НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

ФАКУЛЬТЕТ ІНФОРМАТИКИ І ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ

КАФЕДРА ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ

**Курсова робота**

з дисципліни **«**Системне програмування 2**»**

Виконав:

студент 3 курсу гр. ІO-91

Діденко Владислав

Варіант 8

Перевірив:

Павлов В. Г.

Київ 2021 р.

Національний технічний університет України

“Київський політехнічний інститут імені Ігоря Сікорського”

Факультет (інститут) інформатики та обчислювальної техніки

( повна назва )

Кафедра обчислювальної техніки

( повна назва )

Освітньо-кваліфікаційний рівень бакалавр

Спеціальність 123 «Комп’ютерна інженерія»

*(шифр і назва)*

***ЗАВДАННЯ***

НА КУРСОВУ РОБОТУ СТУДЕНТУ

Діденка Владислава Віталійовича

*(прізвище, ім’я, по батькові)*

1. Тема роботи «Розробка компілятора підмножини команд мови Python, що містять арифметичні дії і дужкові форми»

керівник роботи Павлов В.Г. к.т.н.**,** доцент

( прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

2. Строк подання студентом роботи 14 грудня 2021 р.

3. Вхідні дані до роботи

* вхідна мова: Python;
* цільова мова: Assembler x86;
* типи конструкцій: арифметичні дії, розгалуження, цикли, функції.

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

* розробити лексичний аналізатор мови Python;
* розробити синтаксичний аналізатор мови Python;
* розробити генератор коду мови Assembler x86.

5. Перелік графічного матеріалу;

* блок-схема компілятора;
* скріншоти вхідного коду;
* скріншоти токенів;
* скріншоти згенерованого коду;
* скріншоти запуску.

6. Дата видачі завдання 15 жовтня 2021 р

**Завдання**:

1. Варіант завдання та вхідна мова програмування визначається номером у списку групи. Кожному варіанту відповідає номер алгоритму. Знайти необхідний алгоритм, реалізувати його на мові програмування відповідного варіанту та отримати множину результатів контрольних прикладів, як валідних, так й тих, що містять помилку у вхідному коді.
2. Програма реалізації алгоритму на вхідної мові повинна містити коментарі, що описують роботу цього алгоритму.
3. Для перевірки кожного валідного результату навести відповідні математичні обрахунки з проміжними результатами.
4. Розробити програму компілятора, яка здійснює генерацію асемблерного коду MASM32, відповідно до вхідної програми.
5. Реалізувати обробку коментарів у компіляторі та вивід результату у віконному інтерфейсі.
6. Здійснити перевірку тотожності результатів виконання асемблерних програм контрольнім результатам, отриманих у пункті 1.
7. За підсумками роботи зробити висновки.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Варіант** | **Номер алгоритму** | **Вхідна мова** |  | **Варіант** | **Номер алгоритму** | **Вхідна мова** |
| **1** | 4 | C |  | **16** | 2 | Python |
| **2** | 6 | Python |  | **17** | 6 | C |
| **3** | 1 | C |  | **18** | 4 | Python |
| **4** | 3 | Python |  | **19** | 5 | C |
| **5** | 10 | C |  | **20** | 1 | Python |
| **6** | 5 | Python |  | **21** | 8 | C |
| **7** | 2 | C |  | **22** | 7 | Python |
| **8** | 8 | Python |  | **23** | 12 | C |
| **9** | 7 | C |  | **24** | 10 | Python |
| **10** | 9 | Python |  | **25** | 9 | C |
| **11** | 11 | C |  | **26** | 11 | Python |
| **12** | 12 | Python |  | **27** | 3 | C |
| **13** | 13 | C |  | **28** | 13 | Python |
| **14** | 14 | Python |  | **29** | 14 | C |
| **15** | 15 | C |  | **30** | 15 | Python |

Зміст

[Вступ 6](#_Toc90500676)

[Огляд методів побудови системних програм 7](#_Toc90500677)

[Опис роботи програми за блок-схемою 7](#_Toc90500678)

[Лістинг програми компілятору 11](#_Toc90500679)

[Program.cs 11](#_Toc90500680)

[AstTree.cs 16](#_Toc90500681)

[Ast.cs 17](#_Toc90500682)

[Expressions.cs 18](#_Toc90500683)

[Statements.cs 19](#_Toc90500684)

[MyEnumerator.cs 20](#_Toc90500685)

[CustomExceptions.cs 25](#_Toc90500686)

[INamespace.cs 26](#_Toc90500687)

[Результати перевірки працездатності програми 26](#_Toc90500688)

[Висновок 30](#_Toc90500689)

[Список використаної літератури 31](#_Toc90500690)

Вступ

В даній роботі буде побудована програма компілятора яка з вхідного коду на мові Python генерує асемблерний код MASM32. Даний компілятор буде побудований на мові C#, через її об’єкто-орієнтовність, статичну типізацію та зручність у використанні. Робота програми буде проходити в 3 етапи: перший – виділення токенів з вхідного коду завдяки лексеру, другий – синтаксичний аналіз набору отриманих токенів та побудова абстрактного синтаксичного дерева (AST) завдяки парсеру, третій – генератор asm коду, що приймає абстрактне синтаксичне дерево з парсеру та в заготовлений шаблон вставляє згенерований код в заздалегідь підготовлені місця.

Це завдання є актуальним через те, що в його основі лежить задача аналізу вхідних даних, а ця задача часто зустрічається у програмуванні. Також ця задача корисна через те, що дає можливість розібратися с інструментом, який ми використовуємо кожен день, а це точно ніколи не завадить, і найбільш корисним це завдання для програмістів, що збираються писати програми на мовах низького рівня, або, можливо, писати самі компілятори чи нові язики програмування.

Огляд методів побудови системних програм

Для написання цього компілятора було використано принцип Incremental Approach, тобто ітеративний підхід. Його ідея полягає в тому, щоб написати компілятор з мінімальними можливостями, і поступово додавати до нього новий функціонал, на кожному етапі отримуючи повноцінно робочий компілятор. Перевагами даного методу є легкість його реалізіції, зручність тестування, а також можливість швидко додавати функціонал, також даний метод є доволі зручним для спільної розробки ПЗ. Недоліками цього методу є те, що найчастіше ПЗ написане покладаючись на цей метод є більш ‘примітивним’, ніж написане завдяки іншим методам, через те, що деякий функціонал важко добре написати, через бажання зробити ітеративний шаг якомога меншим.

Сьогодні використовуються різні методи компіляції, що обираються в залежності від поставлених задач, такими є:

* метод крос-трансляції, в основі якого лежить ідея перекладу коду однієї мови в код іншої на додатковому пристрої, а потім ‘переклад’ основному пристрою. Цей метод використовується, коли пристрій, на якому буде виконуватися програма, має обмеження в ресурсах, тому для покращення його роботи ми надаємо йому код написаний спеціально для нього. Основною проблемою цього методу є необхідність в додатковому пристрою;
* метод компіляції на віртуальній машині, в основі якого лежить ідея того, щоб додати посередника в етап компіляції, ми генеруємо код для віртуальної машини, що дає можливість писати програму не турбуючись про пристрій, на якому програма буде виконуватися. Таким чином працює JVM. Проблемою цього методу є повільність;
* та інші

Опис роботи програми за блок-схемою

Програма складається з 4 сутностей:

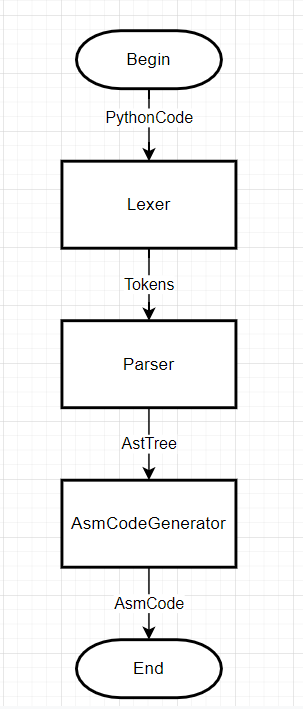
* Program – сутність, що представлена класом Program та є точкою входу програми, в ній ми знаходимо файл з вхідним кодом, створюємо екземпляр класу Lexer, та передаємо в конструктор код на мові Python в форматі string, після цього викликаємо публічний метод Tokenize() у створеного об’єкту, тепер при звернені до властивості Tokens ми отримаємо список токенів. Далі створюємо екземпляр класу Parser, та передаємо в конструктор список токенів, після цього викликаємо публічний метод Parse() у створеного об’єкту, тепер при звернені до властивості AstTree ми отримаємо екземпляр класу AstTree, в якому зберігається наше абстрактне синтаксичне дерево. Далі створюємо екземпляр класу AsmCodeGenerator, та передаємо в конструктор синтаксичне дерево, після цього викликаємо публічний метод GenerateAsm(), тепер при звернені до властивості AsmCode ми отримаємо asm код в форматі string. Потім, ми викликаємо метод WriteCodeToFile(), який створює файл типу asm. Таким чином на кінці програми ми отримуємо файл з згенерованим асемблерним кодом MASM32, за умови, що програма закінчила своє виконання успішно. Інакше, в консоль буде виведено повідомлення про помилку з її описом, та місцем знаходження.
* Lexer – сутність, представлена класами Lexer, Token, вона є частиною компілятора, що відповідає за лексичний аналіз. Ця сутність потрібна, щоб з вхідного коду виділити лексеми (токени). Даний етап є найбільш ‘важким’ для пристрою, через те, що ми перевіряємо кожен символ, а також порівнюємо зі списком відомих лексеру конструкцій, таким чином ми аналізувати код по частинам та виділяти певні токени. Список токенів, що підтримуються у даній реалізації компілятора містить лише ті токени, які використовуються в даній реалізації алгоритму, їх список:
  + FuncDefinition – токен, що відповідає ключовому слову def
  + Identifier – токен, що відповідає будь-якій послідовності букв (саме букв), що не є іншими токенами, тобто, це імена
  + OpenBracket – токен, що відповідає символу (
  + CloseBracket – токен, що відповідає символу )
  + Colon – токен, що відповідає символу :
  + Newline – токен, що відповідає переходу на наступну строку, тобто \n для Unix платформ, та \r \n для інших
  + Indent – токен, що відповідає відступу, в Python можна використовувати різні види відступів, проте цей компілятор вміє працювати лише з табами, тобто \t, та чотирма пробілами
  + Return – токен, що відповідає ключовому слову return
  + Multiply – токен, що відповідає оператору множення, тобто символу \*
  + Divide – токен, що відповідає оператору ділення, тобто символу /
  + Dedent – токен, що відповідає зменшенню відступу, тобто зменшенню рівню вкладеності.
  + WhileLoop – токен, що відповідає ключовому слову while
  + NotEqual – токен, що відповідає послідовності символів !=
  + IfCondition – токен, що відповідає ключовому слову if
  + Greater – токен, що відповідає символу >
  + Assignment – токен, що відповідає оператору присвоєння, тобто cимволу =
  + Subtract – токен, що відповідає оператору віднімання, тобто символу -
  + ElseCondition – токен, що відповідає ключовому слову else
  + Comma – токен, що відповідає символу ,
  + PrintOperator – токен, що відповідає ключовому слову print
  + IntegerNumber – токен, що відповідає будь-якій послідовності цифр
* Parser – сутність, представлена класами Parser, AstTree, AstNode, його нащадками Statement, Expression, та в свою чергу їх нащадками, а саме Statement:
  + ExpressionStatement
  + IfStatement
  + ElseStatement
  + BlockStatement
  + AssignStatement
  + WhileLoopStatement
  + FuncStatement
  + ReturnStatement
  + PrintStatement

Expression:

* + CallExpression
  + ConditionalExpression
  + VarExpression
  + ConstExpression
  + BinaryOperationExpression

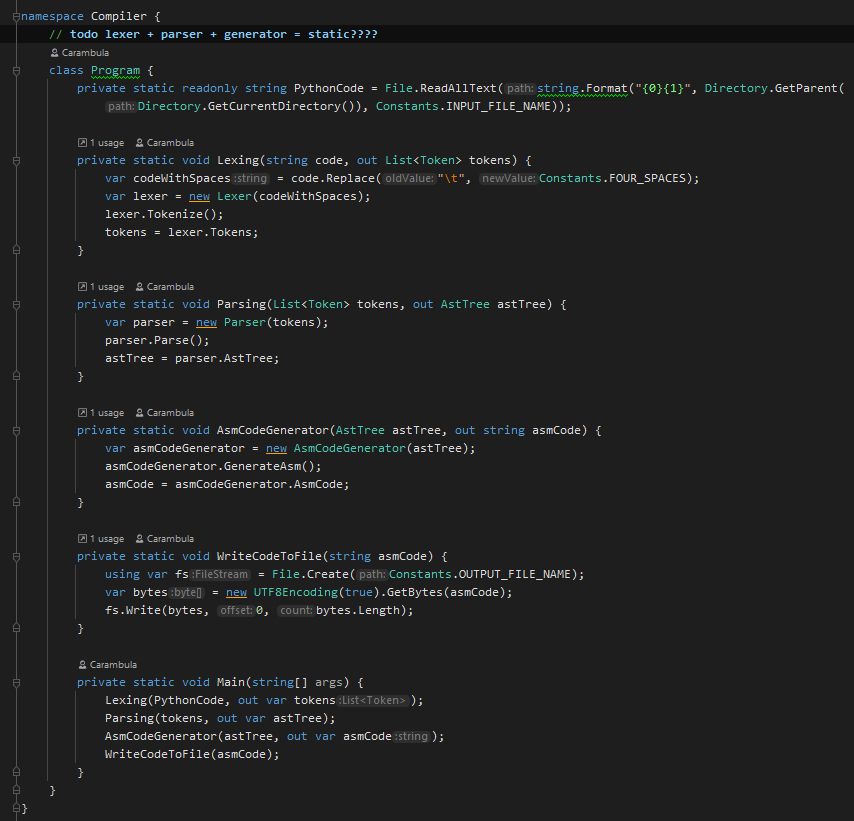
Завданням цієї сутності є побудова абстрактного синтаксичного дерева з токенів отриманих від лексера, саме ця сутність аналізує зміст написаного.

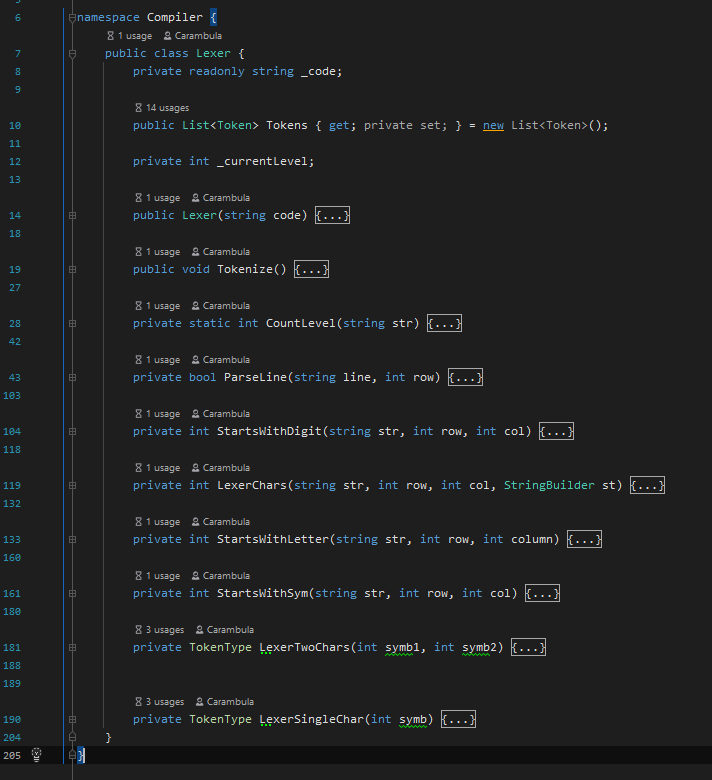
* AsmGenerator – сутність, представлена класом AsmCodeGenerator, який отримує абстрактне синтаксичне дерево, та завдяки заздалегідь підготовленим шаблонам asm кода, генерує код на мові Assembler x86.

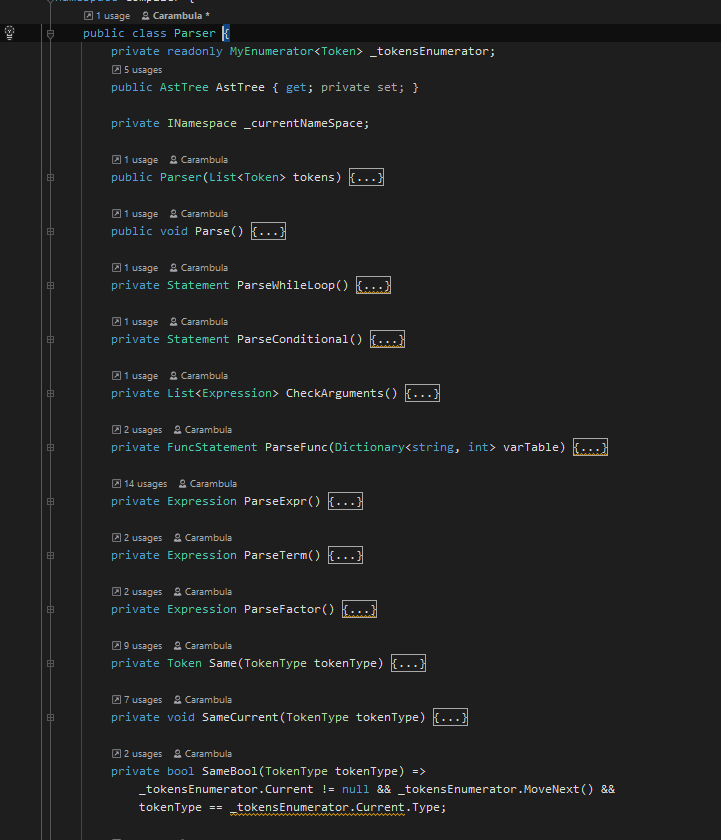


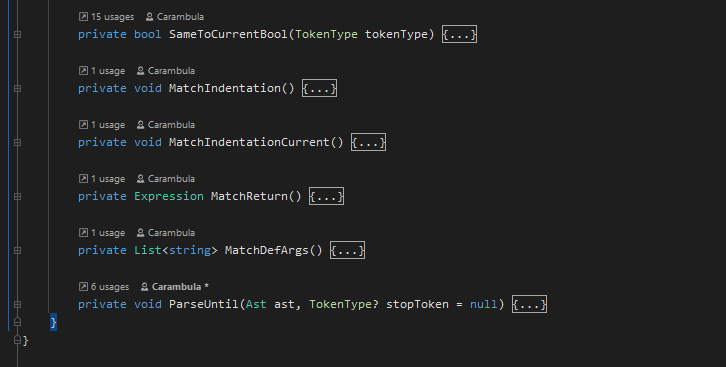
Лістинг програми компілятору

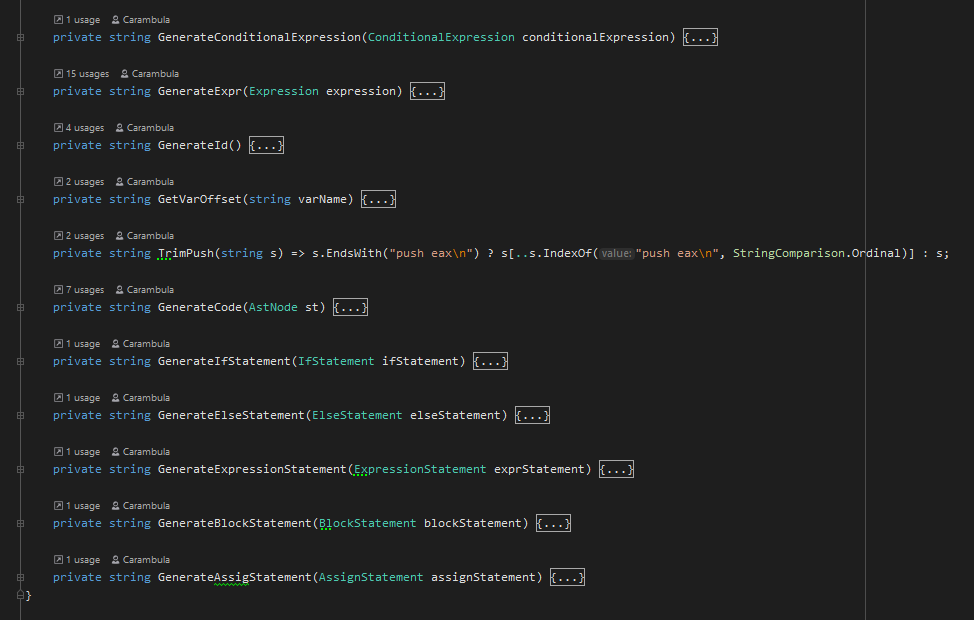
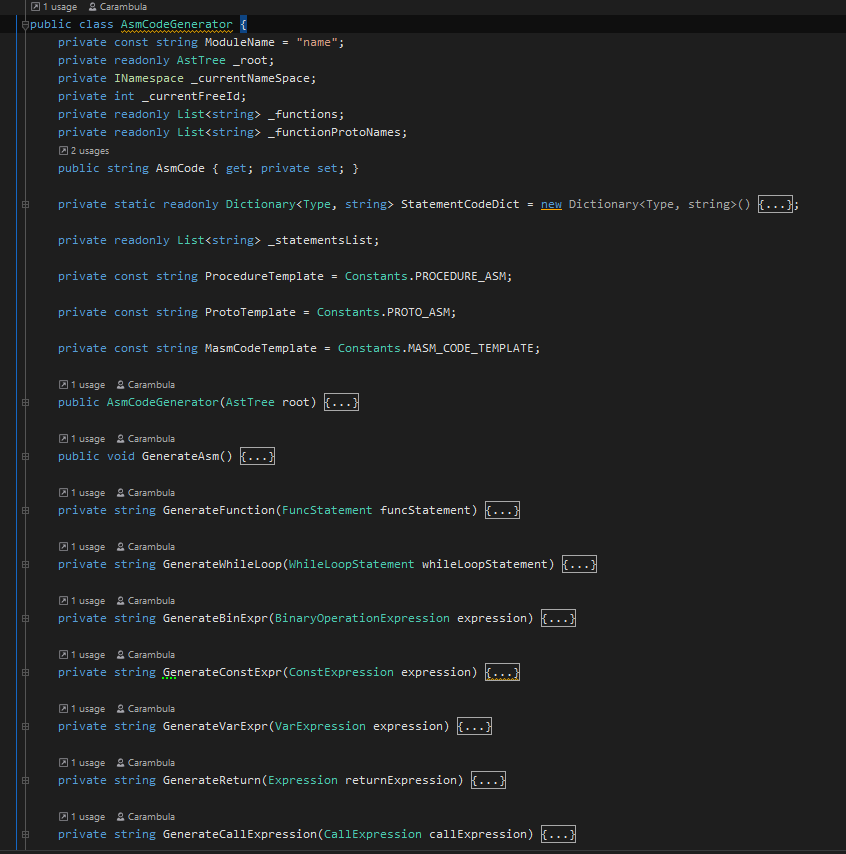
Program.cs



Lexer.cs

Parser.cs

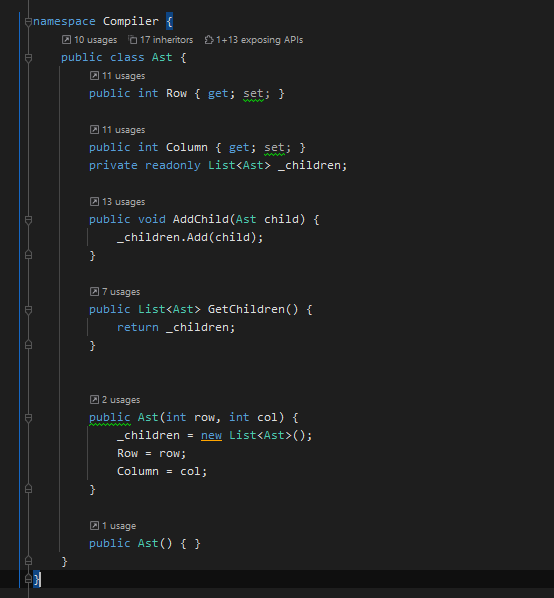


AsmGenerator.cs

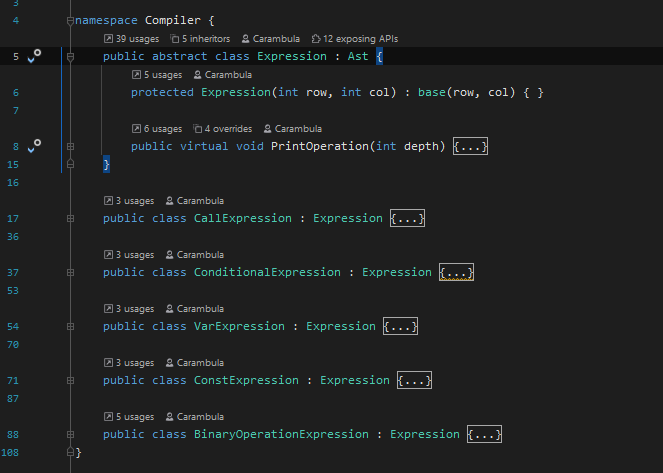
AstTree.cs



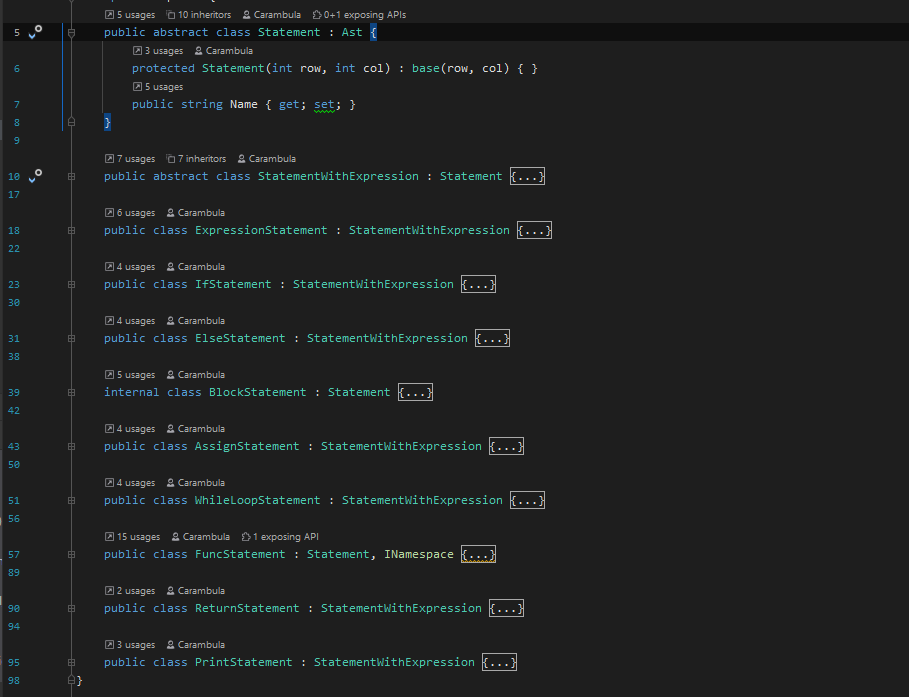
Ast.cs



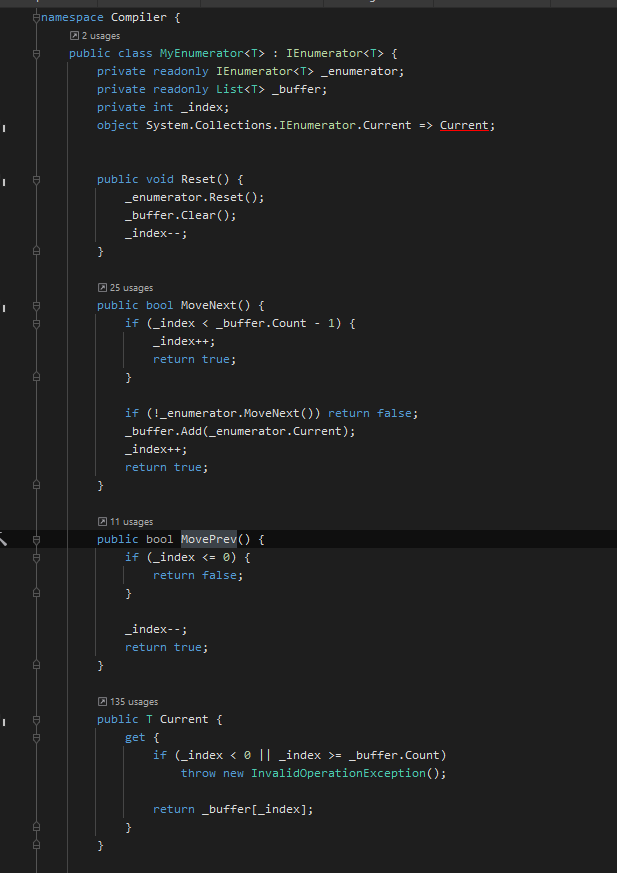
Expressions.cs

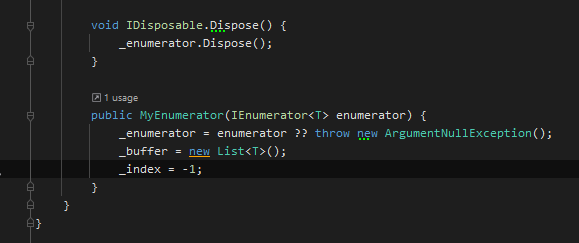


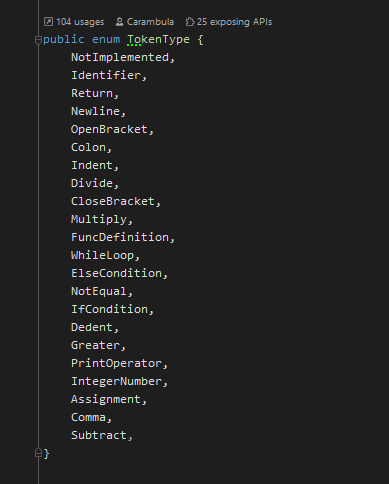
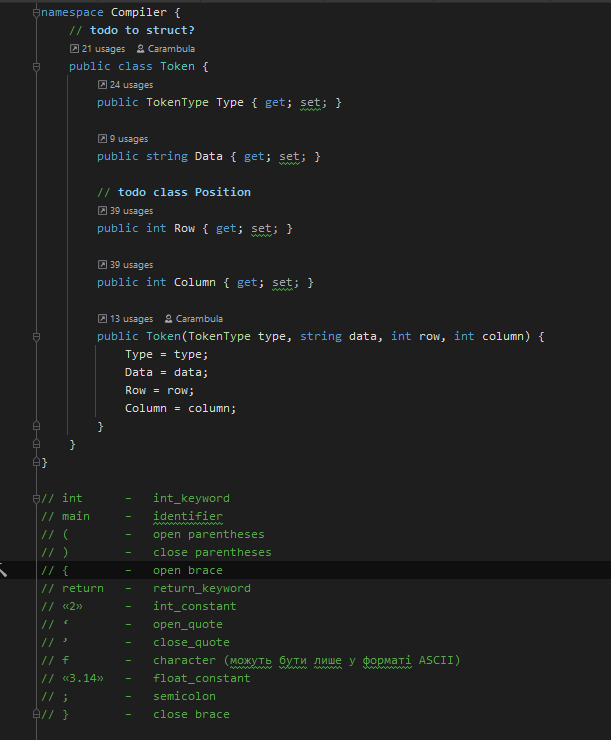
Statements.cs

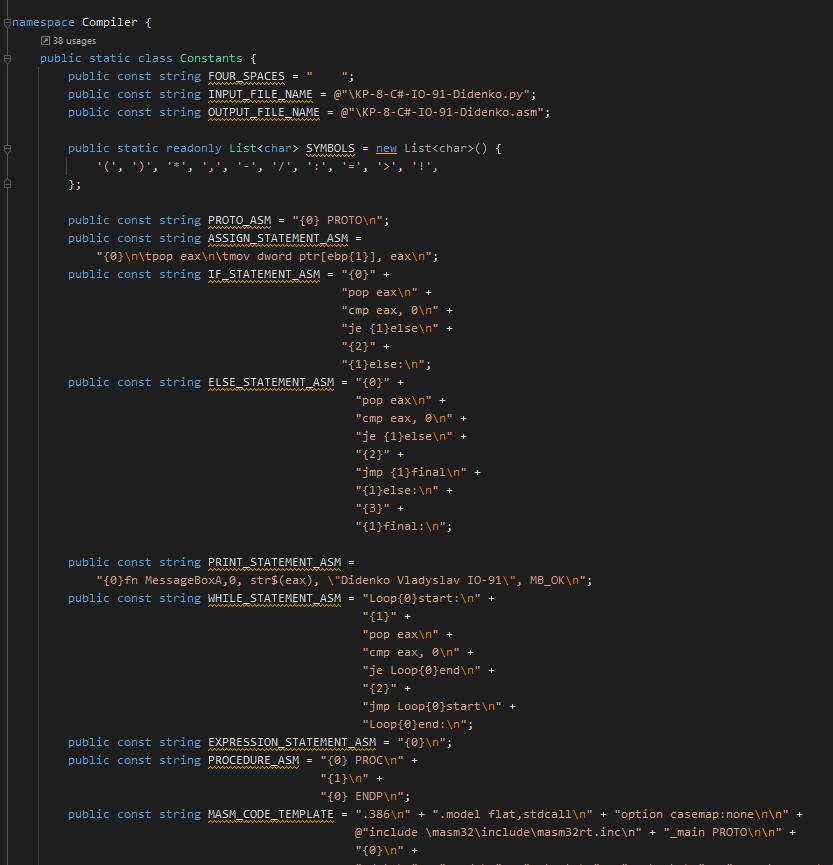


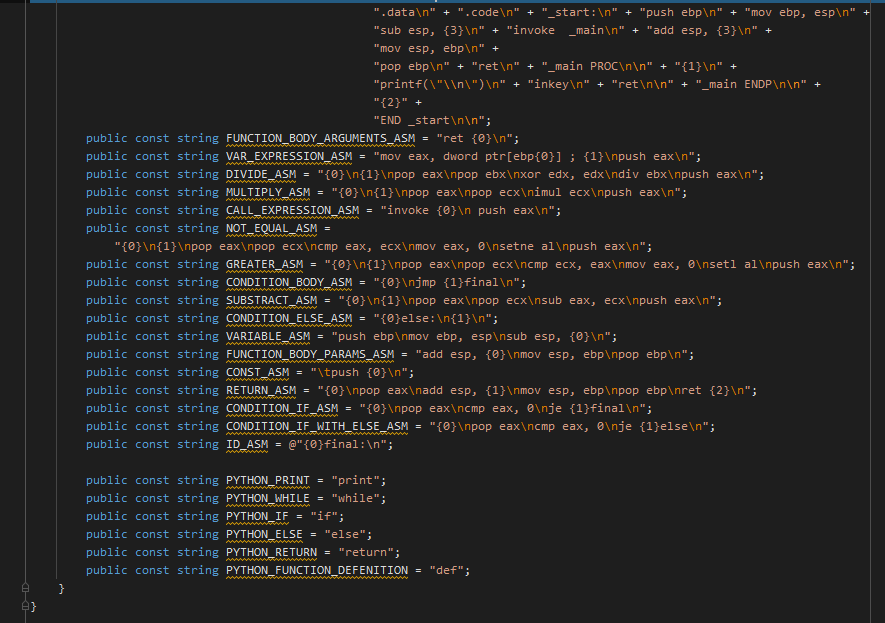
MyEnumerator.cs



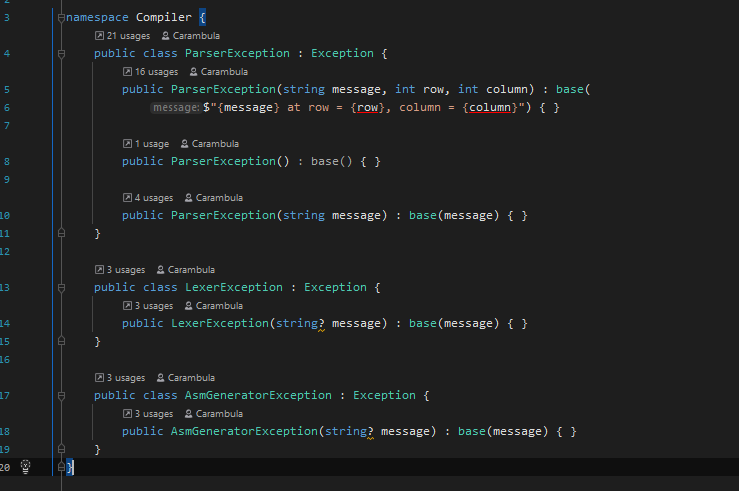


Token.cs

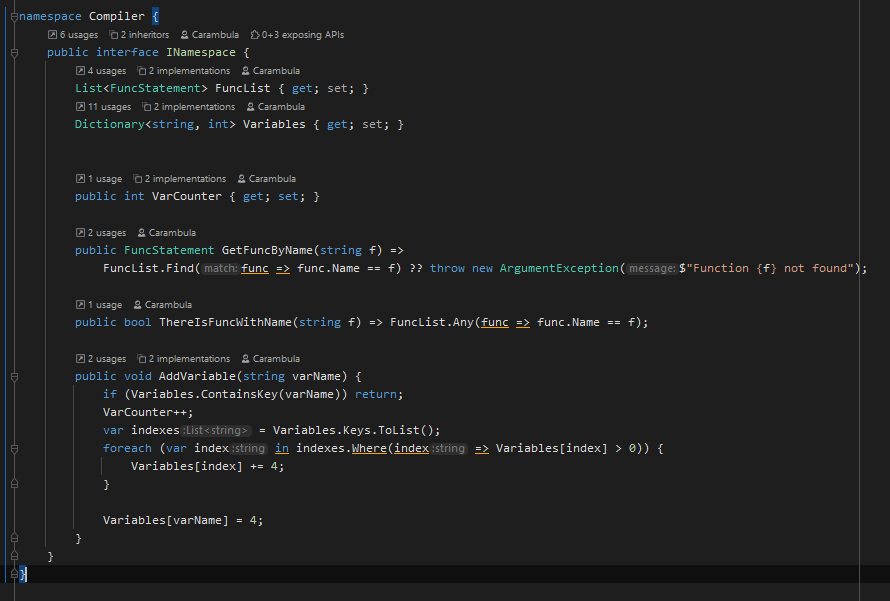
Constants.cs



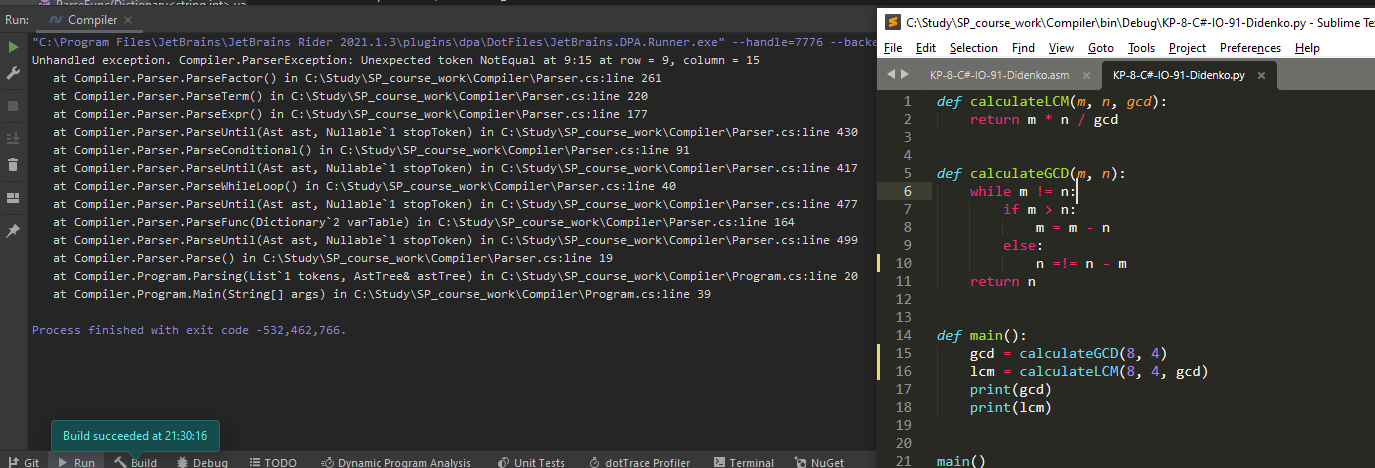
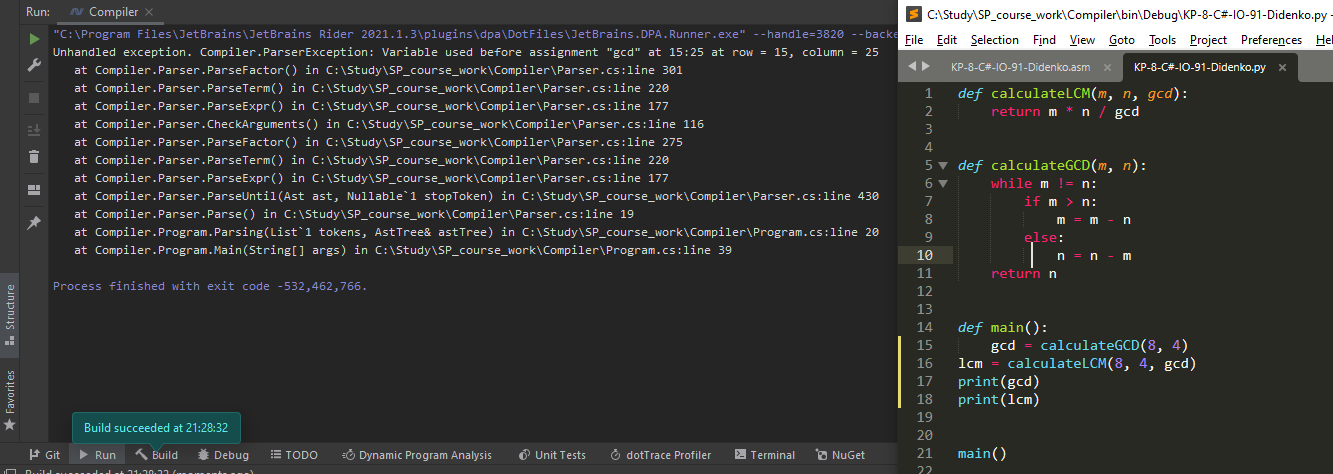
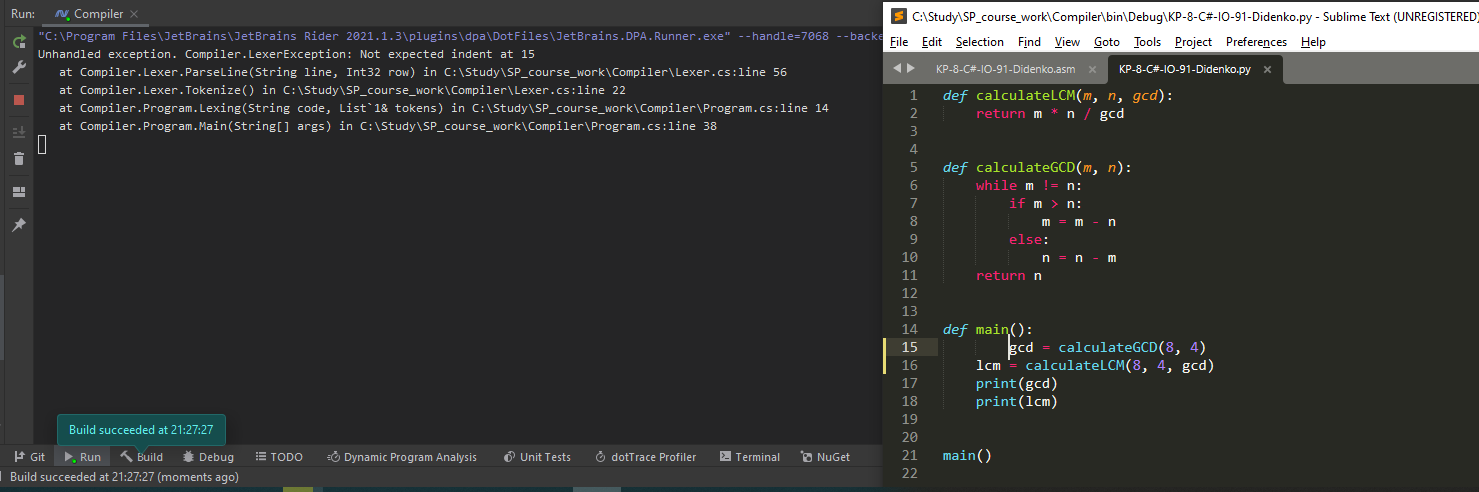
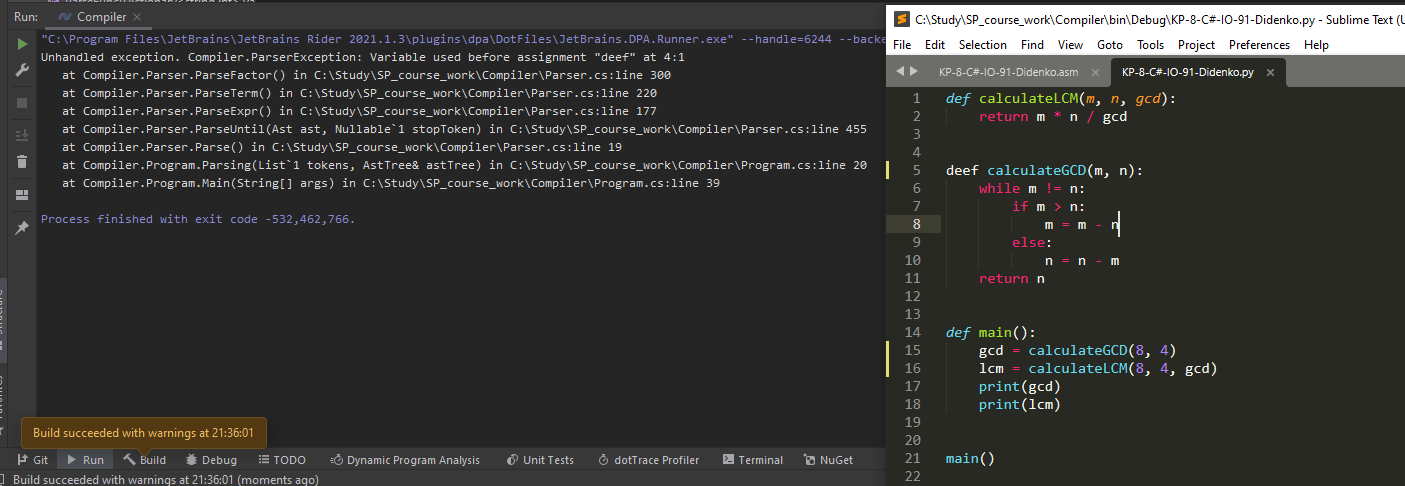
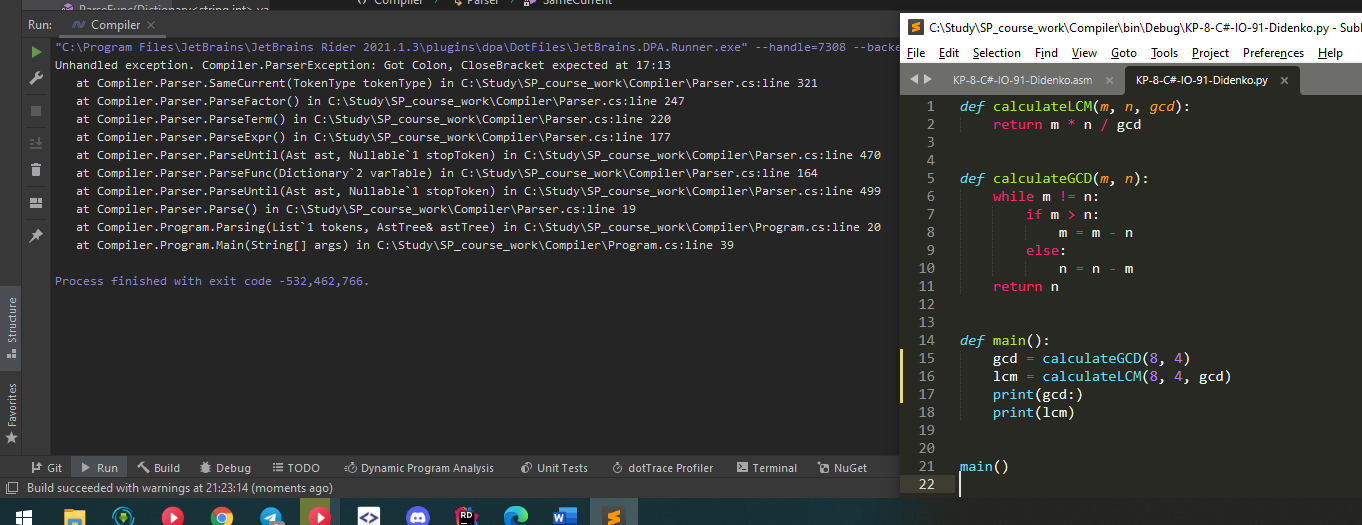
CustomExceptions.cs



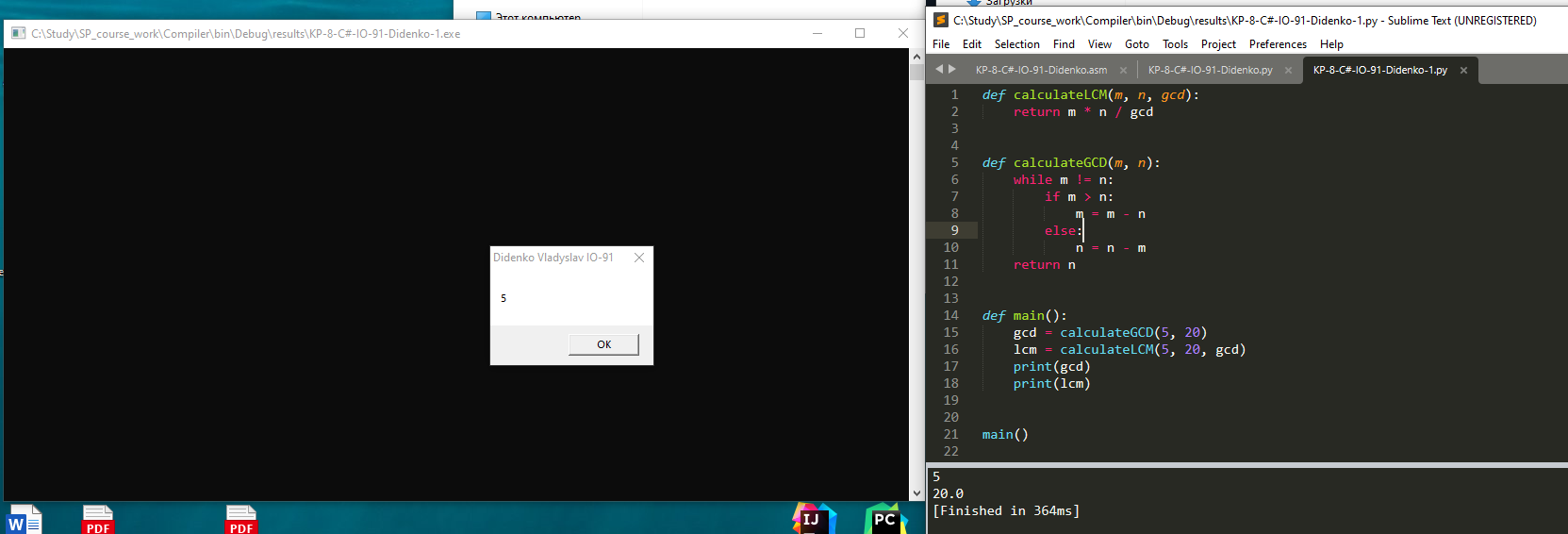
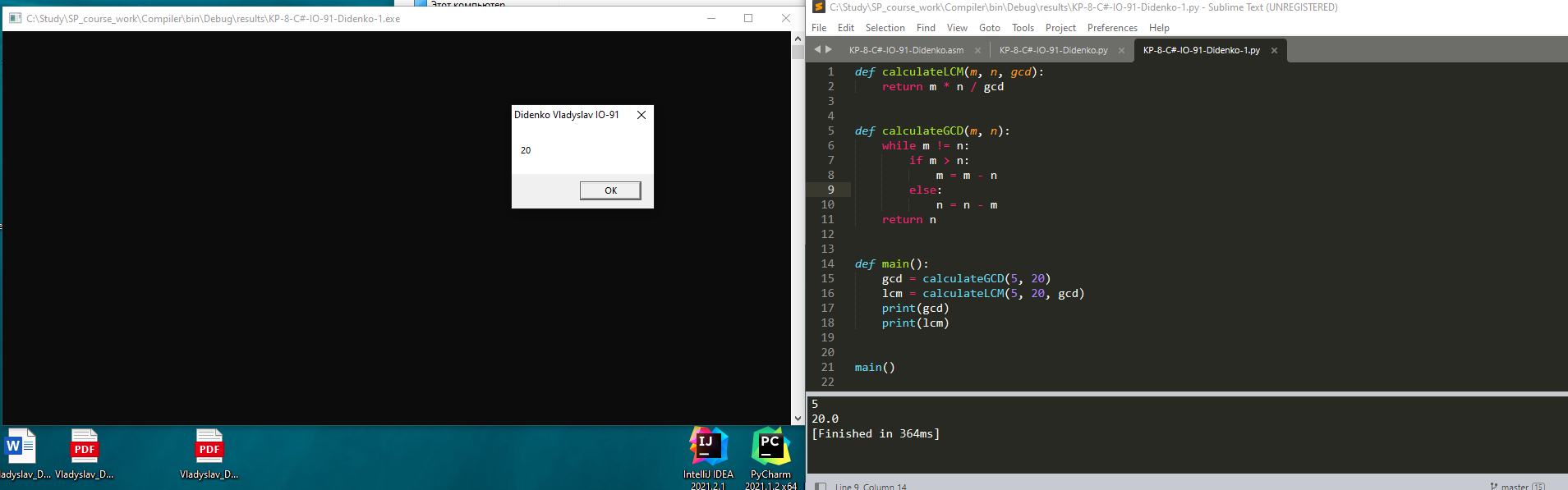
INamespace.cs

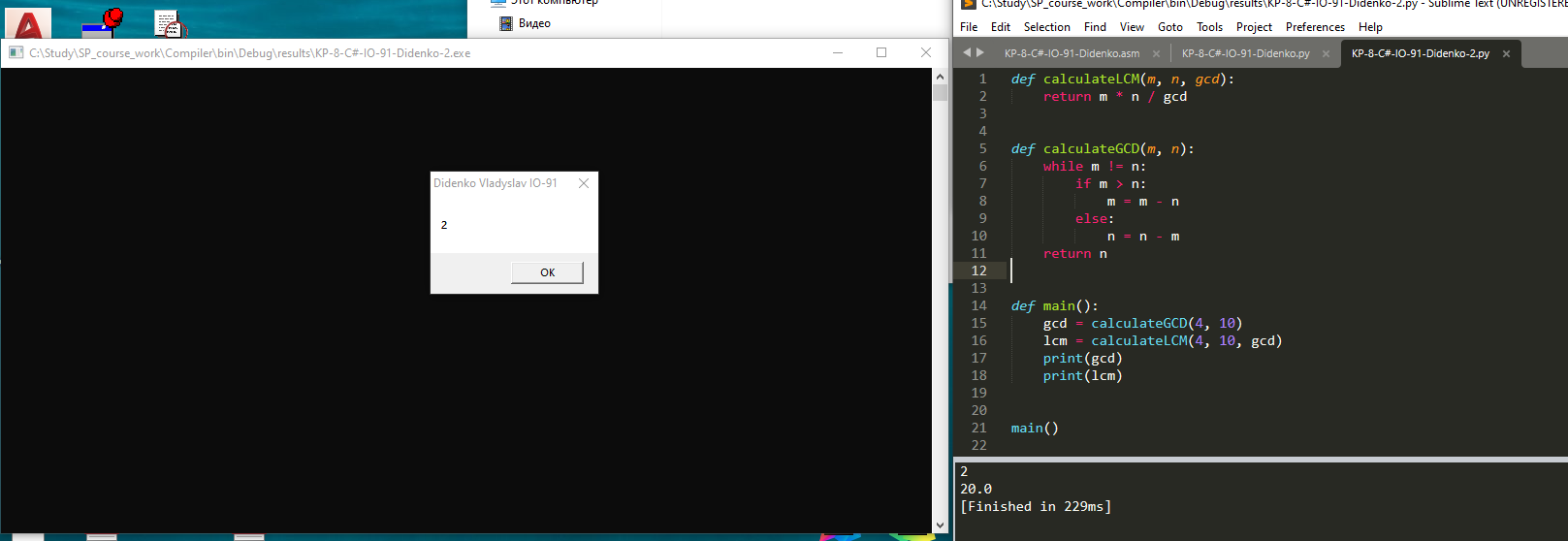
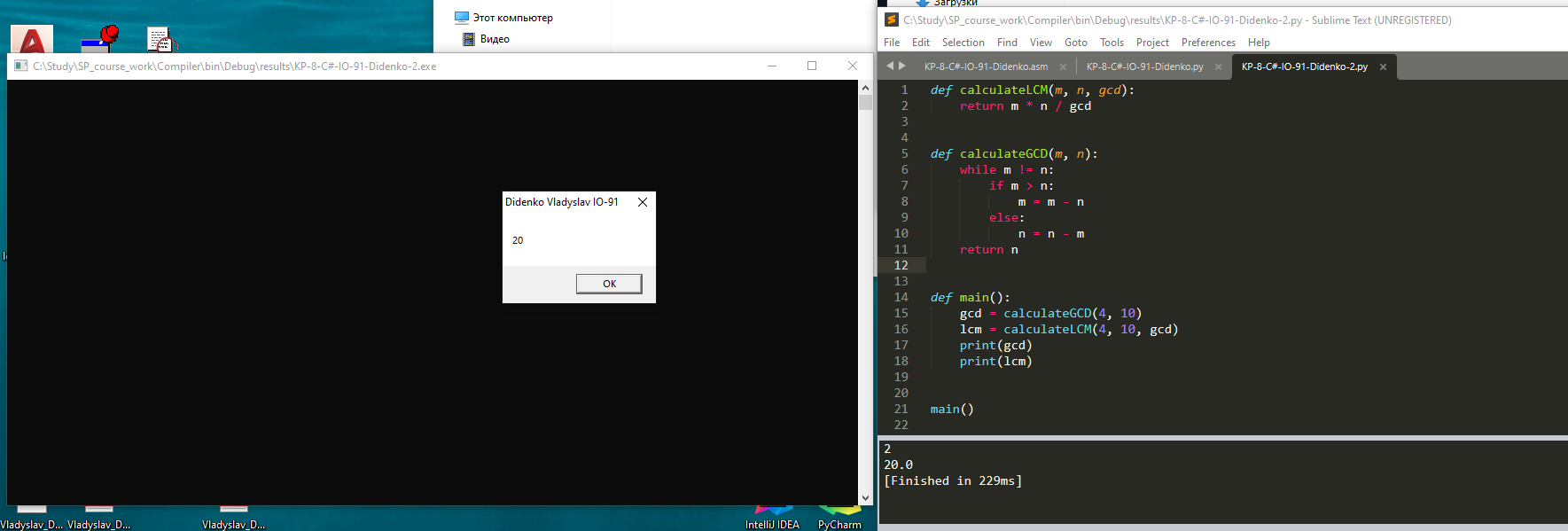


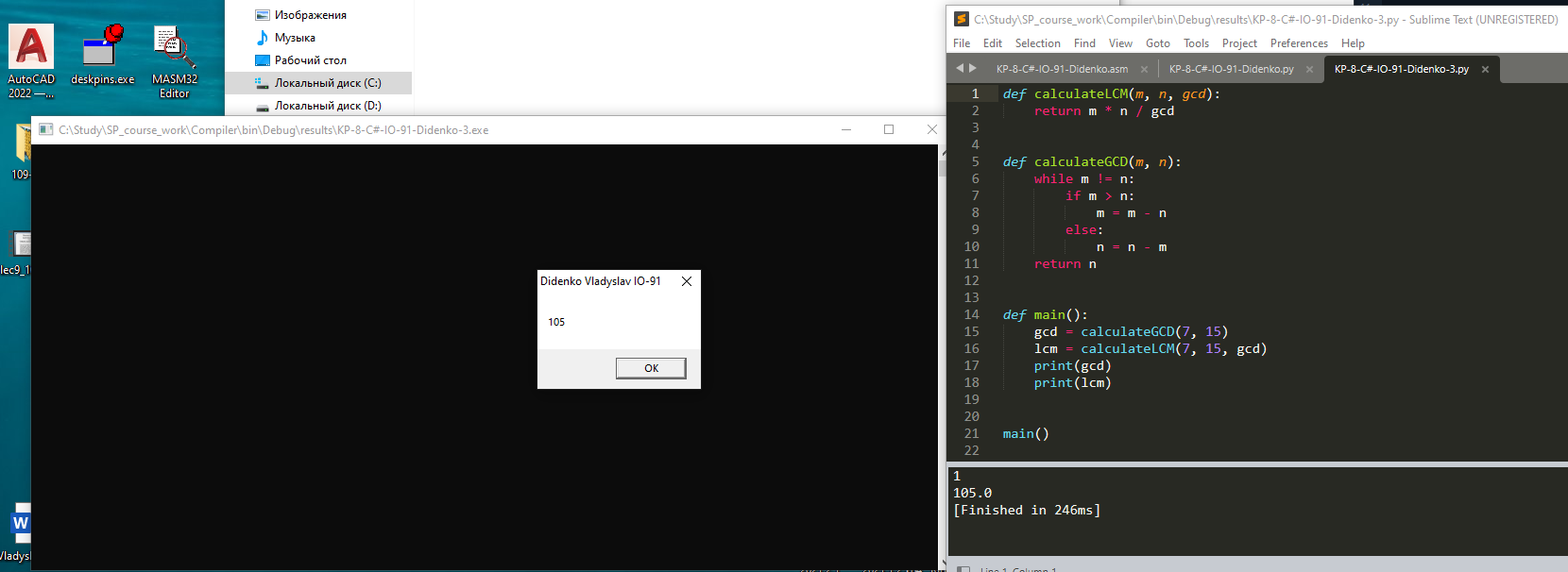
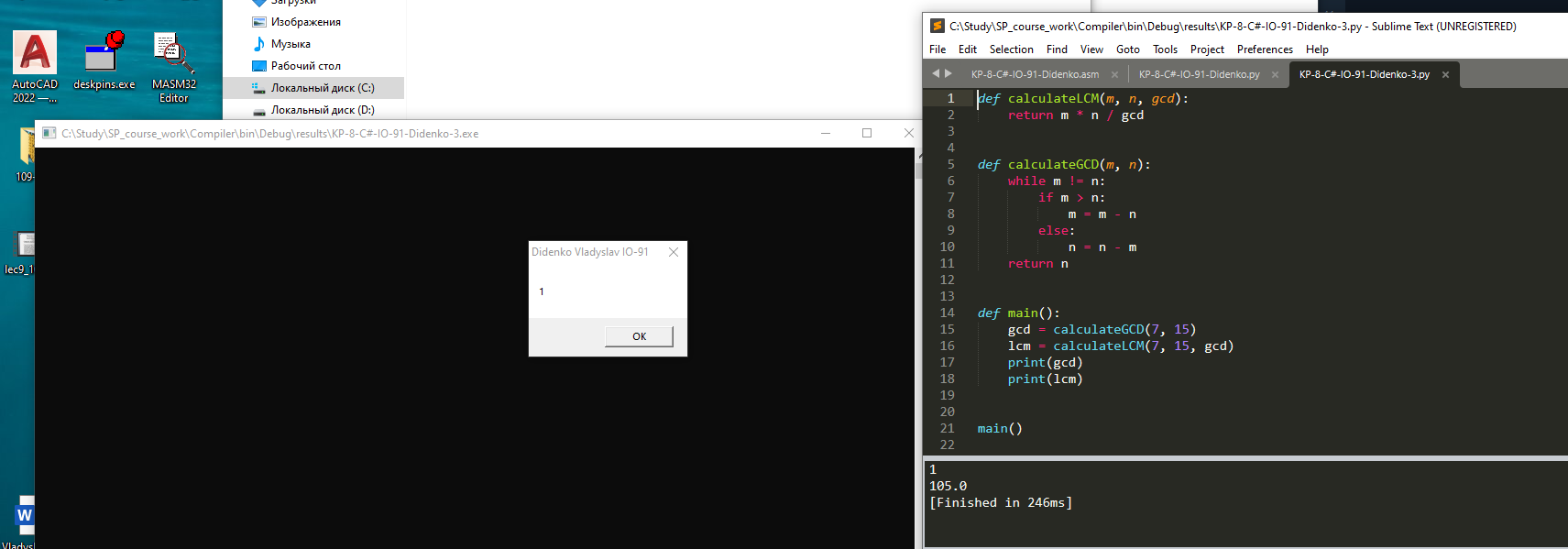
Результати перевірки працездатності програми

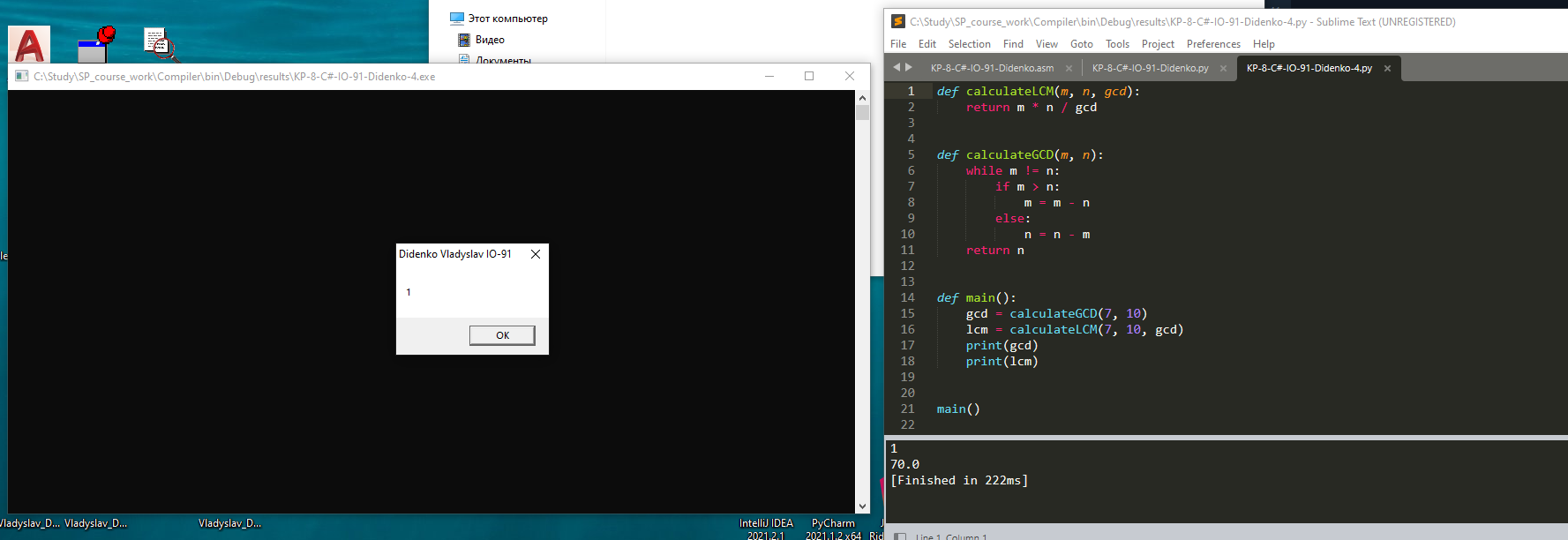
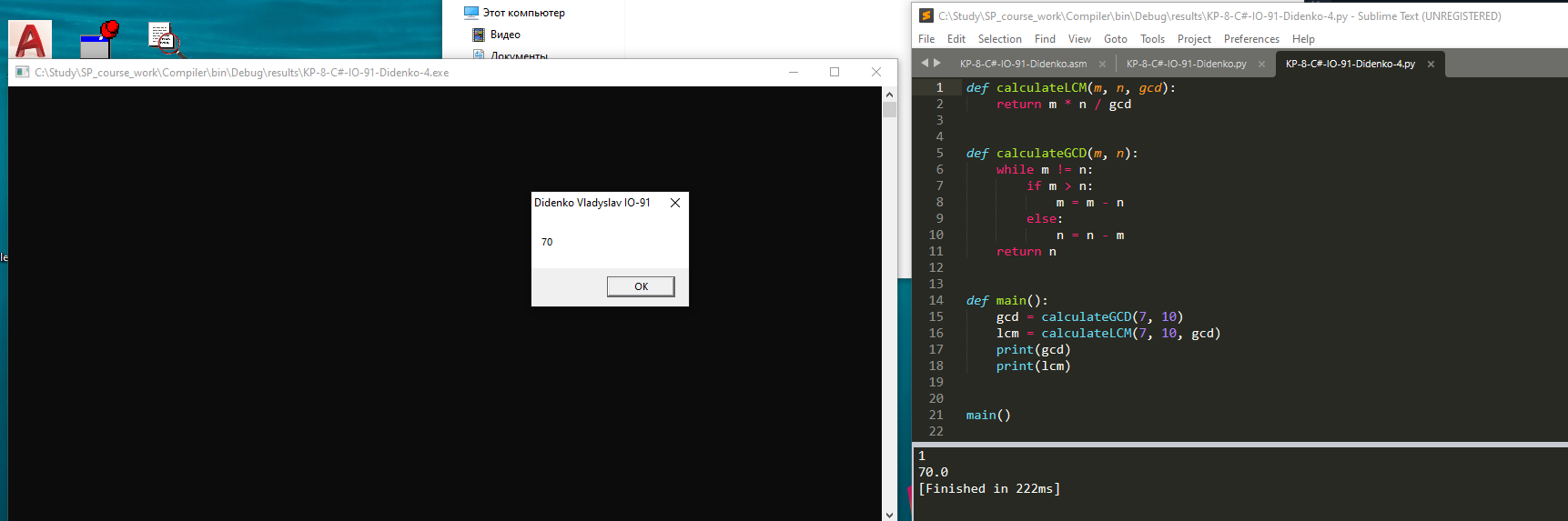
Помилки:

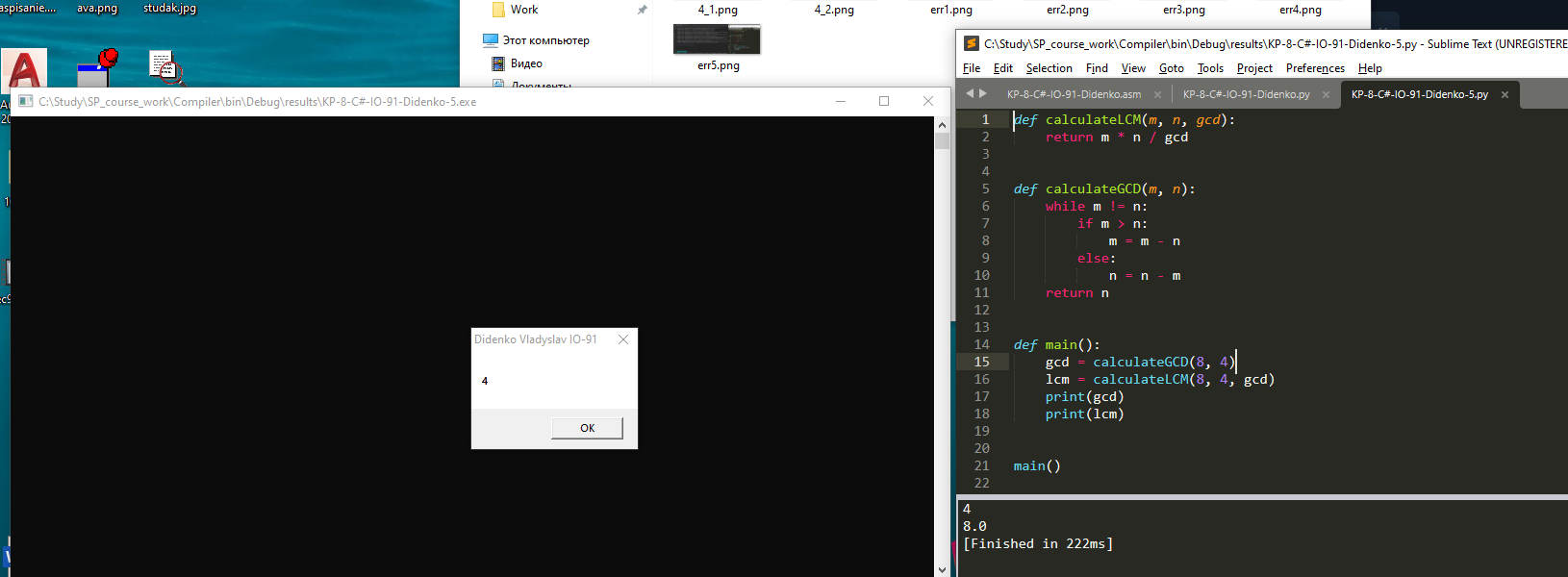
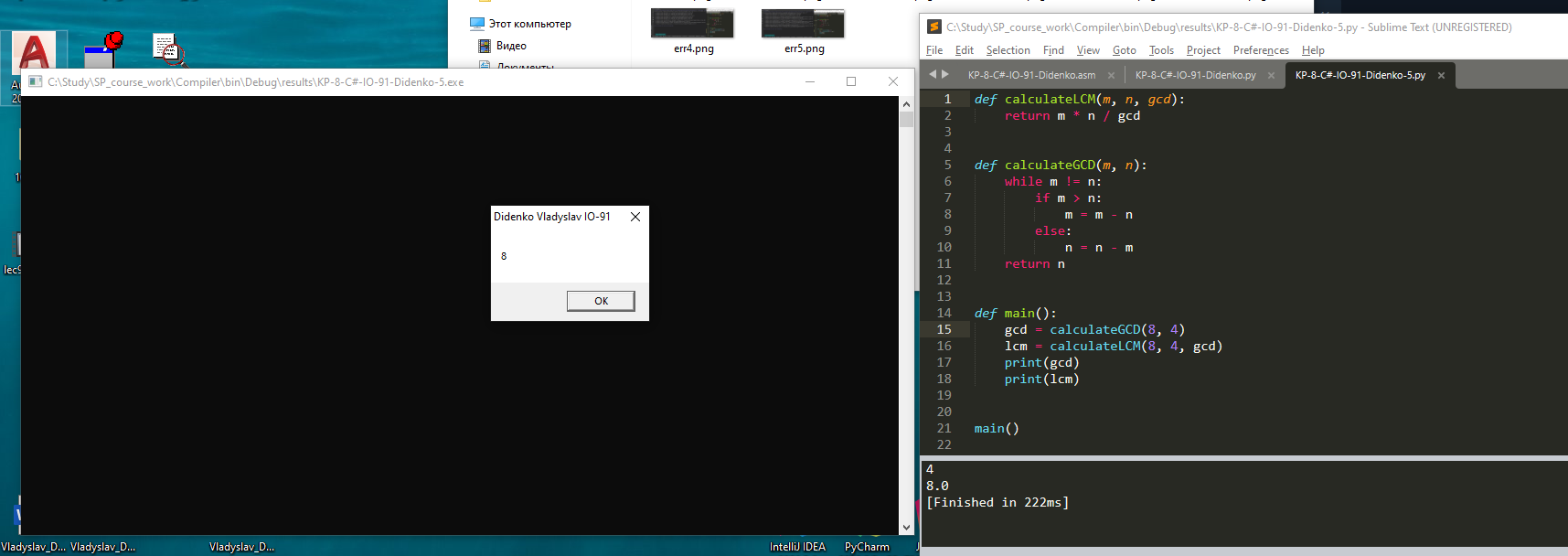
Успішне виконання програми:

НСК(5, 20) = 20  
НСД(5, 20) = 5

НСК(4, 10) = 20  
НСД(4, 10) = 2

НСК(7, 15) = 105  
НСД(7, 15) = 1

НСК(7, 10) = 70  
НСД(7, 10) = 1

НСК(8, 4) = 8  
НСД(8, 4) = 4

Висновок

В процессі написання компілятору було вивчено та засвоєно теорія побудови компіляторів, побудована тестова програма, підтверджено правильність результатів шляхом тестування. При побудові було реалізовано такі арифметичні дії:

* віднімання
* ділення

Такі логічні оператори:

* не дорівнює
* більше

Завдяки тому, що проект було побудовано за принципом Incremental Approach, подальша реалізація різних конструкцій є доволі простою, проте якщо проводити деякий аналіз на етапі лексичного аналізу, то можна було б оптимізувати найважкіший етап побудови компілятора.   
На даний момент програма не вміє розпізнавати імена, що мають у назві символ \_, через те, що як ім’я сприймаються лише набори букв, цю проблему можна вирішити шляхом додавання додаткових умов при розбиванні вхідного коду на токени, також програма може правильно визначати відступи, тільки якщо вони є табуляцією або чотирма пробілами. Можливим рішенням є знаходження найменшої послідовності пробілів, та прийняття їх, як величину одного відступа.

Список використаної літератури

1. Windows Assembly Programming Tutorial. URL: https://doc.lagout.org/operating%20system%20/Windows/winasmtut.pdf
2. Making Compiler Design Relevant for Students who will (Most Likely) Never Design a Compiler. URL: https://www2.cs.arizona.edu/classes/cs453/fall20/DOCS/teaching\_compilers.pdf
3. What's the difference between a statement and an expression in Python? URL: <https://www.quora.com/Whats-the-difference-between-a-statement-and-an-expression-in-Python-Why-is-print-%E2%80%98hi%E2%80%99-a-statement-while-other-functions-are-expressions>
4. X86 Assembly/X86 Architecture. URL: https://en.wikibooks.org/wiki/X86\_Assembly/X86\_Architecture#EFLAGS\_Register
5. Parsing expressions by precedence climbing. URL: https://eli.thegreenplace.net/2012/08/02/parsing-expressions-by-precedence-climbing