

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

До захисту допущено:

Завідувач кафедри

Сергій СТИРЕНКО

(підпис)

“ ” 2024 р.

Дипломний проєкт

на здобуття ступеня бакалавра

за освітньо-професійною програмою “Інженерія програмного забезпечення
комп’ютерних систем”

спеціальності 121 “Інженерія програмного забезпечення”

на тему: Захищений калькулятор з асинхронними діями

Виконав : студент 4 курсу, групи ІІ-03
(шифр групи)

Сенчуков Ілля Ігорович

(прізвище, ім’я, по батькові)

(підпис)

Керівник доцент, к.т.н. Павлов В. Г.

(посада, науковий ступінь, вчене звання, прізвище та ініціали)

(підпис)

Консультант (нормоконтроль) доцент, к.т.н., Волокита А. М.

(назва розділу)

(посада, вчене звання, науковий ступінь, прізвище та ініціали)

(підпис)

Рецензент професор кафедри ІСТ, д.т.н. Корнієнко Б.Я.

(посада, науковий ступінь, вчене звання, прізвище та ініціали)

(підпис)

Засвідчую, що у цьому дипломному
проєкті немає запозичень з праць інших
авторів без відповідних посилань.

Студент

(підпис)

Київ - 2024 р.

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Рівень вищої освіти - перший (бакалавр)

Освітньо-професійна програма

“Інженерія програмного забезпечення комп’ютерних систем”

спеціальності 121 “Інженерія програмного забезпечення”

ЗАТВЕРДЖУЮ

Завідувач кафедри

Сергій СТИРЕНКО

(підпис)

“ ” _____ 2024 р.

ЗАВДАННЯ

на бакалаврський дипломний проєкт студента

Сенчукова Ілля Ігоровича

1. Тема проєкту Захищений калькулятор з асинхронними діями
керівник проєкту Павлов Валерій Георгійович, к.т.н.,
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від 27 травня 2024 року №2102-с
2. Термін здачі студентом закінченого проєкту 8 червня 2024 р.
3. Вихідні дані до проєкту технічна документація, теоретичні дані.
4. Зміст розрахунково-пояснювальної записки (перелік питань, які розробляються)
Розділ 1. Огляд предметної області
Розділ 2. Огляд алгоритмів та технологій для розробки системи.
Розділ 3. Процес розробки системи.
Розділ 4. Прикладні аспекти застосування додатку.

5. Перелік графічного матеріалу (з точним позначенням обов'язкових креслень) ER-діаграма (функціональна схема), схема взаємодії компонентів системи (структурна схема), алгоритм аутентифікації (принципова схема), .

6. Консультанта проєкту, з вказівкою розділів проєкту, які до них вносяться

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Нормоконтроль	Волокита А. М.		

7. Дата видачі завдання «1» грудня 2023 р.

Календарний план

№ п/п	Найменування етапів дипломного проєкту	Терміни виконання етапів проєкту	Примітки
1.	<i>Затвердження теми проєкту</i>	<i>30.10.2023-15.12.2023</i>	
2.	<i>Вивчення та аналіз завдання</i>	<i>16.12.2023-15.01.2024</i>	
3.	<i>Розробка архітектури та загальної структури системи</i>	<i>16.01.2024-25.02.2024</i>	
4.	<i>Розробка структур окремих підсистем</i>	<i>26.02.2024-5.03.2024</i>	
5.	<i>Програмна реалізація системи</i>	<i>6.03.2024-15.04.2024</i>	
6.	<i>Оформлення пояснювальної записки</i>	<i>15.04.2024-20.05.2024</i>	
7.	<i>Захист програмного продукту</i>	<i>25.04.2024</i>	
8.	<i>Передзахист</i>	<i>23.05.2024</i>	
9.	<i>Захист</i>	<i>20.06.2024</i>	

Студент-дипломник _____ Ілля СЕНЧУКОВ
(підпис)

Керівник проєкту _____ Валерій ПАВЛОВ
(підпис)

АНОТАЦІЯ

У даній роботі було детально розглянуто концепцію створення захищеного калькулятора з асинхронними діями для платформ iOS і macOS. Були проаналізовані основні принципи забезпечення безпеки обчислень та управління асинхронними процесами. На основі цього аналізу було обрано найбільш підходящу архітектуру для розробки додатку. В результаті роботи було створено прототип захищеного калькулятора, який здатен одночасно працювати на iOS і macOS, виконувати асинхронні обчислення, проведено дослідження його ефективності. Розроблений додаток дає можливість безпечно виконувати складні обчислення та працювати з асинхронними задачами. Програмний продукт був розроблений на мові Swift.

Ключові слова: Swift, багатопоточність, iOS, CoreData

ANNOTATION

This work thoroughly examined the concept of creating a secure calculator with asynchronous actions for iOS and macOS platforms. The main principles of ensuring computational security and managing asynchronous processes were analysed. Based on this analysis, the most suitable architecture for the application development was selected. As a result of the work, a prototype of the secure calculator was created, capable of running on both iOS and macOS simultaneously, performing asynchronous calculations. The efficiency of the prototype was studied. The developed application allows for secure execution of complex calculations and handling asynchronous tasks on various devices. The software product was developed in Swift.

Keywords: Swift, multi-threading, iOS, CoreData

ТЕХНІЧНЕ ЗАВДАННЯ
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «Захищений калькулятор з асинхронними діями»

Київ - 2024

ЗМІСТ

НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ	2
ПІДСТАВИ ДЛЯ РОЗРОБКИ.....	2
МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ.....	2
ДЖЕРЕЛА РОЗРОБКИ.....	3
ТЕХНІЧНІ ВИМОГИ.....	3
Вимоги до розробленого продукту	3
Вимоги до програмного забезпечення.....	3
Вимоги до апаратної частини.....	3
ЕТАПИ РОЗРОБКИ	4

					ІАЛЦ.467200.002 ТЗ					
		№ докум.	Підпис	Дата						
Розробив		Сенчуков І.І.			Захищений калькулятор з асинхронними діями Технічне завдання			Літ.	Аркуш	Аркушів
Перевірив		Павлов В.Г.							1	3
								КПІ ім. Ігоря Сікорського, ФІОТ,		
Н. Контр.		Волокита А.М.								
Затвердив		Стіренко С.Г.								

1 НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Дане технічне завдання поширюється на розробку програмного додатку на тему «Захищений калькулятор з асинхронними діями»

Областю застосування цієї системи є фінансові та економічні розрахунки, наукові дослідження, інженерні обчислення, навчальні процеси, а також будь-які інші галузі, де важливо забезпечити високий рівень безпеки даних і підтримку асинхронних операцій для підвищення продуктивності.

2 ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки даної системи є завдання для виконання роботи кваліфікаційно-освітнього рівня «бакалавр інженерії програмного забезпечення», який був затверджений факультетом “Інформатики та обчислювальної техніки” кафедрою обчислювальної техніки Національного технічного Університету України «Київський Політехнічний інститут ім. Ігоря Сікорського».

3 МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою цього проєкту є розробка безпечного калькулятора, здатного виконувати асинхронні обчислення на платформах iOS та macOS. Проєкт спрямований на забезпечення високого рівня безпеки під час виконання обчислень, оптимізацію продуктивності та можливість використання додатку на різних пристроях Apple.

Призначення цього проєкту полягає в створенні інструменту для користувачів, який дозволяє безпечно та ефективно виконувати складні математичні обчислення, керувати асинхронними задачами та забезпечувати високу продуктивність і зручність використання на мобільних і настільних платформах.

					ІАЛЦ.467200.002 ТЗ	Арк.
						2
Зм.	Арк.	№ докум.	Підпис	Дата		

4 ДЖЕРЕЛА РОЗРОБКИ

Джерелом розробки даного дипломного проекту є офіційні документації, публікації та статті в мережі Інтернет на дану тему, науково-технічна література.

5 ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до розробленого продукту

Розроблена система має виконувати такі вимоги:

- Простий і інтуїтивно-зрозумілий інтерфейс системи.
- Підтримка автоматичного розпізнавання формул і виразів.
- Доступ до різних одиниць вимірювання (валюти, довжини, ваги, швидкості тощо) для забезпечення різноманітних потреб у обчисленнях.
- Підтримка збереження та повторного використання змінних у різних обчисленнях.
- Можливість збереження шаблонів для швидкого та ефективного використання в майбутньому.
- Забезпечення високого рівня безпеки даних користувача.
- Надати вичерпну та зрозумілу документацію для розробленого продукту.

5.2. Вимоги до програмного забезпечення

- ОС iOS, iPadOS чи MacOS.

5.3. Вимоги до апаратної частини

- ЦП не менше ніж Intel® Core (TM) i3-2100T.
- ROM не менше ніж 64 ГБ.
- RAM не менше ніж 4 ГБ.

					ІАЛЦ.467200.002 ТЗ	Арк.
						3
Зм.	Арк.	№ докум.	Підпис	Дата		

6 ЕТАПИ РОЗРОБКИ

Назва етапів виконання	Термін виконання
Затвердження теми роботи	30.10.2023-15.12.2023
Вивчення та аналіз завдання	16.12.2023-15.01.2024
Розробка архітектури та загальної структури системи	16.01.2024-25.02.2024
Розробка структур окремих частин системи	26.02.2024-5.03.2024
Програмна реалізація системи	6.03.2024-15.04.2024
Оформлення пояснювальної записки	15.04.2024-20.05.2024

					ІАЛЦ.467200.002 ТЗ	Арк.
						4
Зм.	Арк.	№ докум.	Підпис	Дата		

ПОЯСНЮВАЛЬНА ЗАПИСКА
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «Захищений калькулятор з асинхронними діями»

Київ - 2024

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ	5
ВСТУП	6
РОЗДІЛ 1. ОГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ	8
1.1 Огляд існуючих рішень	8
1.1.1 Soulver.....	8
1.1.2 Omni Calculator	9
1.1.3 Numbr.....	10
1.1.4 Apple Calculator.....	11
1.2 Риси майбутньої розробки.....	13
ВИСНОВОК ДО РОЗДІЛУ 1	15
РОЗДІЛ 2. ОГЛЯД АЛГОРИТМІВ ТА ТЕХНОЛОГІЙ ДЛЯ РОЗРОБКИ СИСТЕМИ	16
2.1 Опис функціональної моделі.....	16
2.1.1 Функціональні можливості.....	16
2.1.2 Інтерфейс користувача.....	16
2.2 Огляд методів парсингу тексту	17
2.2.1 Прості парсери	17
2.2.2 Інфіксні парсери	17
2.2.3 Рекурсивні парсери	18
2.2.4 Парсери зворотної польської нотації.....	18
2.2.5 AST парсери.....	18
2.3 Огляд мов програмування для мобільної розробки	18
2.3.1 Swift	19
2.3.2 Dart.....	21
2.3.3 JavaScript	22
2.4 Графічний інтерфейс мовою Swift.....	25
2.4.1 SwiftUI	26
2.4.2 UIKit.....	26

2.5 База даних.....	26
2.6 Основні функціональні модулі.....	27
2.6.1 Отримання вхідних даних	27
2.6.2 Інтерпретатор.....	27
2.6.3 Безпечне зберігання обчислень.....	28
2.6.4 Захищений доступ до API.....	29
ВИСНОВОК ДО РОЗДІЛУ 2	31
РОЗДІЛ 3. ПРОЦЕС РОЗРОБКИ СИСТЕМИ	32
3.1 Підключення бази даних.....	32
3.1.1 CoreDataStack.....	32
3.1.2 Опис властивостей моделі даних.....	33
3.2 Структуровані запити до API	34
3.3. Розробка інтерпретатора.....	36
3.3.1 StatementLine.....	37
3.3.2 InputLayout	40
3.3.3 Tokenizer.....	41
3.3.4 Lexer.....	43
3.3.5 Parser	45
3.3.6 SyntacticAnalyst.....	48
3.3.7 DependencyGraph	49
3.3.8 EvaluationContext	51
3.3.9 Evaluator	52
3.4 Створення шаблонів обчислень	60
ВИСНОВКИ ДО РОЗДІЛУ 3	63
РОЗДІЛ 4. ПРИКЛАДНІ АСПЕКТИ ЗАСТОСУВАННЯ ДОДАТКУ	65
4.1 Демонстрація практичних можливостей розробленого додатку	65
4.2 Перспективи подальшого розвитку додатку.....	71
ВИСНОВОК ДО РОЗДІЛУ 4	73
ВИСНОВКИ	74

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....	76
ДОДАТОК 1	77
ДОДАТОК 2	79
ДОДАТОК 3	81
ДОДАТОК 4	83

					ІАЛЦ.467200.003 ПЗ	Арк.
						4
Зм.	Арк.	№ докум.	Підпис	Дата		

ПЕРЕЛІК СКОРОЧЕНЬ

UI	(User Interface) Інтерфейс користувача
RPN	(Reverse Polish Notation) Зворотня польська нотація
AST	(Abstract Syntax Tree) Абстрактне синтаксичне дерево
БД	База даних
JWT	JSON Web Tokens
AOT	Ahead-Of-Time
JIT	Just-In-Time

ВСТУП

У сучасному світі, де швидкість та ефективність виконання завдань мають вирішальне значення, використання асинхронних технологій стає критичним аспектом розробки програмного забезпечення. Особливо це стосується мобільних та настільних додатків, які повинні забезпечувати безперебійну роботу та високу продуктивність при обробці великих обсягів даних. Розробка калькулятора з асинхронними діями в такому контексті набуває надзвичайної актуальності.

Сучасні користувачі очікують, що додатки будуть не лише функціональними, але й швидкими, зручними та безпечними у використанні. Це особливо важливо для калькуляторів, які застосовуються у складних математичних обчисленнях, фінансовому аналізі, наукових дослідженнях та інженерних розрахунках. Асинхронні процеси дозволяють виконувати обчислення у фоновому режимі, не блокуючи основний інтерфейс користувача, що значно покращує загальний досвід використання додатку.

Метою цього проєкту є створення калькулятора, здатного виконувати асинхронні обчислення з високим рівнем захищеності. Це дозволить забезпечити високу продуктивність додатку, швидкість обробки даних, зручність у використанні та надійний захист даних користувачів. Важливим аспектом є розробка архітектури додатку, яка підтримуватиме асинхронні процеси та захищатиме дані користувачів.

Завдання проєкту:

1. Вибрати технології та підходи, що забезпечать ефективну обробку даних у фоновому режимі без впливу на основну роботу додатку.
2. Реалізувати функцію створення шаблонів обчислень, яка дозволить користувачам зберігати часто використовувані обчислення і швидко їх повторно використовувати.

					ІАЛЦ.467200.003 ПЗ	Арк.
						6
Зм.	Арк.	№ докум.	Підпис	Дата		

3. Впровадити комплексні заходи безпеки для захисту даних користувачів, включаючи шифрування та біометричну аутентифікацію.
4. Реалізувати механізми аутентифікації та авторизації для запобігання несанкціонованого доступу до API додатку.

					ІАЛЦ.467200.003 ПЗ	Арк.
						7
Зм.	Арк.	№ докум.	Підпис	Дата		

РОЗДІЛ 1. ОГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Огляд існуючих рішень

1.1.1 Soulver

Soulver - один з найпопулярніших калькуляторів з автоматичним розпізнаванням формул з тексту. Цей додаток дозволяє користувачам виконувати обчислення, використовуючи текстовий ввід, що робить його схожим на текстовий редактор з додатковими можливостями. Ви можете вводити слова поряд з обчисленнями, що робить цей калькулятор особливо зручним для створення зрозумілих записів.

Soulver також підтримує конвертацію одиниць вимірювання, що дозволяє легко переводити довжини, ваги, об'єми, валюти та інші одиниці. Крім того, додаток підтримує оголошення змінних, що надає користувачам додаткову гнучкість при виконанні складних обчислень [1]. Наприклад, ви можете оголошувати змінні для зберігання постійних значень і використовувати їх у різних обчисленнях.

Переваги:

- Інтуїтивний інтерфейс, що дозволяє легко вводити та редагувати обчислення.
- Підтримка одиниць вимірювання та їх автоматична конвертація.
- Можливість оголошення змінних для більшої гнучкості в обчисленнях.
- Підтримка синтаксису, подібного до природної мови, що робить обчислення простими та зрозумілими.

Недоліки:

- Відсутність можливості зберегти обчислення як шаблон.
- Обмежена функціональність у порівнянні з деякими спеціалізованими калькуляторами.

# Bills	
Electricity = \$150/month	\$150.00/month
Electricity for 10 months in £	£1,106.33
Gas = £1,000	£1,000.00
Total = £1,106.33 + £1,000.00	£2,106.33

Рисунок 1.1 – Демонстрація Soulver 3.

1.1.2 Omni Calculator

Omni Calculator - відомий веб-застосунок, який містить близько 3 тисячі різних шаблонів обчислень на майже всі життєві випадки. Цей калькулятор призначений для тих, хто потребує швидких і точних розрахунків для різних сценаріїв - від простих до складних.

Omni Calculator дозволяє використовувати готові шаблони для обчислень у різних галузях, таких як фінанси, медицина, будівництво, наука та багато інших [2]. Це робить його надзвичайно зручним для професіоналів, які часто стикаються зі специфічними обчислювальними задачами. Користувачі можуть обирати з готових шаблонів, що забезпечує швидке вирішення типових завдань.

Переваги:

- Велика кількість готових шаблонів обчислень.
- Підтримка різних одиниць вимірювання при обчисленнях.
- Легкість у використанні завдяки великій базі готових рішень.
- Не зберігає ніяких даних користувача

Недоліки:

- Неможливість створити власний шаблон
- Залежність від Інтернету.
- Інтерфейс може бути перевантаженим через велику кількість опцій.

					ІАЛЦ.467200.003 ПЗ	Арк.
						9
Зм.	Арк.	№ докум.	Підпис	Дата		

- Обмежені можливості кастомізації для користувачів з нестандартними потребами.

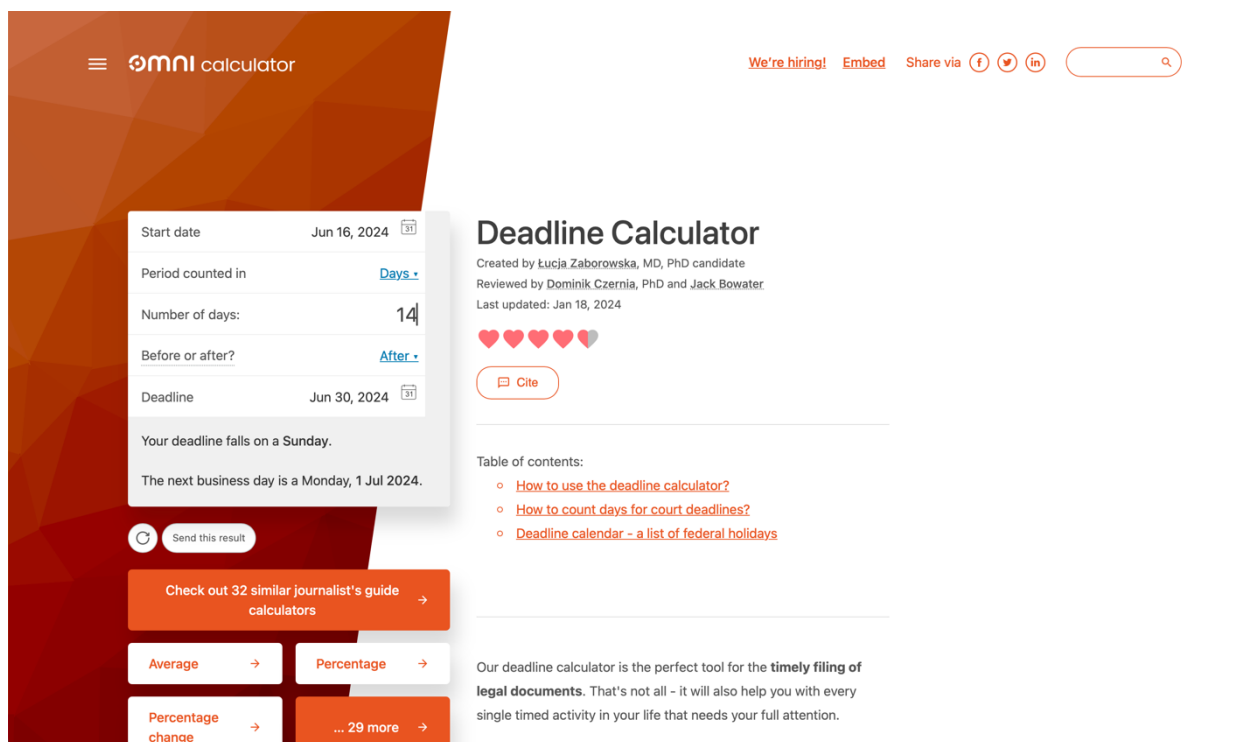


Рисунок 1.2 – Демонстрація Omni Calculator.

1.1.3 Numbr

Numbr - це браузерний калькулятор, який дозволяє користувачам виконувати обчислення з використанням різних валют. Цей калькулятор призначений для тих, хто цінує простоту та доступність, оскільки він не вимагає встановлення додатка і може бути використаний безпосередньо через веб-браузер.

Numbr підтримує базові математичні операції та конвертацію одиниць вимірювання, що робить його корисним для швидких розрахунків у повсякденному житті [3]. Крім того, він має мінімалістичний інтерфейс, що забезпечує легкість і зручність у використанні.

Переваги:

- Легкий доступ через браузер у вигляді розширення.
- Підтримка конвертації валют.

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		10

- Мінімалістичний інтерфейс.
- Можливість використання на будь-якому пристрої з доступом до Інтернету.

Недоліки:

- Залежність від Інтернету.
- Обмежена функціональність порівняно з додатками.
- Відсутність можливості зберігати та ділитися обчисленнями.
- Відсутність одиниць вимірювання, окрім валют.
- Відсутність розширених функцій для професійних користувачів.

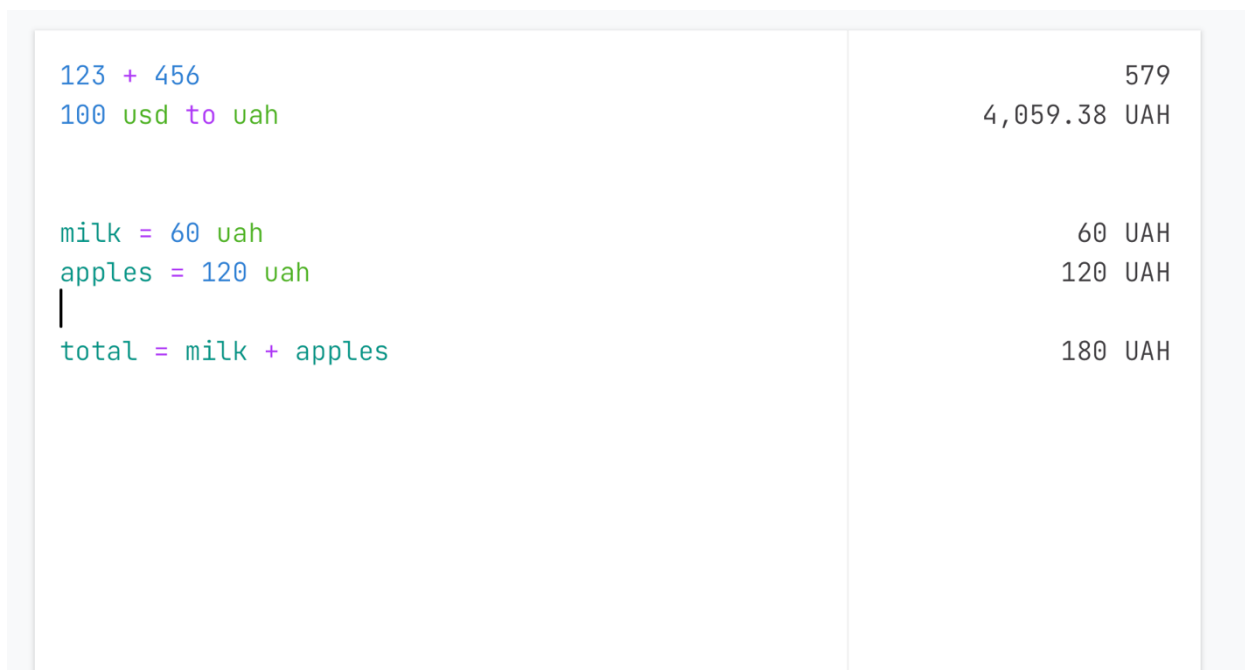


Рисунок 1.3 – Демонстрація роботи Numbrix.

1.1.4 Apple Calculator

Apple Calculator - це стандартний додаток від компанії Apple, який встановлюється на кожен пристрій iOS та macOS. Це один з найпоширеніших калькуляторів, який використовується мільйонами користувачів по всьому світу.

Apple Calculator підтримує всі стандартні математичні операції, такі як додавання, віднімання, множення та ділення. Крім того, він має науковий

режим, який дозволяє виконувати більш складні обчислення, такі як тригонометричні функції, логарифми та експоненти. Додаток відомий своєю простотою у використанні та інтеграцією з іншими додатками Apple.

Переваги:

- Простий в користуванні.
- Підтримка всіх стандартних математичних операцій.
- Інтеграція з операційною системою iOS та macOS, забезпечуючи швидкий доступ.
- Надійність та стабільність роботи.

Недоліки:

- Неможливість зберегти свої обчислення.
- Відсутність підтримки більш складних математичних операцій.

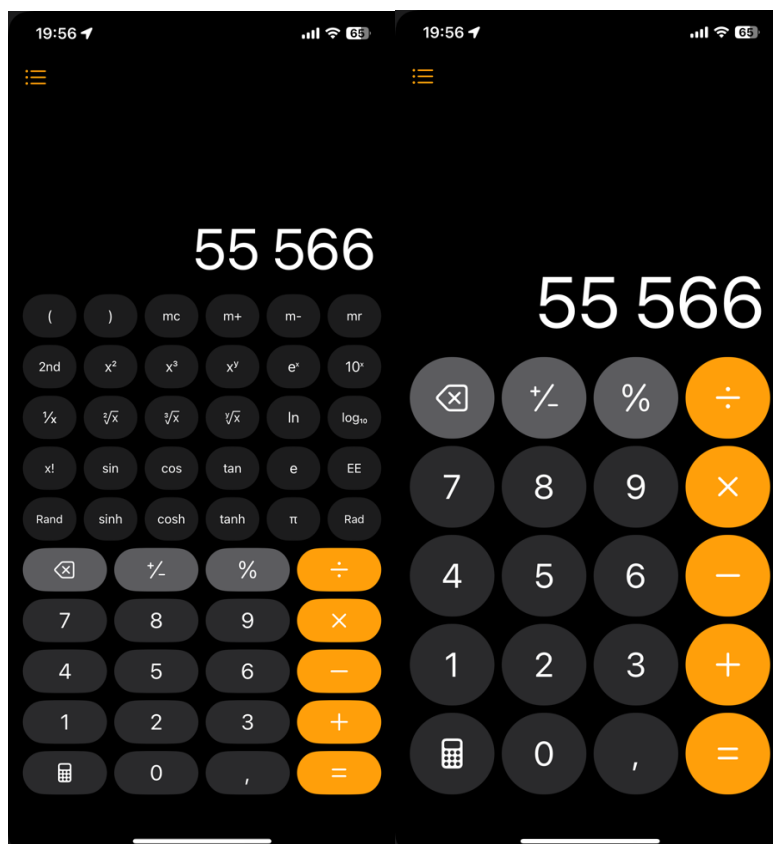


Рисунок 1.4 – Демонстрація Apple Calculator.

					ІАЛЦ.467200.003 ПЗ	Арк.
						12
Зм.	Арк.	№ докум.	Підпис	Дата		

Таблиця 1.1 – Порівняльна таблиця аналогів.

Калькулятор	Soulver	Numbr	Omni Calculator	Apple Calculator
Автоматичне розпізнавання форму	Так	Так	Ні	Ні
Підтримка одиниць вимірювання	Так	Валюти	Так	Ні
Оголошення змінних	Так	Так	Ні	Ні
Збереження обчислень	Так	Ні	Ні	Так
Створення шаблонів	Ні	Ні	Так	Ні
Доступність	iOS, macOS	Веб	Веб	iOS, macOS
Інтернет-залежність	Ні	Так	Так	Ні

1.2 Риси майбутньої розробки

В попередньому пункті ми розглянули аналоги до нашого проєкту, а тепер детально розглянемо, які саме риси майбутньої розробки ми будемо намагатися втілити, щоб зробити наш додаток унікальним і функціональним.

Перш за все, ми плануємо додати можливість користувачам зберігати свої обчислення. На відміну від більшості веб-аналогів, які зберігають дані в базі даних на сервері, наш підхід буде спрямований на зберігання даних безпосередньо на пристрої користувача. Такий підхід не тільки підвищить зручність використання, але й надасть значні переваги з точки зору безпеки, оскільки дані залишаються під повним контролем користувача і не потребують передачі через інтернет.

По-друге, ми інтегруємо можливість створення шаблонів обчислень. Це дозволить користувачам зберігати часто використовувані обчислення і

швидко їх перевикористовувати. Завдяки цьому користувачі зможуть значно заощадити час і зусилля при виконанні рутинних або повторюваних обчислень, що робить наш додаток ще більш зручним і функціональним.

По-третє, ми будемо підтримувати широкий спектр одиниць вимірювання та забезпечимо можливість конвертації між ними. Для цього ми будемо використовувати асинхронні дії, що дозволить швидко і точно виконувати необхідні обчислення без затримок і збоїв. Це особливо важливо для користувачів, які працюють з різними системами одиниць і потребують швидкої конвертації даних у різних форматах.

Нарешті, ми зробимо акцент на безпеці додатку. Пріоритет – забезпечити високий рівень захисту даних користувачів. Наш підхід передбачає використання шифрування, безпечного зберігання інформації та захищеної аутентифікації користувачів, що мінімізує ризики несанкціонованого доступу до даних.

					ІАЛЦ.467200.003 ПЗ	Арк.
						14
Зм.	Арк.	№ докум.	Підпис	Дата		

Висновок до розділу 1

У ході виконання цієї роботи ми провели ґрунтовний і всебічний аналіз існуючих рішень у сфері мобільної розробки. Цей ретельний аналіз дозволив нам не лише виявити основні сильні та слабкі сторони аналогічних додатків, але й глибше зрозуміти поточні тенденції та вимоги ринку. Завдяки цьому ми змогли чітко окреслити пріоритети та визначити найбільш важливі характеристики, які слід було реалізувати в нашому проєкті, щоб зробити його максимально ефективним та конкурентоспроможним.

Аналіз існуючих рішень у сфері калькуляторів показує, що кожен з розглянутих додатків має як спільні риси, так і унікальні особливості. Ці програми розроблені для задоволення різних потреб користувачів, від простих повсякденних обчислень до складних наукових розрахунків.

Аналізуючи кожен аспект роботи наявних рішень, ми зосередилися на ідентифікації ключових елементів, що забезпечують успіх і популярність додатків серед користувачів. Це дало нам змогу не лише перейняти найкращі практики, але й уникнути поширених помилок, характерних для інших продуктів.

					ІАЛЦ.467200.003 ПЗ	Арк.
						15
Зм.	Арк.	№ докум.	Підпис	Дата		

РОЗДІЛ 2. ОГЛЯД АЛГОРИТМІВ ТА ТЕХНОЛОГІЙ ДЛЯ РОЗРОБКИ СИСТЕМИ

2.1 Опис функціональної моделі

2.1.1 Функціональні можливості

Додаток має забезпечувати такі функціональні можливості:

- Обчислення простих математичних виразів: додавання, віднімання, множення та ділення, відсотки.
- Багаторядкові вирази мають бути обчислені по одному рядку.
- Використання одиниць вимірювання.
- Конвертування одиниць вимірювання між собою.
- Декларування та повторне використання змінних.
- Обчислення математичних функцій: $\sin$, $\cos$ тощо.
- Зберігання зашифрованих обчислень в постійній пам'яті пристрою користувача.
- Створення шаблонів обчислень.
- Виконувати асинхронні дії: конвертація валют, отримання погоди тощо.

2.1.2 Інтерфейс користувача

Користувач повинен мати можливість швидко та зручно ввести потрібний математичний вираз і отримати результат у реальному часі. Додаток повинен забезпечувати користувача візуальним зворотним зв'язком, підтверджуючи правильне розуміння введеного виразу.

Результат обчислення повинен бути відображен з правої сторони від того рядка, який створив цей результат.

У разі виникнення помилки, додаток має негайно сповістити користувача про проблему, надавши детальну інформацію та можливості для діагностики і виправлення помилки.

2.2 Огляд методів парсингу тексту

Парсинг математичних виразів є однією з основних задач додатку. Цей процес полягає в аналізі та обробці текстових математичних виразів з метою їхнього перетворення у структуровану форму, що дозволяє виконувати подальші обчислення. Парсери математичних виразів знаходять широке застосування в різноманітних програмних системах, включаючи калькулятори, системи комп'ютерної алгебри, компілятори мов програмування, а також наукові та інженерні додатки.

В цьому розділі розглянемо види парсерів математичних виразів.

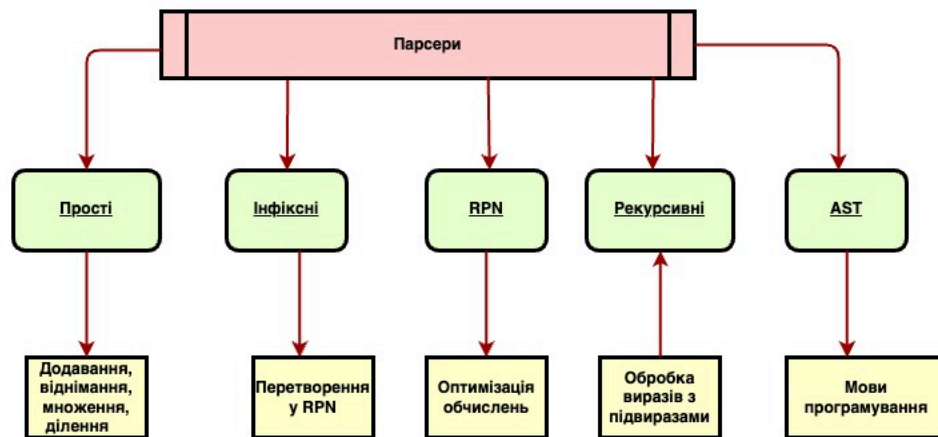


Рисунок 2.1 – Види та застосування парсерів.

2.2.1 Прості парсери

Прості парсери призначені для обробки базових математичних виразів, що включають основні арифметичні операції, такі як додавання, віднімання, множення та ділення. Такі парсери часто використовуються у простих калькуляторах та фінансових додатках. Вони зазвичай реалізуються за допомогою рекурсивного спуску або регулярних виразів.

2.2.2 Інфіксні парсери

Інфіксні парсери обробляють математичні вирази, записані у звичній для нас інфіксній нотації (наприклад, $3 + 5 * 2$). Для реалізації таких парсерів часто

використовують алгоритм сортувального перетворення (Shunting Yard), який дозволяє перетворити інфіксні вирази у зворотну польську нотацію для подальшого обчислення.

2.2.3 Рекурсивні парсери

Рекурсивні парсери використовуються для обробки складних виразів з вкладеними підвиразами та функціями. Вони зазвичай реалізуються за допомогою рекурсивного спуску, де кожна функція парсера відповідає за розбір певного типу підвиразу. Це дозволяє підтримувати оператори з різними пріоритетами та дужки.

2.2.4 Парсери зворотної польської нотації

Парсери зворотної польської нотації (RPN) обробляють вирази, записані у постфіксній нотації (наприклад, $3\ 5\ 2\ *\ +$). Використання RPN дозволяє уникнути необхідності в дужках для визначення порядку операцій, що спрощує обчислення. Такі парсери часто використовуються у системах, де важлива оптимізація обчислень.

2.2.5 AST парсери

AST парсери будують абстрактне синтаксичне дерево для складних виразів та виконують обчислення на основі цього дерева. Вони використовуються у мовах програмування та складних математичних системах, де необхідно підтримувати різноманітні математичні функції, оператори та змінні [4].

2.3 Огляд мов програмування для мобільної розробки

У сучасному світі технологій вибір мови програмування для розробки додатків відіграє вирішальну роль. Мобільні додатки відіграють ключову роль у повсякденному житті мільйонів людей. Від соціальних мереж до фінансових інструментів і розважальних платформ – мобільні додатки забезпечують

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		18

зручний доступ до інформації та сервісів. Вибір правильної мови програмування для розробки мобільних додатків є критично важливим, оскільки це впливає на продуктивність, функціональність і зручність використання додатку.

У цьому пункті ми розглянемо три популярні мови програмування для мобільної розробки: Swift, Dart та JavaScript. Кожна з цих мов має свої унікальні особливості, переваги та недоліки, які роблять їх придатними для різних типів проектів.

2.3.1 Swift

Swift – це сучасна мова програмування, розроблена компанією Apple у 2014 році. Вона призначена для створення додатків на платформах iOS, macOS, watchOS та tvOS. Swift поєднує в собі продуктивність і безпеку, а також простоту і легкість у використанні, що робить її ідеальною для як досвідчених розробників, так і для початківців [7]. Мова активно підтримується Apple і постійно оновлюється, щоб відповідати вимогам сучасних технологій.

Основні переваги:

- Однією з головних переваг Swift є її висока продуктивність. Завдяки використанню компілятора LLVM, Swift забезпечує швидке виконання коду, що позитивно впливає на продуктивність додатків. Крім того, Swift має сучасний синтаксис, який є інтуїтивно зрозумілим і легким для освоєння. Це дозволяє розробникам швидко писати чистий і зрозумілий код.
- Безпека також є ключовою перевагою Swift. Мова включає в себе багато функцій, які допомагають уникати типових помилок програмування. Наприклад, Swift використовує опціональні типи, що дозволяє уникнути проблем, пов'язаних з null-значеннями. Автоматичне управління пам'яттю за допомогою ARC (Automatic

					ІАЛЦ.467200.003 ПЗ	Арк.
						19
Зм.	Арк.	№ докум.	Підпис	Дата		

Reference Counting) забезпечує ефективне використання ресурсів і запобігає витокам пам'яті.

- Swift має потужну екосистему і активну спільноту розробників. Apple надає широкий спектр інструментів і бібліотек для роботи зі Swift, включаючи Xcode – інтегроване середовище розробки, що значно спрощує процес створення додатків. Активна підтримка і регулярні оновлення від Apple гарантують, що Swift залишається актуальною і потужною мовою програмування.

Недоліки:

- Хоча Swift має багато переваг, вона також має деякі недоліки. Одним з них є обмежена кросплатформеність. Swift розроблена в основному для екосистеми Apple, тому її використання обмежене додатками для iOS, macOS, watchOS та tvOS. Це може бути проблемою для розробників, які хочуть створювати додатки для інших платформ, таких як Android.
- Ще одним недоліком Swift є відносна новизна мови. Хоча Swift швидко розвивається і набуває популярності, вона все ще має менше бібліотек і ресурсів у порівнянні з більш старими мовами програмування, такими як JavaScript або Java. Це може створювати певні труднощі для розробників, які шукають специфічні інструменти або бібліотеки.

Незважаючи на ці недоліки, Swift залишається потужним і ефективним інструментом для розробки мобільних додатків на платформах Apple. Її переваги, такі як висока продуктивність, безпека і зручність у використанні, роблять Swift ідеальним вибором для реалізації цього проекту, забезпечуючи високу якість і надійність створених додатків.

2.4 Графічний інтерфейс мовою Swift

					ІАЛЦ.467200.003 ПЗ	Арк.
						20
Зм.	Арк.	№ докум.	Підпис	Дата		

2.3.2 Dart

Dart – це сучасна мова програмування, розроблена компанією Google. Вона була створена у 2011 році з метою забезпечення високої продуктивності і гнучкості при розробці веб-і мобільних додатків. Dart здобула популярність завдяки своїй інтеграції з фреймворком Flutter, який дозволяє створювати кросплатформні додатки для iOS і Android з єдиною кодовою базою. Dart поєднує в собі властивості об'єктно-орієнтованої і функціональної програмування, що робить її потужним інструментом для розробки складних і масштабованих додатків.

Переваги:

- Однією з головних переваг Dart є її здатність забезпечувати кросплатформену розробку. Завдяки фреймворку Flutter, розробники можуть створювати додатки для iOS і Android з єдиною кодовою базою, що значно скорочує час і витрати на розробку. Flutter забезпечує високу продуктивність і плавність інтерфейсу користувача завдяки власному рендерингу, що дозволяє досягти нативного вигляду і відчуття додатків на обох платформах.
- Dart має сучасний і зрозумілий синтаксис, який спрощує процес навчання і розробки. Мова розроблена таким чином, щоб бути легкою у використанні для розробників, знайомих з іншими С-подібними мовами, такими як JavaScript або C#. Це дозволяє швидко освоїти Dart і почати ефективно використовувати її для розробки додатків.
- Іншою важливою перевагою Dart є його висока продуктивність. Завдяки компіляції в нативний код, Dart забезпечує швидке виконання додатків, що особливо важливо для мобільних пристроїв з обмеженими ресурсами. Крім того, Dart підтримує JIT (Just-In-Time) і AOT (Ahead-Of-Time) компіляцію, що дозволяє досягти оптимального балансу між швидкістю розробки і продуктивністю виконання.

					ІАЛЦ.467200.003 ПЗ	Арк.
						21
Зм.	Арк.	№ докум.	Підпис	Дата		

- Активна підтримка Google і велика спільнота розробників забезпечують постійний розвиток мови і фреймворку Flutter. Це гарантує, що Dart залишається актуальною і надійною мовою програмування, яка постійно вдосконалюється і отримує нові функції та можливості.

Недоліки:

- Попри численні переваги, Dart також має деякі недоліки. Один з них полягає у відносній новизні мови. Хоча Dart і Flutter швидко набувають популярності, вони все ще мають менше бібліотек і ресурсів у порівнянні з більш старими і добре відомими мовами програмування, такими як JavaScript. Це може створювати певні труднощі для розробників, які шукають специфічні інструменти або готові рішення.
- Ще одним недоліком є обмежена інтеграція з нативними платформами. Хоча Flutter забезпечує багато вбудованих функцій і віджетів, деякі специфічні платформи API можуть бути менш доступними або вимагати додаткових зусиль для інтеграції. Це може ускладнити розробку додатків, що потребують глибокої інтеграції з нативними функціями пристрою.

Незважаючи на ці недоліки, Dart залишається потужним інструментом для кросплатформної розробки. Його переваги, такі як висока продуктивність, зручний синтаксис і активна підтримка спільноти, роблять його привабливим вибором для розробників, які прагнуть створювати якісні мобільні додатки для різних платформ.

2.3.3 JavaScript

JavaScript – це високорівнева мова програмування, яка спочатку була розроблена для створення інтерактивних веб-сторінок. Вона була створена в

					ІАЛЦ.467200.003 ПЗ	Арк.
						22
Зм.	Арк.	№ докум.	Підпис	Дата		

1995 році компанією Netscape і швидко стала однією з найпопулярніших мов програмування у світі. Завдяки своєму універсальному застосуванню, JavaScript сьогодні використовується не тільки для розробки веб-додатків, але і для створення мобільних додатків за допомогою фреймворків, таких як React Native та Ionic. JavaScript дозволяє розробникам створювати кросплатформні додатки з єдиною кодовою базою, що підтримує як iOS, так і Android.

Переваги:

- Однією з головних переваг JavaScript є його кросплатформеність. За допомогою фреймворків, таких як React Native та Ionic, розробники можуть створювати додатки для різних платформ, використовуючи одну і ту ж кодову базу. Це значно скорочує час розробки і витрати, оскільки один і той самий код можна використовувати для створення додатків для iOS, Android і навіть веб.
- Велика спільнота розробників є ще однією значною перевагою JavaScript. Завдяки своїй популярності, JavaScript має величезну кількість бібліотек, інструментів і ресурсів, що полегшує розробку додатків. Розробники можуть скористатися готовими рішеннями для вирішення поширених задач, що значно прискорює процес розробки і підвищує продуктивність.
- JavaScript також відзначається своєю інтеграцією з веб-технологіями. Це означає, що розробники можуть використовувати один і той самий код для створення як веб-додатків, так і мобільних додатків, що забезпечує узгодженість і повторне використання коду. Це дозволяє швидко розробляти і оновлювати додатки, зменшуючи час на тестування і налагодження.
- Легке навчання є ще однією перевагою JavaScript. Завдяки своїй поширеності і великій кількості навчальних ресурсів, розробники можуть швидко освоїти мову і почати використовувати її для створення додатків. Це робить JavaScript відмінним вибором для новачків, які тільки починають свій шлях у програмуванні.

					ІАЛЦ.467200.003 ПЗ	Арк.
						23
Зм.	Арк.	№ докум.	Підпис	Дата		

Недоліки:

- Попри численні переваги, JavaScript також має деякі недоліки. Один з них полягає у продуктивності. Додатки, створені за допомогою JavaScript, можуть працювати повільніше в порівнянні з нативними додатками, особливо на старіших або менш потужних пристроях. Це пов'язано з тим, що JavaScript виконується в середовищі інтерпретатора, а не компілюється в нативний код.
- Складність налагодження є ще одним недоліком JavaScript. Через асинхронну природу мови, процес відладки може бути складнішим, особливо для новачків. Відстеження помилок і оптимізація коду можуть вимагати більше часу і зусиль у порівнянні з іншими мовами.
- Відсутність деяких нативних функцій також може бути обмеженням JavaScript. Хоча фреймворки, такі як React Native, забезпечують доступ до багатьох функцій пристрою, деякі специфічні платформи API можуть бути менш доступними або вимагати додаткової інтеграції. Це може створювати труднощі для розробників, які потребують глибокого доступу до апаратних можливостей пристрою.

Незважаючи на ці недоліки, JavaScript залишається потужним і популярним інструментом для розробки кросплатформних мобільних додатків. Його переваги, такі як велика спільнота, кросплатформеність і легкість навчання, роблять його відмінним вибором для багатьох проектів.

Для реалізації даної дипломної роботи, однак, більш відповідним вибором є Swift, завдяки його високій продуктивності, безпеці та інтеграції з екосистемою Apple. Вибір Swift для розробки мобільного додатку також має суттєві переваги з точки зору безпеки даних. Використання Swift дозволяє повністю скористатися вбудованими можливостями безпеки, які надають пристрої Apple. Ці можливості включають апаратне шифрування, захищені контейнери для зберігання даних, безпечну аутентифікацію користувачів та

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		24

інші механізми захисту, що забезпечують високий рівень конфіденційності та цілісності даних.

Таблиця 2.1 – Порівняльна таблиця мов програмування.

Параметр	Swift	Dart	JavaScript
Основне застосування	iOS, macOS, watchOS, tvOS	Кросплатформені додатки з Flutter	Кросплатформені додатки з React Native, веб застосунки
Ефективність	Висока, завдяки компіляції в нативний код	Висока, завдяки AOT та JIT компіляції	Нижче в порівнянні з нативними додатками
Безпека	Високий рівень безпеки, вільний доступ до вбудованих функцій захисту Apple	Обмежена кросплатформеністю	Відсутність деяких нативних функцій безпеки
Кросплатформеність	Обмежена до екосистеми Apple	Підтримує всі основні платформи для розробки	Підтримує всі основні платформи для розробки
Обмеження	Обмежена кросплатформеність.	Менша кількість бібліотек порівняно з JavaScript	Можливі проблеми з продуктивністю, складність налагодження
Доступ до вбудованих API	Повний доступ до всіх функцій та API від Apple.	Хороший доступ, але вимагає додаткового налаштування	Можливі обмеження в доступі до специфічних функцій

2.4 Графічний інтерфейс мовою Swift

Одним з основних інструментів для створення графічних інтерфейсів у Swift є SwiftUI - сучасний фреймворк, представлений Apple у 2019 році. SwiftUI дозволяє створювати інтерфейси за допомогою декларативного підходу, що значно спрощує процес розробки та забезпечує більшу гнучкість у створенні інтерфейсів.

2.4.1 SwiftUI

SwiftUI дозволяє описувати інтерфейс за допомогою декларативного синтаксису, що значно спрощує процес розробки та зменшує кількість коду [6]. Основні компоненти інтерфейсу визначаються як структуровані об'єкти, що дозволяє легко зрозуміти та модифікувати структуру додатку. SwiftUI також автоматично забезпечує реактивність інтерфейсу, тобто зміни у стані додатку автоматично відображаються у відповідних компонентах інтерфейсу.

2.4.2 UIKit

UIKit - це традиційний фреймворк для створення інтерфейсів у додатках iOS, який використовується розробниками протягом багатьох років. UIKit пропонує широкий набір інструментів та компонентів для створення складних та інтерактивних інтерфейсів [5]. Хоча SwiftUI набирає популярність, UIKit все ще залишається важливим інструментом для створення інтерфейсів, особливо у випадках, коли потрібно підтримувати старі версії iOS або використовувати специфічні компоненти, які поки що не підтримуються у SwiftUI.

2.5 База даних

CoreData - фреймворк від Apple, який використовується для управління модельними об'єктами та їх збереження у базі даних [8]. Він надає зручний і ефективний спосіб зберігати дані в додатках для iOS, macOS, tvOS та watchOS. CoreData дозволяє працювати з об'єктами високого рівня та абстрагується від деталей низькорівневого управління базами даних.

2.6 Основні функціональні модулі

2.6.1 Отримання вхідних даних

Додаток на мові Swift завжди починається з `UIViewController` - клас, який відповідає за поведінкою екрану. Він керує життєвим циклом додатка, обробляє події, керує відображенням інтерфейсу.

Для даного випадку, найкращим способом отримання та відображення вхідних даних від користувача буде використання вбудованого в `UIKit` класу - `UITextView`. Цей клас надає багаторядковий редактор, який дозволяє отримувати доволі низькорівневу інформацію про текст в ньому, таку як координати тексту, а також кастомізувати стилі параграфів, абзаців тощо.

2.6.2 Інтерпретатор

Для підтримання принципу єдиної відповідальності, створимо окремий модуль під назвою `Interpreter`. Цей модуль буде займатися всім, що пов'язано з обробкою тексту: парсингом, обчисленням, обміном інформацією з редактором тексту, тощо.

Для зберігання даних, створимо модель `Note`, яка буде відповідати за збереження даних в БД `CoreData`.

Для відображення рядка математичного виразу в кодї, створимо структуру з назвою `StatementLine`. В ній будемо тримати всю необхідну інформацію для того, щоб ефективно проводити обчислення. Згідно функціональних вимог, додаток підтримує багаторядкові обчислення в одному текстовому редактору, тому для того, щоб потім було легше слідкувати за позицією виразу, `StatementLine` має бути двобічно зв'язаним списком. Зберігаємо його в середині інтерпретатора.

Щоб дізнатися, де показувати результат обчислення, потрібно знати координати рядка математичного виразу в тексті. Для цього ми можемо використовувати `UITextView`. Щоб абстрагувати редактор від інтерпретатора, створимо інтерфейс `TextLocationProvider`.

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		27

Далі, спираючись на те, що додаток має підтримувати асинхронні дії, доцільним буде створити окремий модуль всередині інтерпретатора (Evaluator), який буде займатися лише обчисленнями, виконуючи їх на окремому потоці програми. Це викликано тим, що інтервали часу між редагуваннями тексту можуть бути меншими, ніж час на обчислення якоїсь асинхронної дії, тому потрібно знати коли нам не потрібно продовжувати виконувати дану операцію.

2.6.3 Безпечне зберігання обчислень

Для безпечного зберігання обчислень в базі даних будемо використовувати вбудований в CoreData метод шифрування інформації `NSPersistentStoreFileProtectionKey`.

Основний принцип захисту даних в CoreData полягає у використанні шифрування файлів бази даних за допомогою симетричного блочного шифру AES (Advanced Encryption Standard) [9], . Шифрування забезпечує перетворення даних у такий вигляд, що вони стають недоступними для неавторизованих користувачів. Файли зашифровуються з використанням ключа, який зберігається в апаратному модулі безпеки пристрою (Secure Enclave). Secure Enclave є автономним обчислювальним середовищем, яке забезпечує високий рівень безпеки для обробки чутливих даних шляхом використання апаратні засоби для генерації, зберігання та управління криптографічними ключами, що значно підвищує безпеку.

Використання `NSPersistentStoreFileProtectionKey` гарантує, що ваші дані залишаються захищеними навіть у випадку, якщо пристрій потрапить до рук зломисників. Без доступу до ключа шифрування, який зберігається в Secure Enclave, розшифрувати дані практично неможливо.

2.6.4 Захищений доступ до API

Виходячи з того, що додаток не передбачає авторизації користувачів, потрібно обмежити доступ до API тільки для користувачів додатку. Для цього скористаємося вбудованою бібліотекою DeviceCheck від Apple.

DeviceCheck дозволяє ідентифікувати пристрої користувачів і встановлювати унікальні маркери, що забезпечує додатковий рівень захисту [10]. Цей інструмент допомагає запобігти несанкціонованому доступу до API, оскільки лише пристрої з встановленим додатком можуть успішно проходити перевірку. DeviceCheck забезпечує:

1. Генерацію унікальних маркерів для кожного пристрою, що дозволяє ідентифікувати його під час кожного запиту до API.
2. Зберігання маркерів на сервері, що дозволяє перевіряти автентичність запитів.
3. Обмеження доступу до API для неавторизованих пристроїв, запобігаючи потенційним загрозам безпеці.

DeviceCheck генерує токен пристрою користувача, який потрібно підтвердити через запит до серверів Apple. Цей процес, хоча і забезпечує високий рівень безпеки, є досить ресурсномістким і займає значну кількість часу для кожного запиту до нашого сервера. Через це виникає необхідність у впровадженні більш ефективного алгоритму аутентифікації користувачів, який зменшить навантаження на сервер і прискорить процес аутентифікації.

Для досягнення цієї мети ми можемо використати JWT, що забезпечить швидку, надійну і безпечну аутентифікацію. JWT є сучасним стандартом для передачі інформації між двома сторонами у компактному та закодованому вигляді. Основна перевага JWT полягає в тому, що токени містять всю необхідну інформацію про користувача в зашифрованій формі, що дозволяє уникнути повторних запитів до серверів для кожної аутентифікації.

Цей метод дозволяє нашому додатку значно знизити навантаження на сервер, оскільки верифікація токенів може здійснюватися локально без

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		29

необхідності постійного звернення до зовнішніх серверів. Це забезпечує більш швидку і плавну роботу додатку, покращує користувацький досвід та підвищує загальну продуктивність системи.

Використання JWT також спрощує процес управління аутентифікацією та забезпечує високий рівень безпеки завдяки використанню надійних алгоритмів шифрування. Це робить JWT ідеальним вибором для наших потреб у ефективній та безпечній аутентифікації користувачів, значно підвищуючи зручність і швидкість роботи додатку.

					ІАЛЦ.467200.003 ПЗ	Арк.
						30
Зм.	Арк.	№ докум.	Підпис	Дата		

Висновок до розділу 2

Розробка захищеного калькулятора з асинхронними діями є важливим і складним завданням, яке вимагає використання сучасних технологій та підходів. Використання мови програмування Swift дозволяє досягти високої продуктивності, безпеки та зручності у розробці, що робить її відмінним вибором для створення такого додатку. Swift пропонує потужні інструменти для роботи з асинхронними операціями, що дозволяє виконувати складні обчислення у фоновому режимі, забезпечуючи плавний користувацький досвід.

Графічний інтерфейс відіграє ключову роль у додатку, забезпечуючи зручність та ефективність взаємодії користувача з програмою. Використання сучасного фреймворку SwiftUI надає можливість створювати інтерфейси за допомогою декларативного підходу, що значно спрощує процес розробки та зменшує кількість необхідного коду. SwiftUI забезпечує реактивність інтерфейсу, автоматично відображаючи зміни у стані додатку в його компонентах. У випадках, коли необхідно підтримувати старі версії iOS або використовувати специфічні компоненти, які ще не підтримуються у SwiftUI, UIKit залишається важливим інструментом для створення інтерфейсів.

Парсинг математичних виразів є однією з основних задач додатку. Вибір підходу до парсингу залежить від складності виразів та вимог до продуктивності. Прості парсери підходять для базових арифметичних операцій, тоді як інфіксні парсери, рекурсивні парсери, парсери зворотної польської нотації та AST парсери забезпечують обробку складніших виразів.

РОЗДІЛ 3. ПРОЦЕС РОЗРОБКИ СИСТЕМИ

У ході попереднього аналізу алгоритмів та технологій була визначена необхідна архітектура додатку для задоволення функціональних вимог. У цьому розділі детально розглянемо процес імплементації, включаючи інтеграцію обраних технологій, оптимізацію обчислювальних процесів, а також заходи щодо забезпечення безпеки даних користувачів.

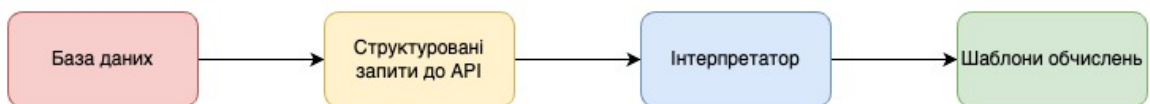


Рисунок 3.1 – Етапи розробки.

3.1 Підключення бази даних

3.1.1 CoreDataStack

Створимо сінглтон CoreDataStack для звернень в базу даних CoreData. Фрагмент коду з реалізацією зображен на рисунку 3.2.

```
final class CoreDataStack {  
    nonisolated(unsafe) static let shared = CoreDataStack()  
  
    var viewContext: NSManagedObjectContext {  
        container.viewContext  
    }  
  
    var backgroundContext: NSManagedObjectContext {  
        let context = container.newBackgroundContext()  
        context.automaticallyMergesChangesFromParent = true  
        return context  
    }  
  
    let spotlightDelegate: SpotlightDelegate  
  
    let container: NSPersistentContainer  
  
    nonisolated init() {  
        container = NSPersistentCloudKitContainer(name: "Multi")  
  
        guard let description = container.persistentStoreDescriptions.first else { fatalError() }  
  
        description.type = NSSQLiteStoreType  
        description.shouldInferMappingModelAutomatically = true  
        description.setOption(true as NSNumber, forKey: NSPersistentStoreRemoteChangeNotificationPostOptionKey)  
        description.setOption(true as NSNumber, forKey: NSPersistentHistoryTrackingKey)  
        description.setOption(FileProtectionType.complete as NSObject,  
                               forKey: NSPersistentStoreFileProtectionKey) // encryption  
  
        container.loadPersistentStores(completionHandler: { description, error in  
            if let error = error as NSError? {  
                fatalError("Unresolved error \(error), \(error.userInfo)")  
            }  
        })  
        container.viewContext.automaticallyMergesChangesFromParent = true  
        container.viewContext.mergePolicy = NSMergePolicy.mergeByPropertyObjectTrump  
  
        let coordinator = container.persistentStoreCoordinator  
        spotlightDelegate = .init(forStoreWith: description, coordinator: coordinator)  
        spotlightDelegate.startSpotlightIndexing()  
    }  
}
```

Рисунок 3.2 – Імплементація класу CoreDataStack

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		32

Цей код створює клас CoreDataStack, який служить для управління Core Data в додатку. Основною особливістю цього класу є те, що він реалізує Singleton патерн, що забезпечує наявність лише одного екземпляра CoreDataStack протягом усього життєвого циклу додатку. Це досягається за допомогою статичної властивості shared.

У класі визначено два контексти управління об'єктами: viewContext і backgroundContext. viewContext використовується для головних операцій з даними у основному потоці, тоді як backgroundContext служить для обробки даних у фоновому режимі, забезпечуючи асинхронну роботу та не блокуючи інтерфейс користувача.

Контексти viewContext і backgroundContext налаштовані на автоматичне об'єднання змін від батьківського контексту, що забезпечує узгодженість даних між основним і фоновим потоками.

Використовується опція FileProtectionType.complete для захисту файлів за допомогою шифрування.

3.1.2 Опис властивостей моделі даних

Для зберігання обчислень створюємо таблицю в базі даних зі структурою, зображеною на рисунку 3.3.

Attributes

Attribute

^

Type

D

createdOn

Date

⌵

S

text

String

⌵

S

title

String

⌵

D

updatedOn

Date

⌵

+

—

Рисунок 3.3 – Модель обчислення в БД

Властивості:

- `text` – цей атрибут є надзвичайно важливим, оскільки він відповідає за збереження самих обчислень у вигляді рядка. Це дозволяє користувачам зберігати всі деталі своїх математичних операцій у зручному текстовому форматі, що забезпечує простоту збереження та подальшого використання.
- `title` – цей атрибут виконує функцію збереження назви обчислення, що є дуже корисним для користувачів, які хочуть швидко знаходити свої збережені обчислення.
- `createdOn` – цей атрибут зберігає дату і час, коли обчислення було збережене.
- `updatedOn` – цей атрибут зберігає дату і час останньої модифікації обчислення.

3.2 Структуровані запити до API

Для захищених запитів до серверу створимо обгортку над вбудованою в Swift структурою `URLSession`. Фрагмент імплементації зображена на рисунку 3.4.

					ІАЛЦ.467200.003 ПЗ	Арк.
						34
Зм.	Арк.	№ докум.	Підпис	Дата		

```

public struct RestService {

    public var request: URLRequest

    var startTime: CFAbsoluteTime?

    init(descriptor: RestServiceDescriptor) {
        self.request = .init(url: descriptor.url,
                              cachePolicy: descriptor.cachePolicy,
                              timeoutInterval: descriptor.timeoutInterval)
        request.httpMethod = descriptor.method.rawValue
    }
}

public extension RestService {

    consuming func fetch(completion: @escaping (Response) -> Void) {
        startTime = CFAbsoluteTimeGetCurrent()
        let dataTask = URLSession.shared.dataTask(with: request) { data, response, error in
            if let error {
                logError("Request error: \(error)")
                return completion(.init(data: data, error: error))
            }

            guard let response = response as? HTTPURLResponse else {
                logError("Bad request")
                return completion(.init(data: data, error: NetworkError.badResponse(response)))
            }

            guard 200...299 == response.statusCode else {
                if let data, let dataString = String(data: data, encoding: .utf8) {
                    logError("Status code: \(response.statusCode)\n\((dataString)")
                } else {
                    logError("Status code: \(response.statusCode)")
                }
                return completion(.init(data: data, error: NetworkError.errorCode(response.statusCode)))
            }

            logInfo("")
            completion(.init(data: data, error: nil))
        }
        dataTask.resume()
    }

    func fetch() async -> Response {
        await withCheckedContinuation { continuation in
            fetch { response in
                continuation.resume(returning: response)
            }
        }
    }
}

```

Рисунок 3.4 – Імплементація обгортки над URLRequest.

Структура RestService є основою для створення HTTP-запиту. Вона приймає параметр descriptor типу RestServiceDescriptor (рисунок 3.5), який містить всю необхідну конфігурацію для запиту, таку як заголовки, політика кешування, тайм-аут і метод запиту. Завдяки цьому підходу, ми можемо легко налаштовувати запити відповідно до потреб нашого додатку.

```

public protocol RestServiceDescriptor {
    var url: URL { get }
    var method: HTTPMethod { get } /// default: GET
    var cachePolicy: URLRequest.CachePolicy { get } /// default: `useProtocolCachePolicy`
    var timeoutInterval: TimeInterval { get } /// default: `60`
}

```

Рисунок 3.5 – Структура RestServiceDescriptor

Тепер створюємо модифікатор сервісу, який буде додавати токен пристрою девайса, згенерований за допомогою бібліотеки DeviceCheck, у HTTP заголовок при кожному запиті на сервер. Імплементація зображена на рисунку 3.6.

```
import DeviceCheck

struct DeviceTokenModifier: RestServiceModifier {

    func modify(service: inout RestService) async {
        let token = try? await DCDevice.current.generateToken()
        guard let token else { return }
        service.request.addValue(token.base64EncodedString(), forHTTPHeaderField: "Device-Check-Token")
    }
}
```

Рисунок 3.6 – Код модифікатору RestService

3.3. Розробка інтерпретатора

Інтерпретатор є основною структурою в цьому додатку, яка відіграє ключову роль у його функціонуванні. Ця структура відповідає за управління всіма процесами, забезпечуючи їхню координацію та взаємодію. Інтерпретатор виконує центральну функцію в архітектурі додатку, забезпечуючи безперебійну роботу та ефективну обробку даних. Зробимо оголошення класу Interpreter (рисунок 3.7).

```

public class Interpreter {

    private let queue = DispatchQueue(label: "com.beaxhem.Parser.Interpreter", attributes: .concurrent)

    private let evaluator: Evaluator

    private let sync = InterpreterSynchronization()

    private var inputLayout = InputLayout()

    private let programContext: ProgramContext

    private let dependencyGraph: DefaultDependencyGraph

    private weak var delegate: InterpreterDelegate?

    private weak var offsetHead: StatementLine?

    public let benchmark = Benchmark()

    let statementLine: StatementLine

    public init(
        evaluator: Evaluator? = nil,
        currencyRatesChart: CurrencyRatesChart,
        weatherRepository: WeatherInfoRepository,
        timeRepository: TimeRepository,
        statementLine: StatementLine = .init()
    ) {
        self.programContext = .init(currencyChart: currencyRatesChart,
                                     weatherRepository: weatherRepository,
                                     timeRepository: timeRepository)
        self.dependencyGraph = .init(programContext: programContext)
        self.evaluator = evaluator ?? DefaultEvaluator(programContext: programContext)
        self.statementLine = statementLine
        self.evaluator.delegate = self
        self.statementLine.dependencyGraph = dependencyGraph
    }
}

```

Рисунок 3.7 – Оголошення класу Interpreter

Інтерпретатор буде ключовим класом у нашому додатку, який відіграє роль фасаду, об'єднуючи всі окремі системи інтерпретатора в єдину інтегровану структуру. Це дозволить нам приховати складність і деталі реалізації окремих компонентів, забезпечуючи простий і зручний інтерфейс для взаємодії з інтерпретатором. Одна з найважливіших систем, яку інкапсулюватиме клас інтерпретатора, – це текстове представлення обчислень.

3.3.1 StatementLine

У додатку підтримуються багаторядкові обчислення, що накладає певні вимоги на обробку тексту. Зокрема, нам потрібно оптимізувати процес

					ІАЛЦ.467200.003 ПЗ	Арк.
						37
Зм.	Арк.	№ докум.	Підпис	Дата		

обробки, щоб розділяти текст на окремі рядки для поліпшення продуктивності та зручності користувача. Це означає, що інтерпретатор повинен бути здатним ефективно працювати з великими обсягами текстових даних, розпізнавати структуру обчислень і коректно розбивати їх на рядки.

Враховуючи те, що обчислення в нашому додатку надходять поступово, ми повинні мати можливість збирати повний текст обчислення з окремих змін. Це вимагає від нас реалізації механізму, який дозволяє постійно оновлювати текст обчислення у міру надходження нових даних і водночас розділяти його на окремі рядки.

Для вирішення цієї задачі створимо спеціальний клас `StatementLine` (рисунок 3.8), який буде реалізований у вигляді двозв'язного списку (`DoublyLinkedList`). Цей клас буде відповідати за обробку текстових даних розділення їх на рядки. Основна функція `StatementLine` полягатиме в тому, щоб відстежувати моменти, коли потрібно розділяти текст, коли він був модифікован, і забезпечувати коректне розділення на рядки.

```

public final class StatementLine: @unchecked Sendable {

    public var positionHasChanged: Bool { ... }

    public var valueHasChanged: Bool { ... }

    public let textContainer: TextContainer

    public private(set) var bounds: CGRect?

    weak var prev: StatementLine?
    public internal(set) var next: StatementLine? {
        didSet {
            next?.prev = self
        }
    }

    public var evaluationContext: EvaluationContext? {
        didSet {
            oldValue?.cancelBulkTasks()
        }
    }

    public weak var dependencyGraph: DependencyGraph?

    public var exportedVariable: VariableIdentifier?

    public var needsLine = false

    public var ast: ParsedStatement?

    public init(value: String = "", location: Int = 0) {
        self.textContainer = .init(value: value, location: location)
        self.textContainer.statementLine = self
    }
}

```

Рисунок 3.8 – Структура StatementLine.

Структура LinkedList є ідеальним вибором для реалізації StatementLine, оскільки вона забезпечує ефективну вставку та видалення елементів, що особливо важливо при обробці поступово надходячих даних. Завдяки цьому підходу, ми зможемо забезпечити високу продуктивність і надійність роботи нашого додатку.

Клас StatementLine буде відігравати важливу роль не лише в обробці та розділенні тексту на рядки, але й у визначенні необхідних координат для

відображення результатів обчислень кожного рядка. Ця функція є критично важливою для забезпечення коректного і зручного візуального представлення обчислень у додатку.

3.3.2 InputLayout

Для того, щоб визначити ті рядки, в яких відбулися зміни при останній модифікації тексту, ми впровадимо клас InputLayout (рисунок 3.9). Цей клас матиме важливе значення в архітектурі нашого додатку, оскільки він буде відповідальний за виявлення змін у тексті та отримання координат для відображення результатів обчислень.

Клас InputLayout відіграватиме ключову роль у забезпеченні коректного відображення текстових змін. Кожного разу, коли в тексті відбуваються зміни, цей клас буде аналізувати новий стан тексту, порівнюючи його з попереднім, щоб виявити, в яких саме рядках сталися модифікації. Це дозволить нам точно визначати області тексту, які потребують оновлення на екрані, що значно підвищить ефективність роботи додатку. Крім того, InputLayout буде відповідати за обчислення координат для відображення результатів текстових обчислень.

```

struct InputLayout {
    var textLocationProvider: TextLocationProvider?

    func findInvalidatedLines(head: StatementLine) -> InputInvalidationContext {
        let context = InputInvalidationContext()
        head.invalidateIfNeeded(context: context)
        return context
    }

    @MainActor func invalidatePosition(head: StatementLine, invalidateLayout: Bool) {
        guard let textLocationProvider else { return }
        var didNotAlreadyChange = false

        for line in head.iter() {
            let didChange = line.askForBounds(locationProvider: textLocationProvider)
            if !invalidateLayout && !didChange {
                if didNotAlreadyChange {
                    return
                }
                didNotAlreadyChange = true
            }
        }
    }
}

```

Рисунок 3.9 – Імплементція InputLayout.

3.3.3 Tokenizer

Створимо клас Tokenizer (рисунок 3.10), який буде виконувати важливу функцію розбивання текстового рядка на окремі синтаксичні елементи, або токени. Ці токени можуть бути різного типу, наприклад, числа, слова, символи тощо. Процес токенізації є ключовим етапом обробки тексту, оскільки він дозволяє структурувати і аналізувати дані більш ефективно.

```

final class Tokenizer {

    var iterator: InputIterator

    init(input: String) {
        self.iterator = .init(input: input)
    }

    func next() -> TokenBit? {
        while let char = iterator.next() {
            switch char {
                case " ", "\n", "\t", "\r":
                    continue
                default:
                    let strategy = strategy(for: char)
                    var builder = TokenBuilder(strategy: strategy, iterator: iterator)
                    return builder.build(startingWith: char)
            }
        }
        return nil
    }

    func strategy(for char: Unicode.Scalar) -> TokenBuilderStrategy {
        switch char {
            case "=":
                return FunctionBodyBuilderStrategy()
            case "-":
                return NegativeNumberBuilderStrategy()
            case "0":
                return NumberBuilderStrategy(startsWithZero: true)
            case "1"... "9":
                return NumberBuilderStrategy(startsWithZero: false)
            default:
                return StringBuilderStrategy()
        }
    }
}

```

Рисунок 3.10 – Фрагмент імплементації класу Tokenizer

Клас Tokenizer буде відповідати за розбиття тексту на токени шляхом аналізу кожного символу в рядку і визначення його типу. Під час цього процесу ми будемо розділяти текст на окремі символи і, залежно від того, який це символ, застосовувати певну стратегію токенизації.

Процес токенизації:

1. Розділення на символи: Спочатку ми розділимо текстовий рядок на окремі символи. Це дозволить нам аналізувати кожен символ окремо і визначати його тип.

					ІАЛЦ.467200.003 ПЗ	Арк.
						42
Зм.	Арк.	№ докум.	Підпис	Дата		

2. Вибір стратегії токенизації: Для кожного символу ми визначимо відповідну стратегію токенизації. Наприклад, якщо символ є числом, ми будемо додавати наступні символи до цього токена, поки не зустрінемо пробіл або літеру. Аналогічно, якщо символ є літерою, ми будемо продовжувати додавати символи до цього токена, поки не зустрінемо пробіл або число. Цей підхід дозволяє коректно визначати межі кожного токена і забезпечувати точність токенизації.

3. Обробка спеціальних символів: Крім чисел і літер, ми також будемо обробляти спеціальні символи, такі як знаки пунктуації, математичні оператори тощо. Кожен з цих символів буде оброблятися окремо, щоб забезпечити правильне розбиття тексту на токени.

3.3.4 Lexer

Після завершення процесу токенизації, необхідно визначити лексичне значення кожного токена. Для цього ми створимо клас `Lexer`, який буде відповідати за інтерпретацію та класифікацію токенів.

Клас `Lexer` відіграватиме ключову роль у визначенні лексичних значень токенів після токенизації. Він буде використовувати словник для зберігання всіх підтримуваних додатком токенів і їхніх відповідних значень. Це дозволить швидко і ефективно порівнювати кожен токен з наявними значеннями у словнику.

Створимо перелік значень, які токени можуть мати (рисунки 3.11, 3.12, 3.13).

```
enum TokenKind: Equatable {  
    case literal(Literal)  
    case measurementUnit(MeasurementUnitType)  
    case `operator`(OperatorKind)  
    case newline  
    case variableAssignment  
}
```

Рисунок 3.11 – Перелік значень токенів.

```
public enum OperatorKind: Equatable, Hashable, Sendable {
    case prefix(PrefixOperator)
    case infix(InfixOperator)
    case postfix(PostfixOperator)

    case helper(HelperOperator)
    case special(SpecialOperator)
}
```

Рисунок 3.12 – Види операторів.

```
public enum InfixOperator: Equatable, Hashable, Sendable {
    case plus, minus, multiply, divide, exponent, modulo
    case convert
}
```

Рисунок 3.13 – Інфіксі оператори.

Процес лексичного аналізу:

1. Порівняння зі словником: Спочатку Lexer буде порівнювати значення кожного токена з елементами в словнику. Якщо значення токена знайдено у словнику, Lexer присвоює цьому токenu відповідне лексичне значення.
2. Конвертація в число: Якщо значення токена не знайдено у словнику, Lexer намагається конвертувати токен в число. Це важливо для обробки числових даних, таких як константи або числові змінні. Якщо токен успішно конвертується в число, його лексичне значення визначається як числове.
3. Визначення як рядок: Якщо токен не вдалося знайти у словнику і він не може бути конвертований в число, Lexer визначає його як рядкове значення. Такі токени можуть використовуватися для змінних, імена яких складаються з літер або комбінації літер і цифр.

```

func token(from bit: inout TokenBit) -> Token {
    if let kind = TOKENS[bit.value] {
        return .init(position: bit.position, kind: kind)
    } else if let kind = TOKENS_LOWER[bit.value.lowercased()] {
        return .init(position: bit.position, kind: kind)
    } else if let number = number(&bit.value) {
        return .init(position: bit.position, kind: .literal(.number(number)))
    }
    return .init(position: bit.position, kind: .literal(.string(bit.value)))
}

```

Рисунок 3.14 – Фрагмент імплементації класу Lexer

3.3.5 Parser

Після отримання першого токена рядка ми можемо розпочати процес побудови абстрактного синтаксичного дерева (AST). Для цього ми створимо спеціалізований клас Parser. Процес побудови дерева складається з трьох основних частин: визначення типу виразу, побудови зворотної польської нотації (RPN), і створення самого дерева.

Коли Parser отримує перший токен рядка, він починає з визначення типу виразу. Це дозволяє зрозуміти, яку операцію хоче виконати користувач. Відповідно до функціональних вимог, додаток повинен підтримувати оголошення змінних та обчислення математичних виразів. Тому, якщо перший токен рядка є текстом, ми припускаємо, що користувач хоче оголосити змінну. Якщо наступний токен є знаком рівності (=), ми підтверджуємо, що це оголошення змінної. Усі наступні токени розглядаються як частина математичного виразу.

```

func parseStatement(token: Token, line: StatementLine) -> ParsedStatement {
    let position = TextPosition(location: line.textContainer.text.location + token.position.location,
                                length: token.position.length)

    switch token.kind {
    case .literal(.string(let literal)):
        return possibleVariableDeclaration(literal: literal,
                                            position: position,
                                            line: line)

    default:
        guard let iterator else { fatalError() }
        let parser = ExpressionParser(iterator: iterator, programContext: programContext)
        return parser.parse(line: line)
    }
}

```

Рисунок 3.15 – Фрагмент визначення типу виразу.

Після визначення типу виразу ми переходимо до важливого етапу – побудови зворотної польської нотації (RPN). Цей етап є критичним, оскільки він забезпечує коректне і ефективне перетворення виразу, що дозволяє спростити подальшу побудову абстрактного синтаксичного дерева (AST). Для цього ми створимо спеціальний клас `ExpressionParser`.

Клас `ExpressionParser` відіграватиме ключову роль у процесі перетворення інфіксного виразу у зворотну польську нотацію. Цей клас аналізує значення токенів, їхню позицію відносно сусідніх токенів, а також їхні пріоритети. Він використовує спеціалізований алгоритм, який дозволяє ефективно обробляти вирази і формувати зворотну польську нотацію.

Процес побудови зворотної польської нотації починається з аналізу кожного токена у виразі. Це включає визначення типу токена (число, оператор, дужка тощо) і його пріоритету. Для побудови зворотної польської нотації `ExpressionParser` використовує структуру даних, відому як стек. Стек дозволяє зберігати оператори і дужки під час аналізу виразу, забезпечуючи правильний порядок операцій.

Коли `ExpressionParser` зустрічає числовий токен, він одразу додає його до вихідної послідовності RPN. Це дозволяє забезпечити, що числа залишаються у правильному порядку у кінцевому виразі. При зустрічі оператора `ExpressionParser` перевіряє його пріоритет порівняно з операторами, що вже знаходяться у стеку. Якщо пріоритет оператора нижчий або рівний пріоритету оператора на вершині стеку, оператори знімаються зі стеку і додаються до вихідної послідовності, доки не зустрінеться оператор з нижчим пріоритетом. Потім новий оператор додається до стеку.

Обробка дужок має свої особливості. Відкриваюча дужка додається до стеку без перевірки пріоритету. Закриваюча дужка викликає процес знімання операторів зі стеку і додавання їх до вихідної послідовності, доки не зустрінеться відкриваюча дужка, яка видаляється зі стеку. Після обробки всіх токенів у виразі, усі залишкові оператори у стеку знімаються і додаються до

					ІАЛЦ.467200.003 ПЗ	Арк.
						46
Зм.	Арк.	№ докум.	Підпис	Дата		

вихідної послідовності RPN. Це забезпечує, що всі операції враховані і знаходяться у правильному порядку.

Використання зворотної польської нотації має кілька важливих переваг. По-перше, RPN спрощує обчислення виразів, оскільки усуває потребу у дужках для визначення порядку операцій. Це означає, що вирази можуть оброблятися зліва направо, що значно спрощує побудову AST.

Крім того, RPN дозволяє уникнути проблем, пов'язаних з неоднозначністю операцій, забезпечуючи єдиний і чіткий порядок обчислень. Це робить алгоритм більш ефективним і надійним, що є критично важливим для коректної роботи додатку.

```
func prepare(line: StatementLine) -> PrepareResult {
    var prevToken: Token?
    while let token = iterator.currentValue, token.kind != .newline {
        switch token.kind {
        case .operator(.special(.input)):
            preparation.addOperator(token: token, style: .function)
        case .measurementUnit:
            if let prev = prevToken {
                if case .operator(let operation) = prev.kind,
                    operation.isOperational {
                } else {
                    preparation.addOutput(token: .init(position: .init(), kind: .operator(.helper(.lazy))), style: .default)
                }
            }
            preparation.addOutput(token: token, style: .measurementUnit)
        case .literal(.number):
            preparation.addOutput(token: token, style: .default)
        case .sectionSeparator:
            preparation.addSectionSeparator(token: token)
        case .literal(.string(let ident)):
            preparation.stringStrategy(for: ident, token: token, line: line)
        case .operator(.prefix(.function(.array))), .operator(.prefix(.function)):
            preparation.addOperator(token: token, style: .function)
        case .operator(.helper(.parentheses(.normal(.left)))):
            preparation.addOperator(token: token, style: .helper)
        case .operator(.helper(.parentheses(.normal(.right)))):
            preparation.rightNormalParenthesesStrategy(token: token)
        case .operator(.helper(.parentheses(.square(.left)))):
            preparation.addOperator(token: token, style: .helper)
        case .operator(.helper(.parentheses(.square(.right)))):
            preparation.rightSquareParenthesesStrategy(token: token)
        case .literal(.string(let id)) = next.kind {
            iterator.seekNext()
            preparation.addTempVariable(id)
            preparation.addSyntaxHighlighting(style: .variable, for: next)
        } else {
            preparation.addTempVariable("i")
        }
        case .operator(.helper(.comma)):
            preparation.defaultOperatorStrategy(operatorKind: .helper(.comma), token: token)
            preparation.addSyntaxHighlighting(style: .helper, for: token)
        case .operator(let operatorKind):
            preparation.defaultOperatorStrategy(operatorKind: operatorKind, token: token)
        case .newline, .variableAssignment:
            break
        }
        prevToken = token
        iterator.seekNext()
    }
}
```

Рисунок 3.16 – Фрагмент формування RPN.

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		47

3.3.6 SyntacticAnalyst

Завершальний етап у процесі побудови абстрактного синтаксичного дерева (AST) полягає у використанні зворотної польської нотації (RPN). Для цього ми створимо клас SyntacticAnalyst, який буде відповідальним за побудову AST, забезпечуючи правильну послідовність математичних операцій. Вузлами дерева можуть бути числові значення, текстові значення або математичні операції.

Процес побудови абстрактного синтаксичного дерева починається з ініціалізації. SyntacticAnalyst приймає на вхід RPN, згенеровану ExpressionParser. Це забезпечує необхідну основу для подальшого аналізу і побудови дерева.

Після цього SyntacticAnalyst приступає до аналізу токенів. Він послідовно розглядає кожен токен у RPN, визначаючи його тип. Токен може бути числовим значенням, текстовим значенням або математичною операцією. Цей аналіз дозволяє класифікувати токени і зрозуміти їх роль у виразі.

Наступним кроком є створення вузлів дерева. Для кожного токена створюється відповідний вузол. Якщо токен є числом або текстом, він стає листом дерева. Листи дерева представляють кінцеві значення, які не потребують подальшої декомпозиції. Якщо ж токен є математичною операцією, він стає вузлом, що з'єднує інші вузли. Такі вузли виконують роль операцій, які об'єднують числові або текстові значення.

Після створення вузлів SyntacticAnalyst приступає до побудови самого дерева. Вузли з'єднуються між собою відповідно до порядку операцій і пріоритетів, визначених у RPN. Це забезпечує правильну ієрархію операцій у виразі, дозволяючи виконувати обчислення у правильній послідовності.

```

struct SyntacticAnalyst {
  func analyze(prepareResult: inout ExpressionParser.PrepareResult, line: StatementLine) -> (any Evaluatable)? {
    case .measurementUnit(let unit):
      let unit = MeasurementUnit(numerator: unit)
      guard let prev = prevToken(),
            prev.kind == .operator(.helper(.lazy)) else {
        tokensStack.append(MeasurementUnitExpression(measurementUnit: unit))
        continue
      }

      guard !tokensStack.isEmpty else {
        tokensStack.append(MeasurementUnitExpression(measurementUnit: unit))
        continue
      }

      let lazy = LazyValue(tokensStack.removeLast(), measurementUnit: unit) // {prev_value || 1}$

      guard let prev = tokensStack.last as? LazyValue,
            prev.measurementUnit?.isAdjacent(to: unit) ?? false else { // check if you can collapse two lazy values
        tokensStack.append(lazy)
        continue
      }

      if let next = nextToken(),
         case .operator = next.kind {
        // There is an operation between two collapsable values
        // Example: 100 km + 200 meters
        tokensStack.append(lazy)
      } else {
        // Two collapsable values are waiting to be collapsed
        // Example: 6 feet 1 inch
        tokensStack.append(MathExpression(lhs: tokensStack.removeLast(), operation: .plus, rhs: lazy))
      }

      case .operator(.prefix(.at)): // @ {something}
        guard let value = tokensStack.popLast() else { continue }
        tokensStack.append(LazyValue(value, measurementUnit: nil))
      case .operator(.postfix(let function)): // {something} !
        guard let parameter = tokensStack.popLast() else { continue }
        tokensStack.append(PostfixFunctionTerm(function: function, parameter: parameter))
      case .operator(.infix(let operation)): // {lhs} + {rhs}
        guard let operand2 = tokensStack.popLast(),
              let operand1 = tokensStack.popLast() else {
          continue
        }
  }
}

```

Рисунок 3.17 – Фрагмент імплементації будування AST.

3.3.7 DependencyGraph

Оскільки наш додаток підтримує змінні і дозволяє користувачам редагувати їх значення, важливо оптимізувати процес обчислення, щоб розуміти, коли значення змінної було змінено, і обчислювати лише ті вирази, які використовують цю змінну. Для цього ми будемо будувати граф всіх виразів, що експортують змінні, а також виразів, які використовують ці змінні. Ми оголосимо клас `DependencyGraph`, який буде виконувати цю функцію.

Клас `DependencyGraph` відіграватиме ключову роль у забезпеченні ефективності обчислень в додатку. Цей граф буде відслідковувати залежності між змінними і виразами, які їх використовують. Завдяки цьому ми зможемо визначати, які вирази потребують переобчислення у випадку зміни значення змінної. При редагуванні тексту виразів, `DependencyGraph` перевірятиме, чи

було змінено значення якоїсь змінної. Це дозволяє виявити, які саме вирази залежать від зміненої змінної і потребують переобчислення. Процес створення і використання графу залежностей включає декілька важливих кроків.

Кожен раз, коли користувач редагує текст виразів, DependencyGraph оновлює свій стан. Він перевіряє, чи було змінено значення якоїсь змінної, і якщо так, то визначає, які вирази залежать від цієї змінної. Цей підхід дозволяє значно скоротити обсяг обчислень, оскільки ми переобчислюємо лише ті вирази, які дійсно залежать від оновленої змінної. При редагуванні тексту виразів, DependencyGraph здійснює кілька основних операцій. Спочатку він аналізує кожен вираз, щоб визначити, чи містить він змінні, чиї значення були оновлені. Якщо така змінна виявлена, граф відзначає цей вираз як такий, що потребує переобчислення. Далі, DependencyGraph перевіряє всі вирази, що залежать від оновленої змінної, і додає їх до списку для переобчислення. Це дозволяє забезпечити, що всі залежні вирази будуть коректно оновлені після зміни значення змінної.

```
func analyze(statement: ParsedStatement, visited: inout Set<StatementLine>) -> DependencyAnalysisResult {
    guard let statementLine = statement.statementLine else {
        Logger.dependencyGraph.error("Statement has no connection to its line")
        return .init(reachedTheBottom: false)}

    register(vertex: statementLine, to: statement.usedVariables)

    let dependants = dependantsOf(line: statementLine, visited: &visited)

    if let newExportedVariable = statement.exportedVariable { // is variable source
        let old = statementLine.exportedVariable
        statementLine.exportedVariable = newExportedVariable

        if old != newExportedVariable {
            return .init(reachedTheBottom: true)
        }
    } else { // normal expression
        let wasVariableSource = statementLine.exportedVariable != nil
        statementLine.exportedVariable = nil
        if wasVariableSource {
            removeSource(line: statementLine)
        }
    }

    return .init(reachedTheBottom: false, dependants: dependants)
}
```

Рисунок 3.18 – Фрагмент будування графу залежностей.

3.3.8 EvaluationContext

Переходимо до етапу обчислення виразів. Після того, як вирази були визначені для обчислення, ми обгортаємо їх у спеціальний контекст обчислення, який називається `EvaluationContext`. Ця обгортка виконує кілька важливих функцій, які значно спрощують і оптимізують процес обчислення. По-перше, `EvaluationContext` використовується для кешування результатів обчислень, що дозволяє уникнути повторних обчислень одних і тих самих виразів, тим самим підвищуючи ефективність роботи додатку. По-друге, цей контекст забезпечує зручний спосіб спілкування з іншими компонентами системи, дозволяючи їм отримувати доступ до результатів обчислень і використовувати їх у своїй роботі.

`EvaluationContext` бере на себе керування всім процесом обчислення. Це особливо важливо при виконанні асинхронних операцій, де обчислення можуть бути розподілені між різними потоками. Використання контексту обчислення дозволяє координувати ці операції і забезпечує, що всі обчислення виконуються в правильному порядку і з урахуванням всіх залежностей.

Крім того, `EvaluationContext` дозволяє відстежувати стан обчислень у реальному часі. Це означає, що якщо під час обчислення виникає помилка або необхідно виконати додаткові дії, система може негайно реагувати і вносити необхідні корективи. Це робить процес обчислення більш гнучким і надійним.

Важливою перевагою використання `EvaluationContext` є його здатність інтегруватися з іншими частинами системи. Це забезпечує цілісність і узгодженість даних у всій системі, оскільки всі компоненти отримують доступ до актуальних і точних результатів обчислень.

```

public final class EvaluationContext: @unchecked Sendable {

    public var didChange = false

    private let dispatchQueue = DispatchQueue(label: "com.calculoapp.Calculo.EvaluationContext")

    private let programContext: ProgramContext

    private var result: Evaluation?

    private var delegate: EvaluationContextDelegate?

    private var evaluation: (any Evaluatable)?

    private var bulkResults: [BulkOperation.ID: BulkOperationResult] = [:]

    private var currentBulkOperations: Set<BulkOperationController> = []

    public init(programContext: ProgramContext, delegate: EvaluationContextDelegate?) {
        self.programContext = programContext
        self.delegate = delegate
    }
}

```

Рисунок 3.19 – Оголошення класу EvaluationContext.

3.3.9 Evaluator

Отримавши всі вирази, ми починаємо планувати ефективну послідовність їх обчислення. Якщо необхідно обчислити лише один вираз, процес не потребує жодних додаткових операцій для коректного знаходження його значення. Однак, часто виникають ситуації, коли кілька виразів залежать від однієї змінної. Для розв’язання таких випадків ми оголошуємо клас Evaluator. Цей клас відповідає за заздалегідь сплановану послідовність обчислення виразів, забезпечуючи ефективність і паралелізованість процесу.

Evaluator буде ретельно аналізувати всі залежності між виразами і створювати оптимізований план обчислень. Всі вирази розбиваються на рівні, кожен з яких обчислює свої вирази паралельно. На першому рівні будуть обчислюватися ті вирази, які не мають жодних залежностей від інших виразів. Це дозволяє розпочати процес обчислення з найпростішого рівня, де кожен вираз може бути обчислений незалежно.

Після завершення обчислень на першому рівні, Evaluator переходить до обчислень на наступному рівні. На цьому рівні знаходяться вирази, які залежать від результатів обчислень попереднього рівня. Такий підхід дозволяє

					ІАЛЦ.467200.003 ПЗ	Арк.
						52
Зм.	Арк.	№ докум.	Підпис	Дата		

ефективно використовувати паралельне виконання, розпаралелюючи обчислення там, де це можливо. Це значно підвищує продуктивність і знижує час, необхідний для завершення обчислень.

Evaluator також враховує можливі зміни в значеннях змінних під час процесу обчислення. Якщо під час виконання обчислень змінюється значення якоїсь змінної, всі вирази, що залежать від цієї змінної, автоматично позначаються як такі, що потребують переобчислення. Це забезпечує точність і узгодженість результатів, оскільки всі залежні вирази будуть оновлені відповідно до нових значень змінних.

```
func levelledEvaluation(of statements: [ParsedStatement], evaluate: ( [ParsedStatement]) -> Void) {
    var statements = Set(statements)

    var parsedStatementsMap: [StatementLine: ParsedStatement] = [:]
    var exporting: [VariableIdentifier: ParsedStatement] = [:]
    for statement in statements {
        if let variable = statement.exportedVariable {
            exporting[variable] = statement
        }
        if let statementLine = statement.statementLine {
            parsedStatementsMap[statementLine] = statement
        }
    }

    while !statements.isEmpty {
        var dependenciesCounter: [StatementLine: Int] = [:]

        for statement in statements {
            guard let statementLine = statement.statementLine else { continue }

            dependenciesCounter[statementLine] = 0

            for variable in statement.usedVariables where exporting[variable] != nil {
                dependenciesCounter[statementLine, default: 0] += 1
            }
        }

        var toEvaluate: [ParsedStatement] = []
        for (sourceLine, dependantsCount) in dependenciesCounter {
            guard dependantsCount == 0,
                let parsedStatement = parsedStatementsMap[sourceLine] else {
                continue
            }

            if let exportedVariable = parsedStatement.exportedVariable {
                exporting.removeValue(forKey: exportedVariable)
            }
            parsedStatementsMap.removeValue(forKey: sourceLine)
            toEvaluate.append(parsedStatement)
            statements.remove(parsedStatement)
        }

        if toEvaluate.isEmpty {
            break
        }
        evaluate(toEvaluate)
    }
}
```

Рисунок 3.20 – Планування порядку обчислень.

Якщо під час обчислення виразу виникає потреба виконати асинхронну дію, наприклад, отримати актуальний курс обміну валют, такий вираз додається в чергу на обчислення. Цим процесом керує клас `AsyncTasksContext`. Цей клас спеціалізується на виконанні асинхронних операцій і зберігає вирази, які необхідно обчислити після отримання результатів асинхронних дій. Крім того, `AsyncTasksContext` управляє виразами, які залежать від тих, що очікують завершення асинхронних операцій.

`AsyncTasksContext` забезпечує ефективне управління асинхронними діями, необхідними для обчислення виразів. Наприклад, якщо вираз потребує отримання зовнішніх даних, цей клас ініціює асинхронний запит до відповідного сервісу і додає вираз до черги на обчислення. Це дозволяє іншим частинам додатку продовжувати роботу без затримок, пов'язаних з очікуванням результатів асинхронних запитів.

Після завершення асинхронної операції і отримання необхідних даних, `AsyncTasksContext` автоматично виконує вирази, які залежали від цих даних. Це забезпечує безперервність і узгодженість обчислень, оскільки всі вирази, що залежать від результатів асинхронних дій, обчислюються своєчасно і правильно. Крім того, цей клас зберігає і відслідковує всі залежні вирази, що дозволяє ефективно керувати обчисленнями навіть у складних сценаріях з численними взаємозалежностями.

`AsyncTasksContext` також відповідає за оптимізацію процесу обчислення, розподіляючи ресурси для виконання асинхронних дій і забезпечуючи, щоб обчислення відбувалися якомога швидше і ефективніше. Це особливо важливо в умовах великої кількості одночасних запитів, коли необхідно мінімізувати час очікування і забезпечити високу продуктивність системи та коли декілька виразів одночасно чекають на виконання іншого виразу.

Цей підхід дозволяє зменшити навантаження на основні потоки додатку і забезпечити швидке та ефективне обчислення виразів, що потребують асинхронних дій. Завдяки `AsyncTasksContext`, наш додаток може виконувати складні обчислення, які включають зовнішні запити, без втрати продуктивності і зберігаючи високу точність результатів.

```
class AsyncTasksContext {
    private let queue = DispatchQueue(label: "com.calculoapp.Calculo.AsyncTasksContext", attributes: .concurrent)
    private var local: [VariableIdentifier: [Weak<EvaluationContext>]] // waiting for variables
    private var external: [any EvaluationExternalTask]
    weak var delegate: AsyncTasksContextDelegate?

    nonisolated init(
        local: [VariableIdentifier: [Weak<EvaluationContext>]] = .init(),
        external: [any EvaluationExternalTask] = [],
        delegate: AsyncTasksContextDelegate? = nil
    ) {
        self.local = local
        self.external = external
        self.delegate = delegate
    }
}
```

Рисунок 3.21 – Оголошення класу `AsyncTaskContext`.

Одночасний доступ до `AsyncTasksContext` можуть мати кілька виразів, які виконуються на різних потоках. З огляду на це, цей клас потребує використання методів синхронізації потоків для забезпечення коректної роботи. Щоб досягти цього, ми використовуємо `DispatchQueue` у тих місцях, де можливий одночасний доступ. Цей підхід використовує чергу для синхронізації потоків, що дозволяє уникнути конфліктів і забезпечити послідовність обробки даних.

Враховуючи можливість одночасного доступу до ресурсів `AsyncTasksContext` з різних потоків, важливо гарантувати, що дані залишаються узгодженими і цілісними. Використання `DispatchQueue` дозволяє нам ефективно керувати доступом до спільних ресурсів. Цей метод забезпечує, що тільки один потік має доступ до критичної секції коду в будь-який момент часу, що усуває можливість конфліктів і даних, що перекриваються.

DispatchQueue працює шляхом створення черги завдань, які виконуються послідовно. Коли потік хоче отримати доступ до ресурсу, він додає своє завдання до черги. Це забезпечує, що завдання виконуються в порядку, в якому вони були додані, і ніколи не виконуються одночасно. Такий підхід дозволяє нам уникнути проблем, пов'язаних з одночасним доступом до спільних ресурсів, і забезпечити коректність обчислень.

У реалізації AsyncTasksContext ми використовуємо DispatchQueue для синхронізації доступу до критичних секцій коду, таких як додавання нових виразів до черги обчислень або оновлення результатів асинхронних дій. Це гарантує, що всі операції з ресурсами виконуються у визначеному порядку, без конфліктів і помилок.

Крім того, використання DispatchQueue дозволяє нам легко масштабувати систему, додаючи нові завдання до черги без ризику порушення узгодженості даних. Це особливо важливо в умовах великої кількості одночасних операцій, коли потрібно забезпечити високу продуктивність і надійність системи.

```

extension AsyncTasksContext: EvaluationContextDelegate {

    func saved(variable identifier: VariableIdentifier, from context: EvaluationContext) {
        let contexts: [Weak<EvaluationContext>] = queue.sync {
            guard let contexts = local.removeValue(forKey: identifier) else {
                return []
            }
            return contexts
        }

        delegate?.run(contexts: contexts)
    }

    func add(externalTask task: any EvaluationExternalTask, from context: EvaluationContext) {
        queue.async(flags: .barrier) { [weak self, task] in
            guard let self else { return }
            let idx = external.firstIndex(where: { $0.isEqual(to: task) })
            if let idx {
                external[idx].add(context: context)
            } else {
                var task = task
                task.add(context: context)
                external.append(task)
                start(task: task)
            }
        }
    }

    func wait(for identifier: VariableIdentifier, context: EvaluationContext) {
        queue.async(flags: .barrier) { [weak self, identifier] in
            self?.local[identifier, default: []].append(.init(context))
        }
    }
}

```

Рисунок 3.22 – Приклад використання DispatchQueue.

Однією з критичних секцій алгоритму інтерпретатора є синхронізація передачі змін тексту виразів та їх обчисленням. Оскільки інтерпретатор отримує зміни до того, як завершиться внутрішня обробка тексту всередині UITextView, яка займає певний час, і саме обчислення виразів також потребує деякого часу, можуть виникати дві ситуації:

1. Внутрішня обробка тексту завершується швидше, ніж інтерпретація виразів.
2. Внутрішня обробка тексту завершується пізніше, ніж інтерпретація виразів.

Це важливо для отримання правильних координат результатів виконання виразів. Оскільки неможливо передбачити, яким шляхом піде виконання у конкретному випадку, ставимо обмеження, що зміни в текст можна вносити лише після завершення виконання обчислень попередніх змін.

Це дозволяє уникнути конфліктів і забезпечити правильну обробку тексту та обчислень.

Для реалізації цього підходу ми використаємо два м'ютекси `NSLock` всередині структури `InterpreterSynchronization` (рисунок 3.22). Перший м'ютекс буде відповідати за внутрішню обробку тексту всередині `UITextView`. Цей м'ютекс гарантує, що жодні зміни не можуть бути внесені, поки текст ще обробляється. Другий м'ютекс буде контролювати процес обчислення виразів всередині інтерпретатора. Він забезпечує, що жодні нові обчислення не почнуться, поки попередні обчислення не завершаться.

Цей підхід дозволяє ефективно синхронізувати два критичні процеси і уникнути конфліктів. Коли відбувається зміна тексту виразів, перший м'ютекс блокується до завершення обробки тексту. Це гарантує, що нові зміни не будуть внесені, поки текст ще обробляється. Після завершення обробки тексту, м'ютекс розблоковується, дозволяючи виконати наступні кроки.

Одночасно другий м'ютекс блокується на час обчислення виразів. Це гарантує, що нові обчислення не почнуться, поки попередні не завершаться. Коли обчислення завершуються, м'ютекс розблоковується, дозволяючи інтерпретатору обробити нові зміни.

Цей метод забезпечує узгодженість і точність обробки тексту і обчислень, знижуючи ризик виникнення помилок. Завдяки використанню двох м'ютексів, ми можемо гарантувати, що всі зміни і обчислення виконуються послідовно і без конфліктів, забезпечуючи стабільну і надійну роботу додатку.

```

struct InterpreterSynchronization {

    private let inputProcessingLock = NSLock()

    private let inputEvaluatingLock = NSLock()

    func startInnerProcessing() {
        inputEvaluatingLock.lock()
    }

    func finishInnerProcessing() {
        inputEvaluatingLock.unlock()
    }

    func startTextProcessing() {
        inputProcessingLock.lock()
    }

    func finishTextProcessing() {
        inputProcessingLock.unlock()
    }

    func ensurePrevEvaluationFinished() {
        inputEvaluatingLock.lock()
        inputEvaluatingLock.unlock()
    }

    func ensureTextProcessed() {
        inputProcessingLock.lock()
        inputProcessingLock.unlock()
    }

}

```

Рисунок 3.23 – Структура синхронізації інтерпретатора.

```

func inputWillChange(string: String, range: NSRange) {
    benchmark.start()
    sync.startTextProcessing()
    sync.startInnerProcessing()
    queue.async { [weak self] in
        guard let self else { return }
        if string.isEmpty && range == .init(location: 0, length: 0) {
            sync.finishInnerProcessing()
            return sync.finishTextProcessing()
        }
        statementLine.didChange(string: string, range: range)
        evaluate()
    }
}

@MainActor func inputDidChange(invalidateLayout: Bool = false) {
    sync.ensurePrevEvaluationFinished()
    inputLayout.invalidatePosition(head: offsetHead ?? statementLine, invalidateLayout: invalidateLayout)
    sync.finishTextProcessing()
}

```

Рисунок 3.24 – Алгоритм синхронізації.

```

func evaluate() {
    let context = inputLayout.findInvalidatedLines(head: statementLine)
    offsetHead = context.offsetLineHead
    evaluate(lines: context.invalidatedLines)
}

func evaluate(lines: [StatementLine]) {
    let parser = Parser(programContext: programContext)
    programContext.invalidate(lines: lines)

    let parsedStatements = parser.parse(lines: lines)
    let dependencies = dependencyGraph.analyze(statements: parsedStatements)
    let parsedDependencies = parser.parse(lines: dependencies)
    dependencyGraph.reregister(statements: parsedDependencies)

    let statements = parsedStatements + parsedDependencies

    evaluator.evaluate(statements: statements)

    applyInitialResults(parsedStatements.flatMap { $0.syntaxHighlighting })
}

```

Рисунок 3.25 – Алгоритм обробки виразів.

3.4 Створення шаблонів обчислень

Після завершення розробки інтерпретатора ми можемо перейти до розробки системи шаблонів у нашому додатку. Для цього ми створимо клас Template, який буде відігравати ключову роль у роботі з текстовими обчисленнями, зокрема виявляти і обробляти спеціальні оператори, що позначають введення або виведення інформації в шаблоні. Головною задачею класу Template є пошук у тексті обчислень спеціальних операторів, таких як input та output.

Клас Template здійснюватиме сканування тексту, шукаючи оператори input та output. Оператор input позначає місце, де очікується введення значення користувачем, а після нього слідує рядок із назвою очікуваного значення. Оператор output, відповідно, позначає місце виводу результату обчислення шаблону.

Сканування тексту відбувається таким чином: при кожній зміні тексту парсер шукає в ньому ці оператори. Якщо оператор виявлено, відповідна StatementLine позначається як місце, де очікується введення значення

					ІАЛЦ.467200.003 ПЗ	Арк.
						60
Зм.	Арк.	№ докум.	Підпис	Дата		

користувачем, або як місце виводу результату обчислення шаблону. Після цього клас Template просто шукає ці помічені рядки тексту, полегшуючи процес роботи з шаблонами.

```
public class Template {  
  
    public var title = "Template"  
  
    public var isComplete: Bool {  
        (!inputs.isEmpty && !outputs.isEmpty)  
    }  
  
    public private(set) var inputs: [Input]  
  
    public private(set) var outputs: [NumericalOutput]  
  
    public weak var delegate: TemplateDelegate?  
  
    weak var head: StatementLine!  
  
    let interpreter: Interpreter  
  
    convenience init(head: StatementLine, interpreter: Interpreter) {  
        var inputs: [Input] = []  
        var outputs: [NumericalOutput] = []  
  
        for line in head.iter() {  
            guard let ast = line.ast else { continue }  
  
            inputs += ast.inputs  
            if let label = ast.outputLabel {  
                outputs.append(.init(label: label, line: .init(line)))  
            }  
        }  
        self.init(head: head, interpreter: interpreter, inputs: inputs, outputs: outputs)  
    }  
}
```

Рисунок 3.26 – Структура класу Template.

Для введення значень у шаблоні використовується наступний метод. Коли створюється шаблон, клас Template також запам'ятовує розташування очікуваного введення значень. Під час введення даних користувачем клас замінює місце вводу на введені значення. Важливо зазначити, що це рішення має певні обмеження. Головним недоліком є те, що введені значення не може бути довшим за очікуване місце введення. Довжину можна регулювати кількістю пробілів між оператором input та назвою очікуваного значення.

```

@MainActor func setInput(_ input: String, at idx: Int) {
    let position = inputs[idx].position
    let input = input.padRight(to: position.length)

    interpreter.inputWillChange(string: input, range: position.range)
    interpreter.inputDidChange()
}

```

Рисунок 3.27 – Функція введення значення в шаблон обчислення.

					ІАЛЦ.467200.003 ПЗ	Арк.
						62
Зм.	Арк.	№ докум.	Підпис	Дата		

Висновки до розділу 3

Під час розробки нашого додатку ми зробили значний крок вперед у створенні надійної та ефективної системи для обробки текстових виразів і математичних обчислень. Завдяки ретельно продуманій архітектурі та впровадженню ключових компонентів, ми змогли створити потужний інструмент, який відповідає сучасним вимогам і забезпечує високу продуктивність.

У ході роботи ми розробили та інтегрували кілька основних класів, кожен з яких виконує свою критичну роль у загальному процесі інтерпретації і обчислення. Цей підхід дозволив нам досягти високого рівня гнучкості і масштабованості системи. Кожен з компонентів був ретельно спроектований та оптимізований для виконання своїх завдань, що дозволило нам створити узгоджену та ефективну структуру, здатну обробляти складні математичні вирази з високою точністю і швидкістю.

Архітектура нашого додатку була розроблена з урахуванням сучасних тенденцій і потреб користувачів, що дозволяє забезпечити максимальну продуктивність навіть у складних умовах. Використання передових технологій і методів програмування дозволило нам досягти високої ефективності та надійності системи.

Завдяки гнучкій і модульній архітектурі, наш додаток легко адаптується до змін і розширень, що забезпечує його довговічність і актуальність. Ми впровадили рішення, які дозволяють системі ефективно працювати у багатопотоковому середовищі, обробляючи одночасно велику кількість запитів без втрати продуктивності. Це особливо важливо для додатків, що працюють з великими обсягами даних і потребують високої швидкості обробки.

Одним з ключових аспектів розробки була безпека додатку. Ми впровадили сучасні методи захисту даних, включаючи шифрування і безпечне зберігання інформації, що гарантує конфіденційність і цілісність даних

					ІАЛЦ.467200.003 ПЗ	Арк.
						63
Зм.	Арк.	№ докум.	Підпис	Дата		

користувачів. Система аутентифікації і авторизації забезпечує захист від несанкціонованого доступу, що є критично важливим для збереження конфіденційної інформації. Крім того, регулярні аудити безпеки і оновлення системи дозволяють своєчасно виявляти і усувати потенційні загрози.

Таким чином, ми створили потужний і надійний інструмент, який відповідає найвищим стандартам якості, продуктивності і безпеки, здатний задовольнити потреби навіть найвимогливіших користувачів. Ми впевнені, що наш додаток стане цінним інструментом для всіх, хто потребує ефективних рішень для обробки текстових виразів і математичних обчислень, забезпечуючи при цьому високий рівень захисту даних.

					ІАЛЦ.467200.003 ПЗ	Арк.
						64
Зм.	Арк.	№ докум.	Підпис	Дата		

РОЗДІЛ 4. ПРИКЛАДНІ АСПЕКТИ ЗАСТОСУВАННЯ ДОДАТКУ

4.1 Демонстрація практичних можливостей розробленого додатку

Після запуску додатку користувач миттєво опиняється на головному екрані, де перед ним відкривається зручний та інтуїтивно зрозумілий інтерфейс калькулятора. Тут він може легко розпочати вводити математичні вирази та розрахунки, які йому потрібні.

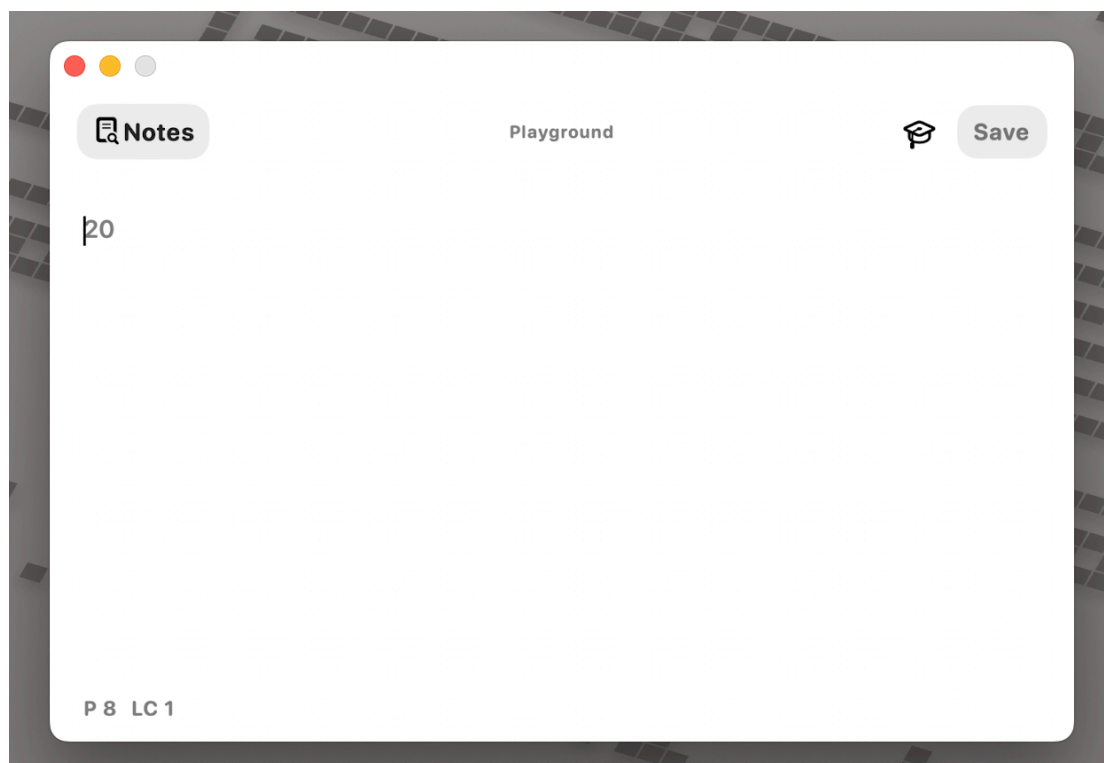


Рисунок 4.1 – Головний екран додатку.

Користувач має можливість вводити найрізноманітніші математичні вирази, використовуючи зручний інтерфейс нашого додатку. Після того як вираз буде обчислено, результат відобразиться безпосередньо після введеного виразу, забезпечуючи швидкий доступ до необхідної інформації виразу.

Приклад на рисунку 4.2.



Рисунок 4.2 – Введення математичних виразів.

Користувач нашого додатку має можливість працювати з широким спектром одиниць вимірювання, майже всі з яких належать до Міжнародної системи одиниць (SI). Крім того, додаток підтримує використання кодів валют згідно зі стандартом ISO 4217, що дозволяє виконувати фінансові розрахунки з використанням офіційних кодів валют різних країн. Це робить додаток універсальним інструментом як для наукових, так і для фінансових розрахунків, забезпечуючи користувачам максимальну зручність та точність.

Приклад на рисунку 4.3.

					ІАЛЦ.467200.003 ПЗ	Арк.
						66
Зм.	Арк.	№ докум.	Підпис	Дата		



Рисунок 4.3 – Демонстрація одиниць вимірювання.

Додаток надає користувачам можливість оголошувати змінні, які можуть бути використані в подальших розрахунках. Змінні є імутабельними, тобто після їх оголошення вони не можуть бути змінені в процесі обчислень, що гарантує стабільність і передбачуваність результатів. Проте, якщо користувач вирішить змінити значення змінної, додаток автоматично перераховує всі вирази, де ця змінна була використана. Це забезпечує точність і актуальність всіх обчислень без необхідності повторного введення даних.

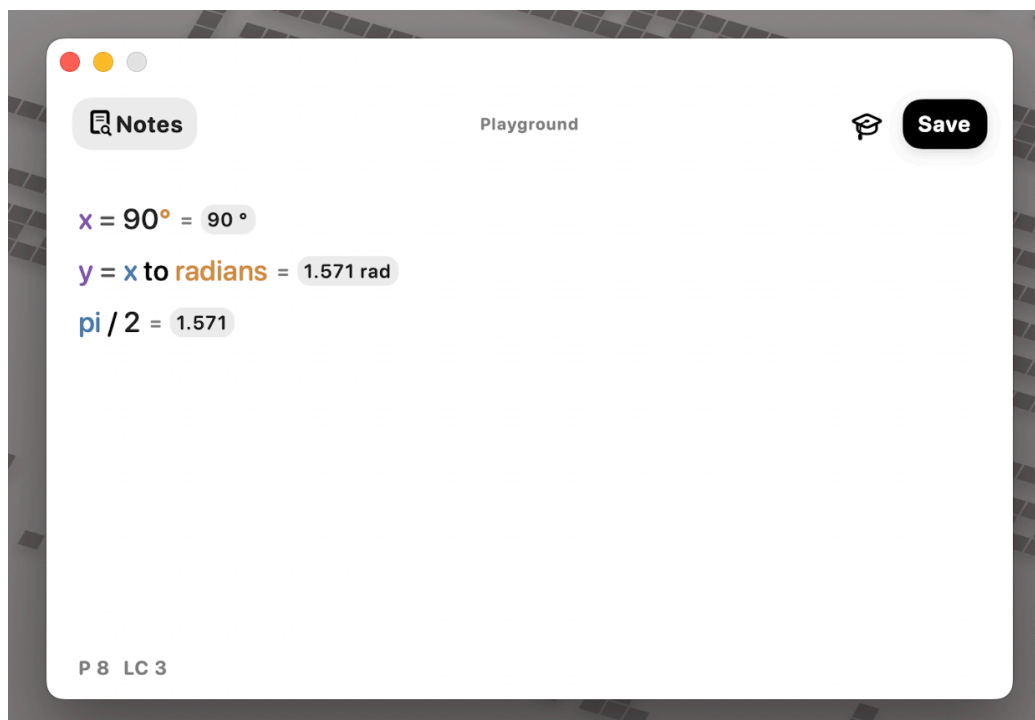


Рисунок 4.4 – Демонстрація змінних в додатку.

Використовуючи оператори введення (input) та виведення (output), користувач має можливість створювати власні шаблони. Це може бути надзвичайно корисним для нього в тих випадках, коли необхідно автоматизувати повторювані обчислення або зберегти певну послідовність дій для майбутнього використання. Створюючи такі шаблони, користувач може значно спростити свою роботу, скоротити час на виконання завдань та підвищити ефективність процесу обчислень.

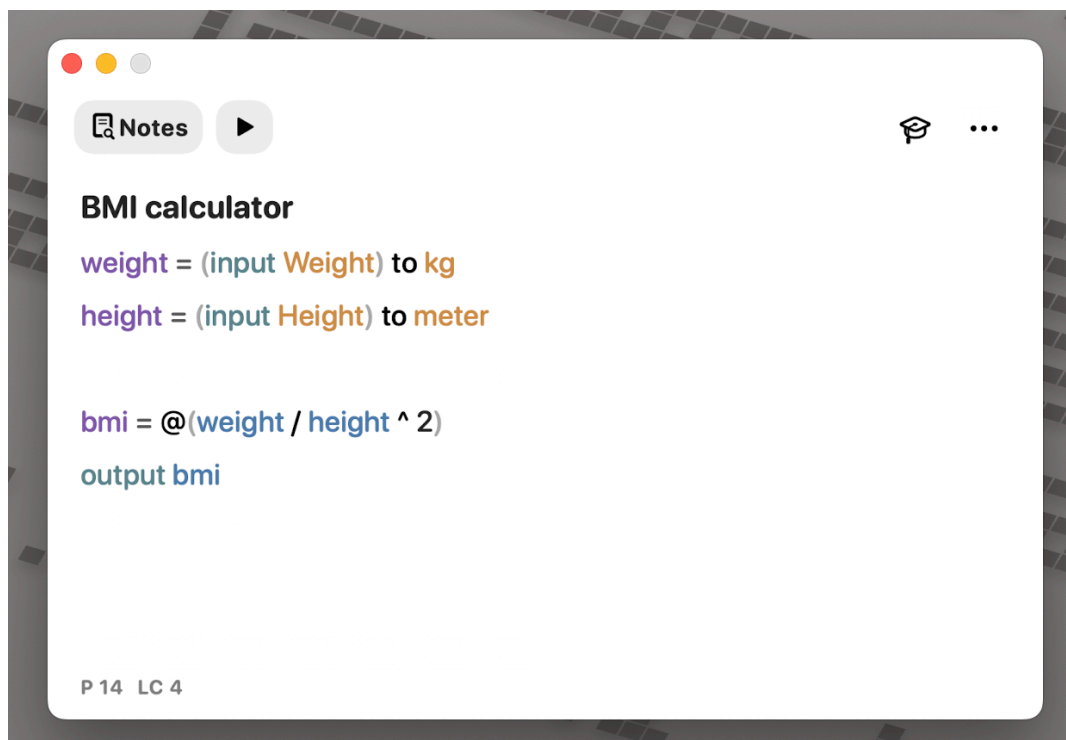


Рисунок 4.5 – Приклад оголошення шаблону.

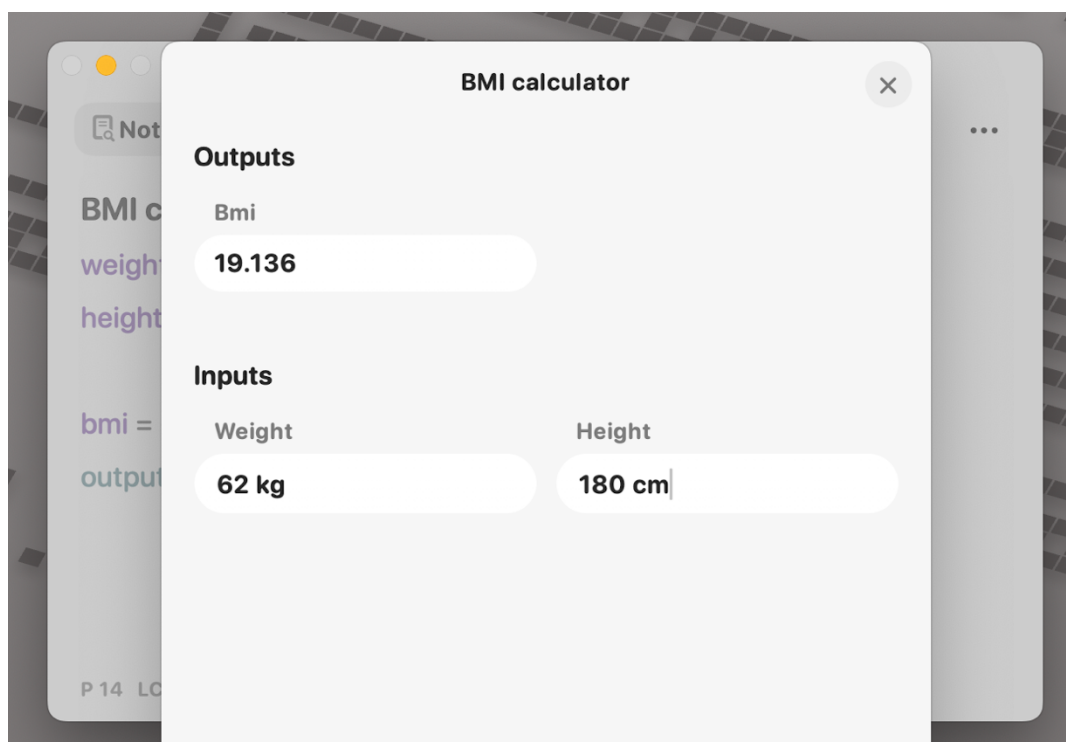


Рисунок 4.6 – Демонстрація використання шаблону.

Користувач нашого додатку має можливість зберігати свої обчислення, просто натиснувши кнопку “Зберегти” (Save). Після цього всі збережені обчислення будуть доступні на спеціальному екрані, де він зможе легко їх переглядати. Для створення нової сторінки з обчисленнями користувачеві потрібно натиснути кнопку “Ігровий майданчик” (Playground), що дозволить йому розпочати новий набір обчислень у чистому просторі, без необхідності видаляти або змінювати попередні результати. Це забезпечує зручність у роботі з обчисленнями та дозволяє легко організувати й управляти ними.

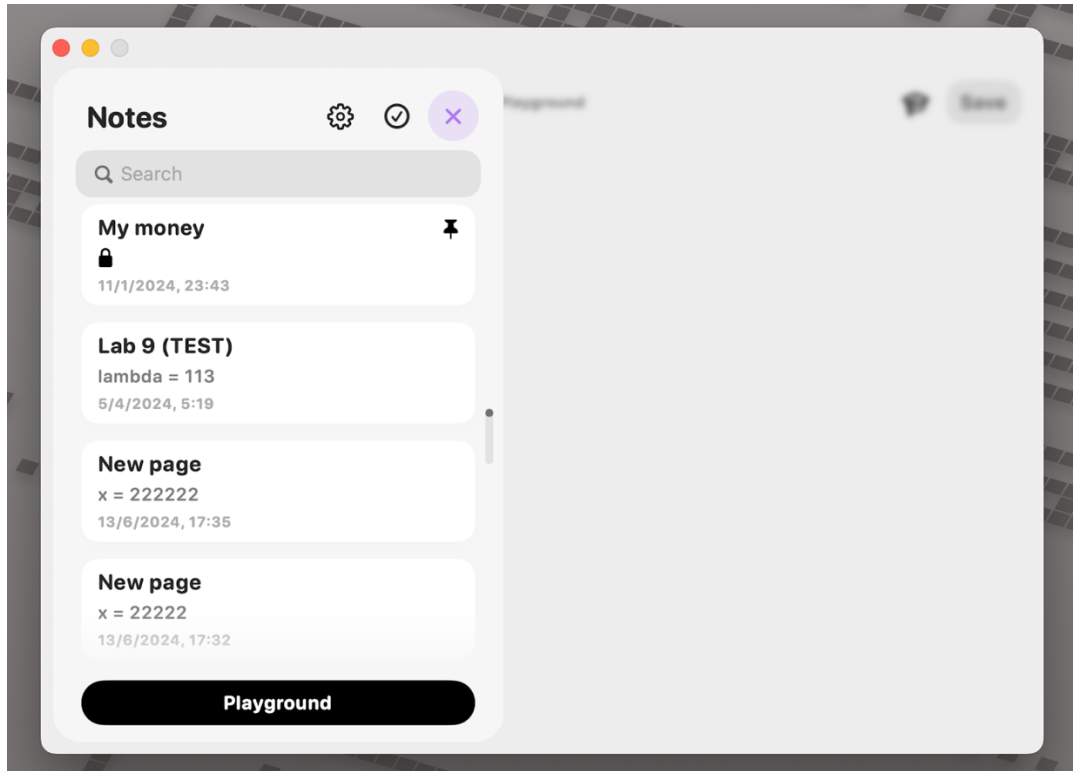


Рисунок 4.7 – Екран збережених обчислень.

У додатку також передбачена функція захисту збережених обчислень. Ця функція дозволяє блокувати доступ до обчислень, що зберігаються, забезпечуючи додатковий рівень безпеки. При спробі відкрити заблоковане обчислення користувач повинен пройти процедуру аутентифікації, використовуючи біометричні дані. В залежності від типу пристрою, це може бути сканування відбитка пальця або розпізнавання обличчя. Ця функція

гарантує, що тільки авторизовані користувачі зможуть отримати доступ до своїх конфіденційних даних, що додає додатковий рівень захисту і спокою для користувачів нашого додатку.

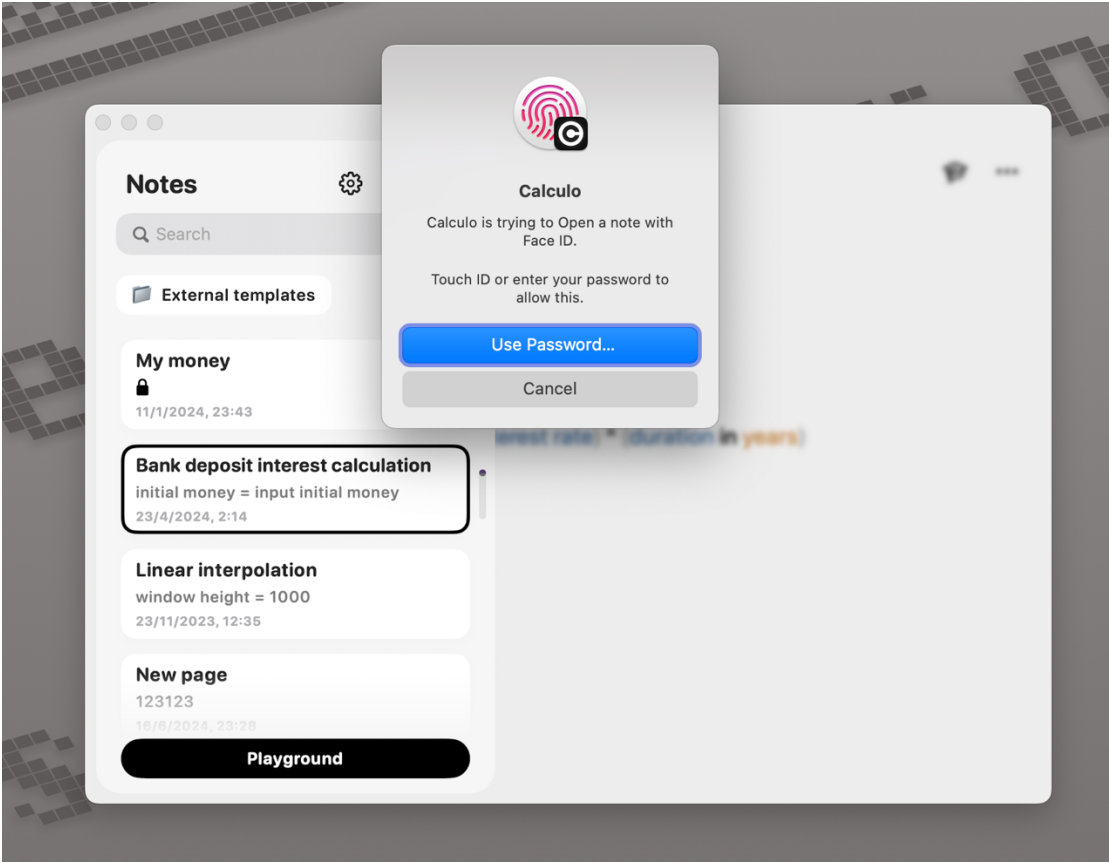


Рисунок 4.8 – Відкривання заблокованого обчислення.

4.2 Перспективи подальшого розвитку додатку

Для подальшого розвитку та підвищення ефективності додатку необхідно реалізувати такі покращення:

- Додати можливість створення та використання масивів, що дозволить виконувати більш складні обчислення і маніпуляції з даними.
- Забезпечити можливість швидкого і зручного поширення результатів обчислень через різні канали, такі як електронна пошта, соціальні мережі або месенджери.

- Впровадити опції для зміни кольорів підсвітки синтаксису, що дозволить кожному користувачеві налаштувати інтерфейс до своїх вподобань та потреб.
- Реалізувати можливість повторного використання обчислень з попередньо збережених результатів, що дозволить створювати нові вирази на основі вже наявних даних.
- Підвищити інтуїтивну зрозумілість, зручність та привабливість інтерфейсу.

					ІАЛЦ.467200.003 ПЗ	Арк.
						72
Зм.	Арк.	№ докум.	Підпис	Дата		

Висновок до розділу 4

У цій роботі ми детально розглянули основні функції та можливості нашого додатку для обчислень. Від інтуїтивно зрозумілого інтерфейсу, який дозволяє швидко і легко вводити математичні вирази, до підтримки змінних і автоматичного перерахунку, наш додаток пропонує ефективні рішення для різноманітних обчислювальних задач. Підтримка широкого спектру одиниць вимірювання SI та валютних кодів ISO 4217 забезпечує гнучкість і точність у наукових і фінансових розрахунках.

Також були розглянуті подальші кроки для розвитку додатку. Впровадження цих покращень значно підвищить функціональність та зручність у користуванні. Завдяки цьому додаток відповідатиме найвищим стандартам якості, забезпечуючи максимальну ефективність і комфорт для всіх користувачів. Наші подальші кроки спрямовані на розширення можливостей, вдосконалення інтерфейсу та оптимізацію процесів, що дозволить створити інструмент, який повністю задовольнить потреби сучасних користувачів.

Однією з найважливіших рис нашого додатку є його високий рівень безпеки. Ми впровадили функцію блокування збережених обчислень, що забезпечує додатковий захист даних користувачів. При спробі відкрити заблоковане обчислення, користувач повинен пройти біометричну аутентифікацію, використовуючи сканування відбитка пальця або розпізнавання обличчя, в залежності від типу пристрою. Це гарантує, що доступ до конфіденційних даних мають лише авторизовані користувачі, що значно підвищує рівень безпеки та захисту інформації. Використання вбудованих функцій безпеки пристроїв Apple, таких як апаратне шифрування і захищене зберігання даних, забезпечує додатковий рівень захисту, роблячи наш додаток надійним і безпечним для всіх користувачів.

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		73

ВИСНОВКИ

У рамках цього дипломного проєкту успішно розроблено калькулятор з асинхронними діями, що забезпечує ефективну обробку великих обсягів даних і високу продуктивність. Проєкт реалізував усі поставлені завдання, досягнувши значних результатів у кількох ключових аспектах.

По-перше, розроблена архітектура додатку забезпечує ефективну обробку даних у фоновому режимі без негативного впливу на основну роботу програми. Це стало можливим завдяки ретельному вибору сучасних технологій та підходів, що дозволяють підтримувати безперебійну роботу додатку.

По-друге, реалізована функція створення шаблонів обчислень значно підвищила зручність використання додатку. Користувачі тепер можуть зберігати часто використовувані обчислення і швидко їх повторно використовувати, що економить час і спрощує виконання рутинних задач.

По-третє, впроваджені комплексні заходи безпеки забезпечують надійний захист даних користувачів. Використання шифрування та біометричної аутентифікації гарантує, що доступ до конфіденційних даних мають лише авторизовані користувачі, що значно підвищує рівень захисту інформації.

По-четверте, реалізація механізмів аутентифікації та авторизації для захисту API додатку дозволяє запобігти несанкціонованому доступу до функціональних можливостей програми. Це забезпечує додатковий рівень безпеки і дозволяє контролювати доступ до важливих даних та функцій.

Ці особливості роблять калькулятор не лише потужним інструментом для професіоналів, але й зручним для повсякденного використання. Завдяки цьому, додаток може ефективно використовуватися у різних сферах, забезпечуючи користувачам інтуїтивно зрозумілий інтерфейс та широкий функціонал, що відповідає їхнім потребам.

					ІАЛЦ.467200.003 ПЗ	Арк.
						74
Зм.	Арк.	№ докум.	Підпис	Дата		

Завдяки інтеграції асинхронних процесів та впровадженню високих стандартів безпеки, створений калькулятор став надійним, функціональним та зручним інструментом для користувачів у різних галузях, включаючи науку, інженерію, фінанси та бізнес. Додаток відповідає сучасним вимогам і очікуванням, забезпечуючи високу задоволеність користувачів та конкурентоспроможність на ринку програмного забезпечення.

					ІАЛЦ.467200.003 ПЗ	Арк.
						75
Зм.	Арк.	№ докум.	Підпис	Дата		

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Soulver [Електронний ресурс] – Режим доступу до ресурсу: <https://soulver.app>
2. Numbr [Електронний ресурс] – Режим доступу до ресурсу: <https://numbr.dev/>
3. Omni calculator - Terms and Conditions of Use [Електронний ресурс] – Режим доступу до ресурсу: <https://www.omnicalculator.com/privacy-policy>
4. Ахо А. В., Лам М. С., Сеті Р., Уллман Дж. Д. Компілятори: Принципи, методи та інструменти. 2-ге видання. - Бостон: Addison-Wesley, 2006. - С. 209-300.
5. UIKit. [Електронний ресурс] - Режим доступу до ресурсу: <https://developer.apple.com/documentation/uikit>
6. SwiftUI. [Електронний ресурс] - Режим доступу до ресурсу: <https://developer.apple.com/xcode/swiftui/>
7. The Swift Programming Language. Guided Tour. [Електронний ресурс] - Режим доступу до ресурсу: <https://docs.swift.org/swift-book/documentation/the-swift-programming-language/guidedtour/>
8. Core Data. [Електронний ресурс] - Режим доступу до ресурсу: <https://developer.apple.com/documentation/coredata/>
9. Advanced Encryption Standard [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/Advanced_Encryption_Standard
10. Device Check [Електронний ресурс] – Режим доступу до ресурсу: <https://developer.apple.com/documentation/devicecheck>

ДОДАТОК 1

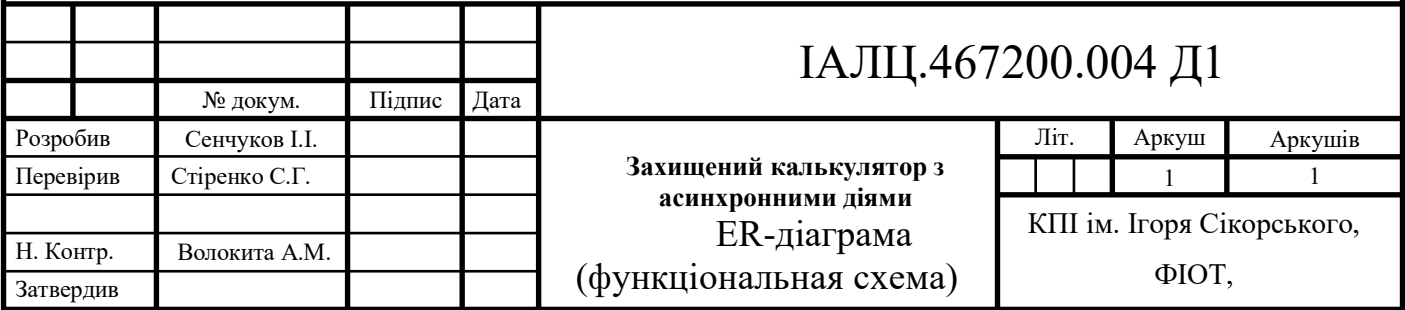
Захищений калькулятор з асинхронними діями

**ER-діаграма
(функціональна схема)**

ІАЛЦ.467200.004 Д1

Аркушів 1

Київ – 2024 р.



ДОДАТОК 2

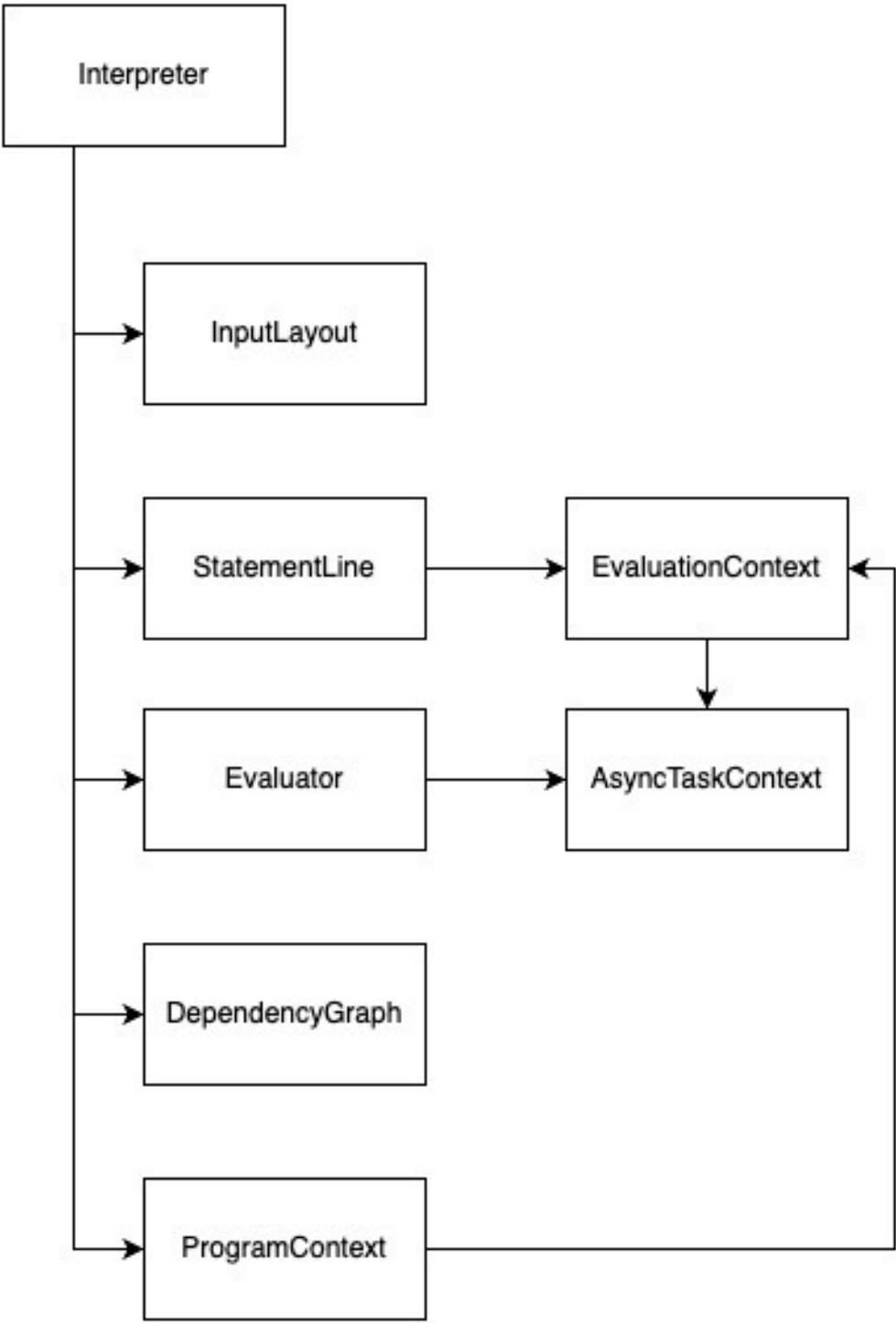
Захищений калькулятор з асинхронними діями

**Схема взаємодії компонентів системи
(структурна схема)**

ІАЛЦ.467200.005 Д2

Аркушів 1

Київ – 2024 р.



					ІАЛЦ.467200.005 Д2				
		№ докум.	Підпис	Дата					
Розробив	Сенчуков І.І.				Захищений калькулятор з асинхронними діями Схема взаємодії компонентів системи (структурна схема)		Літ.	Аркуш	Аркушів
Перевірив	Стіренко С.Г.							1	1
							КПІ ім. Ігоря Сікорського, ФІОТ,		
Н. Контр.	Волокита А.М.								
Затвердив									

ДОДАТОК 3

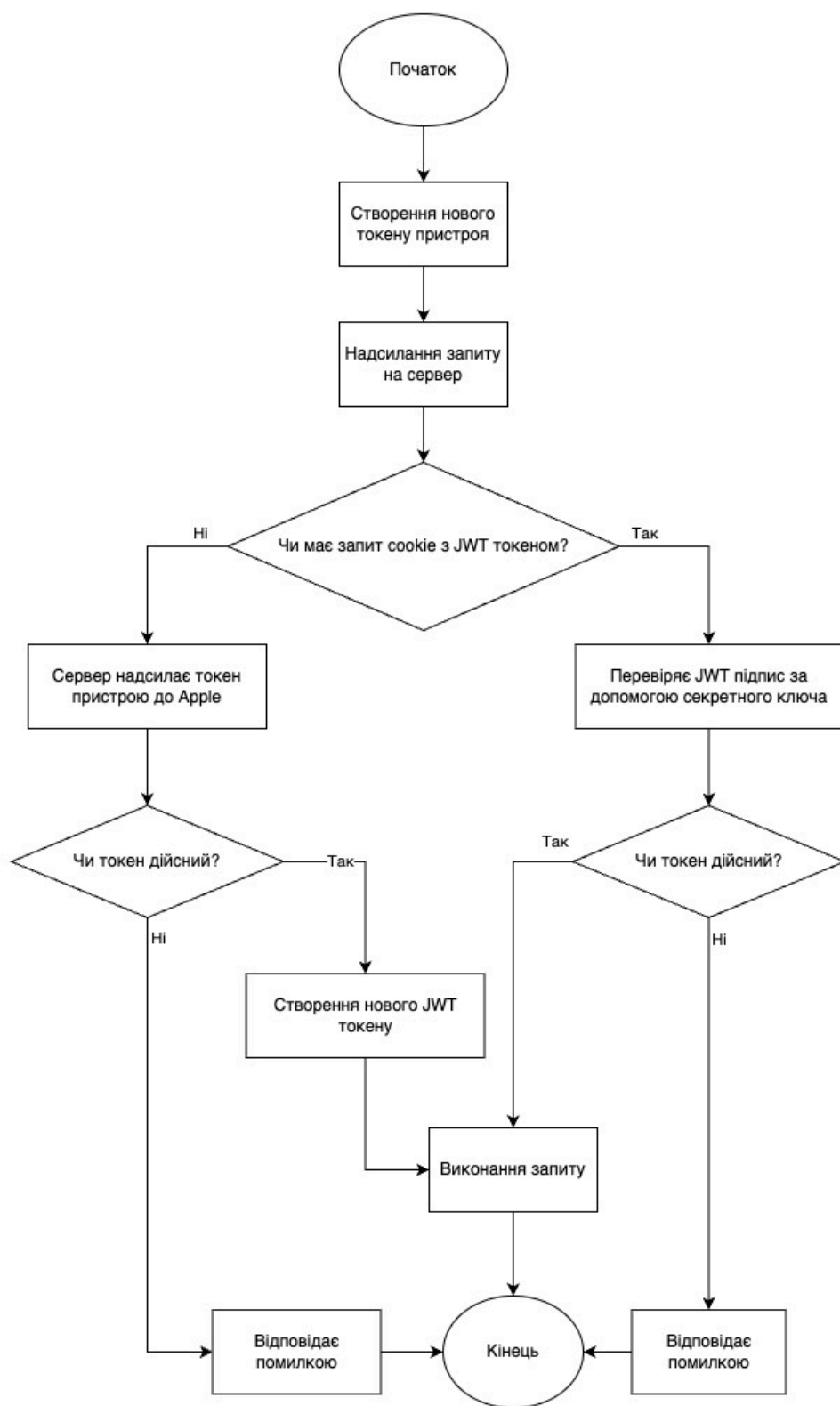
Захищений калькулятор з асинхронними діями

**Алгоритм аутентифікації
(принципова схема)**

ІАЛЦ.467200.006 ДЗ

Аркушів 1

Київ – 2024 р.



					ІАЛЦ.467200.007 Д4			
		№ докум.	Підпис	Дата	Захищений калькулятор з асинхронними діями Текст програмного коду			
Розробив	Сенчуков І.І.							
Перевірив	Стіренко С.Г.							
Н. Контр.	Волокита А.М.							
Затвердив					Літ. Аркуш Аркушів 1 9 КПІ ім. Ігоря Сікорського, ФІОТ,			

ДОДАТОК 4

Захищений калькулятор з асинхронними діями

Текст програмного коду

ІАЛЦ.467200.007 Д4

Аркушів 9

Київ – 2024 р.

```
import Foundation
import Parser
import OSLog

protocol AsyncTasksContextDelegate: AnyObject {
    func run(contexts: [Weak<EvaluationContext>])
}

final class AsyncTasksContext: Sendable {

    private let queue = DispatchQueue(label:
"com.calculoapp.Calculo.AsyncTasksContext", attributes: .concurrent)

    private nonisolated(unsafe) var local: [VariableIdentifier:
[Weak<EvaluationContext>]] // waiting for variables

    private nonisolated(unsafe) var external: [any EvaluationExternalTask]

    private let bulkEvaluator = BulkCalculationsEvaluator()

    nonisolated(unsafe) weak var delegate: AsyncTasksContextDelegate?

    nonisolated init(
        local: [VariableIdentifier: [Weak<EvaluationContext>]] = .init(),
        external: [any EvaluationExternalTask] = [],
        delegate: AsyncTasksContextDelegate? = nil
    ) {
        self.local = local
        self.external = external
        self.delegate = delegate
    }

    deinit {
        bulkEvaluator.stop()
    }
}
```

					ІАЛЦ.467200.007 Д4			
		№ докум.	Підпис	Дата				
Розробив	Сенчуков І.І.				Захищений калькулятор з асинхронними діями Текст програмного коду	Літ.	Аркуш	Аркушів
Перевірив	Стіренко С.Г.						1	9
						КПІ ім. Ігоря Сікорського, ФІОТ,		
Н. Контр.	Волокита А.М.							
Затвердив								

```

extension AsyncTasksContext: EvaluationContextDelegate {

    func saved(variable identifier: VariableIdentifier, from context:
EvaluationContext) {
        let contexts: [Weak<EvaluationContext>] = queue.sync {
            guard let contexts = local.removeValue(forKey: identifier) else {
                return []
            }
            return contexts
        }

        delegate?.run(contexts: contexts)
    }

    func add(externalTask task: any EvaluationExternalTask, from context:
EvaluationContext) {
        queue.async(flags: .barrier) { [weak self, task] in
            guard let self else { return }
            let idx = external.firstIndex(where: { $0.isEqual(to: task) })
            if let idx {
                external[idx].add(context: context)
            } else {
                var task = task
                task.add(context: context)
                external.append(task)
                start(task: task)
            }
        }
    }

    func wait(for identifier: VariableIdentifier, context: EvaluationContext) {
        queue.async(flags: .barrier) { [weak self, identifier] in
            self?.local[identifier, default: []].append(.init(context))
        }
    }

    func performBulk(operation: BulkOperation, from context: EvaluationContext) -
> BulkOperationController {
        bulkEvaluator.addOperation(operation, from: context) { [weak self, weak
context] in
            self?.delegate?.run(contexts: [.init(context)])
        }
    }
}

```

```

}

private extension AsyncTasksContext {

    func start(task: any EvaluationExternalTask) {
        Task.detached { [weak self] in
            await task.performTask()

            guard let self,
                let finishedTaskIdx = external.firstIndex(where: { $0.isEqual(to: task) })
        else {
            return
        }

        let finishedTask = external.remove(at: finishedTaskIdx)
        for context in finishedTask.contexts {
            context.data?.refreshBulk()
        }
        delegate?.run(contexts: .init(finishedTask.contexts))
    }
}

}

public struct ParsedStatement: Sendable {
    nonisolated(unsafe) static let empty = ParsedStatement(exportedVariable: nil,
        usedVariables: [], syntaxHighlighting: [], needsLine: false)

    let id = UUID()

    public var didChange = true
    public var isGraph = false
    public var inputs: [Input] = []
    public var outputLabel: String? = nil
    public var graphLabel: String? = nil
    public var statement: (any Evaluatable)?
    public let exportedVariable: VariableIdentifier?
    public let usedVariables: [VariableIdentifier]
    public let syntaxHighlighting: [SyntaxHighlighting]
    public let needsLine: Bool
    public nonisolated(unsafe) weak var statementLine: StatementLine?

}

```

```

extension ParsedStatement: Hashable {

    public func hash(into hasher: inout Hasher) {
        hasher.combine(id)
        hasher.combine(exportedVariable)
        hasher.combine(usedVariables)
    }

}

extension ParsedStatement: Equatable {

    public static func == (lhs: ParsedStatement, rhs: ParsedStatement) -> Bool {
        lhs.hashValue == rhs.hashValue
    }

}

public class Parser {

    private(set) var iterator: PeekIterator<TokenPackager>!

    let programContext: ProgramContext

    public init(programContext: ProgramContext) {
        self.programContext = programContext
    }

    public func parse(lines: [StatementLine]) -> [ParsedStatement] {
        os_signpost(.begin, log: logger, name: "Parse lines")
        defer {
            os_signpost(.end, log: logger, name: "Parse lines")
        }
        var result: [ParsedStatement] = []
        for line in lines {
            autoreleasepool {
                if var ast = line.ast {
                    ast.didChange = false
                    result.append(ast)
                } else if let statement = parse(line: line) {
                    line.ast = statement
                    result.append(statement)
                }
            }
        }
    }
}

```

```

    }
  }
  return result
}

}

private extension Parser {

  func parse(line: StatementLine) -> ParsedStatement? {
    initialize(input: line.textContainer.text.value)

    guard let firstToken = iterator?.currentValue else {
      line.needsLine = false
      return nil
    }

    let parsedStatement = parseStatement(token: firstToken, line: line)
    line.needsLine = parsedStatement.needsLine

    if parsedStatement.statement != nil {
      if let exportedIdentifier = parsedStatement.exportedVariable {
        programContext.registerVariable(for: exportedIdentifier, source: line)
      }
    }
    return parsedStatement
  }

  func parseStatement(token: Token, line: StatementLine) -> ParsedStatement {
    let position = TextPosition(location: line.textContainer.text.location +
      token.position.location,
      length: token.position.length)
    switch token.kind {
    case .operator(.special(.graph)):
      return graphStatement(operatorPosition: position, line: line)
    case .operator(.special(.output)):
      return outputStatement(operatorPositon: position, line: line)
    case .literal(.string(let literal)):
      return possibleVariableDeclaration(literal: literal,
        position: position,
        line: line)
    default:
      guard let iterator else { fatalError() }

```

```

        let parser = ExpressionParser(iterator: iterator, programContext:
programContext)
        return parser.parse(line: line)
    }
}

}

private extension Parser {

    func graphStatement(operatorPosition: TextPosition, line: StatementLine) ->
ParsedStatement {
        nextToken() // skip "graph"

        let graphSyntax = SyntaxHighlighting(style: .function, position:
operatorPosition)

        guard let nextToken = iterator.currentValue else {
            return .init(exportedVariable: nil,
                usedVariables: [],
                syntaxHighlighting: [graphSyntax],
                needsLine: false)
        }

        let statement = parseStatement(token: nextToken, line: line)

        guard let evaluable = statement.statement else { return statement }

        let syntaxHighlighting = [graphSyntax] + statement.syntaxHighlighting

        let label: String? = if let letStatement = evaluable as? LetStatement {
            letStatement.name.value
        } else if let variable = evaluable as? VariableTerm {
            variable.identifier.value
        } else {
            nil
        }

        return .init(isGraph: true,
            inputs: statement.inputs,
            outputLabel: statement.outputLabel,
            graphLabel: label,
            statement: GraphStatement(value: evaluable),
            exportedVariable: statement.exportedVariable,

```

```

        usedVariables: statement.usedVariables,
        syntaxHighlighting: syntaxHighlighting,
        needsLine: statement.needsLine,
        statementLine: line)
    }

func outputStatement(operatorPositon: TextPosition, line: StatementLine) ->
ParsedStatement {
    nextToken() // skip "output"

    let operatorSyntax = SyntaxHighlighting(style: .function, position:
operatorPositon)

    guard let nextToken = iterator.currentValue else {
        return .init(exportedVariable: nil,
            usedVariables: [],
            syntaxHighlighting: [operatorSyntax],
            needsLine: false,
            statementLine: line)
    }

    let statement = parseStatement(token: nextToken, line: line)

    guard let evaluable = statement.statement else { return statement }

    let syntaxHighlighting = [operatorSyntax] + statement.syntaxHighlighting

    let label = if let letStatement = evaluable as? LetStatement {
        letStatement.name.value
    } else if let variable = evaluable as? VariableTerm {
        variable.identifier.value
    } else {
        "Output"
    }

    return .init(inputs: statement.inputs,
        outputLabel: label,
        statement: OutputStatement(label: label,
            value: evaluable),
        exportedVariable: statement.exportedVariable,
        usedVariables: statement.usedVariables,
        syntaxHighlighting: syntaxHighlighting,
        needsLine: statement.needsLine,
        statementLine: line)

```

```

}

func possibleVariableDeclaration(literal: String, position: TextPosition, line:
StatementLine) -> ParsedStatement {
    guard let equalsToken = iterator.peekValue, equalsToken.kind ==
.variableAssignment else {
        return ExpressionParser(iterator: iterator, programContext:
programContext).parse(line: line)
    }

    nextToken() // skip '='
    guard iterator.currentValue != nil else { return .empty }
    nextToken()

    let parsedStatement = ExpressionParser(iterator: iterator, programContext:
programContext).parse(line: line)
    let identifier = VariableIdentifier(stringLiteral: literal)

    if programContext.variableExists(name: identifier, at: line) {
        return parsedStatement
    }

    let variableSyntax = SyntaxHighlighting(style: .variable,
position: position)
    let equalsSyntax = SyntaxHighlighting(style: .equalsSign,
position: .init(location: line.textContainer.text.location
+ equalsToken.position.location,
length: equalsToken.position.length))
    if let statement = parsedStatement.statement {
        return .init(inputs: parsedStatement.inputs,
statement: LetStatement(name: identifier,
value: statement),
exportedVariable: identifier,
usedVariables: parsedStatement.usedVariables,
syntaxHighlighting: [variableSyntax, equalsSyntax] +
parsedStatement.syntaxHighlighting,
needsLine: parsedStatement.needsLine,
statementLine: line)
    }
    return .init(inputs: parsedStatement.inputs,
exportedVariable: nil,
usedVariables: [],
syntaxHighlighting: [.init(style: .default, position: position)],
needsLine: parsedStatement.needsLine,

```

```

        statementLine: line)
    }
}

extension Parser {

    func initialize(input: String) {
        let iterator = PeekIterator(iterator: Lexer(tokenizer: .init(input: input)))
        self.iterator = .init(iterator: .init(iterator: iterator))
        nextToken()
    }

    func nextToken() {
        iterator?.seekNext()
    }
}

```

					ІАЛЦ.467200.007 Д4	Арк.
						92
Зм.	Арк.	№ докум.	Підпис	Дата		