляє значення статичного поля
0x59	sub	Віднімає значення2 від значення1, повернувши нове значення.
0x75	isinst<клас>	Перевіряє, чи об'єкт є екземпляром класу, що повертає null, або екземпляром цього класу чи інтерфейсу.
0xFE 0x09	ldarg<індекс>	Завантажує аргумент під індексом в стек.
0x02- 0x05	ldarg.0-ldarg3	Часткові випадки попередньої інструкції для значень індексу 0 - 3
0x20	ldc.i4<число>	Завантажує число типу int32 на стек
0x16-0x1E	ldc.i4.0-ldc.i4.8	Часткові випадки попередньої інструкції для значень 0-8
0x94	ldelem.i4	Завантажує елемент із типом int32 з індексом у вершину стека як int32.
0x9A	ldelem.ref	Завантажте елемент з індексом у верхню частину стека як O. Тип O такий самий, як і тип елемента масиву, що завантажується на стек CIL.
0x91	ldelem.u1	Завантажує елемент з типом unsigned int8 за індексом у верх стека як int32.
0x8E	ldlen	Завантажує довжину масиву на стек
0x14	ldnull	Завантажує посилання NULL на стек

Продовження таблиці 2

Опкод	Інструкція	Опис
0xFE 0x0C	ldloc< індекс >	Завантажте локальну змінну за індексом на стек.
0x06- 0x09	ldloc.0- ldloc.3	Часткові випадки попередньої інструкції для значень 0-3
0x7E	ldsfld<поле>	Завантажте значення статичного поляна стек.
0x72	ldstr<рядок>	Завантажує константний рядок на стек
0x79	unbox<тип>	Переносить з кучі на стек зміст типу значення
0x61	xor	Побітовий XOR цілочисельних значень повертає ціле число.

Наприклад, під час трансляції масивів будуть використані такі інструкції, як: newarr, ldelem, stelem та ldlen. Для трансляції виклику функцій – відповідно call та callvirt. Детальніше про це написано в наступному розділі.

ВИСНОВКИ ДО РОЗДІЛУ 2

У цьому розділі було розглянуто етапи розвитку та будову платформи .NET.

Було визначено переваги нової платформи .NET 5 у порівнянні з старою версією .NET Framework.

Відомо, що платформа .NET має такі основні складові:

1) загальна система типів;
2) базова бібліотека класів (BCL);
3) мова CIL;
4) віртуальна машина CLR;

Особливо увагу було приділено мові CIL та віртуальній машині CLR.

Визначено, що основними складовими віртуальної машини CLR є JIT компілятор та збирач сміття.

Мова CIL є об'єктно-орієнтованою та заснованою на стеку. Було досліджено, як виконуються компіляція мови CIL у мову асемблера та проаналізовані основні інструкції цієї мови.

Також, варто зазначити, що у цьому розділі наведена таблиця в алфавітному порядку з усіма інструкціями, які будуть використані при дослідженні способів трансляції мови Typescript у CIL, які описані в наступному розділі.

3. СПОСОБИ ТРАНСЛЯЦІЇ МОВИ TYPESCRIPT У ПРОМІЖНУ МОВУ CIL ПЛАТФОРМИ .NET

3.1. Спосіб трансляції декларації змінних, простих типів даних та операцій над ними

Розглянемо систему типів мови програмування Typescript. Вона налічує такі типи даних:

- boolean – найбільш базовий тип даних. Може мати значення true / false, яке називають булевим значенням;
- number – як і в JavaScript, усі числа в TypeScript є значеннями з плаваючою комою;
- bigint – частковий випадок для великих цілих чисел;
- string – рядковий тип даних, константи відокремлюються подвійними або одинарними лапками;
- any – будь-який тип;
- void – відсутність типу, використовується для вказування того, що функція не повертає результат;
- enum – перелічення;
- масиви та кортежі;

З простих типів даних було вирішено дослідити трансляцію таких типів: boolean, number, string, any та void. Масиви будуть розглянуті далі, у відповідному підрозділі. Семантична відповідність базовим типам платформи .NET наведена у таблиці 3. Варто зазначити, що було вирішено зробити невелике спрощення. Будемо вважати, що тип number відповідає цілим числам. Даний підхід ніяк не впливає на особливість трансляції чисел у мову CIL платформи .NET (за виключенням того, що будуть згенеровані інструкції для цілих чисел), але спростить аналіз введених даних.

Таблиця 3 – Відповідність базових типів мови Typescript до типів платформи .NET

Базовий тип мови Typescript	**Базовий тип платформи .NET**
boolean	System.Boolean
number	System.Int32
string	System.String
any	System.Object
void	System.Void

Декларацію змінних простих типів даних в мові Typescript можна описати такою БНФ (повна БНФ досліджуваної підмножини мови Typescript наведена у додатку 3):

<variable-declarations-statement> --> <variable-declarations>;

<variable-declarations> --> <declaration-keyword> <declarations-list>

<declarations-list> --> <declaration> | <declaration>,<declarations-list>

<declaration> --> <identifier> <type-annotation>? <initializer>?

< type-annotation > --> :<attribute>

<initializer> --> =<expression>

<attribute> --> boolean | number | string | any

<declaration-keyword> --> var | let | const

Як бачимо, <type-annotation> і <initializer> не обов'язкові. У разі відсутності <type-annotation> тип змінної визначається значенням виразу ініціалізації. Якщо відсутні і <type-annotation> і <initializer>, то значення змінної матиме тип any.

Нехай, маємо програму, в якій оголошується числова змінна зі значенням 5. Після роботи лексичного та синтаксичного аналізаторів отримуємо представлення у вигляді дерева розбору (рисунок 14).

В мові CIL усі локальні змінні оголошуються на початку методу та замість ім'я мають відповідний індекс. Завдяки тому, що вирішено було обрати

бібліотеку Mono.Cecil, можливо перекласти відповідальність вписувати всі змінні та надавати їм індекс на цю бібліотеку.

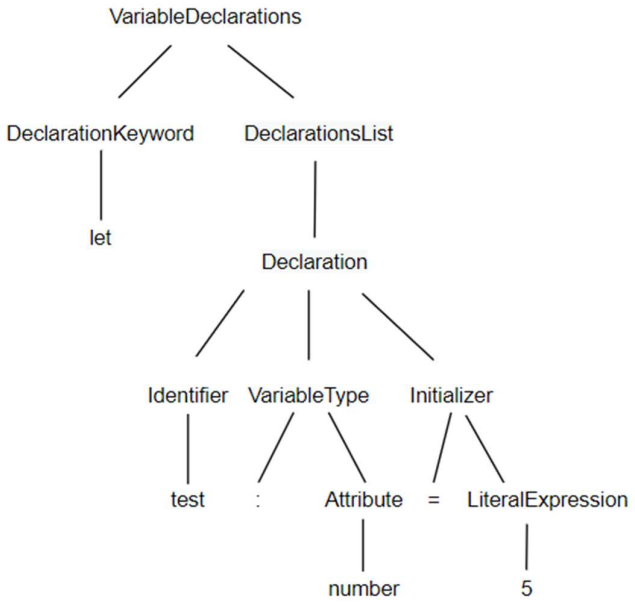

Рисунок 14 – Приклад дерева розбору оголошення числової змінної

Таким чином спосіб трансляції змінних полягає в наступному:

1) знайти відповідність для типу Typescript у платформі .NET;
2) створити VariableDefinition бібліотеки Mono.Cecil с заданим типом та запам’ятати у словник локальних змінних;
3) додати змінну до тіла методу;
4) якщо ініціалізатор відсутній, перейти до пункту 7;
5) виконати генерацію коду для виразу ініціалізації;
6) згенерувати інструкцію запису верхівки стеку у змінну;
7) кінець;

Приклад реалізації способу на мові програмування C# зображено на рисунку 15. Результат роботи методу для заданого прикладу зображено на рисунку 16.

Також розглянемо операції, які можуть виконуватись над типами, що розглядаються. Унарні оператори вказані у таблиці 4, а бінарні – у таблиці 5.

```
1 reference
private void EmitVariableDeclaration(ILProcessor ilProcessor, BoundVariableDeclaration node)
{
    var typeRef = ResolveCLRType(node.Variable.Type);
    var varDef = new VariableDefinition(typeRef);
    _locals.Add(node.Variable, varDef);
    ilProcessor.Body.Variables.Add(varDef);

    if (node.Initializer != null) {
        EmitExpression(ilProcessor, node.Initializer);
        ilProcessor.Emit(OpCodes.Stloc, varDef);
    }

}
```

Рисунок 15 – Генерація CIL інструкцій оголошення змінної

```
.locals init (
  [0] int32 V_0
)

IL_0000: ldc.i4.5
IL_0001: stloc.0      // V_0
```

Рисунок 16 – Згенеровані інструкції для прикладу програми з рисунку 14.

Таблиця 4 – Перелік унарних операторів

Оператор	**Тип**	**Опис**
!	boolean	Логічне не
+	number	Повертає операнд
-	number	Повертає операнд з протилежним знаком.
~	number	Побітове не

Таблиця 5 – Перелік бінарних операторів

Оператор	**Тип операндів**	**Тип результату**	**Опис**
&	boolean	boolean	Побітове І
&	number	number	Побітове І

Продовження таблиці 5

Оператор	**Тип операндів**	**Тип результату**	**Опис**
|	boolean	boolean	Побітове АБО
|	number	number	Побітове АБО
^	boolean	boolean	Виняткова диз'юнкція
^	number	number	Виняткова диз'юнкція
&&	boolean	boolean	Логічне І
||	boolean	boolean	Логічне АБО
==	boolean	boolean	Дорівнює
==	number	boolean	Дорівнює
==	string	boolean	Дорівнює
!=	boolean	boolean	Не дорівнює
!=	number	boolean	Не дорівнює
!=	string	boolean	Не дорівнює
<	number	boolean	Менше
<	string	boolean	Менше
>	number	boolean	Більше
>	string	boolean	Більше
<=	number	boolean	Менше або дорівнює
<=	string	boolean	Менше або дорівнює

Продовження таблиці 5

Оператор	**Тип операндів**	**Тип результату**	**Опис**
>=	number	boolean	Більше або дорівнює
>=	string	boolean	Більше або дорівнює
+	number	number	Додавання
+	string	string	Конкатенація
-	number	number	Віднімання
*	number	number	Множення
/	number	number	Ділення

В загальному випадку, кожному оператору відповідає певна інструкція мови CIL. Так, наприклад, додавання чисел виконується за допомогою інструкції add, віднімання – sub, логічне та побітове I – and, тощо. Усі інструкції перелічені у другому розділі.

Але можна виділити декілька випадків, коли оператор транслюється не в одну інструкцію. Дана особливість відноситься до таких операторів, як “!=”, “>=”, “<=” для типів number, boolean та всіх операторів, які відносяться до типу string.

Спочатку розглянемо оператори порівняння для типів number та boolean. Через відсутність у мові CIL спеціальних інструкцій для операторів “!=”, “>=”, “<=”, трансляція виконується за допомогою заміни їх на еквівалентну логіку. Так, наприклад, “більше або дорівнює” замінюється на “не менше”, “менше або дорівнює” на “не більше”, “не дорівнює” на “результат порівняння на рівність дорівнює false”.

Тобто, спосіб трансляції відповідних операторів для типів number і boolean полягає в генерації таких інструкцій:

1) “!=”:

```
ceq
ldc.i4.0
ceq
```

2) “<=” :

```
cgt
ldc.i4.0
ceq
```

3) “>=” :

```
clt
ldc.i4.0
ceq
```

Для типу string трансляція доступних операторів буде виконана інакше. Через те, що рядок є вказівниковим типом даних, порівняння на рівність має виконуватись за допомогою вбудованої функції Object.Equals [11]. Таким чином спосіб трансляція операторів “==” та “!=” буде схожим до генерації коду для типів number і boolean, але замість першого виклику інструкції ceq буде виконано виклик функції Object.Equals.

Для трансляції операторів порівняння “<”, “<=”, “>”, “>=” використаємо вбудовану функцію string.Compare [11]. Вона повертає позитивне ціле число, якщо перший аргумент більший за другий, 0 у випадку, коли аргументи рівні та негативне число, коли перший аргумент менший за другий.

Згенеровані інструкції за цим способом матимуть вигляд:

1) “<”:

```
call int32 System.String::Compare(string, string)
ldc.i4.0
clt
```

2) “<=” :

```
call int32 System.String::Compare(string, string)
ldc.i4.0
```

```
cgt
ldc.i4.0
ceq
```

3) “>” :

```
call int32 System.String::Compare(string, string)
ldc.i4.0
cgt
```

4) “>=” :

```
call int32 System.String::Compare(string, string)
ldc.i4.0
clt
ldc.i4.0
ceq
```

Спосіб трансляції оператора “+” для рядків, який називається конкатенацією, дещо складніший, тому алгоритм розглянуто в наступному підрозділі.

3.2 Спосіб трансляції конкатенації рядкових виразів

Розглянемо трансляцію конкатенації на прикладі вхідної програми, що зображена на рисунку 17.

```
let a: string = 'test a', b: string = 'test b',
    c: string = 'test c', d: string = 'test d',
    e: string = 'test e';
let result: string;

result = a + b + c + d + e;
```

Рисунок 17 – Вхідна програма на мові Typescript, яка демонструє конкатенацію рядків.

Після роботи лексичного та синтаксичного аналізатора отримуємо представлення поданої програми у вигляді дерева розбору (рисунок 18).

Для конкатенації рядків платформа .NET має вбудовану функцію String.Concat [12], яка виконує конкатенацію переданих до неї рядкових аргументів.

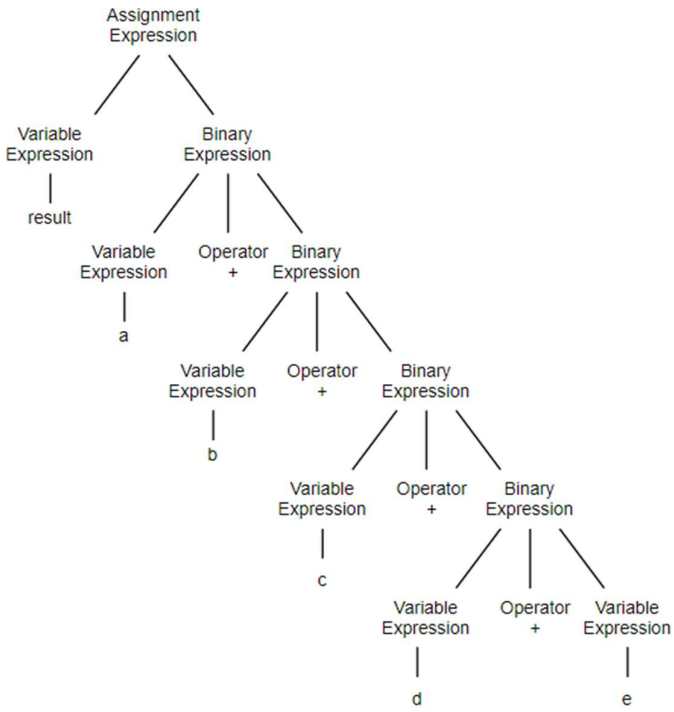

Рисунок 18 – Частина дерева розбору, що містить конкатенацію рядкових змінних.

Звісно, можна згенерувати виклик цієї функції для кожного оператору додавання та передати як аргументи лівий та правий вираз, але така реалізація не буде оптимальною, оскільки доведеться зробити стільки викликів функції, скільки операторів додавання міститься у виразі. У документації зазначено, що функція має декілька варіантів виклику з перевантаженнями: для 2 аргументів, для 3 аргументів, для 4 аргументів та для масиву рядків. Тому згенерований код можна оптимізувати викликом потрібного варіанту перевантаження функції.

Запропонований спосіб трансляції конкатенації рядків полягає в наступному:

1) перетворити у масив дерево бінарних виразів, у яких виконується конкатенація рядків;

2) якщо розмір масиву менше ніж 5, то викликати потрібний варіант перевантаження функції для 2, 3 чи 4 аргументів та перейти до пункту 5, інакше перейти до пункту 3;
3) згенерувати інструкції для створення и заповнення масиву рядковими виразами, які необхідно об’єднати;
4) викликати потрібний варіант перевантаження функції для масиву аргументів;
5) кінець.

Результат виконання алгоритму для обраного прикладу зображено на рисунку 19.

Лістинг функції, яка реалізує запропонований спосіб показано на рисунку 20.

Варто зазначити, що цей алгоритм було опубліковано у власній статті [2].

```
IL_001f: ldc.i4.5
IL_0020: newarr       [System.Runtime]System.String
IL_0025: dup
IL_0026: ldc.i4.0
IL_0027: ldloc.0      // V_0
IL_0028: stelem.ref
IL_0029: dup
IL_002a: ldc.i4.1
IL_002b: ldloc.1      // V_1
IL_002c: stelem.ref
IL_002d: dup
IL_002e: ldc.i4.2
IL_002f: ldloc.2      // V_2
IL_0030: stelem.ref
IL_0031: dup
IL_0032: ldc.i4.3
IL_0033: ldloc.3      // V_3
IL_0034: stelem.ref
IL_0035: dup
IL_0036: ldc.i4.4
IL_0037: ldloc.s      V_4
IL_0039: stelem.ref
IL_003a: call         string [System.Runtime]System.String::Concat(string[])
IL_003f: dup
IL_0040: stloc.s      V_5
```

Рисунок 19 – Згенеровані інструкції для конкатенації рядкових змінних за запропонованим способом для програми, показаної на рис.17.

```
private void EmitStringConcatExpression(ILProcessor ilProcessor, BoundBinaryExpression node)
{
    var nodes = EmitUtils.Flatten(node).ToList();

    switch (nodes.Count)
    {
        case 0:
            ilProcessor.Emit(OpCodes.Ldstr, string.Empty);
            break;

        case 1:
            EmitExpression(ilProcessor, nodes[0]);
            break;

        case 2:
            EmitExpression(ilProcessor, nodes[0]);
            EmitExpression(ilProcessor, nodes[1]);
            ilProcessor.Emit(OpCodes.Call, _stringConcat2Reference);
            break;

        case 3:
            EmitExpression(ilProcessor, nodes[0]);
            EmitExpression(ilProcessor, nodes[1]);
            EmitExpression(ilProcessor, nodes[2]);
            ilProcessor.Emit(OpCodes.Call, _stringConcat3Reference);
            break;

        case 4:
            EmitExpression(ilProcessor, nodes[0]);
            EmitExpression(ilProcessor, nodes[1]);
            EmitExpression(ilProcessor, nodes[2]);
            EmitExpression(ilProcessor, nodes[3]);
            ilProcessor.Emit(OpCodes.Call, _stringConcat4Reference);
            break;

        default:
            ilProcessor.Emit(OpCodes.Ldc_I4, nodes.Count);
            ilProcessor.Emit(OpCodes.Newarr, ResolveCLRType(TypeSymbol.String));

            for (var i = 0; i < nodes.Count; i++)
            {
                ilProcessor.Emit(OpCodes.Dup);
                ilProcessor.Emit(OpCodes.Ldc_I4, i);
                EmitExpression(ilProcessor, nodes[i]);
                ilProcessor.Emit(OpCodes.Stelem_Ref);
            }

            ilProcessor.Emit(OpCodes.Call, _stringConcatArrayReference);
            break;
    }
}
```

Рисунок 20 – Лістинг функції, яка генерує IL інструкції для конкатенації.

3.3. Спосіб трансляції масивів

Для трансляції масивів мови Typescript у мову CIL окремо розглянемо трансляцію таких функціональних можливостей:

1) оголошення змінних з типом масив;
2) виділення пам’яті для масивів;
3) ініціалізація масивів початковими елементами;
4) доступ до комірки масиву.

Оголошення змінної з типом масив нічим не відрізняється від оголошення змінної, описаної у попередньому розділі, за виключенням того, що БНФ буде мати складніший вигляд, а саме – буде доповнено можливі значення <type-annotation> . Доповнена БНФ має наступний вигляд:

<type-annotation> --> :<type-clause>

<type-clause> --> <attribute> | <array-type>

<array-type> --> <type-clause> []

<attribute> --> boolean | number | string | any

У платформі .NET за масиви відповідає тип System.Array. Бібліотека Mono.Cecil дозволяє перетворити .NET тип у тип відповідного масиву за допомогою методу TypeReference.MakeArrayType. Таким чином, остаточна реалізація методу, який відповідає за перетворення типу мови Typescript у відповідний тип платформи .NET, зображена на рисунку 21. Варто зазначити, що словнику _knownTypes відповідає таблиця 3.

```
11 references
private TypeReference ResolveCLRType(TypeSymbol type)
{
    if (type is ArrayTypeSymbol arrT)
    {
        var arrType = ResolveCLRType(arrT.ElementType);
        return arrType.MakeArrayType();
    }

    return _knownTypes[type];
}
```

Рисунок 21 – Лістинг функції, яка повертає відповідний .NET тип для типу Typescript.

Як і в багатьох інших мовах програмування, Typescript дозволяє створити пустий масив певного розміру, або проініціалізувати його певними значеннями. Розглянемо БНФ для цих виразів:

<array-creation> --> new Array <generic-type>? (<expression>?)

<array-initializer> --> [<expression-list>?]

<generic-type> --> < <type-clause> >

<expression-list> --> <expression> | <expression>, <expression-list>

Хоча створення масиву є підмножиною загального випадку виклику конструктора об’єкта, але через те, що способи трансляції об’єктів не розглядаються, було вирішено винести це в окреме правило.

Для створення масивів мова CIL має спеціальну інструкцію newarr. Таким чином спосіб трансляції створення масиву полягає у наступному:

1) знайти відповідність типу у платформі .NET для якого створюємо масив;
2) помістити значення <expression> на стек, що відповідає за розмір масиву;
3) викликати інструкцію newarr зі значенням типу, знайденого в пункті 1.

Реалізація способу на мові C# зображена на рисунку 22.

```
private void EmitArrayCreationExpression(ILProcessor ilProcessor, BoundArrayCreationExpression node)
{
    EmitExpression(ilProcessor, node.Size);
    ilProcessor.Emit(OpCodes.Newarr, ResolveCLRType(node.ElementType));
}
```

Рисунок 22 – Лістинг функції, яка генерує CIL інструкції створення масиву.

Спосіб трансляції ініціалізації масиву початковими значеннями полягає у наступному:

1) створюється масив такого ж самого розміру, що і розмір переданого <expression-list> за способом, описаним вище;
2) для кожного елементу генеруються інструкції для встановлення елементу за індексом.

Спосіб встановлення елементу за індексом містить такі кроки:

1) на стек завантажується значення індексу;

2) генерується інструкція встановлення елементу. Але потрібно пам’ятати, що в залежності від типу обирається певна інструкція - для вказівникових типів генерується інструкція stelem.ref, для чисел stelem.i4, для булевих значень stelem.i1.

Реалізацію функції, яка відповідає за генерацію ініціалізації масиву, зображено на рисунку 23.

```
private void EmitArrayInitializationExpression(ILProcessor ilProcessor, BoundArrayInitializationExpression node)
{
    ilProcessor.Emit(OpCodes.Ldc_I4, node.Values.Length);
    ilProcessor.Emit(OpCodes.Newarr, ResolveCLRType(node.ElementType));

    for (var i = 0; i < node.Values.Length; i++)
    {
        ilProcessor.Emit(OpCodes.Dup);
        ilProcessor.Emit(OpCodes.Ldc_I4, i);
        EmitExpression(ilProcessor, node.Values[i]);

        var type = node.Values[i].Type;
        if (type.IsReference)
        {
            ilProcessor.Emit(OpCodes.Stelem_Ref);
        }
        else if (type == TypeSymbol.Number)
        {
            ilProcessor.Emit(OpCodes.Stelem_I4);
        }
        else
        {
            ilProcessor.Emit(OpCodes.Stelem_I1);
        }
    }
}
```

Рисунок 23 – Лістинг функції, яка генерує CIL інструкції ініціалізації масиву.

Доступ до комірки масиву генерується схожим чином, але замість інструкції stelem, генерується ldelem, тобто:

1) на стек завантажується значення індексу;
2) генерується інструкція встановлення елементу ldelem в залежності від типу.

Лістинг функції, яка генерує інструкції доступу до елементу, зображено на рисунку 24.

```
1 reference
private void EmitIndexAccessExpression(ILProcessor ilProcessor, BoundIndexAccessExpression node)
{
    EmitExpression(ilProcessor, node.Expression);
    EmitExpression(ilProcessor, node.Index);

    var type = node.Type;
    if (type.IsReference)
    {
        ilProcessor.Emit(OpCodes.Ldelem_Ref);
    }
    else if (type == TypeSymbol.Number)
    {
        ilProcessor.Emit(OpCodes.Ldelem_I4);
    }
    else
    {
        ilProcessor.Emit(OpCodes.Ldelem_U1);
    }
}
```

Рисунок 24 – Лістинг функції, яка генерує CIL інструкції доступу до елементу масиву.

Доцільно відзначити, що хоча в даній дисертації і не розглядається доступ до полів об’єкту, ця функціональність буде додана у тестовий транслятор на етапі лексичного та синтаксичного аналізу з метою дозволити реалізувати можливість зчитування розміру масиву.

Тобто, спосіб зчитування розміру масиву полягає в наступному:

1) на етапі семантичного аналізу поле length, яке відповідає за розмір масиву, буде перетворено в окрему вершину дерева (для інших типів буде виводитись помилка про те, що тип не має заданого поля);
2) завантажити масив на стек;
3) транслювати вершину дерева розбору, яка відповідає за розмір масиву, у відповідну інструкцію мови CIL – ldlen.

3.4. Спосіб трансляції умовних конструкцій if-else

Як і кожна мова програмування, Typescript має керуючі конструкції. До них відносяться конструкції if-else та switch. У даній магістерській дисертації було вирішено розглянути саме конструкцію if-else.

БНФ конструкції, що розглядається, має такий вигляд:

<if-statement> --> if (<expression>) <statement>

<if-else-statement> --> if (<expression>) <statement> else <statement>

<statement> --> <block-statement> | <if-statement> | <if-else-statement> | <expression-statement>

<block-statement>--> { <statements-list>? }

<expression-statement> --> <expression>;

<statements-list> --> <statement> | <statement><statements-list>

Мова програма CIL має примітивні інструкції керування потоком команд, а саме:

1) безумовний перехід (br <мітка>);
2) умовний перехід, якщо значення умови true (brtrue<мітка>);
3) умовний перехід, якщо значення умови false (brfalse<мітка>);

З метою полегшення трансляції умовної конструкції <if-else> пропонується після семантичного аналізу переписати відповідні вершини дерева розбору. Основна ідея полягає у тому, щоб ввести поняття оператора переходу, схожого на принцип роботи відповідних операторів у мові CIL. У мові програмування C++, C# та інших є конструкція goto <мітка>. Введемо відповідну конструкцію і отримаємо такі додаткові вершини дерева розбору:

1) LabelStatement – відповідає за оголошення мітки;
2) GotoStatement – відповідає за безумовний перехід до мітки;
3) ConditionalGotoStatement – відповідає за умовний перехід до мітки;

Ідея перетворення дерева полягає в наступному алгоритмі:

1) якщо вираз if-else має частину else перейти до пункту 5, інакше до пункту 2;
2) створити мітку з унікальним ідентифікатором;
3) створити вершину умовного переходу до створеної мітки з протилежною умовою до заданої;
4) створити вершину блоку зі створеними виразами у пунктах 2-3 та виразом тіла if у порядку: оператор переходу, тіло, мітка. Перейти до пункту 10;
5) створити мітку початку гілки else з унікальним ідентифікатором;

6) створити мітку кінця конструкції з унікальним ідентифікатором;
7) створити вершину умовного переходу до мітки else з протилежною умовою до заданої;
8) створити вершину умовного переходу до мітки end;
9) створити вершину блоку зі створеними виразами у пунктах 5-8 та виразами конструкції if-else у порядку: умовний перехід до гілки else, вираз then, перейти до кінця, вираз else, мітка кінець;
10) кінець.

Схематично принцип роботи способу зображено на рисунку 25, а реалізація на мові C# на рисунку 26.

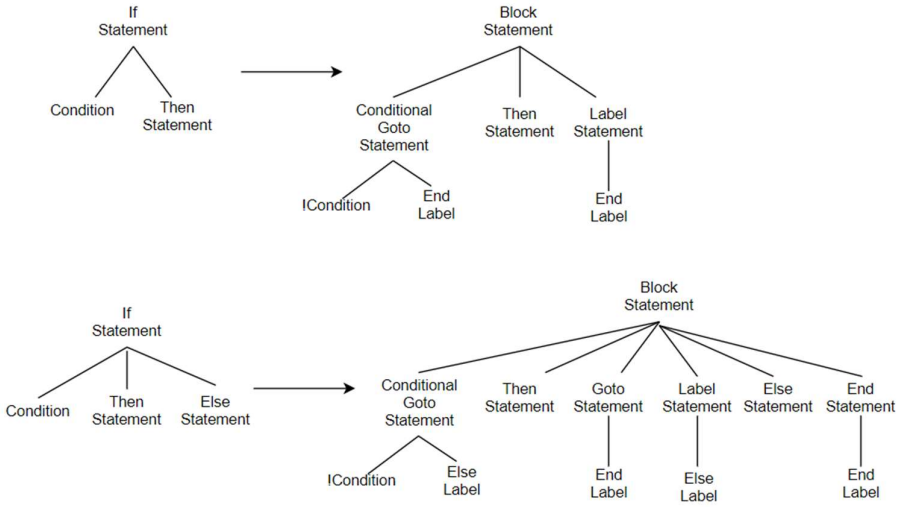

Рисунок 25 – Схематичне зображення перетворення умовної конструкції if-else.

Через те, що в мові CIL іменовані мітки не використовуються, а мітка являє собою індекс інструкції, до якої потрібно перейти, запропонований спосіб трансляції перетвореного дерева програми полягає в наступному:

1) трансляція вершини оголошення мітки записує у словник відповідність імені мітки до індексу поточної інструкції;
2) трансляція безумовного/умовного переходу виконується шляхом додавання до словнику, який зберігає відповідність індексу інструкції поточного оператора goto до імені мітки, генерацією інструкцій для умови, якщо вона наявна та

інструкції br, brtrue чи brfalse в залежності від типу переходу з тимчасовим індексом цільової інструкції;

3) після трансляції підпрограми виконуємо прохід по словнику з пункту 2 та виправляємо відповідні інструкції переходу правильним індексом зі словника, описаного в пункті 1.

```
protected override BoundStatement RewriteIfStatement(BoundIfStatement node)
{
    if (node.ElseStatement == null)
    {
        var endLabel = GenerateLabel();
        var gotoFalseStatement = new BoundConditionalGotoStatement(endLabel, node.Condition, false);
        var endLabelStatement = new BoundLabelStatement(endLabel);
        var result = new BoundBlockStatement(ImmutableArray.Create(
            gotoFalseStatement,
            node.ThenStatement,
            endLabelStatement));
        return RewriteStatement(result);
    }
    else
    {
        var elseLabel = GenerateLabel();
        var endLabel = GenerateLabel();

        var gotoFalseStatement = new BoundConditionalGotoStatement(elseLabel, node.Condition, false);
        var gotoEndStatement = new BoundGotoStatement(endLabel);
        var elseLabelStatement = new BoundLabelStatement(elseLabel);
        var endLabelStatement = new BoundLabelStatement(endLabel);
        var result = new BoundBlockStatement(ImmutableArray.Create(
            gotoFalseStatement,
            node.ThenStatement,
            gotoEndStatement,
            elseLabelStatement,
            node.ElseStatement,
            endLabelStatement));

        return RewriteStatement(result);
    }

}
```

Рисунок 26 – Лістинг функції перетворення умовної конструкції if-else.

На рисунку 27 зображено реалізацію пунктів 1-2 відповідними методами генератора коду.

```
1 reference
private void EmitGotoStatement(ILProcessor ilProcessor, BoundGotoStatement node)
{
    _fixLabels.Add((ilProcessor.Body.Instructions.Count, node.Label));
    ilProcessor.Emit(OpCodes.Br, Instruction.Create(OpCodes.Nop));
}

1 reference
private void EmitConditionalGotoStatement(ILProcessor ilProcessor, BoundConditionalGotoStatement node)
{
    EmitExpression(ilProcessor, node.Condition);

    _fixLabels.Add((ilProcessor.Body.Instructions.Count, node.Label));

    var opCode = node.JumpIfTrue ? OpCodes.Brtrue : OpCodes.Brfalse;
    ilProcessor.Emit(opCode, Instruction.Create(OpCodes.Nop));
}

1 reference
private void EmitLabelStatement(ILProcessor ilProcessor, BoundLabelStatement node)
{
    _labels.Add(node.Label, ilProcessor.Body.Instructions.Count);
}
```

Рисунок 27 – Лістинг функцій, які відповідають за пункт 1-2 описаного способу

3.5. Способи трансляції циклічних конструкцій

Мова Typescript має декілька видів циклічних конструкцій. До них відносяться: цикл з передумовою (while), цикл з післяумовою (do-while), цикл з лічильником (for) та два види циклів по колекції (for-of та for-in). For-of використовується для ітерації по елементам масиву, а for-in – по ключам об’єктів (які в Javascript та Typescript є словниками). Кожен з цих циклів також підтримує конструкцію дострокового виходу з циклу (break) та пропуск ітерації (continue). У даній магістерській дисертації буде розглянуто способи трансляції усіх циклів, окрім циклу for-in, так як способи трансляції об’єктів не розглядаються.

3.5.1. Спосіб трансляції циклу while

БНФ конструкції while має такий вигляд:

<while-statement> --> while (<expression>) <statement>

<loop-statement> --> <while-statement> | <do-while-statement> | <for-statement> | <for-of-statement>

<statement> --> <block-statement> | <if-statement> | <if-else-statement> | <loop-statement> | <expression-statement> | <break-statement> | <continue-statement>

<break-statement> --> break;

<continue-statement> --> continue;

Через те, що для трансляції циклу while буде використано такі самі інструкції CIL, як і для трансляції if-else, то використаймо спосіб схожий на описаний у попередньому розділі. Основна мета – звести трансляцію циклу до вже реалізованої трансляції виразу goto.

До того, як виконати переписування вершини while необхідно на етапі семантичного аналізу спочатку замінити усі break та continue на відповідні вирази goto. Для цього заведемо у семантичному аналізаторі стек для відстеження глибини вкладених циклів. Перед початком аналізу вершини while, генеруємо та додаємо до стеку мітки виходу з циклу та пропуску ітерації. Під час проходу по під-дереву якщо зустрічаємо вираз break чи continue, замінюємо його

на безумовний перехід до відповідної мітки, яку беремо з вершини стеку. При завершені аналізу відповідного циклу, виштовхуємо вершину стека. Після цього, можна приступити до переписування циклу за допомогою виразів умовного та безумовного переходів.

Спосіб перетворення вершини while дерева розбору полягає в наступному:

1) генеруємо вирази оголошення міток для break та continue;
2) створюємо вираз умовного переходу до мітки break з оберненою умовою до умови циклу;
3) створюємо вираз безумовного переходу до мітки продовження ітерації;
4) замінюємо вершину while вершиною блоку виразів, з доданими виразами у такому порядку: оголошення мітки continue, умовний перехід, тіло циклу, безумовний перехід до continue, оголошення мітки break.

Принцип роботи способу зображено на рисунку 28, а реалізація на мові C# – на рисунку 29.

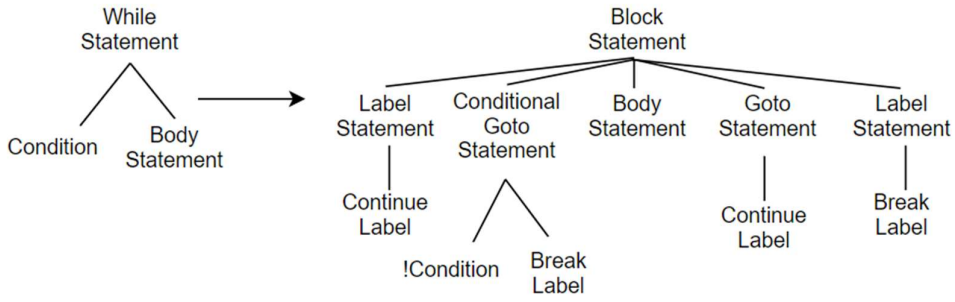

Рисунок 28 – Схематичне зображення перетворення циклу while.

Після перетворення ми отримуємо дерево розбору з вершинами, для яких вже реалізовано трансляцію у інструкції мови CIL, тому трансляцію циклу while можна вважати виконаною.

```
protected override BoundStatement RewriteWhileStatement(BoundWhileStatement node)
{
    var continueLabel = node.ContinueLabel;
    var continueLabelStatement = new BoundLabelStatement(continueLabel);
    var breakLabel = node.BreakLabel;
    var breakLabelStatement = new BoundLabelStatement(breakLabel);

    var gotoBreakIfFalseStatement = new BoundConditionalGotoStatement(breakLabel, node.Condition, false);
    var gotoContinueStatement = new BoundGotoStatement(continueLabel);

    var result = new BoundBlockStatement(ImmutableArray.Create<BoundStatement>(
        continueLabelStatement,
        gotoBreakIfFalseStatement,
        node.Body,
        gotoContinueStatement,
        breakLabelStatement
        ));

    return RewriteStatement(result);
}
```

Рисунок 29 – Лістинг функції перетворення циклу while.

3.5.2. Спосіб трансляції циклу do-while

Цикл з післяумовою має майже ідентичний спосіб трансляції до циклу while. Відмінність буде лише у місці знаходження виразу умовного переходу і мітки до якої він переходить.

БНФ циклічної конструкції, що розглядається має такий вигляд:

<do-while-statement> --> do <statement> while (<expression>)

Спосіб перетворення вершини do-while полягає в наступному:

1) генеруємо вирази оголошення міток для break та continue;
2) створюємо вираз умовного переходу до мітки continue з умови циклу;
3) замінюємо вершину do-while вершиною блоку виразів, з доданими виразами у такому порядку: оголошення мітки continue, тіло циклу, умовний перехід, оголошення мітки break.

Принцип роботи способу зображено на рисунку 30, а реалізація на мові C# – на рисунку 31.

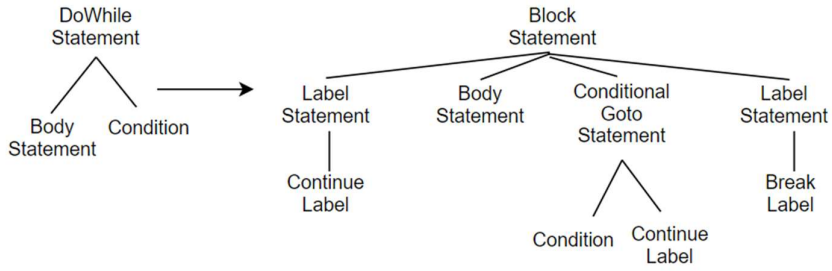

Рисунок 30 – Схематичне зображення перетворення циклу do-while.

```
2 references
protected override BoundStatement RewriteDoWhileStatement(BoundDoWhileStatement node)
{
    var continueStatement = node.ContinueLabel;
    var continueLabelStatement = new BoundLabelStatement(continueStatement);
    var breakLabel = node.BreakLabel;
    var breakLabelStatement = new BoundLabelStatement(breakLabel);

    var gotoContinueIfTrueStatement = new BoundConditionalGotoStatement(continueStatement, node.Condition, true);

    var result = new BoundBlockStatement(ImmutableArray.Create<BoundStatement>(
        continueLabelStatement,
        node.Body,
        gotoContinueIfTrueStatement,
        breakLabelStatement
        ));

    return RewriteStatement(result);
}
```

Рисунок 31 – Лістинг функції перетворення циклу do-while.

3.5.3 Спосіб трансляції циклу for

Трансляція циклу for дещо складніша, ніж у розглянутих конструкцій у попередніх підрозділах. Дане твердження випливає з відповідної БНФ:

<for-statement> --> for (<for-initialize>?; <expression>?; <expression-list>?)
<statement>

<for-initialize> --> <variable-declarations> | < expression-list >

Як бачимо, складність полягає у тому, що оголошення циклу має більше логічних частин у порівнянні з вже розглянутими. І такий цикл одразу перетворити на представлення через goto буде складніше. Тому ми можемо виконати перетворення в декілька етапів. Спочатку перетворити цикл for у відповідний цикл while, а потім - перетворити вже реалізованим способом через вирази goto.

Спосіб перетворення циклу for у while полягає в наступному:

1) створити тіло циклу while з тіла циклу for, оголошення мітки continue та finals виразів, якщо вони наявні;
2) створити цикл while з умовою циклу for, якщо вона наявна, або замінити її на true;
3) створити блок виразів з оголошення змінних/ініціалізаторів циклу for та згенерованого циклу while;
4) застосувати запропонований спосіб перетворення циклу while.

Принцип роботи способу зображено на рисунку 32, а реалізація на мові C# – на рисунку 33.

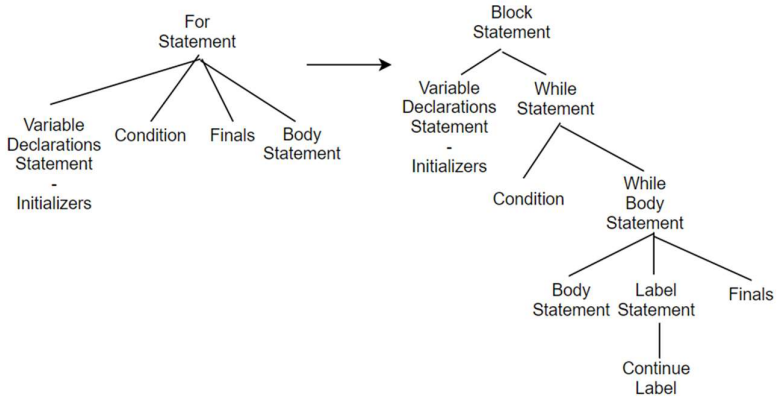

Рисунок 32 – Схематичне зображення перетворення циклу for.

```
protected override BoundStatement RewriteForStatement(BoundForStatement node)
{
    var whileBodyBuilder = ImmutableArray.CreateBuilder<BoundStatement>();
    whileBodyBuilder.Add(node.Body);

    var continueLabelStatement = new BoundLabelStatement(node.ContinueLabel);
    whileBodyBuilder.Add(continueLabelStatement);

    if (node.Finals.HasValue) {
        foreach (var final in node.Finals.Value)
            whileBodyBuilder.Add(new BoundExpressionStatement(final));
    }

    var whileBody = new BoundBlockStatement(whileBodyBuilder.ToImmutable());
    var condition = node.Condition ?? new BoundLiteralExpression(true);
    var whileStatement = new BoundWhileStatement(condition, whileBody, node.BreakLabel, GenerateLabel());

    var forBlockBuilder = ImmutableArray.CreateBuilder<BoundStatement>();
    if (node.VariableDeclaration != null)
        forBlockBuilder.Add(node.VariableDeclaration);
    if (node.Initializers.HasValue) {
        foreach (var initializer in node.Initializers.Value)
            forBlockBuilder.Add(new BoundExpressionStatement(initializer));
    }

    forBlockBuilder.Add(whileStatement);
    var result = new BoundBlockStatement(forBlockBuilder.ToImmutable());

    return RewriteStatement(result);
}
```

Рисунок 33 – Лістинг функції перетворення циклу for.

3.5.4. Спосіб трансляції циклу for-of

Розглянемо спосіб трансляції циклу по ітерації по елементам масиву. БНФ циклічної конструкції має такий вигляд:

<for-of-statement> --> for(<declaration-keyword> <identifier> of <expression>) <statement>

<declaration-keyword> --> var | let | const

Є два основних підходи до перетворення цього циклу. Перетворити у цикл while, а потім переписати через goto, або виконати перетворення спочатку у цикл for.

У даній магістерській дисертації було вирішено перетворити спочатку у цикл for, так як цей підхід має більшу наочність. Час трансляції при такому підході буде більше, але на зовсім невелику різницю.

Спосіб перетворення циклу for-of у for полягає в наступному:

1) якщо <expression> не є змінною, створити вершину декларації локальної змінної з цим значенням;
2) створити вершину декларації лічильника циклу;
3) створити умову з виразом значення лічильнику менше за розмір масиву;
4) створити вираз збільшення лічильника на 1;
5) створити тіло циклу for, як зчитування елементу масиву за індексом у змінну та тіло циклу for-or;
6) створити цикл for з відповідними параметрами;
7) створити блок виразів: декларація локальної змінної (якщо наявна), цикл for;
8) перетворити цикл for;
9) перетворити цикл while;

Принцип роботи способу зображено на рисунку 34, а реалізація на мові C# – на рисунку 35.

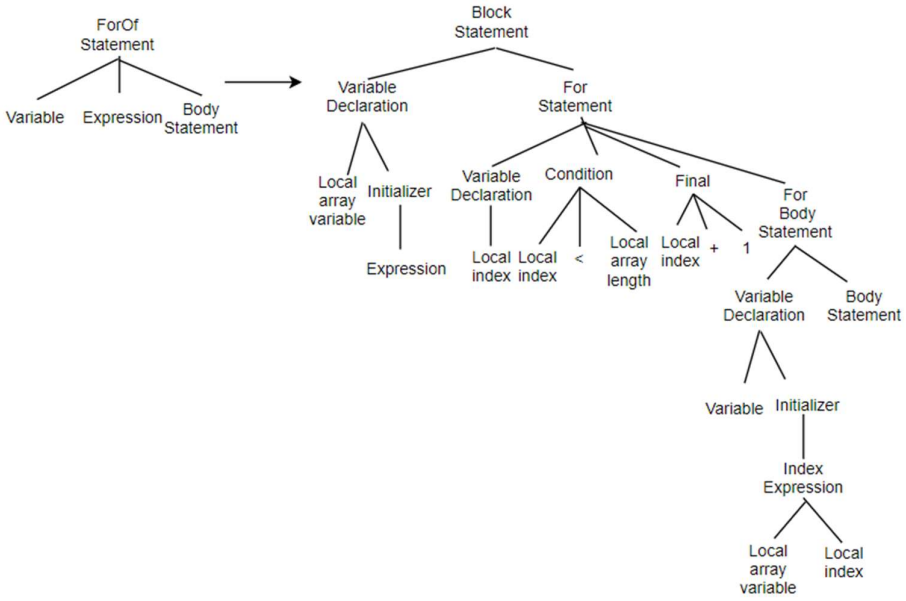

Рисунок 34 – Схематичне зображення перетворення циклу for-of.

```
protected override BoundStatement RewriteForOfStatement(BoundForOfStatement node)
{

    var blockBuilder = ImmutableArray.CreateBuilder<BoundStatement>();
    if (node.Expression is BoundVariableExpression arrExpr)
    {

    }
    else
    {
        var arrVar = new LocalVariableSymbol("$arr", true, node.Expression.Type);
        var arrDecl = new BoundVariableDeclaration(arrVar, node.Expression);
        blockBuilder.Add(arrDecl);

        arrExpr = new BoundVariableExpression(arrVar);
    }

    var indexVar = new LocalVariableSymbol("$index", false, TypeSymbol.Number);
    var indexDecl = new BoundVariableDeclaration(indexVar, new BoundLiteralExpression(0));
    var lessOp = BoundBinaryOperator.Bind(Syntax.SyntaxKind.LessToken, indexVar.Type, TypeSymbol.Number);
    var addOp = BoundBinaryOperator.Bind(Syntax.SyntaxKind.PlusToken, indexVar.Type, TypeSymbol.Number);
    var condition = new BoundBinaryExpression(new BoundVariableExpression(indexVar), lessOp,
        new BoundArrayLengthExpression(arrExpr));
    var final = new BoundAssignmentExpression(new BoundVariableExpression(indexVar),
        new BoundBinaryExpression(new BoundVariableExpression(indexVar), addOp, new BoundLiteralExpression(1)));
    var varDecl = new BoundVariableDeclaration(node.Variable, new BoundIndexAccessExpression(arrExpr,
        new BoundVariableExpression(indexVar)));
    var forBody = new BoundBlockStatement(ImmutableArray.Create(varDecl, node.Body));
    var forStatement = new BoundForStatement(indexDecl, null, condition,
        ImmutableArray.Create<BoundExpression>(final), forBody, node.BreakLabel, node.ContinueLabel);

    blockBuilder.Add(forStatement);

    return RewriteStatement(new BoundBlockStatement(blockBuilder.ToImmutable()));
}
```

Рисунок 35 – Лістинг функції перетворення циклу for-of.

3.6. Спосіб трансляції глобальних функцій

Мова програмування Typescript також підтримує роботу з функціями, а саме з:

- глобальними функціями;
- локальними функціям;
- лямбда функціями;
- методами класів.

У даній магістерській дисертації буде розглянуто саме спосіб трансляції глобальних функцій. БНФ оголошення функцій має наступний вигляд:

<compilation-unit> --> <function-declaration>

<function-declaration> --> function <identifier> (<paramerter-list>?)<type-annotation>? <block-statement>

<parameter-list> --> <parameter> | <parameter>,<parameter-list>

<parameter> --> <identifier><type-annotation>?

<block-statement>--> { <statements-list>? }

<statements-list> --> <statement> | <statement><statements-list>

<statement> --> <expression-statement> | <if-statement> | <if-else-statement> | <loop-statement> | <expression-statement> | <break-statement> | <continue-statement> | <return-statement>

<return-statement> --> return <expression>?;

<expression-statement> --> <expression>;

<expression> --> <call-expression> | <assignment-expression> | <binary-expression>

<call-expression> --> <identifier>(<argument-list>?)

<argument-list> --> <argument> | <argument>, <argument-list>

<argument> --> <expression>

Варто зазначити, що з форми видно, що розглядаються функції зі звичайними параметрами. Функції з параметрами за замовчуванням, та залишковими (rest) параметрами розглядатись не будуть, так як вид параметру не впливає на кінцевий результат, тобто – CIL інструкції.

Для того, щоб виконати трансляцію декларації функції, спочатку виконується лексичний і синтаксичний аналіз введеної програми.

На етапі семантичного аналізу перевіряється коректність оголошених параметрів, коректність типу, що повертається, а також заноситься ім’я функції до таблички ідентифікаторів.

Мова програмування CIL зобов’язує, щоб функція повертала значення в усіх гілках потоку керування, тому на етапі семантичного аналізу доведеться також виконати цю перевірку і попередити користувача про помилку. Інакше, на етапі запуску CLR виконає перевірку та виведете помилку, а виконуваний файл не запуститься.

Для аналізу того, що функція завжди повертає значення, для тіла функції виконаємо побудову графу потоку керування (Control-flow Graph).

Граф потоку керування — множина всіх можливих шляхів виконання програми, представлених у вигляді графу.

Для того, щоб виконати побудову CGF, будемо вважати, що усі вирази окрім виразів переходу, які після переписування у дереві розбору представлені у вигляді виразів goto та міток, будуть відноситись до вершин графу. Умовні вирази будуть відповідати за ребра графу (зі значенням відповідної умови переходу до наступного блоку).

Для тестування коректності побудови графа, було реалізовано експорт у формат dot. DOT – це мова опису графів. Для перегляду файлів було використано програму Graphviz.

Розглянемо граф для тіл двох функцій (рисунок 36). Зліва зображено функцію, яка повертає результат не у всіх випадках, а справа – повертає у всіх випадках. Як бачимо з рисунку, у правильному випадку, усі блоки, які входять у кінцевий, повинні закінчуватись на вираз return.

Тому алгоритм перевірки полягає у наступному:

1) виконується побудова CGF;
2) якщо вхідні ребра відсутні, або вхідна вершина лише початкова, перейти до пункту 5;

3) для кожної вхідної гілки у кінцеву вершину, перевіряємо блок. Якщо хоча б один блок не завершується на return, перейти до пункту 5;
4) повертаємо true;
5) повертаємо false;

Реалізація алгоритму зображена на рисунку 37.

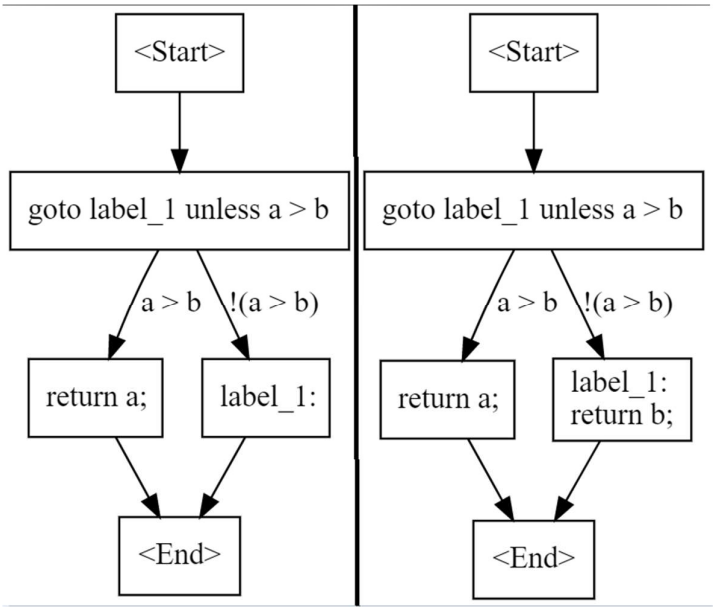

Рисунок 37 – Приклад CGF для неправильної функції (зліва), та правильної (справа)

```
public static bool HasAllReturns(BoundBlockStatement body)
{
    var graph = Create(body);

    if (graph.End.Incoming.Count == 0 || graph.End.Incoming.Count == 1
        && graph.End.Incoming.First().From.IsStart)
        return false;

    foreach (var branch in graph.End.Incoming)
    {
        var lastStatement = branch.From.Statements.Last();
        if (lastStatement.Kind != BoundNodeKind.ReturnStatement)
        {
            return false;
        }
    }

    return true;
}
```

Рисунок 38 – Лістинг функції, яка перевіряє тіло функції на те, що всі кінцеві гілки потоку керування повертають значення

Після семантичного аналізу можемо перейти до трансляції у проміжну мову CIL.

Через відсутність у мові CIL глобальних функцій, ідея трансляції полягає у створенні приватних статичних методів класу Program. Отже, спосіб трансляції декларації функції полягає в наступному:

1) знайти відповідність типу, що повертається, до типу платформи .NET
2) створити MethodDefinition з ім’ям функції та атрибутами static і private
3) для кожного параметру функції, знайти відповідність типу та створити ParameterDefintion;
4) додати ParameterDefinition до MethodDefinition;
5) запам’ятати відповідність згенерованого методу до заданої функції;

Реалізацію способу на мові C# зображено на рисунку 39.

```
private void EmitFunctionDeclaration(FunctionSymbol function)
{
    var clrType = ResolveCLRType(function.Type);
    var method = new MethodDefinition(function.Name, MethodAttributes.Static | MethodAttributes.Private, clrType)

    foreach (var param in function.Parameters)
    {
        var typeRef = ResolveCLRType(param.Type);
        var paramDef = new ParameterDefinition(param.Name, ParameterAttributes.None, typeRef);
        method.Parameters.Add(paramDef);
    }

    _programTypeDefinition.Methods.Add(method);
    _methods.Add(function, method);
}
```

Рисунок 39 – Лістинг функції, яка генерує декларацію глобальної функції на мові CIL

Спосіб трансляції імплементації функції налічує такі кроки:

1) очистити локальні змінні та мітки;
2) виконати трансляцію тіла функції у IL процесор згенерованого методу;
3) виправити мітки, як описано у підрозділі про конструкцію if-else;

Реалізацію способу на мові C# зображено на рисунку 40.

Також у цьому розділі розглянемо спосіб трансляції виклику функції. Варто зазначити, що до тестового транслятора було додано такі вбудовані

функції: alert(message: string) та prompt(): string для того, щоб можливо було реалізувати введення і виведення результату в консоль.

```
private void EmitFunctionImplementation(FunctionSymbol function, BoundBlockStatement body
{
    _locals.Clear();
    _fixLabels.Clear();
    _labels.Clear();

    var method = _methods[function];
    var ilProcessor = method.Body.GetILProcessor();

    foreach(var statement in body.Statements)
        EmitStatement(ilProcessor, statement);

    // replaces out internal labels to
    // IL labels
    foreach (var fix in _fixLabels)
    {
        var targetInstructionIndex = _labels[fix.Target];
        var targetInstruction = method.Body.Instructions[targetInstructionIndex];
        var instructionToFix = method.Body.Instructions[fix.InstructionIndex];
        instructionToFix.Operand = targetInstruction;
    }

    method.Body.OptimizeMacros();
}
```

Рисунок 40 – Лістинг функції, яка генерує імплементацію глобальної функції на мові CIL

У платформі .NET, вбудованому методу alert відповідає Console.Write, а методу prompt – Console.ReadLine.

Спосіб трансляції виклику функції полягає в наступному:

1) завантажити всі аргументи на стек;
2) якщо функція не вбудована, знайти відповідний метод, інакше взяти відповідну функцію платформи .NET;
3) згенерувати інструкцію call для відповідного методу.

Реалізацію способу на мові C# зображено на рисунку 41.

```
private void EmitCallExpression(ILProcessor ilProcessor, BoundCallExpression node)
{
    foreach (var argument in node.Arguments)
        EmitExpression(ilProcessor, argument);

    if (node.Function == BuildInFunctions.Alert)
    {
        ilProcessor.Emit(OpCodes.Call, _consoleWriteReference);
    }
    else if (node.Function == BuildInFunctions.Prompt)
    {
        ilProcessor.Emit(OpCodes.Call, _consoleReadLineReference);
    }
    else
    {
        var method = _methods[node.Function];
        ilProcessor.Emit(OpCodes.Call, method);
    }
}
```

Рисунок 41 – Лістинг функції, яка транслює виклик функції у мову CIL.

ВИСНОВКИ ДО РОЗДІЛУ 3

У даному розділі було запропоновано способи трансляції певної підмножини мови програмування Typescript у проміжну мову CIL платформи .NET, а саме:

1) спосіб трансляції декларації змінних, простих типів даних та операцій над ними;
2) спосіб трансляції конкатенації рядкових виразів;
3) спосіб трансляції масивів;
4) спосіб трансляції умовних конструкцій if-else;
5) способи трансляції таких циклічних конструкцій: while, do-while, for, for-of;
6) спосіб трансляції глобальних функцій;

При аналізі трансляції операцій над типами даних визначено, що у деяких випадках не достатньо однієї CIL інструкції для виконання трансляції операції. До таких операторів належать: “!=”, “>=”, “<=” для типів number, boolean та всі оператори для типу string.

Особливу увагу приділено способу трансляції конкатенації рядкових виразів (оператор “+” для типу string). Він базується на виклику вбудованої у платформу .NET функції – String.Concat для певної кількості аргументів.

Для трансляції масивів було запропоновано способи трансляції таких функціональних можливостей роботи з ними:

- оголошення змінних типу масивів;
- виділення пам’яті для масивів;
- ініціалізація масивів початковими елементами;
- доступ до комірки масиву;

Способи трансляції умовних та циклічних конструкцій базуються на перетворенні проміжного подання програми через штучно введений вираз goto та мітки. Саме цей вираз є семантично найбільш близьким до умовних інструкцій мови CIL. Варто зазначити, що для for та for-of було вирішено виконати поетапне

перетворення, тобто спочатку у цикл while, а вже потім переписати через вираз goto.

Через відсутність у мові CIL глобальних функцій, було вирішено виконати трансляцію у приватні статичні методи класу Program. Також для трансляції було реалізовано перевірку на те, що функція повертає значення у всіх випадках за допомогою побудови та аналізу графа потоку команд. Для тестування написаних програм на мові Typescript, які транслюються у мову CIL, було додано трансляцію вбудованих функцій alert та prompt для взаємодії з консоллю на етапі генерації CIL інструкцій.

4. ТРАНСЛЯТОР ПІДМНОЖИНИ МОВИ TYPESCRIPT НА ОСНОВІ ЗАПРОПОНОВАНИХ СПОСОБІВ

Для підтвердження ефективності запропонованих способів трансляції мови Typescript у проміжну мову CIL платформи .NET було розроблено тестовий транслятор на мові C# з використанням бібліотеки Mono.Cecil. Архітектура транслятора була спроектована на основі аналізу таких джерел [1, 4, 8, 10, 13].

4.1. Структура транслятора

Всю логіку трансляції мови програмування Typescript було вирішено винести в окремий модуль (бібліотеку), щоб мати можливість на основі реалізованого лексичного, синтаксичного та семантичного аналізаторів у подальшому створити розширення для IDE з розумними підказками, автоматичним рефакторингом певних конструкцій, тощо.

В основній програмі реалізована підтримка консольних команд, а також створено простий REPL для тестування та налагодження роботи певних частин транслятора. Виконуваний файл транслятора має назву tsnet (від TypeScript та .NET).

На даний момент серед команд підтримується build. На вхід, як аргументи, їй подаються вхідні файли, що містять програму, написану на мові Typescript. Також вона приймає такі опції:

- --reference | -r <reference> – посилання на залежну бібліотеку;
- --target | -t <target> – цільова платформа. Підтримуються такі значення: net5.0, netcoreapp3.1, .net4.6.1 та інші;
- --output | -o <path> – назва та шлях вихідного файлу.

Якщо запустити транслятор без виклику команди, відкриється інтерактивне середовище для тестування роботи транслятора (рисунок 42). До речі, на рисунку можна помітити приклад роботи підсвітки синтаксису, реалізованої на основі роботи лексичного аналізатора.

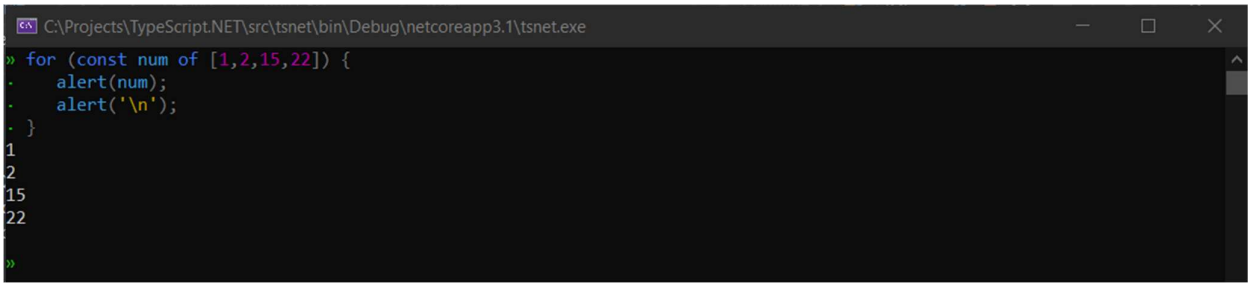

Рисунок 42 – Приклад роботи REPL.

Модуль трансляції мови Typescript у проміжну мову CIL складається з таких основних компонентів: лексичний аналізатор, синтаксичний аналізатор, семантичний аналізатор, оптимізатор та генератор коду. Схема роботи тестового транслятора зображена на рисунку 43.

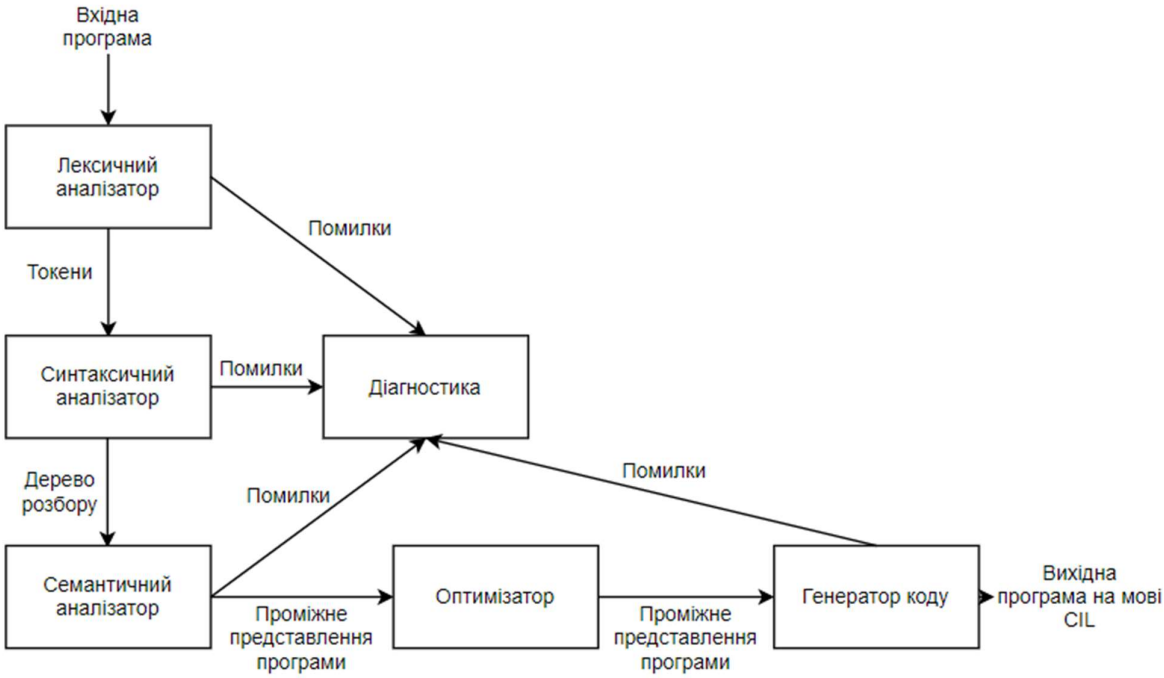

Рисунок 43 – Схема роботи тестового транслятора.

4.1.1. Лексичний аналізатор

За лексичний аналіз вхідної програми відповідає клас Lexer. Результатом його роботи є масив токенів (SyntaxToken).

Lexer має такі методи:

- Lex() – повертає один наступний токен;
- ReadAllTokens() – зчитує всі токени та повертає у вигляді перелічення;
- ReadWhiteSpace() – зчитує пробіли, поки поточний символ є пробілом;
- ReadStringToken(char closeChar) – зчитує строкову константу, отримує на вхід символ, який означає кінець строки (так як у мові Typescript строки можуть буди виділені одинарними чи подвійними лапками);
- ReadNumberToken() – зчитує числову константу;
- ReadIdentifierOrKeyWord() – зчитує ідентифікатор чи ключове слово;

Лексичний аналізатор представляє собою скінченний автомат. Приклад часткової реалізації автомату за допомогою конструкції switch зображено на рисунку 44.

```
case var _ when char.IsDigit(Current):
    ReadNumberToken();
    break;
case var _ when char.IsWhiteSpace(Current):
    ReadWhiteSpace();
    break;
case var _ when char.IsLetterOrDigit(Current):
    ReadIdentifierOrKeyWord();
    break;
default:
    var location = new TextLocation(_text, new TextSpan(_start, 1));
    Diagnostics.ReportBadCharacter(location, Current);
    Next();
    break;
```

Рисунок 44 – Лістинг на мові C# частини станів лексичного аналізатора.

4.1.2. Синтаксичний аналізатор

Результат роботи лексичного аналізатора (набір токенів) передається в синтаксичний аналізатор, який виконує аналіз на синтаксичні помилки та будує дерево розбору.

У даній магістерській дисертації для реалізації синтаксичного аналізу було обрано рекурсивний низхідний алгоритм. Ідея цього алгоритму полягає у тому, що кожному правилу граматики ставиться у відповідність певна функція, яка відповідає за аналіз та побудову вершини дерева для відповідного правила. Якщо

правило містить у собі інші правила, для цих правил викликаються відповідні функції.

У тестовому трансляторі за синтаксичний аналіз відповідає клас Parser. Приклад функцій, які він містить, зображено на рисунку 45.

```
74 references
private SyntaxToken MatchToken(SyntaxKind kind)...

1 reference
private SyntaxToken EatToken(SyntaxKind kind)...

10 references
private SyntaxToken NextToken()...

8 references
private StatementSyntax ParseStatement()...

1 reference
private StatementSyntax ParseReturnStatement()...

1 reference
private StatementSyntax ParseBreakStatement()...

1 reference
private StatementSyntax ParseContinueStatement()...

1 reference
private StatementSyntax ParseEmptyStatement()...

1 reference
private StatementSyntax ParseForOrForOfStatement()...

1 reference
private StatementSyntax ParseForOfStatement()...

1 reference
private StatementSyntax ParseForStatement()...

2 references
private SeparatedSyntaxList<ExpressionSyntax> ParseExpressionList()...

1 reference
private StatementSyntax ParseWhileStatement()...

1 reference
private StatementSyntax ParseDoWhileStatement()...

1 reference
private StatementSyntax ParseIfStatement()...
```

Рисунок 45 –Приклад певної частини методів, які містить Parser.

Розглянемо приклад роботи синтаксичного аналізатора на розборі циклу while. Правило граматики цієї синтаксичної конструкції було описано у попередньому розділі (пункт 3.5.1).

Лістинг функції, яка відповідає за аналіз цієї синтаксичної конструкції зображено на рисунку 46, а приклад відповідної вершини дерева розбору на рисунку 47.

```
1 reference
private StatementSyntax ParseWhileStatement()
{
    var keyword = MatchToken(SyntaxKind.WhileKeyword);
    var openParenthesis = MatchToken(SyntaxKind.OpenParenthesisToken);
    var condition = ParseExpression();
    var closeParenthesis = MatchToken(SyntaxKind.CloseParenthesisToken);
    var body = ParseStatement();
    return new WhileStatementSyntax(_syntaxTree, keyword, openParenthesis, condition, closeParenthesis, body);
}
```

Рисунок 46 – Лістинг функції на мові C#, яка відповідає за синтаксичний аналіз циклу while.

```
4 references
public sealed class WhileStatementSyntax : StatementSyntax
{
    1 reference
    public WhileStatementSyntax(SyntaxTree syntaxTree,
                                SyntaxToken whileKeyword,
                                SyntaxToken openParenthesis,
                                ExpressionSyntax condition,
                                SyntaxToken closeParenthesis,
                                StatementSyntax body)
        : base(syntaxTree)
    {
        WhileKeyword = whileKeyword;
        OpenParenthesis = openParenthesis;
        Condition = condition;
        CloseParenthesis = closeParenthesis;
        Body = body;
    }

    99+ references
    public override SyntaxKind Kind => SyntaxKind.WhileStatement;

    1 reference
    public SyntaxToken WhileKeyword { get; }
    1 reference
    public SyntaxToken OpenParenthesis { get; }
    2 references
    public ExpressionSyntax Condition { get; }
    1 reference
    public SyntaxToken CloseParenthesis { get; }
    2 references
    public StatementSyntax Body { get; }
}
```

Рисунок 47 – Лістинг класу на мові C#, який відповідає за представлення вершини дерева розбору циклу while.

4.1.3. Семантичний аналізатор

Результат роботи синтаксичного аналізатора (дерево розбору) подається на вхід до семантичного аналізатора.

Семантичний аналізатор працює за схожим принципом до синтаксичного аналізатора. За допомогою низхідного рекурсивного алгоритму виконується аналіз кожної вершини дерева розбору. Для відповідної вершини існує певна функція,

поставлена у відповідність. В результаті будується нове дерево розбору, яке відповідає за проміжне представлення програми. Воно вже не містить інформацію про синтаксис та токени, а містить лише необхідну частину для виконання генерації коду.

За реалізацію семантичного аналізу відповідає клас Binder. Приклад певної частини його методів зображено на рисунку 48.

```
6 references
private BoundStatement BindStatement(StatementSyntax syntax, bool isGlobal = false)...

1 reference
private BoundStatement BindStatementInternal(StatementSyntax syntax)...

1 reference
private BoundStatement BindReturnStatement(ReturnStatementSyntax syntax)...

4 references
private BoundStatement BindLoopBody(StatementSyntax body, out BoundLabel breakLabel, out BoundLabel continueLabel)...

1 reference
private BoundStatement BindBreakStatement(BreakStatementSyntax syntax)...

1 reference
private BoundStatement BindContinueStatement(ContinueStatementSyntax syntax)...

3 references
private BoundStatement BindErrorStatement()...

1 reference
private BoundStatement BindEmptyStatement(EmptyStatementSyntax _)...

1 reference
private BoundStatement BindForOfStatement(ForOfStatementSyntax syntax)...

1 reference
private BoundStatement BindForStatement(ForStatementSyntax syntax)...

1 reference
private BoundStatement BindWhileStatement(WhileStatementSyntax syntax)...

1 reference
private BoundStatement BindDoWhileStatement(DoWhileStatementSyntax syntax)...

1 reference
private BoundStatement BindIfStatement(IfStatementSyntax syntax)...

2 references
private BoundStatement BindVariableDeclarationList(VariableDeclarationSyntax syntax)...
```

Рисунок 48 – Приклад певної частини методів, які містить Binder.

Розглянемо приклад роботи синтаксичного аналізатора на попередньому прикладі (аналіз циклу while). Лістинг функції, яка відповідає за аналіз зображено на рисунку 49, а приклад відповідної вершини дерева розбору на рисунку 50.

```
1 reference
private BoundStatement BindWhileStatement(WhileStatementSyntax syntax)
{
    var condition = BindConversion(syntax.Condition, TypeSymbol.Boolean);
    var body = BindLoopBody(syntax.Body, out var breakLabel, out var continueLabel);
    return new BoundWhileStatement(condition, body, breakLabel, continueLabel);
}

4 references
private BoundStatement BindLoopBody(StatementSyntax body, out BoundLabel breakLabel, out BoundLabel continueLabel)
{
    breakLabel = new BoundLabel($"break_{_labelCounter}");
    continueLabel = new BoundLabel($"continue_{_labelCounter}");
    _labelCounter++;

    _loopStack.Push((breakLabel, continueLabel));
    var result = BindStatement(body);
    _loopStack.Pop();
    return result;
}
```

Рисунок 49 – Лістинг функцій на мові C#, які відповідає за семантичний аналіз циклу while.

```
9 references
internal class BoundWhileStatement : BoundLoopStatement
{
    3 references
    public BoundWhileStatement(
        BoundExpression condition,
        BoundStatement body,
        BoundLabel breakLabel, BoundLabel continueLabel) : base(breakLabel, continueLabel)
    {
        Condition = condition;
        Body = body;
    }

    57 references
    public override BoundNodeKind Kind => BoundNodeKind.WhileStatement;

    5 references
    public BoundExpression Condition { get; }
    5 references
    public BoundStatement Body { get; }
}
```

Рисунок 50 – Лістинг класу на мові C#, який відповідає за проміжне представлення циклу while.

4.1.4. Оптимізатор

В реалізації тестового транслятора оптимізатор виконує перетворення для конструкцій, описаних у попередньому розділі, тобто для if-else, while, do-while, for та for-of. У подальшому, в цій частині транслятора можливо буде реалізувати згортку констант, видалення недосяжного коду та інші оптимізації.

Функції оптимізатора виконує клас Lowerer. Він налічує такі методи:

- Lower(FunctionSymbol, BoundStatement) – виконує оптимізації до тіла переданої функції;
- RewriteIfStatement(BoundIfStatement) – переписує умовну конструкцію if-else за алгоритмом, описаним у пункті 3.4;
- RewriteWhileStatement(BoundWhileStatement) – переписує цикл while за алгоритмом, описаним у пункті 3.5.1;
- RewriteDoWhileStatement(BoundDoWhileStatement) – переписує цикл do-while за алгоритмом, описаним у пункті 3.5.2;
- RewriteForStatement(BoundForStatement) – переписує цикл for за алгоритмом, описаним у пункті 3.5.3;
- RewriteForOfStatement(BoundForStatement) – переписує цикл for-of за алгоритмом, описаним у пункті 3.5.4;
- GenerateLabel() – генерує унікальне ім’я мітки.

4.1.5. Генератор коду

Результат роботи семантичного аналізатора та оптимізатора передається на вхід генератора коду. У даному трансляторі роль генератора коду виконує клас Emitter. Структура класу схожа до структури Parser та Binder, тобто за генерацію коду для кожної вершини дерева відповідає певна функція (рисунок 51). Повна реалізація генератора коду міститься у додатку 4.

```
1 reference
private void EmitExpressionStatement(ILProcessor ilProcessor, BoundExpressionStatement node)...

32 references
private void EmitExpression(ILProcessor ilProcessor, BoundExpression node)...

1 reference
private void EmitArrayLengthExpression(ILProcessor ilProcessor, BoundArrayLengthExpression node)...

1 reference
private void EmitArrayInitializationExpression(ILProcessor ilProcessor, BoundArrayInitializationExpression node)...

1 reference
private void EmitArrayCreationExpression(ILProcessor ilProcessor, BoundArrayCreationExpression node)...

1 reference
private void EmitInstanceOfExpression(ILProcessor ilProcessor, BoundInstanceOfExpression node)...

1 reference
private void EmitLiteralExpression(ILProcessor ilProcessor, BoundLiteralExpression node)...

1 reference
private void EmitVariableExpression(ILProcessor ilProcessor, BoundVariableExpression node)...

1 reference
private void EmitAssignmentExpression(ILProcessor ilProcessor, BoundAssignmentExpression node)...

1 reference
private void EmitIndexAccessExpression(ILProcessor ilProcessor, BoundIndexAccessExpression node)...

1 reference
private void EmitUnaryExpression(ILProcessor ilProcessor, BoundUnaryExpression node)...

1 reference
private void EmitBinaryExpression(ILProcessor ilProcessor, BoundBinaryExpression node)...

1 reference
private void EmitStringConcatExpression(ILProcessor ilProcessor, BoundBinaryExpression node)...

1 reference
private void EmitConversionExpression(ILProcessor ilProcessor, BoundConversionExpression node)...

1 reference
private void EmitCallExpression(ILProcessor ilProcessor, BoundCallExpression node)...
```

Рисунок 51 –Приклад певної частини методів, які містить Emitter.

Варто зазначити, що для REPL було реалізовано інтерпретатор замість генератору коду, який виконує проміжне представлення програми методами мови C#. За виконання даної функціональності відповідає клас Evaluator. На рисунку 52 зображено принцип роботи цього класу на прикладі інтерпретації створення масиву.

```
1 reference
private object EvaluateArrayCreationExpression(BoundArrayCreationExpression node)
{
    var size = (int)EvaluateExpression(node.Size);
    var type = ResolveCLRType(node.ElementType);

    return Array.CreateInstance(type, size);
}
```

Рисунок 52 – Створення масиву інтерпретатором Evaluator.

4.2. Інтеграція з утилітою dotnet

Утиліта dotnet постачається разом з .NET Runtime. Вона виконує дві основні функції:

1. Надає команди для роботи з проектами .NET. Наприклад, команда dotnet build виконує збірку проекту. Кожна команда має свої параметри та аргументи. Всі команди підтримують параметр --help, дозволяючи вивести довідку про команди.
2. Запускає додатки для платформи .NET. Для запуску програми необхідно вказати шлях до його файлу .dll. Щоб запустити додаток, необхідно знайти і виконати точку входу, яка в разі використання консольних додатків є методом Main. Наприклад, команда dotnet myapp.dll запускає додаток myapp.

Команда dotnet build підтримує роботу с загальною системою проектів (CPS), розробленою для IDE Visual Studio, що дозволяє дуже зручно налаштовувати збірки різних проектів. Для кожної мови платформи .NET існує спеціальний тип проекту: для мови C# – csproj, VisualBasic – vbproj, F# – fsproj, тощо.

Таким чином команда dotnet build підтримує парсинг файлів проектів і автоматично знаходить всі потрібні залежності. Ми можемо використати це у тестовому трансляторі. Зараз користувачу необхідно власноруч передати залежності через спеціальні опції, але цей недолік буде усунутий.

Створимо у корені папки, де будуть зберігатись проекти Typescript два файли: Directory.build.props та Directory.build.targets. У подальшому зміст цих файлів можливо буде зробити частиною SDK розроблюваного транслятора.

У файлі Directory.build.props вкажемо тип файлів, які підтримуватимуть проекти. В даному випадку це ts. Зміст файлу матиме такий вигляд:

```
<Project>
        <PropertyGroup>
          <DefaultLanguageSourceExtension>.ts</DefaultLanguageSourceExtension>
        </PropertyGroup>
</Project>
```

Зміст файлу Directory.build.targets буде дещо складніший:

```
<Project>
    <Target Name="CreateManifestResourceNames" />
    <Target Name="CoreCompile" DependsOnTargets="$(CoreCompileDependsOn)">
        <Exec Command="dotnet run tsnet.dll -- build @(Compile->'&amp;quot;%(Identity)&amp;quot;', '
') -t:&amp;quot;$(TargetFramework)&amp;quot; -o:&amp;quot;@(IntermediateAssembly)&amp;quot;
@(ReferencePath->'-r:&amp;quot;%(Identity)&amp;quot;', ' ')"
            WorkingDirectory="$(MSBuildProjectDirectory)" />
    </Target>
</Project>
```

Цей файл вказує, що команда dotnet build повинна викликати компілятор tsnet та передати йому файли, які потрібно скомпілювати у вигляді аргументів. Також будуть передані залежності та цільова платформа у вигляді опцій.

Тепер розглянемо проект мови Typescript на прикладі програми, що виводить “Hello world!”. Створимо каталог HelloWorld та проект HelloWorld.tsproj.

Зміст проекту наступний:

```
<Project Sdk="Microsoft.NET.Sdk">
        <PropertyGroup>
            <TargetFramework>net5.0 </TargetFramework>
            <OutputType>exe</OutputType>
        </PropertyGroup>
</Project>
```

Тут вказано, шо проект створюється для платформи .NET 5, а результатом збірки проекту буде виконуваний exe файл. Зміст файлу main.ts зображено на рисунку 53.

```
function main() {
    alert('Hello world!')
}
```

Рисунок 53 –Зміст файлу main.ts.

Запустимо команду dotnet build в каталозі HelloWorld. У результаті бачимо створені каталоги bin та obj. В каталозі bin\Debug\net5.0 міститься тестова програма (рисунок 54). Файл Helloworld.dll є результатом роботи тестового транслятора, який розроблено для дисертації.

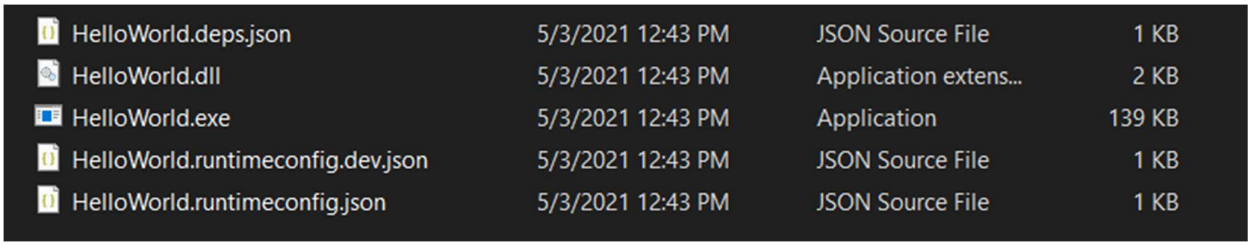

Рисунок 54 – Зміст каталогу, що містить зібрану програму Hello world.

Запустимо файл HelloWorld.exe та побачимо результат виконання програми у консолі (рисунок 55).

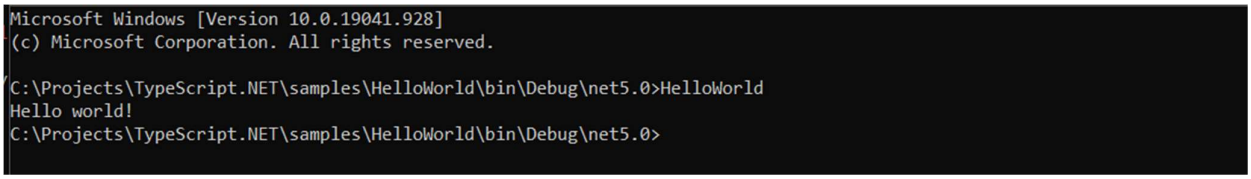

Рисунок 55 – Приклад роботи скомпільованої програми Hello world.

Як бачимо, все працює правильно. Інтеграція тестового транслятора с утилітою dotnet виконана успішно.

ВИСНОВКИ ДО РОЗДІЛУ 4

У даному розділі описано створення тестового транслятора мови програмування Typescript у проміжну мову CIL платформи .NET, виконаного на основі запропонованих способів у попередньому розділі.

Реалізацію транслятора було вирішено розділити на два модулі: модуль з консольною утилітою та модуль, який містить реалізацію трансляції. Це виконано з метою у подальшому використати частини транслятора для реалізації модифікацій для різних IDE, створення аналізатора коду, тощо.

Трансляція складається з таких етапів: лексичний аналіз, синтаксичний аналіз, семантичний аналіз, оптимізація та генерація коду.

Лексичний аналізатор виконано у вигляді кінцевого автомату.

Синтаксичний аналізатор реалізовано з використанням низхідного рекурсивного алгоритму, основна ідея якого полягає у тому, що кожному правилу граматики відповідає певна функція.

На відповідному підході побудовано семантичний аналізатор та генератор коду.

Варто зазначити, що також було реалізовано інтерпретатор, що виконує проміжне представлення програми методами мови C# для тестування усіх частин транслятора перед реалізацією генератора коду.

З метою перекласти відповідальність введення залежних бібліотек під час трансляції з користувача на програму, було вирішено реалізувати інтеграцію з утилітою dotnet. Спосіб базується на використанні загальної системи проектів, яку підтримує команда dotnet build. Таким чином, для проектів tsproj (Typescript project) можна викликати команду dotnet build, яка автоматично запустить розроблений транслятор з переданими необхідними аргументами та опціями.

5. ПОРІВНЯЛЬНИЙ АНАЛІЗ ОТРИМАНОГО КОДУ ТА ТЕСТУВАННЯ ТРАНСЛЯТОРА

5.1. Підходи до порівняльного аналізу отриманого коду

Головною вимогою до коду, що отримується в результаті трансляції мови програмування Typescript у проміжну мову CIL є коректність. Якщо результатом роботи тестового транслятора, який побудовано на основі запропонованих способів, буде програма, яка містить синтаксичну або семантичну помилку мови CIL, віртуальне середовище CLR у результаті запуску програми виведе помилку і не продовжить виконання.

Основним критерієм ефективності отриманого коду у порівнянні з результатом роботи транслятора Jscript, який обрано у якості аналога, є його швидкодія.

Для заміру швидкодії було розроблено консольну програму на мові програмування C#. Вона запускає виконувані файли, які є результатом роботи компіляторів, які порівнюються. Для зменшення ефекту похибок при запуску програм та більшої наочності, деякі приклади, швидкість виконання яких порівнюється, виконано у циклах з великою кількістю ітерацій. Лістинг програми, яка виконує заміри, зображено на рисунку 56.

```
0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        var sw = new Stopwatch();
        foreach (var arg in args)
        {
            using (Process myProcess = new Process())
            {
                myProcess.StartInfo.UseShellExecute = false;
                myProcess.StartInfo.FileName = arg;
                myProcess.StartInfo.CreateNoWindow = true;
                Console.WriteLine("Starting process: {0}", myProcess.StartInfo.FileName);
                myProcess.Start();
                sw.Start();
                myProcess.WaitForExit();
                sw.Stop();
                Console.WriteLine("Ms: {0}", sw.ElapsedMilliseconds);
            }
        }

    }
}
```

Рисунок 56 – Лістинг програми на мові C#, яка виконує запуск та замірює швидкість роботи виконуваних файлів.

Для тестування швидкодії роботи було реалізовано певні програми, які містять конструкції мови, які тестуються у даний момент. Таким чином, наприклад для заміру швидкості роботи конкатенації строкових виразів реалізовано відповідну програму, яка має тільки конкатенацію.

Також, для тестування швидкодії отриманого коду у результаті роботи тестового транслятора, який містить поєднання різних конструкцій мови програмування, було обрано та реалізовано на мові програмування Typescript певний пласт алгоритмів.

Для тестування швидкодії виконання обчислень і циклів було обрано наступні алгоритми:

- ітеративний алгоритм підрахунку факторіалу;
- ітеративний алгоритм пошуку числа Фібоначчі;
- решето Сундарама для знаходження простих чисел;

Для тестування швидкодії виконання обчислень та функцій було обрано наступні алгоритми:

- рекурсивний алгоритм пошук факторіалу;
- рекурсивний алгоритм пошуку числа Фібоначчі.

Варто зазначити, що алгоритми мають таку оцінку складності:

- пошук факторіалу – O(N);
- пошук числа Фібоначчі – O(N), для рекурсивної реалізації – O(e^N) ;
- решето Сундарама – O(NlogN);

5.2. Порівняльний аналіз запропонованих способів з результатом роботи транслятора JScript

Почнемо порівняння швидкодії роботи отриманого CIL коду за запропонованими способами зі способу конкатенації рядкових виразів. Тестова програма на мові програмування Typescript зображена на рисунку 57. Результати тестування зображено на рисунку 58.

```
function main(): void {
    let a: string = 'test a', b: string = 'test b',
        c: string = 'test c', d: string = 'test d',
        e: string = 'test e';

    let result: string;
    for (let i = 0; i < 1000000; i = i + 1)
        result = a + b + c + d + e;
}
```

Рисунок 57 – Вхідна програма для тестування конкатенації рядкових виразів.

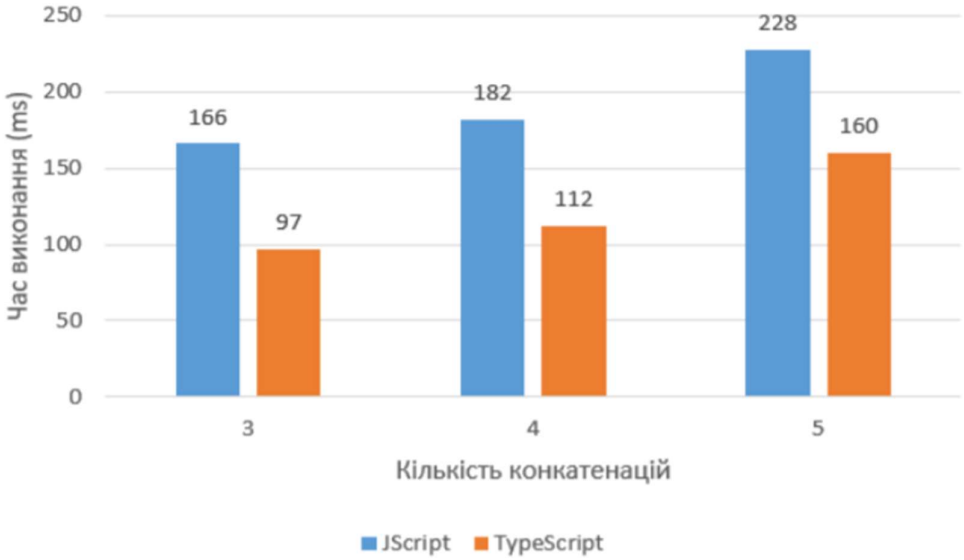

Рисунок 58 – Графік порівняння швидкодії згенерованих інструкцій для конкатенації компілятором JScript та тестовим компілятором Typescript.

Виміри проводилися для конкатенації 3, 4 та 5 рядкових виразів. Як видно з діаграми, код, що згенерований згідно запропонованого способу, працює швидше, ніж результуючий код існуючого аналогу.

Проаналізуємо швидкодію роботи масивів, а конкретно – встановлення елементу за індексом. Використаємо метод порівняння описаний у першому розділі, коли проводився вибір аналогу, порівнюючи результат його роботи з еталонною реалізацією транслятора мови високого рівня у проміжну мову CIL – компілятором C#.

Тестова програма на мові програмування Typescript зображена на рисунку 59. Результати тестування зображено на рисунку 60.

```
function main() {
    const arrTest = new Array<number>(1000000);
    for(let i = 0; i < arrTest.length; i = i + 1) {
        arrTest[i] = i;
    }
}
```

Рисунок 59 – Вхідна програма для тестування встановлення елементу масиву.

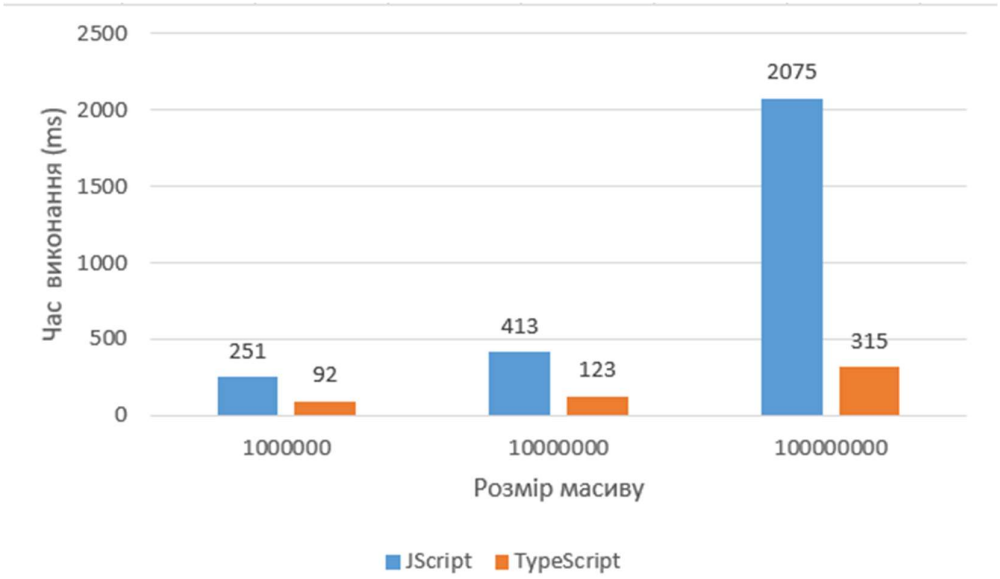

Рисунок 60 – Графік порівняння швидкодії згенерованих інструкцій для встановлення елементу масиву компілятором JScript та тестовим компілятором Typescript.

Як видно з результатів, тестовий транслятор, який використовує запропонований спосіб, генерує CIL код, який працює швидше, ніж обраний аналог. Різниця між швидкодією майже ідентична, до різниці між результатом роботи компіляторів JScript і C#.

Варто зазначити, що отримані часові результати дещо відрізняються у більшу сторону. Це пов’язано з тим, що у першому розділі замір часу

виконувався іншим способом. Було використано клас Stopwatch на рівні програми. Через те, що у реалізованому тестовому трансляторі не має підтримки використання класів стандартної бібліотеки, було вирішено обрати спосіб тестування, описаний у пункті 5.1 і похибка часу з'являється через запуск тестової програми операційною системою.

Тепер виконаємо тестування на запропонованих у попередньому пункті алгоритмів.

Імплементацію ітеративного алгоритму пошуку факторіалу на мові Typescript зображено на рисунку 61. Результат замірів часу для чисел 1000, 100000000, 1000000000 вказано на рисунку 62.

```
function fact(num: number): number {
    let result = 1;
    for(let i = 1; i <= num; i = i + 1) {
        result = result * i;
    }
    return result;
}
```

Рисунок 61 – Функція ітеративного пошуку факторіала на мові Typescript.

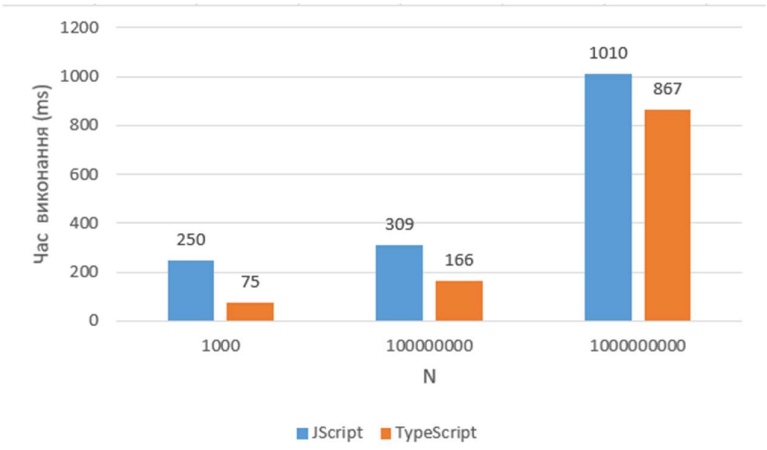

Рисунок 62 – Графік порівняння швидкодії згенерованих інструкцій для ітеративного алгоритму пошуку факторіала компілятором JScript та тестовим компілятором Typescript.

З діаграми видно, що тестовий транслятор мови Typescript генерує більш ефективний код, ніж компілятор JScript.

Розглянемо ітеративний алгоритм пошуку числа Фібоначчі. Реалізація зображена на рисунку 63. Результат замірів часу для чисел 1000, 100000000, 1000000000 показано на рисунку 64.

```
function fib(num: number): number {
    let fib1 = 1, fib2 = 1;

    let fib_sum = 0;
    for (let i = 0; i < num - 2; i = i + 1) {
        fib_sum = fib1 + fib2;
        fib1 = fib2;
        fib2 = fib_sum;
    }

    return fib2;
}
```

Рисунок 63 – Функція ітеративного пошуку числа Фібоначчі на мові Typescript.

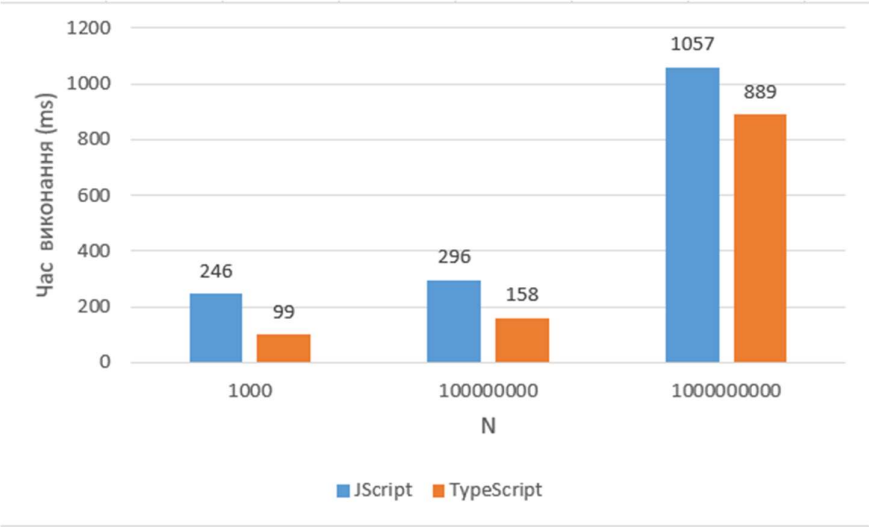

Рисунок 64 – Графік порівняння швидкодії згенерованих інструкцій для ітеративного алгоритму пошуку числа Фібоначчі компілятором JScript та тестовим компілятором Typescript.

Як видно з графіку, результат роботи алгоритму близький до отриманого результату у попередньому аналізі. Це зумовлено однаковою складністю алгоритмів та однаковим набором конструкцій мови, що були використані.

Розглянемо роботу рекурсивних версій цих алгоритмів. Рекурсивна версія пошуку факторіалу на мові Typescript зображена на рисунку 65, результати тестування – на рисунку 66.

```
function fact(num: number): number {
    if (num <= 0)
        return 1;

    return num * fact(num - 1);
}
```

Рисунок 65 – Рекурсивна функція пошуку факторіала на мові Typescript.

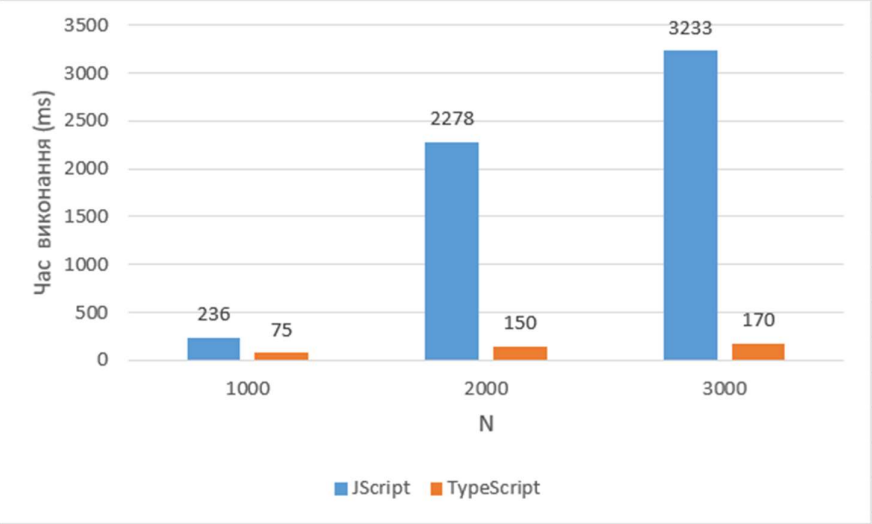

Рисунок 66 – Графік порівняння швидкодії згенерованих інструкцій для рекурсивного алгоритму пошуку факторіала компілятором JScript та тестовим компілятором Typescript.

Як видно з графіку, програма, що є результатом роботи тестового транслятора працює швидше Варто зазначити, що для тестування рекурсивних функцій було обрано менші значення параметрів. Це зумовлено тим, що рекурсія працює повільніше через наявність CIL інструкцій, які записують аргументи функції на стек.

Рекурсивна версія алгоритму пошуку числа Фібоначчі зображена на рисунку 67, а результати тестування – на рисунку 68.

```
function fib(num: number): number {
    if (num == 0)
        return 0;
    if (num == 1)
        return 1;
    return fib(num - 1) + fib(num - 2);
}
```

Рисунок 67 – Рекурсивна функція пошуку числа Фібоначчі на мові Typescript.

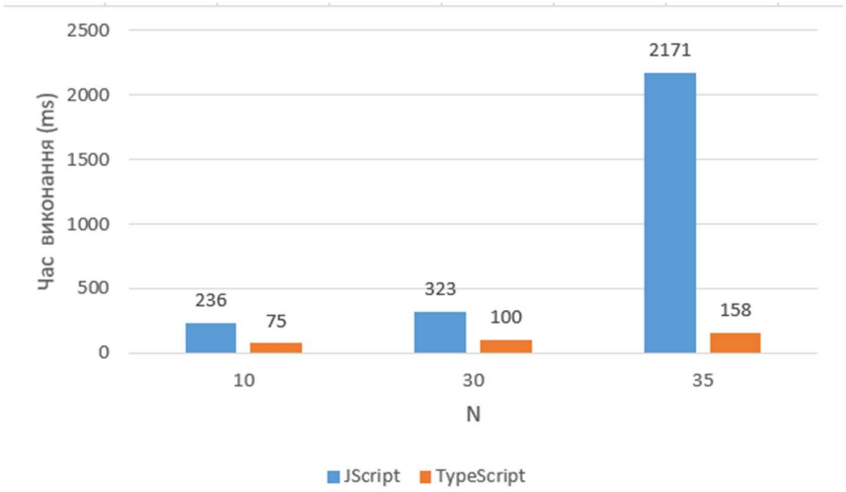

Рисунок 68 – Графік порівняння швидкодії згенерованих інструкцій для рекурсивного алгоритму пошуку числа Фібоначчі компілятором JScript та тестовим компілятором Typescript.

Виконаємо заміри часу для більш складного алгоритму. Розглянемо алгоритм пошуку простих чисел за методом «решето Сундарама». Реалізація цього алгоритму на мові Typescript зображена на рисунку 69. Графік порівняння швидкодії з аналогічною програмою на мові JScript зображено на рисунку 70.

```
function sundaram(n: number): void {
    const A = new Array<boolean>(n);
    let i: number, j: number;
    for (i = 1; i <= A.length; i = i + 1)
        A[i] = true;

    i = 1; j = 1;
    while ((2 * i * j + i + j) <= A.length) {
        while (j <= (n - i) / (2 * i + 1)) {
            A[2 * i * j + i + j] = false;
            j = j + 1;
        }
        i = i + 1;
        j = i;
    }

    for (i = 1; i <= A.length; i = i + 1)
        if (A[i]) {
            alert(2 * i + 1);
            alert(' ');
        }
}
```

Рисунок 69 – Функція пошуку простих чисел методом «решето Сундарама» на мові Typescript.

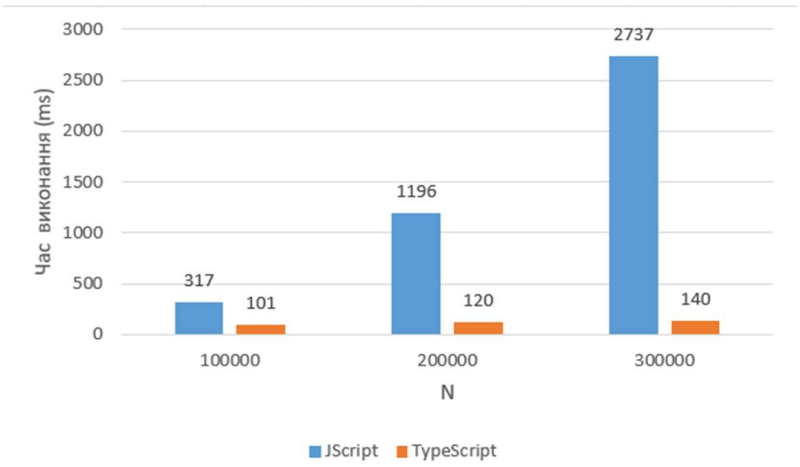

Рисунок 70 – Графік порівняння швидкодії згенерованих інструкцій для пошуку простих чисел методом «решето Сундарама» компілятором JScript та тестовим компілятором Typescript.

Як бачимо, що згенерований код розробленим тестовим транслятором працює швидше і майже не уповільнюється зі збільшенням діапазону пошуку. Згенерований код компілятора JScript сильно уповільнюється зі збільшенням діапазону пошуку через наявність складних інструкцій у результуючій програмі.

ВИСНОВКИ ДО РОЗДІЛУ 5

У даному розділі було виконано порівняння роботи тестового транслятора за запропонованими способами трансляції мови програмування Typescript у проміжну мову CIL платформи .NET та обраного аналога – транслятора JScript.

Основним критерієм порівняння ефективності згенерованого коду CIL обома трансляторами є швидкодія програми.

Для замірювання часу було написано програму на мові програмування C#, яка виконує запуск програм, які є результатом роботи трансляторів, що тестуються, та замірює час.

Для тестування було використано тестові програми, які використовують усі конструкції мови програмування Typescript, які було розглянуто у 3 розділі.

В ході тестування на основі отриманих результатів було визначено, що тестовий транслятор генерує більш ефективний код ніж обраний аналог для відібраних задач. Особливий приріст швидкодії наявний при у програмах, в яких використовувались масиви.

ВИСНОВКИ

В даній магістерській дисертації було досліджено способи трансляції мови програмування Typescript у проміжну мову CIL платформи .NET.

Метою дослідження було розробити більш швидкі способи трансляції умовних конструкцій, циклів, функцій, рядків та масивів та виконати порівняльний аналіз розроблених способів.

В ході виконання магістерської дисертації було виконано аналіз існуючих рішень трансляції мови Typescript у проміжну мову CIL, але через відсутність прямих способів трансляції було вирішено проаналізувати способи трансляції мови Javascript. Результатом аналізу було обрано транслятор Jscript, як аналог до тестового транслятора, який розроблено на основі запропонований способів у цій дисертації.

Результатом дослідження є розроблені способи такої підмножини мови програмування Typescript:

1) спосіб трансляції декларації змінних, простих типів даних та операцій над ними;
2) спосіб трансляції конкатенації рядкових виразів;
3) спосіб трансляції масивів;
4) спосіб трансляції умовних конструкцій if-else;
5) способи трансляції таких циклічних конструкцій: while, do-while, for, for-of;
6) спосіб трансляції глобальних функцій;

Для дослідження ефективності запропонованих способів було розроблено тестовий транслятор на мові програмування C# з використанням бібліотеки Mono.Cecil, який було інтегровано з утилітою dotnet.

При аналізі результатів було визначено, що код згенерований розробленим тестовим транслятором працює швидше, ніж код, який є результатом роботи транслятора Jscript, що підкріплено результатами роботи тестових програм для вирішення певних задач.

Практична цінність отриманих в дисертації результатів полягає в тому, що запропоновані способи трансляції мови Typescript у проміжну мову CIL платформи .NET та розроблений тестовий транслятор підтверджують, що безпосередня трансляція можлива і ефективна. Отримані результати є підставою для створення у майбутньому повноцінного транслятора мови Typescript у платформу .NET, що дозволить збільшити популярність обраних технологій та популяризувати платформу .NET для розробки серверної частини веб-додатків, які мають клієнтську частину, розроблену на мовах програмування Typescript чи Javascript.

У подальшому планується дослідити способи трансляції класів, інтерфейсів та просторів імен мови програмування Typescript у проміжну мову CIL, а також додати підтримку використання бібліотек, розроблених на інших мовах програмування платформи .NET, а саме: C#, F# та VisualBasic. На фінальній стадії, коли повноцінний транслятор буде реалізовано, пропонується додати реалізацію підтримки методів стандартної бібліотеки Javascript роботи з масивами, рядками та іншими типами даних на рівні транслятора.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Альфред В. Ахо, Моника С. Лам, Рави Сети, Джеффри Д. Ульман. Компиляторы: принципы, технологии и инструментарий, 2изд. — М.: Вильямс, 2008.
2. Іваненко А.Р., Марченко О.І Спосіб трансляції конкатенації рядкових виразів мови TypeScript у проміжну мову CIL платформи .NET. *Науковий журнал "Комп'ютерно-інтегровані технології: освіта, наука, виробництво"*, 2021, № 42. С. 137 – 141.
3. Марк Дж. Прайс. C# 7 и .NET Core. Кросс-платформенная разработка для профессионалов. 3-е изд. — СПб.: Питер, 2018.
4. Марченко О.І. Конспект лекцій з дисципліни «Інженерія програмного забезпечення-1. Основи проектування трансляторів», Київ 2013.
5. Марченко О.І., Іваненко А.Р. Аналіз способів трансляції мови TypeScript у проміжну мову CIL платформи .NET, *XIII конференція молодих вчених ПМК-2020 року* : тези доповіді, м.Київ, 2020.
6. Рихтер Дж. CLR via C#. Программирование на платформе Microsoft .NET Framework 4.5 на языке C#. 4-е изд., СПб.:Питер, 2013
7. Jason Bock, CIL Programming: Under the Hood of .NET, New York, 2002.
8. John I. Moore, Jr. Introduction to Compiler Design. Ab Object-Oriented Approach Using Java.
9. Nathan Rozentals. Mastering TypeScript - Build enterprise-ready, industrial strength web applications using TypeScript and leading JavaScript Frameworks. — Packt Publishing, 2015.
10. Keight D.Cooper, Linda Torczon, Engineering a Compiler, 2nd Edition, Burlington, 2012.
11. Microsoft documentation [Електронний ресурс] – Режим доступу: https://docs.microsoft.com/en-us/dotnet/api/system.string.concat
12. Mono.Cecil documentation [Електронний ресурс] – Режим доступу: https://cecil.pe/

13. Ronald Mak, Writing Compilers and Interpreters: A Software Engineering Approach, 3rd Edition, Indianapolis, 2009.
14. Serge Lidin, Inside Microsoft .Net IL Assembler, Redmond, 2002.
15. Typescript documentation [Електронний ресурс] – Режим доступу: https://www.typescriptlang.org/
16. Yannis Bres, Bernard Serpette, Manuel Serrano, Compiling Scheme programs to .NET Common Intermediate Language [Електроний ресурс] – Режим доступу: https://www.researchgate.net/publication/213885643_Compiling_Scheme_programs_to_NET_Common_Intermediate_Language.
17. YangSun Lee, SeungWon Na, DaeHoon Hwang, Language Translator for Execution Java programs in .NET [Електроний ресурс] – Режим доступу: https://www.koreascience.or.kr/article/JAKO200411922370411.pdf