Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Інформаційно-аналітичних технологій та менеджменту
(повна назва)
Кафедра Інформатики
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти перший (бакалаврський)

ПРОЄКТУВАННЯ ТА ПРОГРАМНА РЕАЛІЗАЦІЯ ІНТЕГРОВАНОГО СЕРЕДОВИЩА РОЗРОБКИ ТА ІНСТРУМЕНТІВ ПРОГРАМУВАННЯ МАТЕМАТИЧНИХ ОБЧИСЛЕНЬ
(тема)

Виконав:
студент 4 курсу, групи ІТІНФ-19-2
Стешенко В.В.
(прізвище, ініціали)

Спеціальності 122 Комп'ютерні науки
(код і повна назва спеціальності)

Тип програми освітньо-професійна

Освітня програма Інформатика
(повна назва освітньої програми)

Керівник ст. викл. Кіношенко Д.К.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри _____ Кобилін О.А.
(підпис) (прізвище, ініціали)

2023 р.

Харківський національний університет радіоелектроніки

Факультет Інформаційно-аналітичних технологій та менеджменту
(повна назва)

Кафедра Інформатики
(повна назва)

Рівень вищої освіти перший (бакалаврський)

Спеціальність 122 Комп'ютерні науки
(код і повна назва)

Тип програми освітньо-професійна

Освітня програма Інформатика
(повна назва освітньої програми)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«____»_____2023 р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Стешенку Владиславу Віталійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Проєктування та програмна реалізація інтегрованого середовища розробки та інструментів програмування математичних обчислень

затверджена наказом по університету від 15 травня 2023 року № 474 Ст

2. Термін подання студентом роботи до екзаменаційної комісії 29 травня 2023 р.

3. Вихідні дані до роботи наукова-технічна та науково-методична література, матеріали конференцій, мова програмування Kotlin, середовище розробки Android Studio.

4. Перелік питань, що потрібно опрацювати в роботі

1. Аналіз існуючих мов програмування.
2. Моделювання мови програмування, алгоритмів та підходів.
3. Проєктування програмних інструментів.
4. Програмна реалізація спроєктованих інструментів.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри) Актуальність проблеми програмування математичних обчислень, постановка задачі.

6. Консультанти розділів роботи (п.6 включається до завдання за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата
Консультант з дотримання діючих стандартів та норм	Доцент Творошенко І.С.		

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	Отримання завдання на кваліфікаційну роботу	10.04.2023	
2	Аналіз завдання, підбір літератури	11.04.2023-14.04.2023	
3	Аналіз літератури з досліджуваної проблеми	15.04.2023-21.04.2023	
4	Моделювання мови програмування	22.04.2023-28.04.2023	
5	Проєктування програмних інструментів	29.04.2023-07.05.2023	
6	Програмна реалізація	08.05.2023-20.05.2023	
7	Оформлення пояснювальної записки	21.05.2023-27.05.2023	
8	Перевірка на плагіат	28.05.2023	
9	Рецензування	29.05.2023	
10	Підготовка презентації та доповіді	30.05.2023-06.06.2023	
11	Занесення роботи в електронний архів	07.06.2023	
12	Попередній захист кваліфікаційної роботи	07.06.2023	

Дата видачі завдання 10 квітня 2023 р.

Студент _____
(підпис)

Керівник роботи _____ ст. викл. Кіношенко Д.К.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ/ABSTRACT

Пояснювальна записка до кваліфікаційної роботи: 65 с., 56 рис., 30 джерел.

МОВА ПРОГРАМУВАННЯ, ЛЕКСИЧНИЙ АНАЛІЗ, СИНТАКСИЧНИЙ АНАЛІЗ, СЕМАНТИЧНИЙ АНАЛІЗ, КОМПІЛЯТОР, ВІРТУАЛЬНА МАШИНА, МАТЕМАТИЧНІ ОБЧИСЛЕННЯ.

Об’єкт роботи – задача програмування математичних обчислень.

Мета роботи – проєктування і програмна реалізація принципів, підходів, алгоритмів та інструментів, що дозволять вирішувати інформаційні задачі програмування математичних обчислень з використанням загальноприйнятого запису математичних виразів.

В межах кваліфікаційної роботи було проведено аналіз предметної області, спроєктовано та програмно реалізовано мову програмування, інструменти лексичного, синтаксичного і семантичного аналізів, компілятор, віртуальну машину та інтегроване середовище розробки.

PROGRAMMING LANGUAGE, LEXICAL ANALYSIS, SYNTACTIC ANALYSIS, SEMANTIC ANALYSIS, COMPILER, VIRTUAL MACHINE, MATHEMATICAL COMPUTATIONS.

The object of work is the task of programming mathematical computations.

The aim of the work is to design and implement principles, approaches, algorithms, and tools that allow solving information tasks of programming mathematical computations using a commonly accepted notation for mathematical expressions.

Within the scope of the qualification work, an analysis of the subject area was conducted, and a programming language, lexical, syntactic, and semantic analysis tools, a compiler, a virtual machine, and an integrated development environment were designed and implemented.

ЗМІСТ

ВСТУП

Невпинний розвиток науки і техніки постійно збільшував вимоги до існуючих засобів обробки інформації. Складність інформаційних задач підвищувалась, а їх кількість зростала, що, в свою чергу, стимулювало наукові дослідження та технічні розробки в області автоматизації обчислень.

Перші електронно-обчислювальні машини були системами електронно-механічними компонентів, які дозволили автоматизувати виконання типових інформаційних задач. Але вони були дуже складними в обслуговуванні через велику кількість механізації, а швидкість виконання ними арифметичних операцій була низькою [1, 2]. Недоліки в роботі техніки, а також низька зручність її програмування, рухали наукову думку у напрямку зменшення механічної складової в обчислювальній техніці. Наступним етапом розвитку була поява електронно-обчислювальної техніки на електровакуумних лампах. Це дозволило прибрати механічну складову, але актуальними лишалися багато проблем і недоліків, наприклад, складність обслуговування техніки, високі вимоги до енергоживлення та, все ще, низька швидкість обчислень [2].

Революцію в електронно-обчислювальній техніці створила поява транзисторів. Транзистори мають менші вимоги до електроживлення, розміри, тепловиділення і головне – значно надійніші. З часом, вартість виготовлення транзисторів стала низькою, а їх розповсюдженість дозволила надати доступ до автоматизації інформаційних задач людям за межами науково-технічного й інженерного прошарків суспільства [3–5].

Розвиток електронно-обчислювальної техніки стимулював розвиток інженерії програмного забезпечення. Мови програмування, як інструменти керування технікою, розвивалися паралельно з нею. Техніка ставала швидшою, мови програмування ставали більш абстрактними і менше орієнтованими на ефективність обчислень. Техніка ставала розповсюдженою

серед більшої кількості людей, мови програмування, в свою чергу, ставали простішими для використання людьми без профільної освіти чи інженерного досвіду.

Мови програмування пройшли довгий шлях від появи до свого сучасного вигляду, шлях покращення існуючих можливостей та створення нових. Інженерія програмного забезпечення розробляла підходи, практики та парадигми [6–8], які впливали на розвиток мов програмування. Але більшість мов програмування була створена не в процесі аналізу та доопрацювання існуючого досвіду інженерії, а шляхом поєднання можливостей вже існуючих мов чи додавання якоїсь бажаної можливості до конкретної мови. Це створило серйозну технічну кризу, яка гальмує темпи розвитку та зводить інженерію до вирішення проблем в інструментах, а не вирішення актуальних інформаційних задач [9, 10].

Актуальність роботи полягає у необхідності перегляду та дослідження проблеми зручності запису рішення інформаційних алгоритмічних задач у вигляді програмного коду. Ця проблема розглядається у кваліфікаційній роботі на прикладі задачі програмування математичних обчислень [11–15].

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Загальний огляд мов програмування

Мова програмування – формальна мова, що має лексичні, синтаксичні та семантичні правила, які визначають можливості її використання та вимоги до її програмного коду [16]. Мови програмування використовуються як інструменти алгоритмізації та автоматизації виконання інформаційних задач за допомогою програмованої електронно-обчислювальної техніки. На даний момент існує багато мов програмування загального та спеціального призначення, всі вони можуть бути класифіковані за такими, основними для розробки програмного забезпечення, критеріями:

– належність до певної парадигми програмування;

– рівень абстракції програмного коду [7, 8];

– тип виконання програмного коду.

Парадигма мови програмування – це загальна концепція, що визначає основні підходи до проєктування та реалізації програмного забезпечення з використанням певної мови програмування.

Рівень абстракції програмного коду – це загальноприйнятий рівень того, наскільки деталі послідовностей команд виконуваних програмою приховані від їх виклику в програмному коді.

Тип виконання програмного коду – це спосіб яким програмний код обробляється та виконується електронно-обчислювальною технікою.

1.2 Загальний огляд розробки мов програмування

Розробка мови програмування – це комплексний інженерний процес, що складається з послідовності задач на проєктування і програмну реалізації

мови програмування. Таким чином, процес розробки мови програмування складається з таких етапів:

– аналіз проблем та вимог;

– розробка лексики, синтаксису та семантики;

– реалізація інструментів аналізу, обробки та виконання програмного коду.

Процес розробки мови програмування вимагає досвіду в інженерії програмного забезпечення, знань в області теорії мов програмування і пов’язаних з нею напрямках математики та інформатики [16, 17]. Етапи аналізу і теоретичної розробки потребують високого рівня розуміння існуючих напрацювань та підходів, а також спираються на сформоване досвідом програмної інженерії бачення розробників. Крім того, реалізація інструментів аналізу, обробки і виконання програмного коду потребує досвіду в області проєктування та програмної реалізації структур даних і алгоритмів різних рівнів складності.

1.3 Загальний огляд інструментів обробки програмного коду

Інструменти обробки програмного коду – це програмні засоби, що дозволяють перетворити програмний код певної мови програмування у послідовність інструкцій для виконання електронно-обчислювальною машиною. В загальному вигляді, дані інструменти є послідовними у використанні і охоплюють етапи лексичного, синтаксичного, семантичного аналізів, включаючи подальшу трансляцію результатів аналізу в можливі для виконання конструкції.

Інструменти лексичного аналізу використовуються для первинної обробки послідовності символів програмного коду. Лексичний аналіз полягає у визначенні сенсу послідовностей символів, тобто визначенні лексем програмного коду. На наступному етапі обробки послідовність отриманих

лексем підлягає синтаксичному аналізу.

Інструменти синтаксичного аналізу використовуються для отримання структури синтаксичних конструкцій. Синтаксичний аналіз проводиться над послідовністю лексем, а сам процес аналізу полягає у пошуку синтаксичних правил відповідних заданій послідовності. У випадку успішного визначення відповідних правил, послідовність лексем перетворюється на конструкцію, що структуровано відображає програмний код та може використовуватись для семантичного аналізу і подальшої трансляції.

Інструменти семантичного аналізу використовуються для перевірки відповідності програмного коду специфікаціям мови програмування. Зазвичай, семантичний аналіз проводиться над структурою, отриманою на етапі синтаксичного аналізу, та не вносить до неї жодних змін. В його процесі можуть перевірятися коректності областей видимості та відповідність програмного коду системі типів, а також його теоретична обчислюваність.

Інструменти трансляції програмного коду перетворюють структуру коду на виконувану послідовність інструкцій для визначеної електронно-обчислювальної машини. Це може бути компіляцією у послідовність інструкцій машинного коду певної апаратної архітектури, а може бути перетворенням структур у формат, зрозумілий певній архітектурі віртуальних програмних машин для подальшого виконання, ізольовано від апаратного забезпечення.

1.4 Загальний огляд інструментів виконання програмного коду

Інструменти виконання програмного коду використовуються для отримання результату роботи алгоритмів описаних програмним кодом. Цим результатом можуть бути вихідні дані або вплив на зовнішнє середовище та оточення програмного застосунку. Виконання може бути порційним під час аналізу чи повним після аналізу. Перший випадок є класичною

інтерпретацією програмних мов, наприклад, виконання аналізу програмного коду по одному твердженню та виконання їх одразу при успішному результаті аналізу. Другий випадок зазвичай є трансляцією програмного коду в програмний код іншої формальної мови. Це може бути трансляція на іншу мову програмування, трансляція програмного коду у програмні структури для віртуальних машин чи компіляція програмного коду в машинний код апаратної обчислювальної архітектури, адже машинний код також є випадком формальної мови.

Незалежно від того, електронно-обчислювальна машина є програмною чи апаратною – загальноприйнятою архітектурою є розділення програмного забезпечення на операції і операнди з подальшим виділенням двох систем пам’яті [2]. В такому випадку, абстрактний обчислювач отримує інструкції з системи пам’яті операцій і виконує завантажені інструкції над даними, отриманими з системи пам’яті операндів. Порційне чи повне виконання програмного коду є завантаженням програмних структур у системи пам’яті та ініціюючим викликом абстрактного обчислювача.

1.5 Загальний огляд інтегрованих середовищ розробки

Інструменти та засоби введення програмного коду, інструменти його подальшої обробки та виконання у своїх програмних реалізаціях є доволі різними за використанням і своєю поведінкою. Використання цих програмних застосунків може бути складним для користувача і може бути автоматизовано шляхом приховування деталей їх реалізації та взаємодії у більш комплексному програмному застосунку. Ззовні такий програмний застосунок може виглядати як прямий шлях від введення програмного коду до отримання результатів його виконання зі зручним і зрозумілим користувачу керуванням, яке в свою чергу може бути обгорткою над складним програмним інтерфейсом внутрішніх інструментів. Такий

застосунок є інтегрованим середовищем розробки, яке дозволяє розробляти програмне забезпечення використовуючи меншу кількість відокремлених інструментів з простішим інтерфейсом взаємодії [7].

1.6 Аналіз вимог до мови програмування

Мова програмування, яка використовується як інструмент програмування математичних обчислень, може бути спроєктована орієнтуючись на різні парадигми і підходи в програмуванні. Будь-яку з сучасних, повних за Тьюрингом, мов програмування можна використовувати як інструмент у математичних розрахунках, але в більшості випадків це не буде зручним.

В основній своїй масі, сучасні мови програмування є інструментами для роботи зі складними, багатошаровими абстрактними сутностями, важливим для них є структурованість і виразність обмеженого числа визначених розробниками мови мовних конструкцій [7].

Наприклад, класичним представником об’єктно-орієнтованих мов є мова програмування Java. Вона дозволяє реалізовувати складні програмні комплекси, але страждає на багатослівність, а її синтаксис та семантика незручні для запису математичних виразів. Примітивних типів, їх логіки та операторів недостатньо для задач складніших за додавання двох чисел, а в усіх інших випадках мова змушує працювати з об’єктними сутностями, оперуючи ними з використанням описаних для них методів [18]. Для реалізації обчислення факторіалу мовою Java необхідно мати клас, в ньому реалізувати статичний метод і потім викликати його як функцію від числа. По-перше, необхідність описувати класи та методи є завеликим ускладненням для реалізації функції факторіалу. По-друге, синтаксис виразу для обчислення факторіалу не буде мати нічого спільного зі звичним у математиці визначенням через використання знаку *!*.

Лістинг 1.1 Реалізація функції факторіалу мовою Java:

```
public class Factorial {


   public static void main(String[] args) {

      int number = 7;
      System.out.println(factorial(number));
   }

   public static int factorial(int n) {

      int accumulated = 1;
      for (int i = 1; i <= n; i++) {
         accumulated *= i;
      }
      return accumulated;
   }
}
```

Аналогічні проблеми є не менш виразними у випадку використання функціональних мов програмування. Не дивлячись на те, що концептуально мови функціональної парадигми максимально наближені до математичних ідей, вони не кращі для запису математичних виразів. Наприклад, мова програмування Haskell [19] дозволяє описати функцію для обчислення факторіалу, але для відповідності його ітеративному визначенню необхідно використовувати хвостову рекурсію, таким чином, програмний код стає багатослівним і заплутаним, а виклик функції факторіалу у виразі все ще не відповідає загальноприйнятому та очікуваному *!*.

Лістинг 1.2 Реалізація факторіалу мовою Haskell:

```
factorial :: Integer -> Integer
factorial n = go 1 n
   where
      go accumulated 0 = accumulated
      go accumulated n = go (accumulated * n) (n - 1)

main :: IO ()
main = do
   let n = 7
      result = factorial n
   putStrLn $ show result
```

Мова програмування, що використовується як інструмент запису математичних виразів, повинна мати доволі просту систему типів [20] і модель обчислення, які не будуть обтяжувати програмний код багатослівністю і складною структурою. Крім того, вона обов'язково повинна мати інструменти для побудови власних синтаксичних конструкції, відповідних виклику функцій у виразах, що дозволить використовувати функцію факторіалу як *n!*, а не *factorial(n)* чи *factorial n*.

Якщо розглядати поняття функції у розумінні, звичному для інженерів та розробників програмного забезпечення, а не спеціалістів у математиці, то під функцією зазвичай прийнято розуміти іменовану ідентифікатором послідовність тверджень, кожне наступне з яких спирається або на параметри функції, або на одне з попередніх тверджень. При цьому, в більшості мов програмування, функція може або змінювати стан екземпляру оточення, в якому вона описана, або повертати певний її результат від аргументів, переданих за параметрами, або робити і те, і інше. Якщо ж розглядати поняття функції у розумінні більш наближеному до математики, то функція є

інструментом відображенням множини значень екземплярів аргументів її параметрів у множину значень екземплярів даних типу, який ця функція повертає [21]. В такому випадку функція не змінює стан екземпляру її оточення, вона працює лише зі своїми аргументами, а будь-яка складна функція є лише композицією викликів простих функцій.

Для мови програмування, яка має використовуватись в якості інструмента програмування математичних обчислень, можна зробити такі висновки:

– одиницею програмного коду є функція, концепція функції мови програмування повинна бути наближена в своїй реалізації до концепції функції в математиці [19];

– функція завжди повинна повертати результат своїх внутрішніх обчислень, у випадку якщо функція повертає екземпляр певного базового типу, тоді цей тип можна не вказувати у оголошенні функції;

– функція завжди повинна бути композицією інших функцій, тобто по своїй суті функція є іменованим обчислюваним виразом, який використовує аргументи зі свого оточення;

– система типів повинна будуватись на кількох основних для математичних обчислень типах даних, складні типи мають будуватись як поєднання основних [20];

– як і для будь-якої іншої мови програмування, програмний код повинен мати початкову функцію, результат виконання якої і є результатом всіх обчислень [16].

1.7 Постановка задачі

Таким чином, розробка мови програмування математичних обчислень є актуальним завданням з причин необхідності перегляду та дослідження проблеми зручності запису рішень інформаційних алгоритмічних задач у

вигляді програмного коду.

Об’єктом роботи є задача програмування математичних обчислень.

Метою роботи є проєктування і програмна реалізація принципів, підходів, алгоритмів та інструментів, що дозволять вирішувати інформаційні задачі програмування математичних обчислень з використанням загальноприйнятого запису математичних виразів.

Для досягнення мети необхідно вирішити такі завдання:

– провести аналіз предметної області;

– провести дослідження проблем та недоліків мов програмування;

– розробити підходи, принципи і алгоритми які дозволять використовувати нетиповий синтаксис виразів мови програмування;

– провести моделювання мови програмування;

– розробити підходи, принципи і алгоритми які дозволять використовувати нетиповий синтаксис виразів мови програмування;

– провести проектування інструментів лексичного, синтаксичного та семантичного аналізів;

– провести проектування віртуальної машини, компілятора та інтегрованого середовища розробки.

– програмно реалізувати розроблені підходи, принципи і алгоритми, а також спроектовані програмі інструменти.

2 МОДЕЛЮВАННЯ МОВИ ПРОГРАМУВАННЯ, АЛГОРИТМІВ, ПІДХОДІВ І ПРИНЦИПІВ РОБОТИ

2.1 Моделювання мови програмування

Базовим типом для інструменту програмування математичних обчислень може бути числовий тип, а ідентифікатором точки входу послідовність символів main. Початкова функція повинна не мати параметрів виклику. В такому випадку оголошення та визначення початкової функції може мати максимально скорочений вигляд.

Лістинг 2.1 Приклад оголошення та визначення функції main:

```
#main => ‘2 + 2’
```

Таке оголошення функції, з визначенням у вигляді короткої форми, може бути незручним у випадку високої складності виразу, з яким функція пов’язується. Тоді мова програмування повинна мати оператор композиції, який дозволить покроково описувати функцію, виражаючи її через послідовність виразів з викликами інших функцій.

Лістинг 2.2 Приклад оголошення та визначення функції main:

```
#main => {
  @main <- ‘2’
  @main <- ‘@main + 2’
}
```

Використовуючи мову програмування, не можна програмно реалізувати функції, які не є композицією інших функцій. Це означає, що мова програмування повинна мати вбудовані функції, реалізація яких прихована і

залежить від засобів подальшої обробки та виконання програмного коду. Оголошення та визначення вбудованих функцій може бути виконане з використанням ключового слова *builtIn*.

Лістинг 2.3 Приклад оголошення та визначення вбудованої функції:

```
#add: (a, b) => builtIn { }
```

Функція add оголошена як функція від двох параметрів числового типу, a і b. Тип який вона повертає не вказується явно, тобто це є базовий числовий тип. Її реалізація визначається як вбудована. Функція add, описана у такому вигляді, не надає жодної інформації щодо синтаксичної конструкції, яка відповідає її виклику, а значить використовувати її можна лише як виклик функції у виразах більшості мов програмування.

Лістинг 2.4 Приклад використання вбудованої функції:

```
#add: (a, b) => builtIn { }
#main => ‘add(2, 2)’
```

Для того щоб функцію *add* можна було використовувати зі звичним для нас записом у вигляді символу +, її оголошення потрібно пов’язати з виразом від її параметрів, який в подальшому буде виступати в ролі патерну для засобів обробки та виконанання програмного коду. Мова програмування повинна мати конструкцію для перевизначення виразу функції із синтаксисом, подібним до синтаксису виразу звичайних математичних обчислень.

Лістинг 2.5 Приклад перевизначення синтаксису виклику функції:

```
#add: (a, b) = ‘a + b’ => builtIn { }
#main => ‘2 + 2’
```

В такому випадку вираз функції *main* порівнюється з виразами описаних функції і визначається як виклик функції *add*, де перша двійка передається як аргумент параметру *a*, а друга двійка передається як аргумент параметру *b*.

Для мови програмування, яка проєктується як інструмент програмування саме математичних обчислень, оптимальним рішенням є вибір конкатенації для оголошення та визначення типів ідентифікаторів. Такий підхід дозволяє зручно вказувати типи, при цьому не додає багатослівності та зайвої структуризації, яка для програмного забезпечення такого розміру не потрібна.

Оголошення функції без оголошення параметрів повинне визначати, що функція не приймає аргументи, а результатом її роботи є екземпляр числового типу.

Лістинг 2.6 Приклад оголошення типу функції:

```
#function => ‘0’
```

Оголошення функції з оголошенням параметрів, але без оголошення типу, повинне визначати, що функція приймає аргументи, а результатом її роботи є екземпляр числового типу.

Лістинг 2.7 Приклад оголошення типу функції:

```
#function: (a, b) => ‘a + b’
```

Оголошення функції з оголошенням параметрів, які є функціями без параметрів, повинне визначати, що функція приймає функції, а результатом її роботи є екземпляр числового типу.

Лістинг 2.8 Приклад оголошення типу функції:

#function: (a(), b()) => ‘a() + b()’

Функція повинна мати можливість приймати аргументи, параметри яких є числами, функціями, масивами, матрицями або множинами:

Лістинг 2.9 Приклад оголошення типу функції:

#function: (a, b(n), c[10], d[3, 3], e{}) => ‘...’

Функція повинна мати можливість повертати як результат своєї роботи екземпляри чисел, функцій, масивів, матриць або множин:

Лістинг 2.10 Приклад оголошення типу функції:

#function1: () => ‘...’

#function2: ()(n) => ‘...’

#function3: ()[10] => ‘...’

#function4: ()[3, 3] => ‘...’

#function5: (){} => ‘...’

Функція повинна мати можливість приймати і повертати екземпляри складних поєднаних типів, наприклад, приймати матрицю функцій від параметру і розміру 5×5 та матрицю чисел розміру 5×5, а як результат своєї роботи повертати множину функцій від параметру n типу матриці чисел розміру 3×3, які повертають матрицю 10×10 масивів функцій від параметрів *a* та *b* розміру 100.

Лістинг 2.11 Приклад оголошення типу функції:

```
#function: (f[5, 5](i), m[5, 5]){}(n[3, 3])[10, 10][100](a, b) => '...'
```

Функція повинна мати можливість оголошувати та визначати функції всередині себе і використовувати їх в подальшому, у виразах, тобто передавати як аргументи в інші функції, а також повертати як результат своєї роботи.

Лістинг 2.12 Приклад оголошення типу функції:

```
#function: ()() => {
        #f => '5'
        @function <- 'f'
}

#main => {
   #f => 'function()'
   @main <- 'f()'
}
```

Окрім того, функцію може бути визначено анонімно у виразі без оголошення ідентифікатора.

Лістинг 2.13 Приклад анонімного оголошення та визначення функції:

```
#main => {
   #f: ()(i) => '{ i -> i + i }'
   @main <- 'f()(5)'
}
```

2.2 Лексичний аналіз програмного коду

З опису вимог до можливостей мови програмування та наведених прикладів її синтаксису необхідно розробити модель її лексики та підхід до реалізація інструментів лексичного аналізу.

Програмний код, який в подальшому обробляється інструментами мови програмування, є послідовністю символів визначеного мовою алфавіту. Зазвичай, множина символів цього алфавіту (рис. 2.1) невелика і обмежена стандартним набором символів для представлення слів латинської мови, чисел та набором спеціальних символів які виступають в ролі операторів, роздільників. Крім того, програмний код, задля легкості читання людиною, має бути чітко структурованим і поділеним на блоки, тому алфавіт мови включає в себе символи форматування, такі як пробіли, табуляції та переноси строк.

В результаті лексичного аналізу програмного коду, за допомогою інструментів мови програмування, виділяється послідовність лексем, що спрощує подальшу обробку тексту [17]. Послідовність лексем в подальшому передається до інструментів синтаксичного аналізу. Тобто, на етапі лексичного аналізу, програмний код повинен бути чітко поділеним на лексеми та не мати символів без визначених лексем чи лексем, в яких перетинаються множини індексів символів.

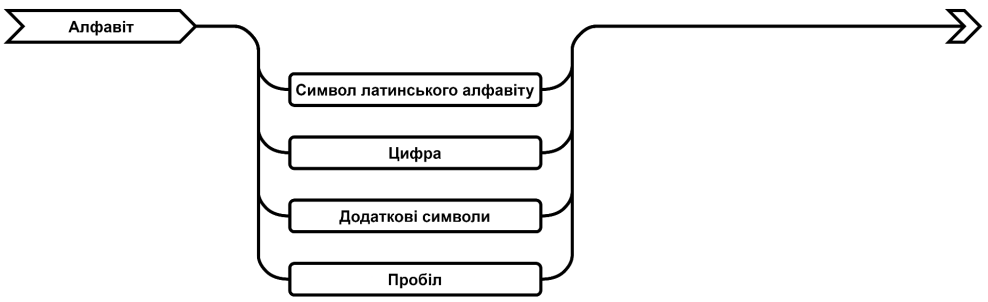

Рисунок 2.1 – Синтаксична діаграма алфавіту

Виходячи з того, що головною метою розробки мови програмування є покращення запису математичних виразів та створення можливостей для використання будь-якого синтаксису роботи з функціями, схема розділення лексем на групи та одиниці відрізняється від прийнятої зазвичай (рис. 2.2). Наприклад, математичний вираз цілком виділяється як окрема лексема для подальшої обробки аналізатором виразів, адже на етапі лексичного аналізу відсутня інформація щодо існуючих патернів функцій, а їх порівняння сильно підвищує структурну та алгоритмічну складності лексичного аналізу.

Загалом, процес лексичного аналізу можна представити як задачу виділення лексем п’яти основних типів:

– ідентифікатори;

– оператори;

– роздільники;

– вирази;

– пробіли.

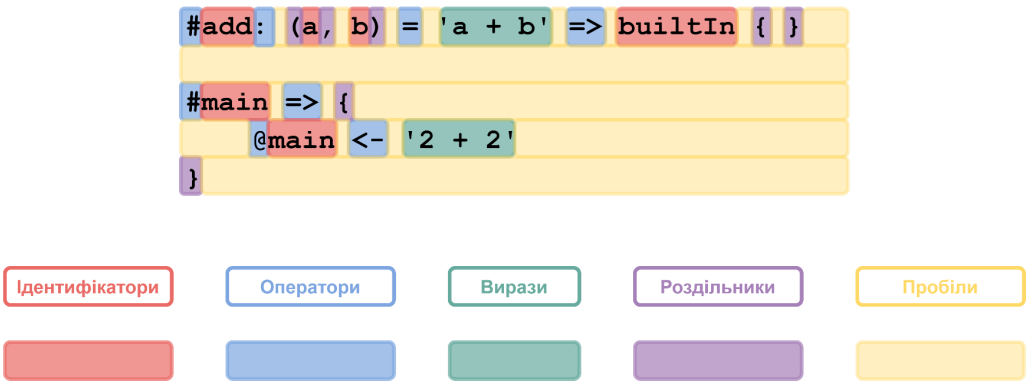

Рисунок 2.2 – Групи лексем на прикладі програмного коду

В групу лексем ідентифікатори входить лише одна лексична одиниця, сам ідентифікатор. Ідентифікатор виступає в ролі імен для функцій, параметрів та їх аргументів. Ідентифікатор повинен мати ненульову довжину та складатись із символів які входять в множину символів слів латинської

мови, в множину символів цифр, або в одиничну множину символу нижнього підкреслення (рис. 2.3).

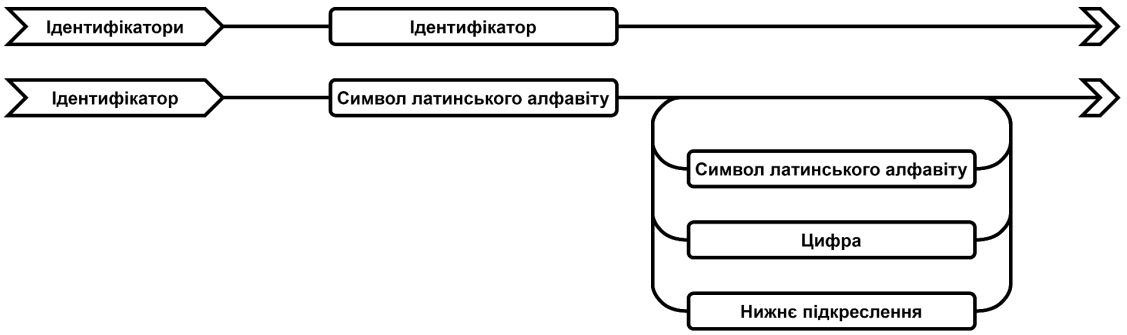

Рисунок 2.3 – Синтаксична діаграма групи лексем ідентифікатори

В групу лексем оператори (рис. 2.4) входить кілька лексичних одиниць, кожна з яких представляє собою один із операторів мови програмування:

– оголошення;

– параметризації;

– перевизначення;

– визначення;

– посилання;

– композиції.

В групу лексем роздільники входить кілька лексичних одиниць, кожна з яких використовується для розділення лексичних одиниць інших груп. Це дає можливість створити логічний контекст, що визначає різні синтаксис та семантику для одних лексем, а також підвищити легкість сприйняття програмного коду людиною. В групу входять такі роздільники:

– відкриття параметрів;

– закриття параметрів;

– кома;

– відкриття вбудованої структури;

– закриття вбудованої структури;

– відкриття блоку;

– закриття блоку (рис. 2.5).

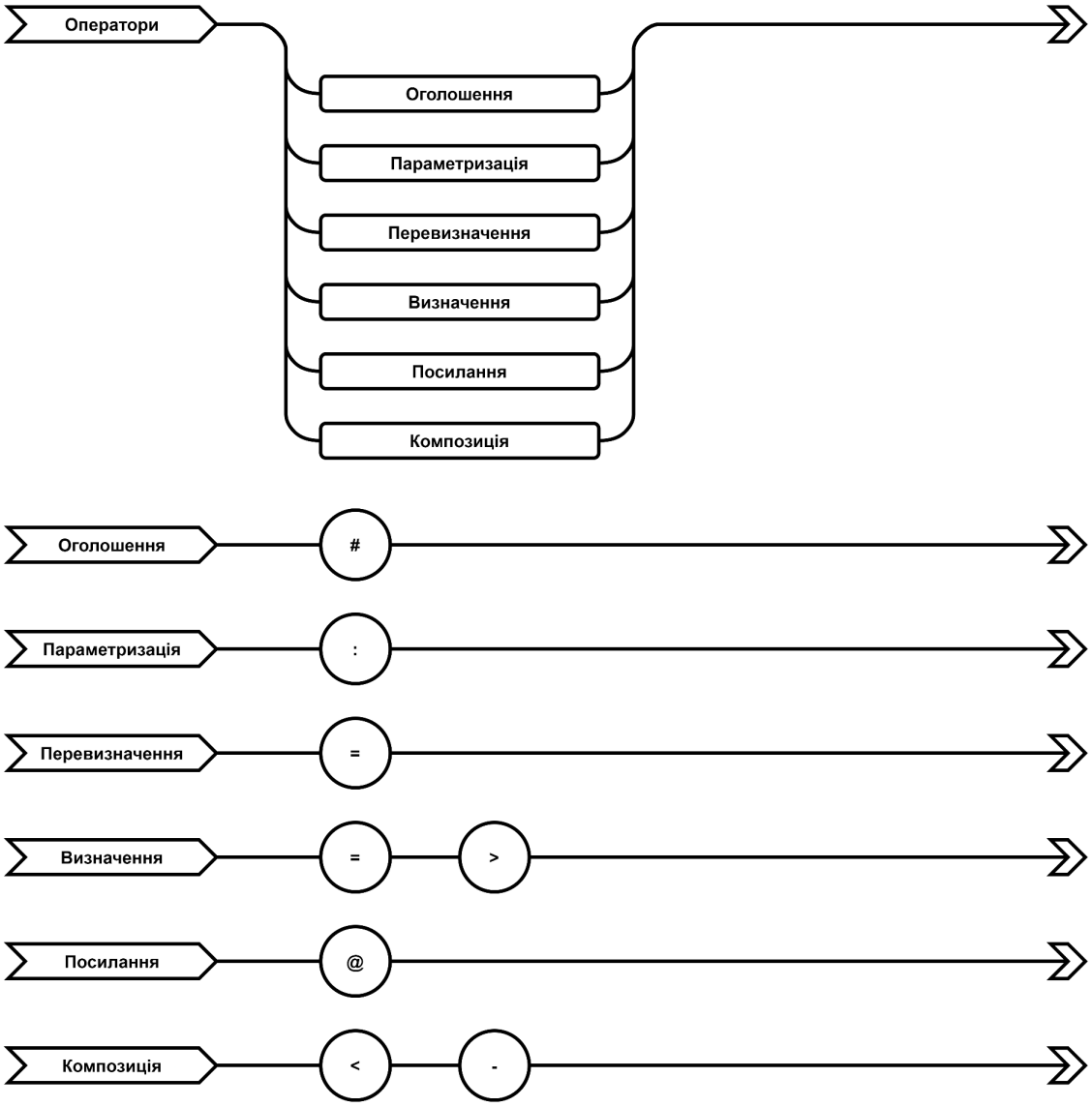

Рисунок 2.4 – Синтаксична діаграма групи лексем оператори

Група лексем вирази є одиничною множиною лексеми вираз. Вираз повинен мати чітко окреслені початок і кінець, так як синтаксис патернів функцій може бути довільним, що створює проблеми для виділення його лексеми. Вираз починається та закінчується одинарними лапками, кількість символів між одинарними лапками повинна бути ненульовою. В подальшому,

частина виразу між лапками виділяється як його вміст і передається аналізатору виразів (рис. 2.6).

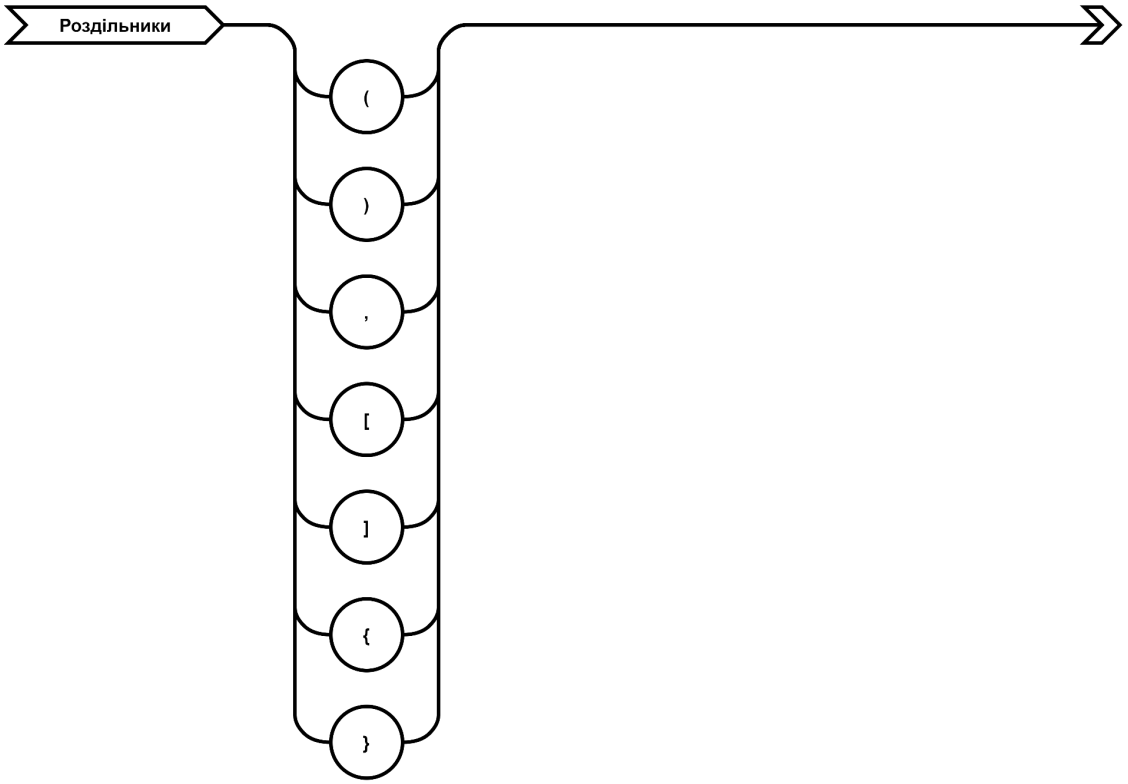

Рисунок 2.5 – Синтаксична діаграма групи лексем роздільники

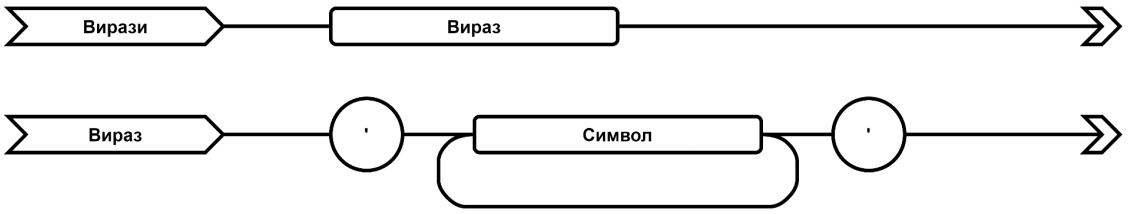

Рисунок 2.6 – Синтаксична діаграма групи лексем вирази

В групу лексем пробіли (рис. 2.7) входять лексеми, що використовуються для візуальної структуризації та розділення програмного коду, але жодним чином не впливають на його синтаксис та семантику. Це такі одиничні лексеми як:

– пробіл;

– відступ;

– перенос строки.

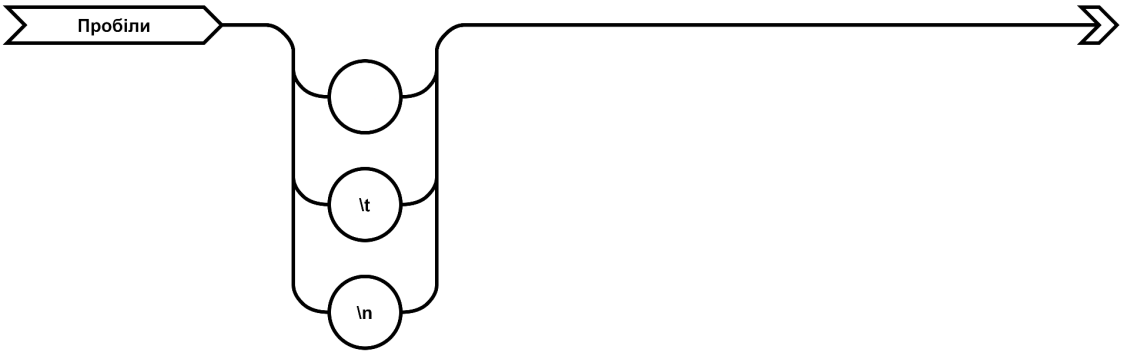

Рисунок 2.7 – Синтаксична діаграма групи лексем пробіли

Для розділення послідовності символів програмного коду на послідовність лексем необхідно спроєктувати і реалізувати інструменти лексичного аналізу (рис. 2.8). В умовах аналізу програмного коду можна зробити певні спрощення принципу роботи лексичного аналізатора, орієнтуючись на те, що послідовність символів програмного коду є кінцевою, а лексика мови програмування достатньо проста, що спрощує виділення лексем та їх оцінку [22].

Рисунок 2.8 – Загальна схема лексичного аналізатора

Примітивом будови лексичного аналізатора можна виділити аналізатор одиничної лексеми. Аналізатор одиничної лексеми це програмна сутність яка

обробляє послідовність символів програмного коду та виділяє в них лексеми одного типу (рис. 2.9).

Рисунок 2.9 – Загальна схема аналізатора одиничної лексеми

Таким чином, лексичний аналізатор може виступати обгорткою над множиною аналізаторів одиничних лексем (рис. 2.10), свого роду їх композицією. Лексичний аналізатор направляє послідовність символів до кожного з аналізаторів одиничних лексем та накоплює результати виділення лексем кожного типу для подальшої оцінки та формування послідовності вихідних лексем.

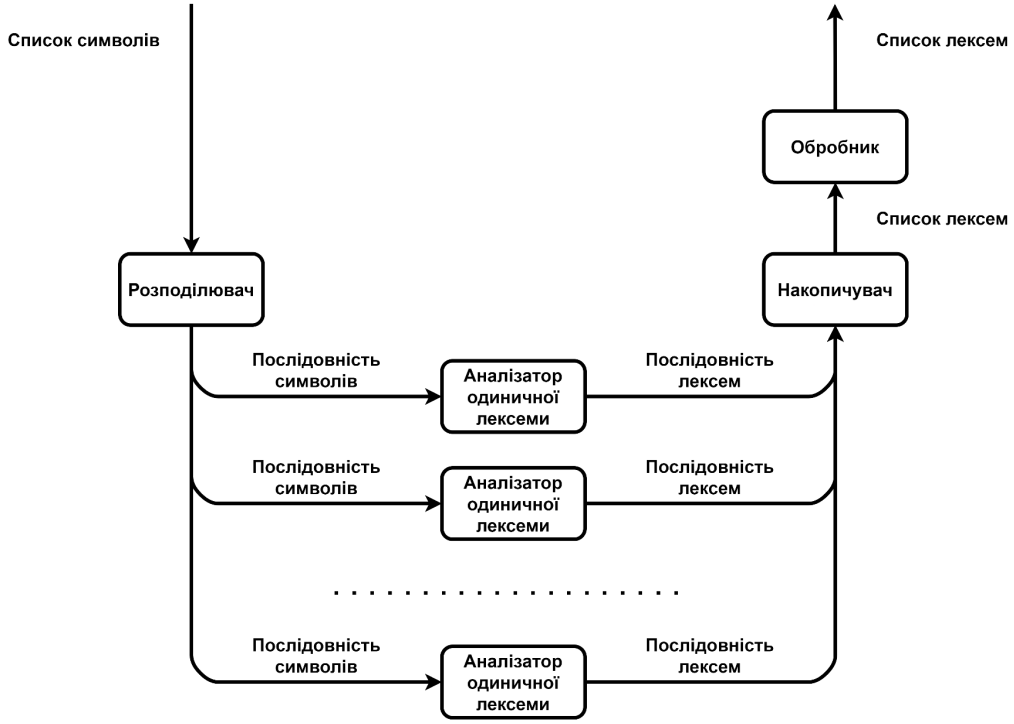

Рисунок 2.10 – Внутрішня схема лексичного аналізатора

В послідовності символів для кожного елементу можна виділити його положення у вигляді індексу, в такому випадку результати аналізу одиничної лексеми можна зафіксувати як послідовність символів визначеної довжини, починаючи з визначеного синтаксису. Структура даних отримана в результаті накопичення вихідних даних кількох аналізаторів одиничних лексем може бути візуалізована схемою, яка в горизонтальному напрямку має шкалу індексів, а у вертикальному напрямку мітки аналізаторів одиничних лексем (рис. 2.11).

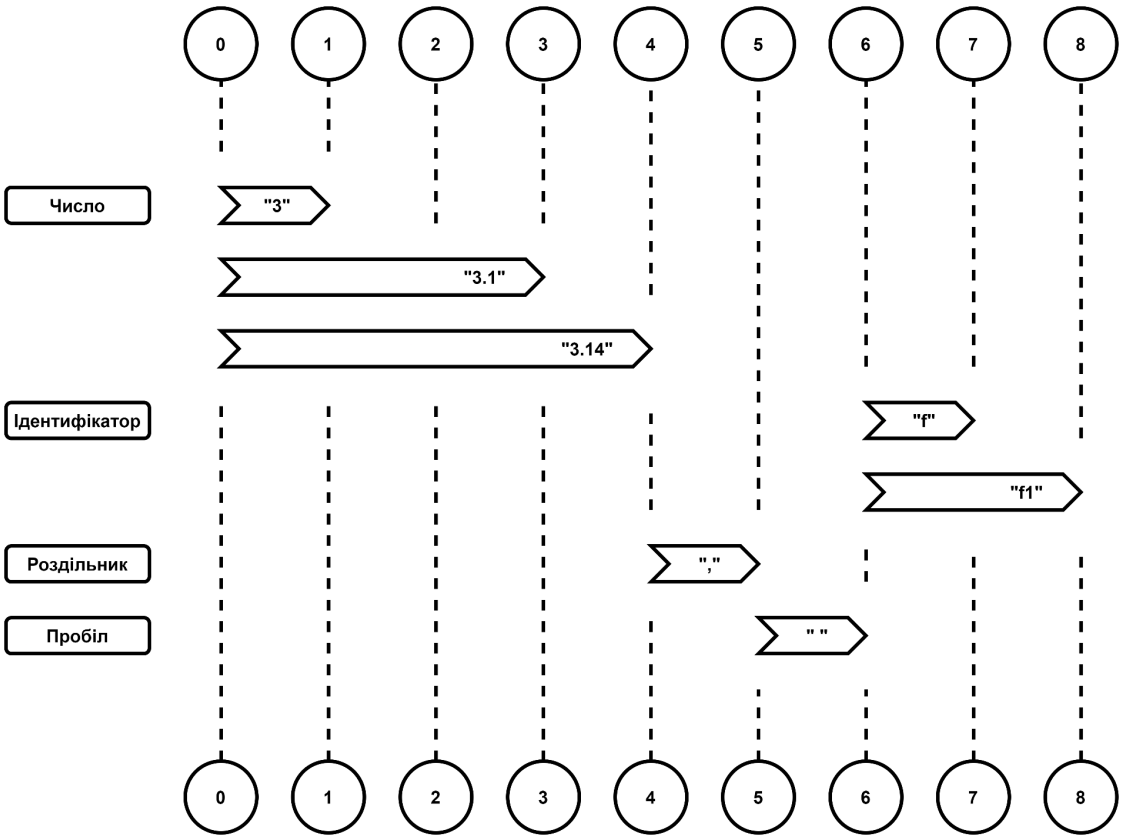

Рисунок 2.11 – Візуалізація списку лексем до оцінки

Отриманий список визначених лексем необхідно оцінити для формування вихідної послідовності лексичного аналізатору. В результаті оцінки повинен бути сформований список (рис. 2.12), одиничні лексеми в якому не перетинаються за множинами індексів своїх символів [18, 23]. Крім

того, об’єднана множина індексів всіх одиничних лексем після оцінки повинна включати в себе всі індекси символів вхідної послідовності [21].

В загальному вигляді, алгоритм оцінки списку одиничних лексем можна визначити як алгоритм пошуку неперервної послідовності найдовших лексем [21, 23].

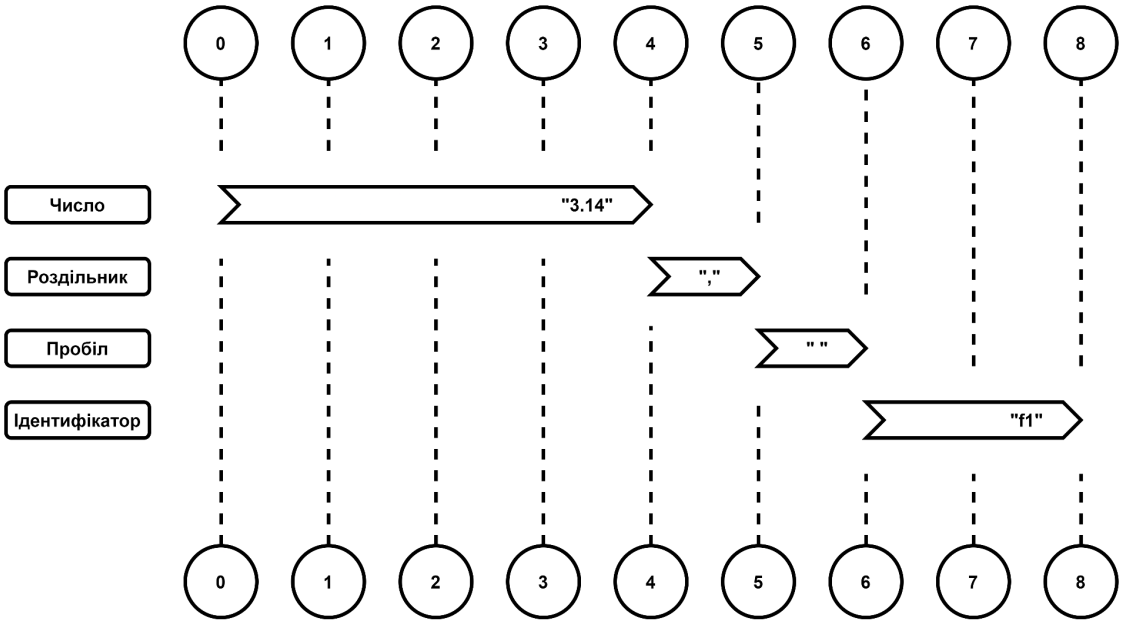

Рисунок 2.12 – Візуалізація списку лексем після оцінки

2.3 Синтаксичний аналіз програмного коду

Синтаксичний аналіз є важливим етапом аналізу програмного коду будь-якої мови програмування, вихідним результатом процесу синтаксичного аналізу є програмні сутності, зручні для їх подальшої обробки засобами програмування. Синтаксичний аналіз проводиться над послідовністю лексем, а на виході синтаксичного аналізатору отримують, зазвичай, дерево з усіма даними про програмний код [16, 17, 22, 24, 25].

Виходячи з того, що граматику мови програмування, яка проєктується,

можна представити як LL(1) граматику, вдалим рішення є використання алгоритму рекурсивного синтаксичного аналізу. В такому випадку синтаксичний аналізатор (рис. 2.13) проєктується як набір програмних сутностей, кожна з яких аналізує надану їй послідовність лексем та намагається співставити її з визначеним синтаксичним правилом [16, 22]. Для цього необхідно визначити синтаксичні правила мови програмування і описати алгоритм синтаксичного аналізу в загальному вигляді.

Рисунок 2.13 – Загальна схема синтаксичного аналізатора

Програмний код можна представити як послідовність оголошень та визначень функцій (рис. 2.14), при цьому відсутність опису функцій також має бути коректною ситуацію для синтаксичного аналізатора. Хоч семантика мови і вимагає наявність початкової функції, це не має значення на етапі синтаксичного аналізу.

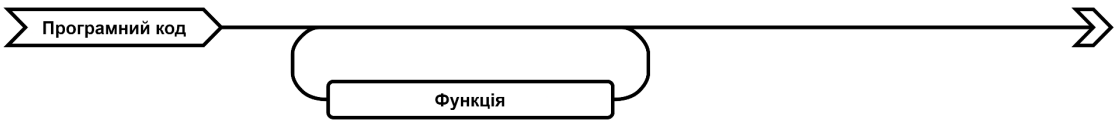

Рисунок 2.14 – Синтаксична діаграма програмного коду

Функція як основна одиниця програми синтаксично є складною двокомпонентною конструкцією. Опис функції у програмному коді складається з оголошення та визначення (рис. 2.15). Оголошення функції обов’язково повинне мати ідентифікатор та не обов’язково може мати

оголошення типу і перевизначення виразу. Функція обов’язково повинна мати визначення, при цьому визначення повинне бути одним з трьох можливих видів:

– вбудоване(приховане) визначення функції;

– визначення функції в короткому вигляді;

– визначення функції в повному вигляді.

В короткому вигляді визначення функції складається з виразу, в повному – з послідовності тверджень.

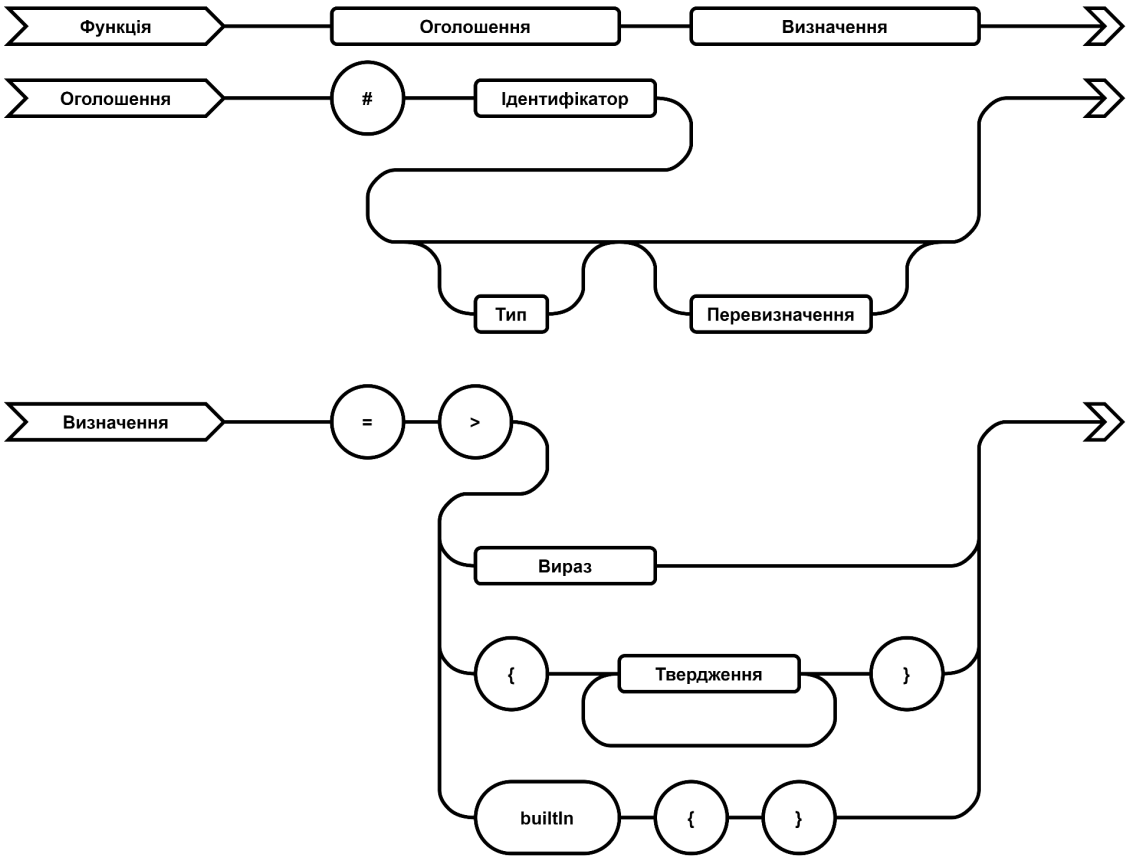

Рисунок 2.15 – Синтаксична діаграма функції

Кожне твердження у визначенні функції є або оголошення іншої функції, або композицією виразу функції у визначенні якої воно знаходиться (рис. 2.16).

Алгоритм синтаксичного аналізу полягає у рекурсивному виклику

функцій на послідовності лексем, кожна з яких співставляє послідовність лексем зі своїм синтаксичним правилом та визначає яку функцію викликати наступною. Наступна функція обирається шляхом розгляду послідовності лексем на одну вперед. Якщо функція не може співставити послідовність лексем зі своїм синтаксичним правилом, тоді синтаксичний аналіз закінчується з помилкою невідповідності програмного коду синтаксису мови програмування.

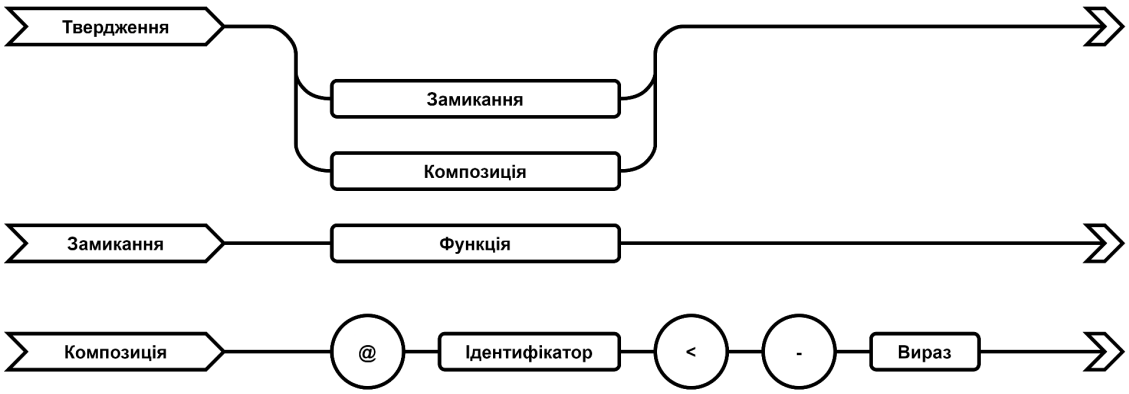

Рисунок 2.16 – Синтаксична діаграма твердження

2.4 Семантичний аналіз програмного коду

На етапі семантичного аналізу необхідно перевірити коректність програмного коду та його відповідність специфікаціям мови програмування. Семантичний аналізатор проводить аналіз структури, отриманої в результаті роботи синтаксичного аналізатору (рис. 2.17). Варто зазначити, що певні перевірки програмний код проходить впродовж попередніх етапів аналізу і на етапі семантичного аналізу вони не є потрібними. Наприклад, значення ідентифікатору функції в структурі отриманій після синтаксичного аналізу гарантовано є коректним з точки зору специфікації мови програмування. Але необхідно перевірити чи є коректним ідентифікатор з огляду на вже

оголошені ідентифікатори в області видимості. Також гарантовано коректними є конструкції визначення типів, але необхідно перевірити чи всі послідовності зміни типів під час обчислень є коректними в деревах виразів.

У випадку мови програмування з довільним синтаксисом виразів, аналіз виразів є важливою частиною саме семантичного аналізу. Аналіз необхідно провести на основі інформації із синтаксичних структур, при успішному виділенні викликів функцій необхідно побудувати дерево виразів та провести подальший його семантичний аналіз для перевірки усіх типів.

Рисунок 2.17 – Загальна схема семантичного аналізатору

2.5 Аналіз виразів програмного коду

На етапі лексичного аналізу, вирази мови програмування виділяються як окремі лексеми і в подальшому, під час синтаксичного аналізу, вираз не аналізується, а зберігається у вигляді послідовності символів. Причиною цього є відсутність необхідної інформації для визначення використаних у виразі функцій.

Під час семантичного аналізу для кожного виразу визначається область видимості, тобто перелік дозволених для виклику функцій та дозволених для використання ідентифікаторів [17]. Для кожної з сутностей області видимості необхідно згенерувати патерн. Патерн є послідовністю змінних та постійних частин, тобто виразів та роздільників (рис. 2.18).

Загалом, процес аналізу полягає у рекурсивному порівнянні виразу з усіма можливими комбінаціями відповідності (рис. 2.19) для кожного

патерну.

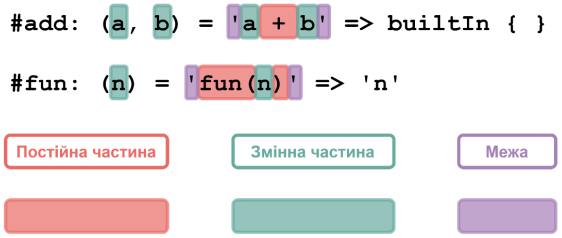

Рисунок 2.18 – Приклад патернів як послідовностей частин

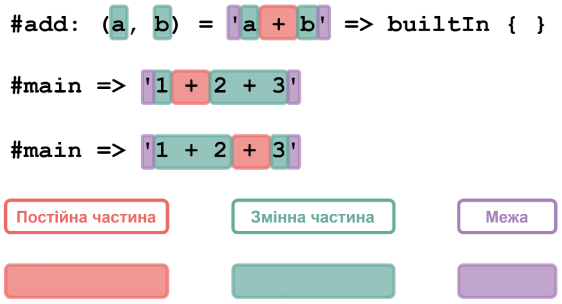

Рисунок 2.19 – Приклад різних комбінацій патерну

Якщо вираз є відповідним певному патерну зі змінними частинами, то змінні частини виділяються як вирази і аналізуються рекурсивно далі. Вираз вважається дійсним в результаті аналізу якщо всі його змінні частини є дійсними в результаті аналізу (рис. 2.20).

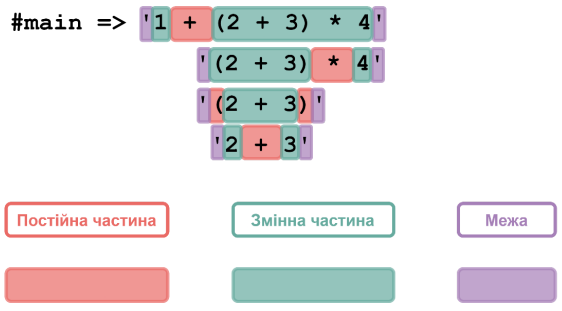

Рисунок 2.20 – Приклад аналізу виразу шляхом перевірки патернів

Для дійсного виразу будується дерево виразу (рис. 2.21), вузли якого є викликом функцій, а листки числами або ідентифікаторами з області видимості, що передаються у функції.

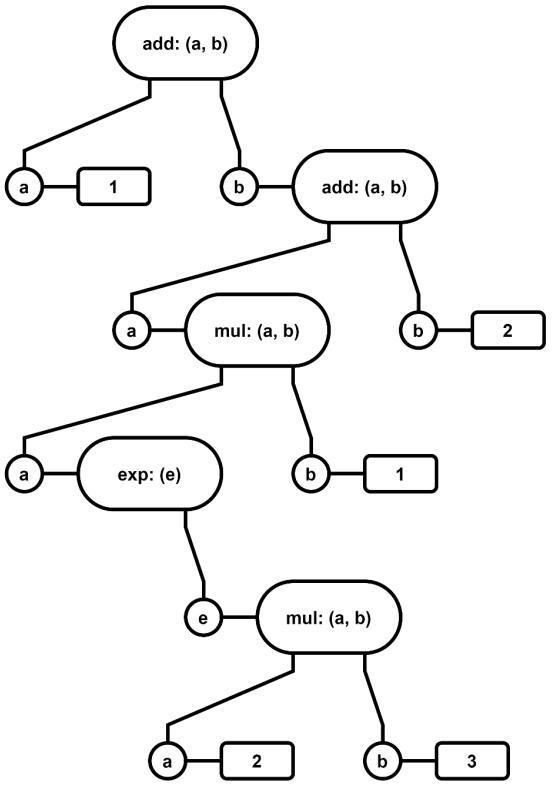

Рисунок 2.21 – Дерево виразу після визначення дійсних патернів

2.6 Компіляція та виконання програмного коду

Для розробки підходу компіляції програмного коду необхідно визначити якими будуть інструменти виконання програм, адже від цього залежить вигляд очікуваного результату роботи компілятора.

З огляду на те, що спроєктована мова програмування задумана як інструмент зручного запису математичних виразів, має сенс реалізувати середовище виконання програм як віртуальну машину на одній з розповсюджених мов програмування [25, 26]. Це дозволить вбудувати аналіз,

компіляцію та виконання програм у програмне забезпечення по типу інтегрованих середовищ розробки чи інженерних калькуляторів.

Також, варто зазначити, що мова програмування була спроєктована як мова без можливості створення користувачем складних структур даних, а всі екземпляри даних з якими виконується робота є імутабельними за своєю поведінкою [19, 20]. Крім того, в мову не закладені можливості паралельної обробки даних, а це також значно спрощує розробку інструментів компіляції та виконання.

Виходячи з вищенаведеного, оптимальним варіантом є проєктування віртуальної машини як регістрової віртуальної машини з класичним розділенням пам’яті на пам’ять операндів та пам’ять операцій (рис. 2.22).

Рисунок 2.22 – Будова віртуальної машини

Так як основною одиницею програмного коду є функція, то пам’ять операцій можна спроєктувати як віртуальну таблицю функцій [26]. У зв’язку з тим, що функції не мають побічних ефектів і є лише композицією інших функцій, віртуальну таблицю функцій можна представити як віртуальну таблиці іменованих ідентифікаторами виразів (рис. 2.23).

При цьому пам’ять операндів можна представити як таблицю операндів, кожен з яких є або числом, або посиланням на комірку таблиці пам’яті операцій (рис. 2.24).

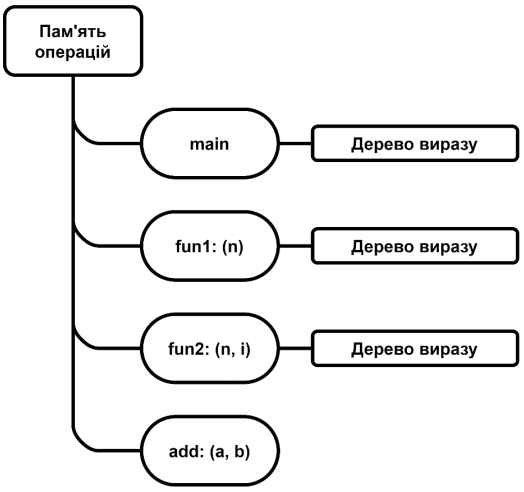

Рисунок 2.23 – Будова пам’яті операцій

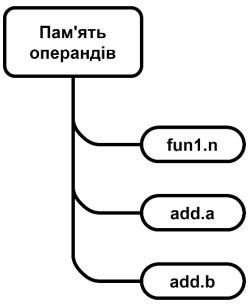

Рисунок 2.24 – Будова пам’яті операндів

При такій архітектурі віртуальної машини процес компіляції програмного коду є перетворенням структур отриманих під час попередніх етапів аналізу у дві таблиці пам’яті (рис. 2.25).

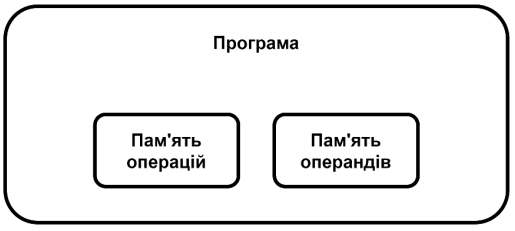

Рисунок 2.25 – Будова програми

Під час компіляції (рис. 2.26) необхідно провести композицію тверджень [24] кожної функцій у вираз і провести розподілення ідентифікаторів регістрів. Посилання на кожну функцію з пам’яті операцій необхідно помістити до пам’яті операндів. При такому підході виконання функції є отриманням функції з пам’яті операцій і заповненням відповідних їй параметрів у памя’ті операндів з подальшим виконанням дерева виразів.

Рисунок 2.26 – Загальна схема компілятору

2.7 Інтегроване середовище розробки програмного коду

Інтегроване середовище розробки це програмний інструмент, який повинен поєднати в собі всі етапи розробки програмного забезпечення [7]. Середовище повинне надати засоби введення програмного коду і його виконання для отримання результату обчислень, при цьому всі етапи аналізу та компіляції мають виконуватись автоматично і приховано від користувача (рис. 2.27).

Рисунок 2.27 – Загальна схема процесів у середовищі

3 ПРОГРАМНА РЕАЛІЗАЦІЯ І ТЕСТУВАННЯ

3.1 Програмна реалізація лексичного аналізатора

Основною одиницею лексичного аналізу є символ алфавіту формальної мови. Послідовність таких символів програмного коду перетворюється лексичним аналізатором на послідовність лексем мови програмування. Символ алфавіту бажано розробити як окремий інтерфейс символу, таке рішення дозволить збільшити семантичне навантаження типу даних, що покращить розуміння програмного коду людиною [7], а також надасть можливості до більш зручного розширення поведінки [6, 8]. Виходячи з того, що екземпляр символу алфавіту, по суті, є лише контейнером для зберігання числового коду символу, у певній систему кодування, з метою спрощення структури програмного коду і зменшення його розміру, символ алфавіту розроблено як клас даних (рис. 3.1).

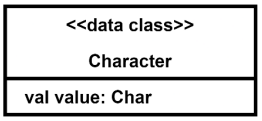

Рисунок 3.1 – Діаграма класу даних символу алфавіту

Для зручності реалізації лексичного аналізатора, екземпляр символу алфавіту повинен мати поведінку, що дозволяє перевіряти його на відповідність одному символу чи групам символів, це є важливим для лексичного аналізу. З огляду на те, що ця поведінка є додатковою та не залежить від внутрішньої реалізації символу алфавіту, прийняте рішення додати її шляхом використання патерну декоратор [6] (рис. 3.2).

Враховуючи те, що ключові для лексичного аналізу символи та їх

послідовності є сталими, то необхідно сформувати сховище констант, найбільш вдалим варіантом є використання патерну одиночка, тобто класу-об’єкту [27, 28] (рис. 3.3).

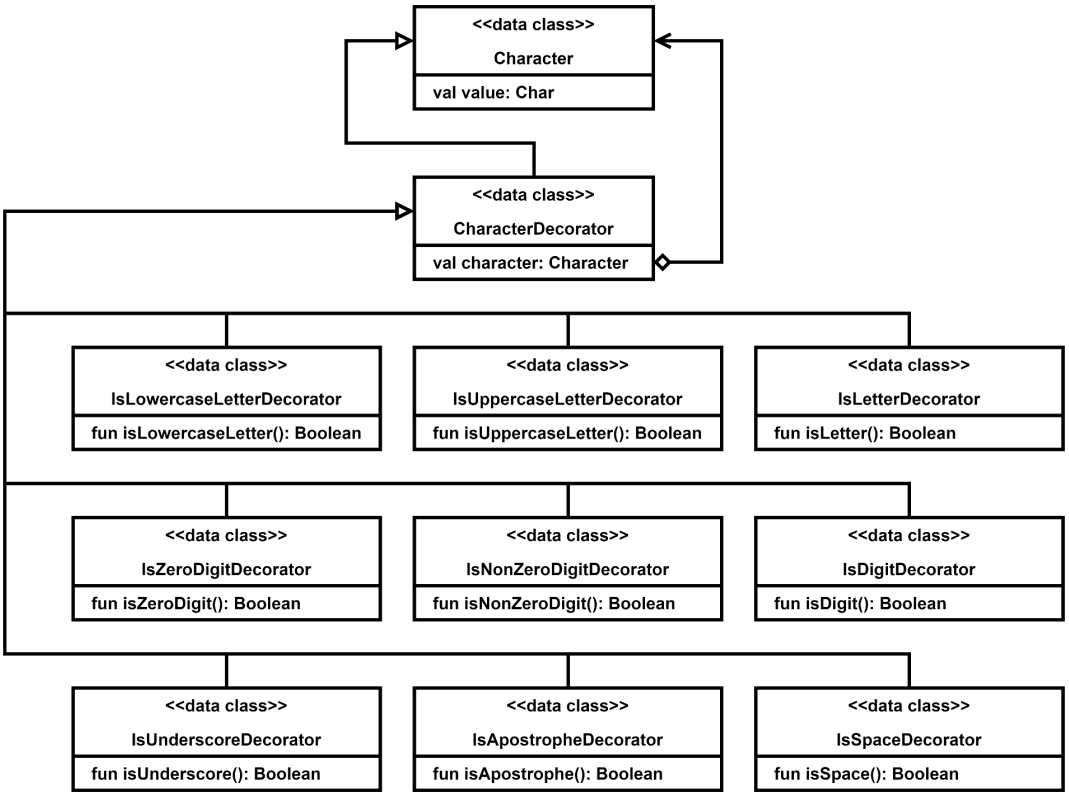

Рисунок 3.2 – Діаграма класів декораторів символу алфавіту

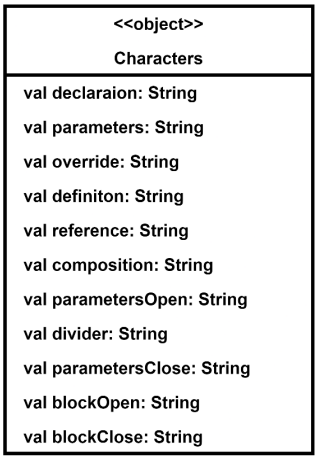

Рисунок 3.3 – Діаграма класу-об’єкту сховища констант

Лексема є контейнером, що зберігає в собі позицію визначеної аналізатором послідовності та її значення. Для зручності подальшого синтаксичного аналізу, тип лексеми реалізовано за поведінкою (рис. 3.4) як алгебраїчний [20], що дозволяє визначити її вид не за значенням, а за ідентифікатором типу при подальшій обробці [27].

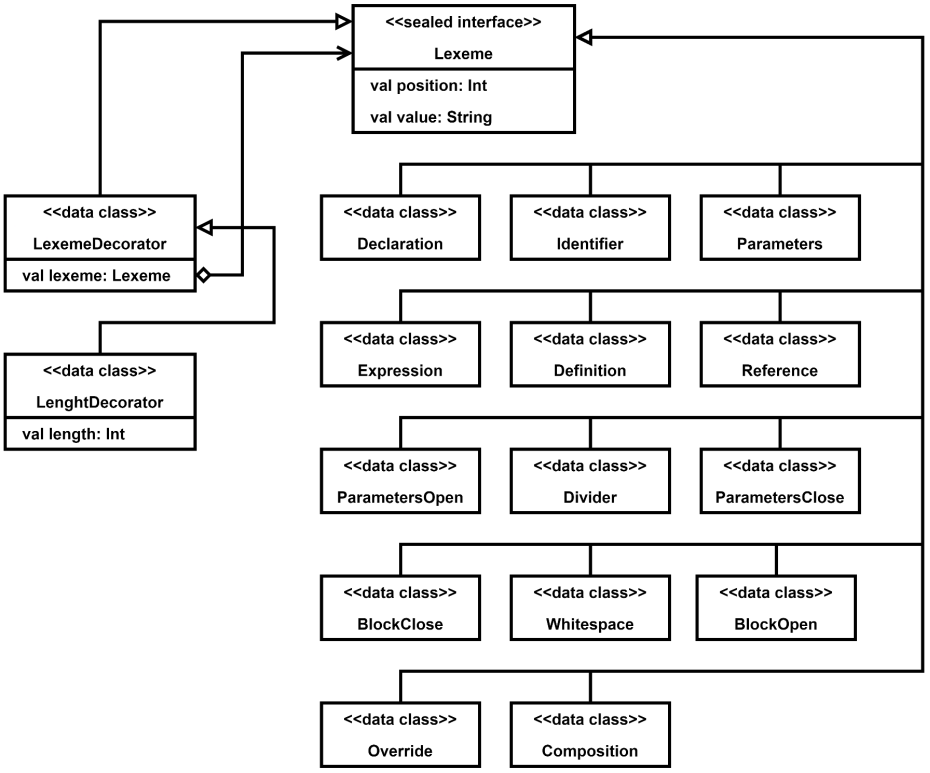

Рисунок 3.4 – Діаграма інтерфейсу лексеми та її нащадків

Лексичний аналізатор за своїм інтерфейсом виконано як програмну сутність з одним методом, який приймає послідовність символів алфавіту і повертає як результат аналізу список лексем алгебраїчного типу (рис. 3.5).

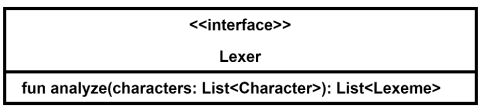

Рисунок 3.5 – Діаграма інтерфейсу лексичного аналізатору

3.2 Програмна реалізація синтаксичного аналізатора

Синтаксичний аналізатор реалізовано у вигляді рекурсивного алгоритму синтаксичного аналізу [16, 17]. Для реалізації цього алгоритму послідовність лексем програмного коду зручно представити у вигляді екземпляру класу з поведінкою патерну ітератор [7], який дозволяє отримати значення одного елементу попереду без зміни позиції (рис. 3.6). Такий ітератор виконано як інтерфейс і його реалізацію. Для виконання лексичного аналізу з його використанням необхідна додаткова поведінка у вигляді методів перевірки типу наступних лексем і методів запиту наступної лексеми визначеного типу. Цю задачу виконано шляхом використання патерну декоратор.

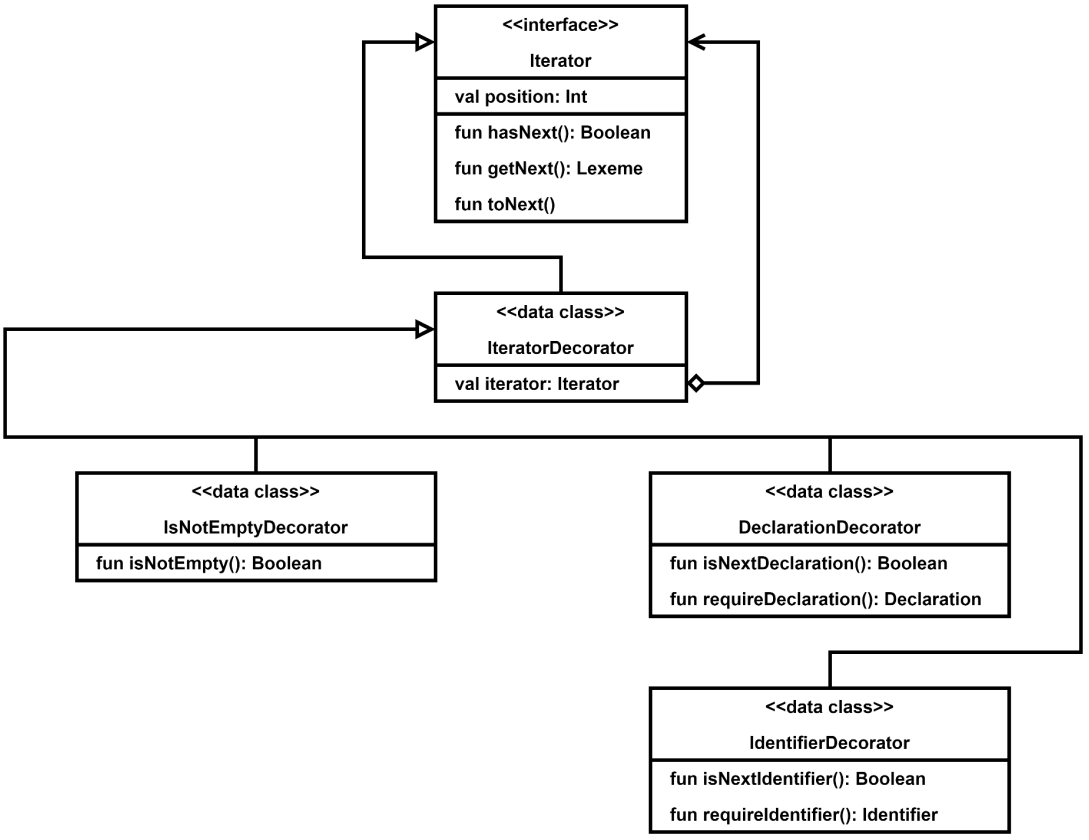

Рисунок 3.6 – Діаграма ітератора лексем і його класів декораторів

Результатом роботи синтаксичного аналізатора є структура програмного коду, яка є деревом екземплярів структур різних типів синтаксичних конструкцій мови програмування. Необхідно описати інтерфейси структур та їх реалізації, для спрощення програмного коду цю задачу виконано з використання класів даних (рис. 3.7).

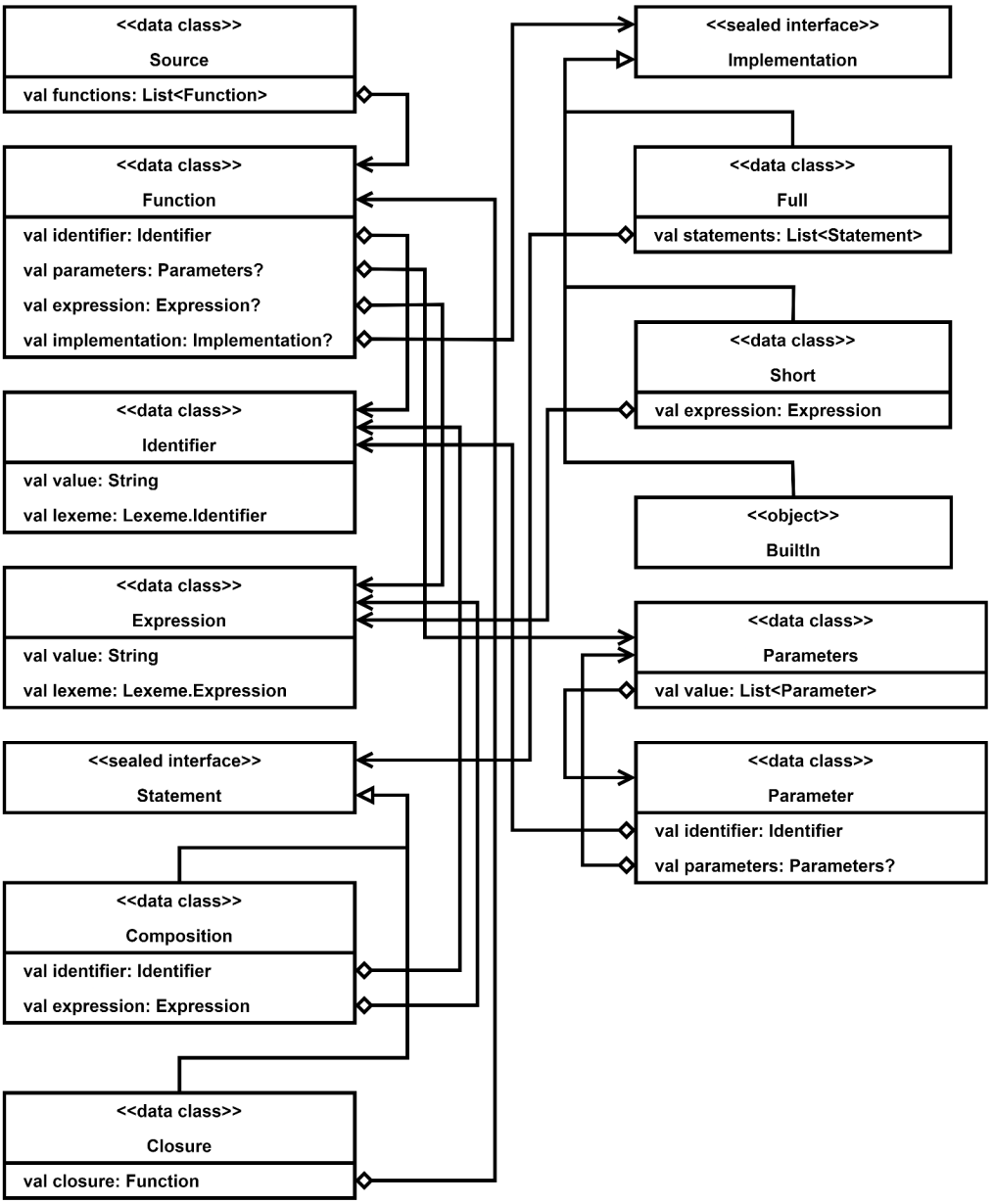

Рисунок 3.7 – Діаграма класів даних структур синтаксичних конструкцій

Синтаксичний аналізатор за своїм інтерфейсом виконано як програмну сутність з одним методом, який приймає список лексем і повертає результатом розбору структуру програмного коду (рис. 3.8).

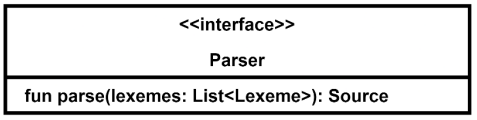

Рисунок 3.8 – Діаграма інтерфейсу синтаксичного аналізатора

3.3 Програмна реалізація віртуальної машини

Пам’ять операцій віртуальної машини розроблено у вигляді класу даних таблиці [26], яка є контейнером для мапи процедур (рис. 3.9). Кожна з процедур пов’язується в таблиці з відповідним їй ідентифікатором. В свою чергу, процедуру розроблено як клас даних, що складається з ідентифікатору процедури та виконуваного для неї дерева виразу. Дерево виразу розроблено як клас даних, тобто контейнер вузлу алгебраїчного типу даних. Кожен з вузлів розроблено як клас даних одного з можливих видів елементу дерева виразу.

Пам’ять операндів віртуальної машини розроблено у вигляді класу даних пам’яті [28], яка є контейнером для мапи регістрів (рис. 3.10). Кожен з регістрів в мапі пов’язується з його адресою. Регістри розроблено як класи даних одного з можливих нащадків алгебраїчного типу.

Пам’ять операндів і пам’ять операцій використовуються не лише як сховища даних та команд для віртуальної машини в процесі її роботи, а також як компоненти сформованої для віртуальної машини програми [22]. Програму розроблено у вигляді класу даних (рис. 3.11).

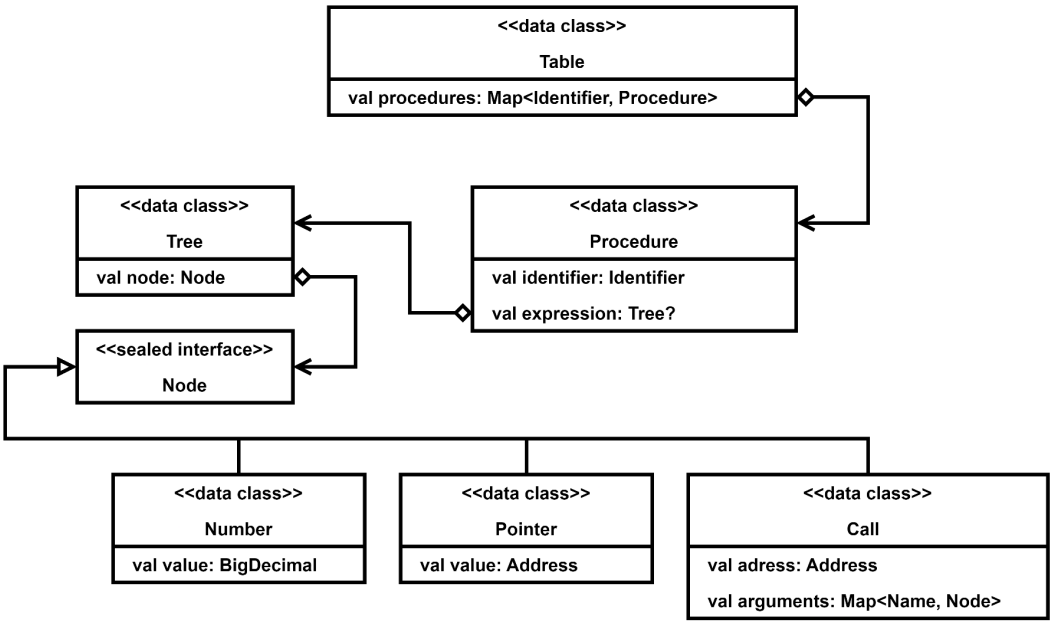

Рисунок 3.9 – Діаграма програмних сутностей пам’яті операцій

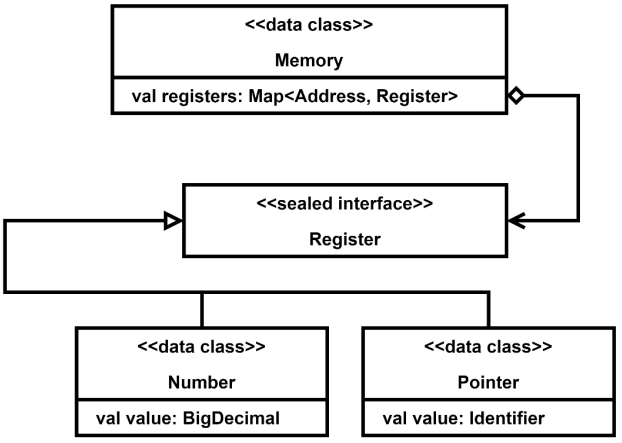

Рисунок 3.10 – Діаграма програмних сутностей пам’яті операндів

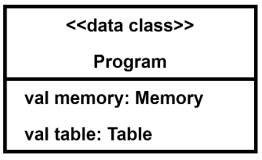

Рисунок 3.11 – Діаграма класу даних програми

Віртуальну машину розроблено як програмну сутність, що складається з інтерфейсу та його реалізації. Інтерфейс віртуальної машини складається з одного метода, який приймає сформовану для неї програму і повертає результат обчислень (рис. 3.12).

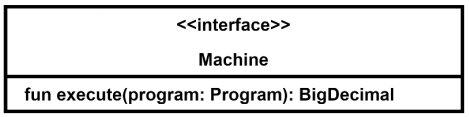

Рисунок 3.12 – Діаграма інтерфейсу віртуальної машини

3.4 Програмна реалізація компілятора

Компілятор розроблено як програмну сутність (рис 3.13), яка приймає структуру програмного коду, отриману на етапі синтаксичного аналізу, у сформовану програму для віртуальної машини. Інтерфейс компілятора має один метод, що виконує дане перетворення в процесі своєї роботи.

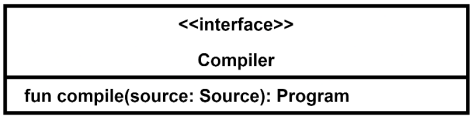

Рисунок 3.13 – Діаграма інтерфейсу компілятора

В процесі компіляції алгоритм рухається послідовно конструкціями мови програмування, а також рекурсивно вглиб. Елементи які пройдені алгоритмом на момент обробки наступної конструкції формують область видимості конструкцій програмного коду [17]. Область видимості розроблено програмною сутністю з інтерфейсом, який має адресу та ідентифікатор, а також властивості та поведінку, що дозволяють вносити зміни (рис. 3.14).

Елемент області видимості розроблено як сутність алгебраїчного типу, його нащадками є класи даних, що відповідають різним видам елементів.

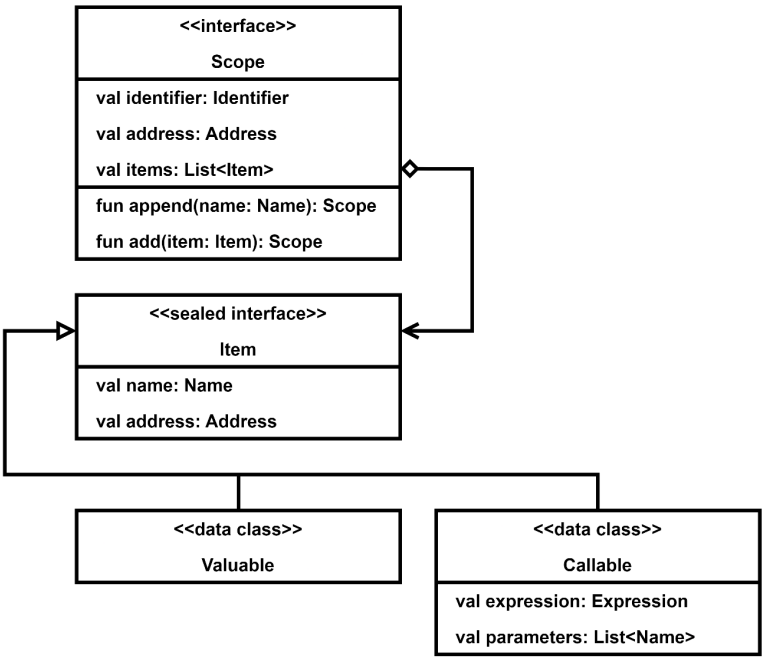

Рисунок 3.14 – Діаграма програмних сутностей області видимості

Компіляцію виразів програмного коду у дерева виразів віртуальної машини розроблено у вигляді програмної сутності аналізатору виразів, інтерфейс якого має метод, який приймає вираз та область видимості, а повертає сформоване дерево (рис. 3.15).

Ключовим внутрішнім елементом є шукач комбінацій виклику у виразах (рис. 3.16), який розроблено як реалізацію інтерфейсу, екземпляр якого приймає вираз, а повертає список виділених можливих комбінацій.

Рисунок 3.15 – Діаграма інтерфейсу аналізатору виразів

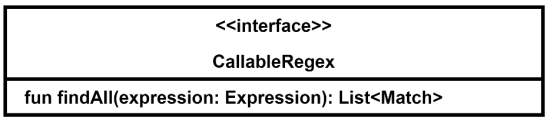

Рисунок 3.16 – Діаграма інтерфейсу шукача комбінацій виразів

3.5 Дизайн інтегрованого середовища розробки

Інтегроване середовище розробки має надавати кінцевому користувачу зручний графічний інтерфейс для сворення, редагування і збереження елементів програмного коду, а також приховувати внутріші деталі обробки програмного коду, спрощуючи отримання результатів виконання програми.

В базовому вигляді інтегроване середовище розробки може мати графічний інтерфейс, що складається з двох блоків (рис. 3.17):

– роботи з текстом програмного коду;

– інструментів обробки програмного коду.

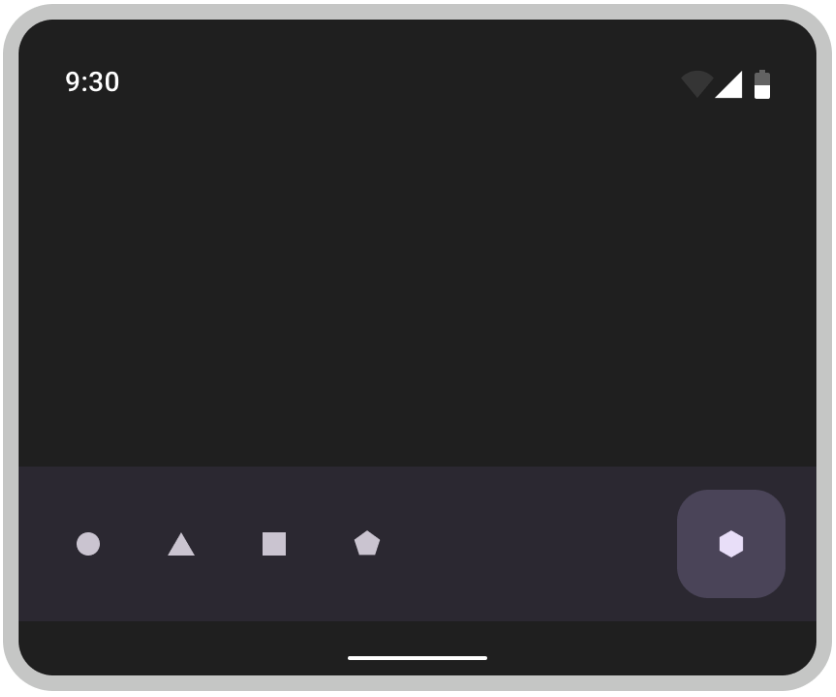

Рисунок 3.17 – Дизайн блоків графічного інтерфейсу

Блок роботи з текстом програмного коду (рис. 3.18) є контейнером для компонентів графічного інтерфейсу, які необхідні для введення тексту програмного коду. В базовому вигляді такий блок має бути контейнером для поля введення тексту з реалізацією автоматизованої підсвітки програмного коду, що позитивно вплине на сприйняття інформації кінцевим користувачем.

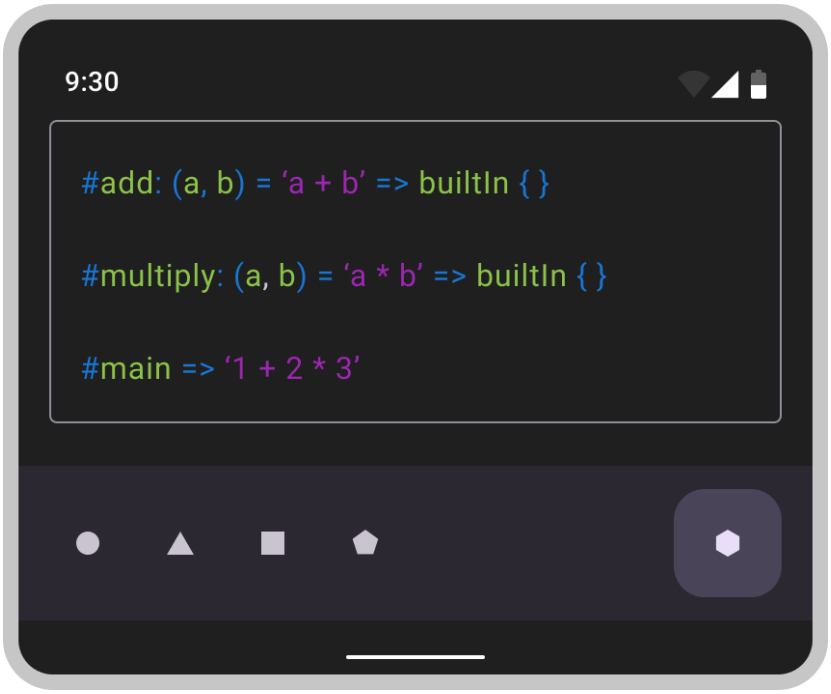

Рисунок 3.18 – Дизайн блоку роботи з текстом програмного коду

Блок інструментів обробки програмного коду (рис. 3.19) є контейнером для компонентів графічного інтерфейсу, які необхідні для внесення автоматизованих змін в текст програмного коду, або для виклику послідовності етапів обробки та виконання програмного коду. В базовому вигляді середовище повинне мати можливості зберігання програмного коду і відновлення його при подільших запусках програмного забезпечення, а також перемикання між збереженими версіями. Крім того, блок повинен мати графічні компоненти для виклику обробки і виконання програмного коду, а також для скасування цього процесу.

Результатом обробки програмного коду може бути результат математичних обчислень, отриманий в процесі виконання програми

віртуальною машиною, або опис помилки, яка виникла в процесі. Повідомлення з результатами можна відобразити у вигляді графічного компоненту діалогу (рис. 3.20).

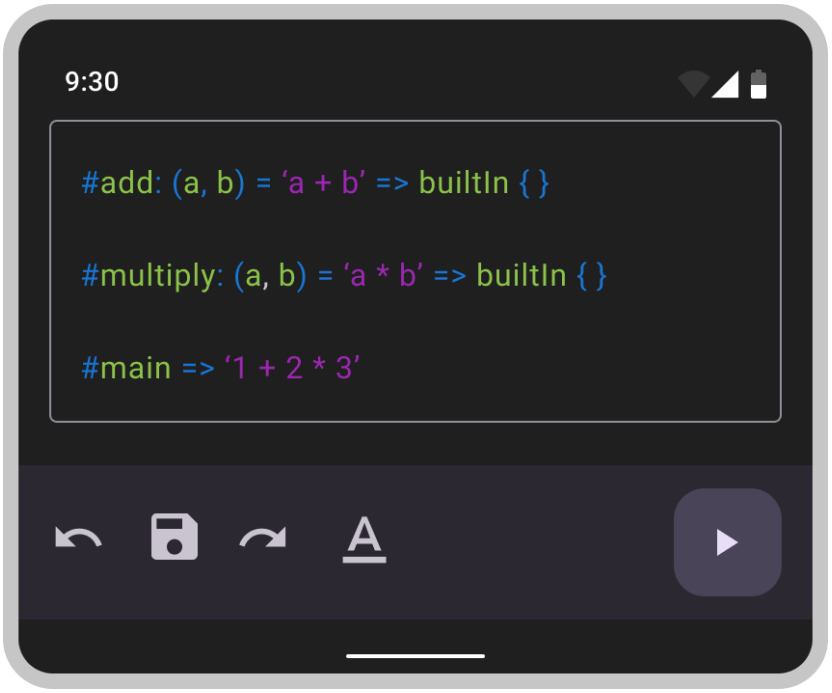

Рисунок 3.19 – Дизайн блоку інструментів обробки програмного коду

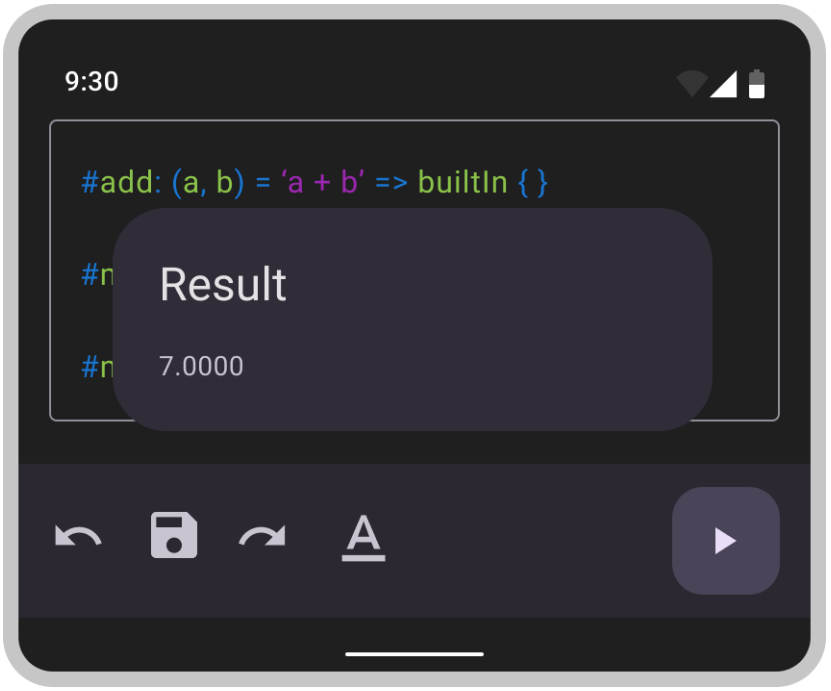

Рисунок 3.20 – Дизайн діалогу результатів

3.6 Програмна реалізація інтегрованого середовища розробки

Інтегроване середовище розроблено як програмний застосунок з графічним інтерфейсом. Архітектура програмної реалізації побудована орієнтуючись на тришарові архітектурні патерни програмних застосунків з односпрямованими потоками даних [8, 29].

Шар відображення інформації, тобто шар керування графічним інтерфейсом, розроблено як програмну сутність, яка обробляє однонаправлений потік станів графічного інтерфейсу та вносить зміни у графічний інтерфейс для відповідності оновленому стану [30]. Стан графічного є екземпляром інтерфейсу описаного у вигляді інтерфейсу алгебраїчного типу, нащадки якого є класами даних різних видів стану графічного інтерфейсу (рис. 3.21).

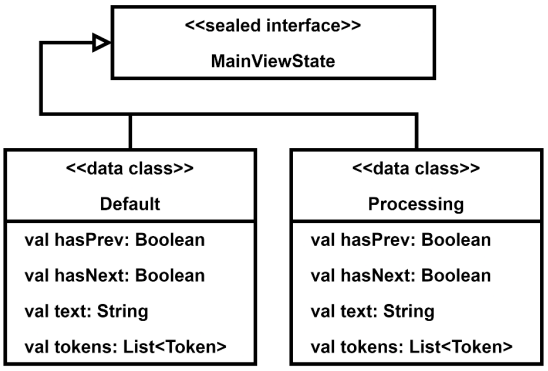

Рисунок 3.21 – Діаграма сутностей стану графічного інтерфейсу

Шар відображення інформації також є обробником вводу користувача, так як він напряму взаємодіє з графічними компонентами та має до них доступ. Отримані події вводу перетворюються в екземпляри інтерфейсу описаного у вигляді інтерфейсу алгебраїчного типу, нащадки якого є класами даних різних подій користувацького вводу (рис. 3.22) та передаються до шару програмної логіки.

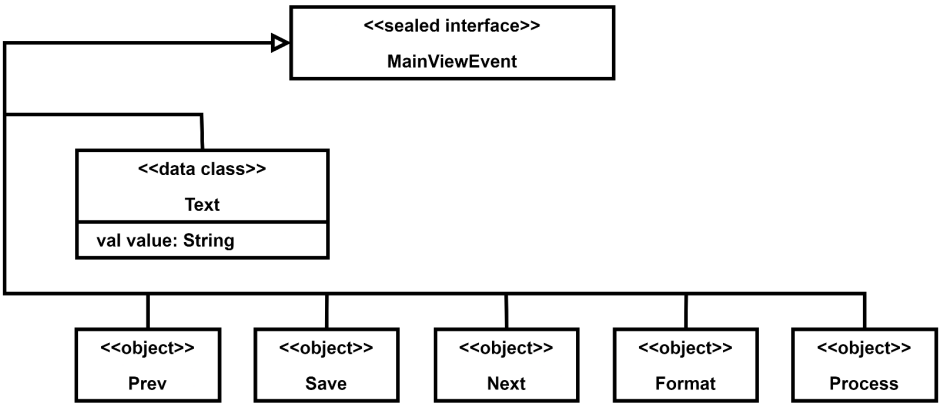

Рисунок 3.22 – Діаграма сутностей подій користувацького вводу

Шар програмної логіки розроблено як програмну сутність з інтерфейсом, який дозволяє надсилати події користувацького вводу, а також отримувати оновлення стану з шару відображення інформації (рис. 3.23). Шар програмної логіки повністю реалізує в собі обробку вводу користувачу, а також реалізує взаємодію з шаром даних [29, 30].

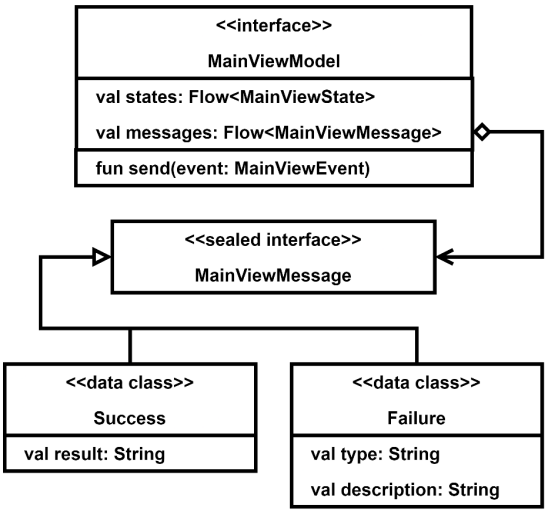

Рисунок 3.23 – Діаграма інтерфейсу шару програмної логіки

Шар даних розроблено як набір програмних сутностей (рис. 3.24), основні з яких є обгортками для розроблених інструментів обробки

програмного коду, або обгортками для системних інструментів зберігання інформації.

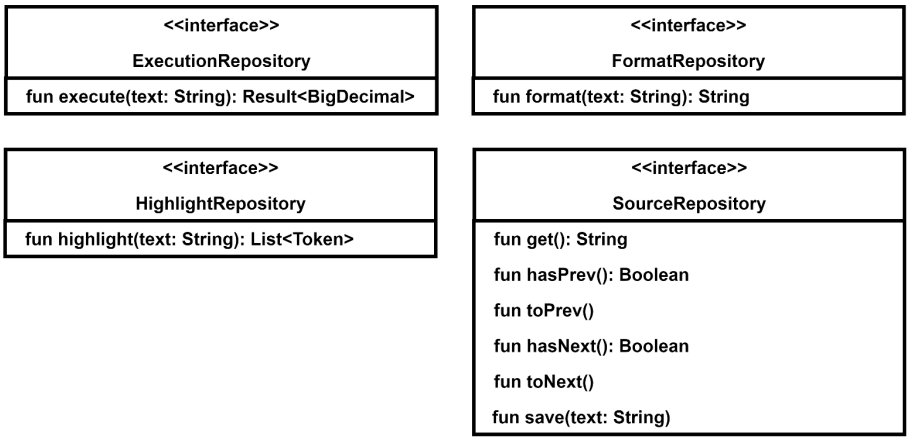

Рисунок 3.24 – Діаграма основних програмних сутностей шару даних

3.7 Тестування

Для тестування реалізованих програмних інструментів розроблено кілька програм. Метою їх розробки є перевірка можливості реалізації програм і алгоритмів з використанням створеної мови програмування та інтегрованого середовища розробки, а також тестування зручності використання загальноприйнятого синтаксису різних математичних функцій у виразах.

Розроблена мова програмування дозволяє виконувати арифметичні обчислення використовуючи вбудовані математичні функції, виклику функцій в цьому випадку все одно аналізуються з огляду на оголошення, як і у випадку функцій визначених користувачем.

Лістинг 3.1 Приклад обчислення значення виразу (рис. 3.25):

#add: (a, b) = 'a + b' => builtIn { }

#subtract: (a, b) = 'a - b' => builtIn { }

*#multiply: (a, b) = 'a * b' => builtIn { }*

#divide: (a, b) = 'a / b' => builtIn { }

#brackets: (expression) = '(expression)' => 'expression'

*#main => '1 + (2 * 3) / 4 - 5'*

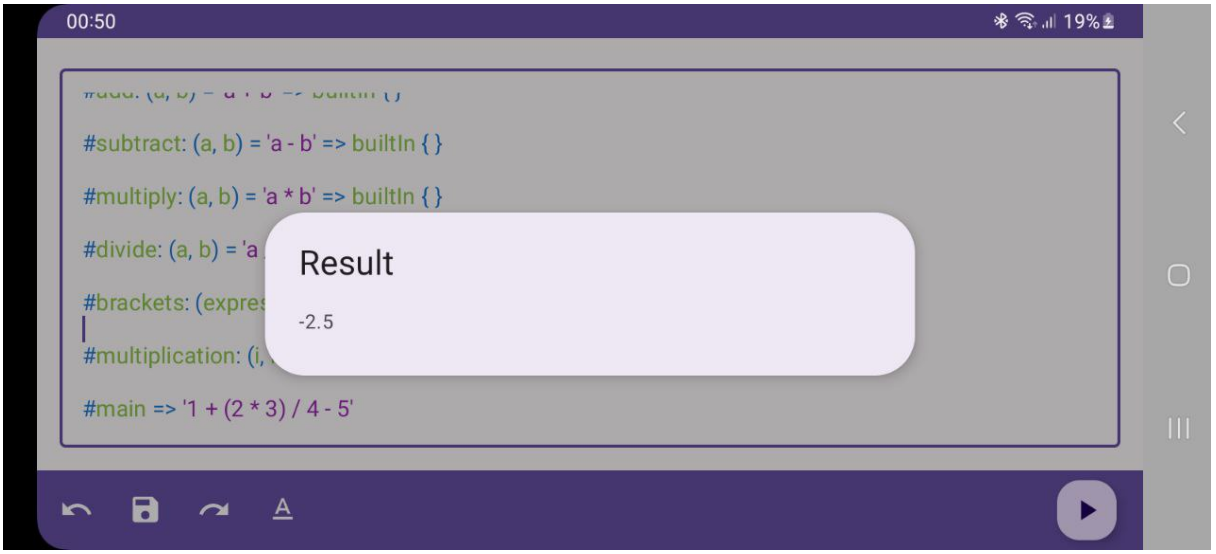

Рисунок 3.25 – Результат виконання програмного коду

Розроблена мова програмування дозволяє користувачу створювати власні математичні функції зі спеціально перевизначеним синтаксисом виклику. Мова надає вбудовані інструменти для побудови ітеративних алгоритмів, це використано на прикладі реалізації функції факторіалу.

Лістинг 3.2 Приклад реалізації операторних дужок (не вбудована реалізація), факторіалу (рис. 3.26) і обчислення виразу:

#add: (a, b) = 'a + b' => builtIn { }

```
#subtract: (a, b) = 'a - b' => builtIn { }

#multiply: (a, b) = 'a * b' => builtIn { }
#divide: (a, b) = 'a / b' => builtIn { }

#brackets: (expression) = '(expression)' => 'expression'

#multiplication: (i, n, lambda(i)) = 'M[i, n]{ lambda }' => builtIn { }
#factorial: (number) = 'number!' => {
   #f: (i) => 'i'
   @factorial <- 'M[1, number]{ f }'
}

#main => '1 + (2 * 3)! / 4 - 5'
```

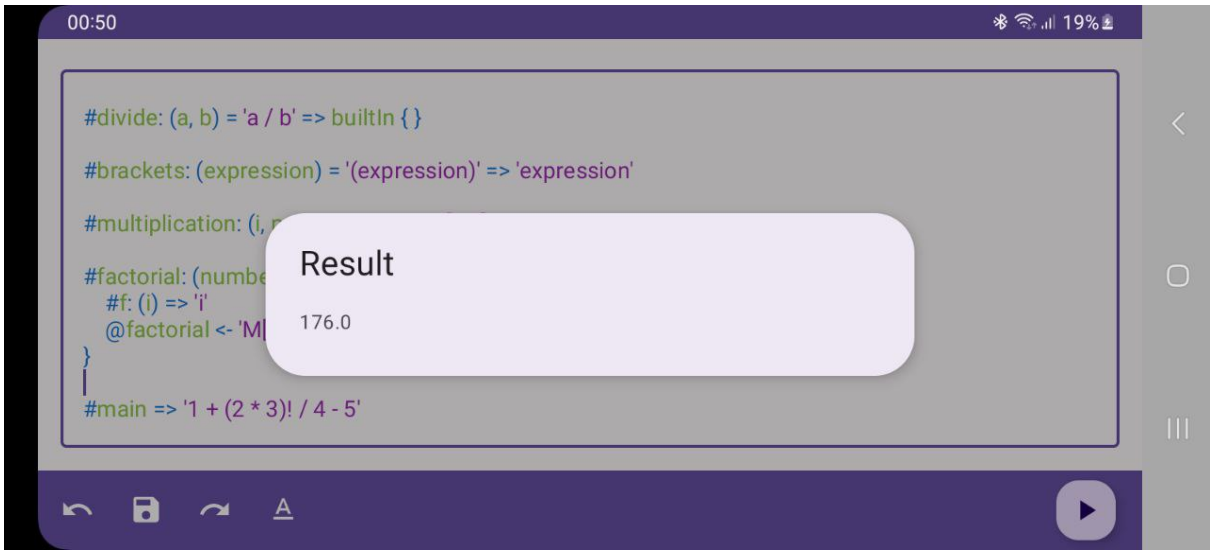

Рисунок 3.26 – Результат виконання програмного коду

Розроблена мова програмування не має обмежень множини символів для синтаксису виклику функцій. Можливість використовувати символи у верхньому індексі показано на прикладі розробленої ітеративної функції обчислення степеня і його скорочень для найбільш розповсюджених

параметрів.

Лістинг 3.3 Приклад реалізації степеня (рис. 3.27) і обчислення виразу:

```
#add: (a, b) = 'a + b' => builtIn { }
#multiplication: (i, n, lambda(i)) = 'M[i, n]{ lambda }' => builtIn { }

#pow: (a, b) = 'a^b' => {
   #p: (i) => 'a'
   @pow <- 'M[1, b]{ p }'
}

#pow2: (a) = 'a²' => 'a^2'
#pow3: (a) = 'a³' => 'a^3'
#pow4: (a) = 'a⁴' => 'a^4'

#main => '2³ + 2⁴'
```

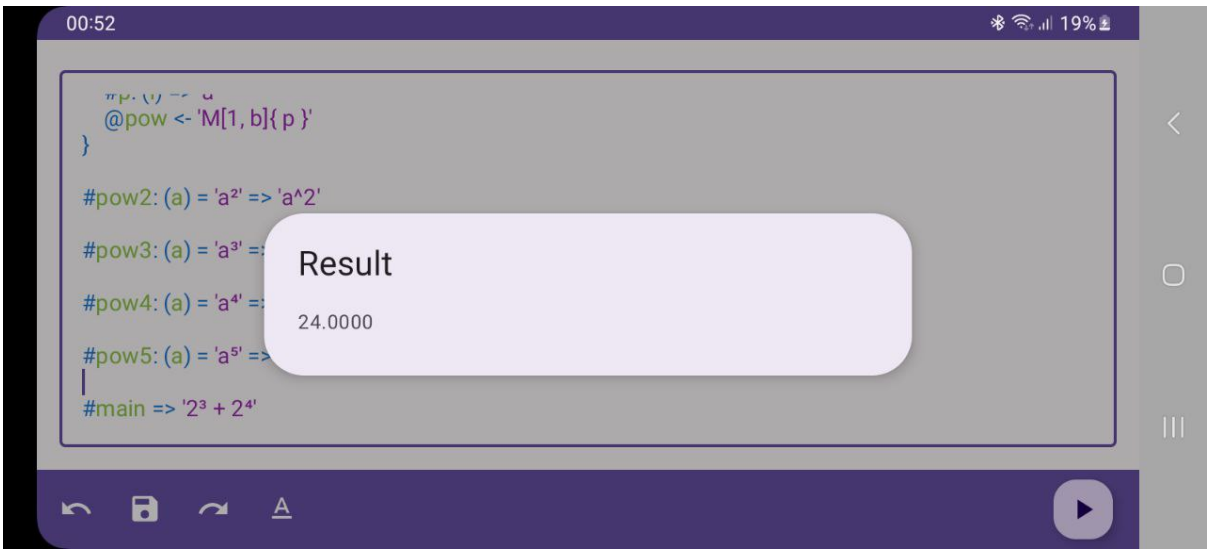

Рисунок 3.27 – Результат виконання програмного коду

Розроблена мова програмування дозволяє реалізовувати чисельні методи засновані на апроксимації, це показано на прикладі реалізації функції

обчислення значення квадратного кореня.

Лістинг 3.4 Приклад реалізації кореня, модулю (рис. 3.28) і обчислення виразу:

```
#subtract: (a, b) = 'a - b' => builtIn { }
#multiply: (a, b) = 'a * b' => builtIn { }
#divide: (a, b) = 'a / b' => builtIn { }

#brackets: (expression) = '(expression)' => 'expression'

#multiplication: (i, n, lambda(i)) = 'M[i, n]{ lambda }' => builtIn { }
#approximation: (n, f(n), epsilon) = 'n >> f[epsilon]' => builtIn { }

#sqrt: (a) = '√a' => {
   #f: (n) => 'n - (n² - a) / (2 * n)'
   @sqrt <- '1 >> f[0.01]'
}

#pow: (a, b) = 'a^b' => {
   #p: (i) => 'a'
   @pow <- 'M[1, b]{ p }'
}

#pow2: (a) = 'a²' => 'a^2'

#abs: (n) = '|n|' => '√n²'

#main => '√16 * |10 - 20|'
```

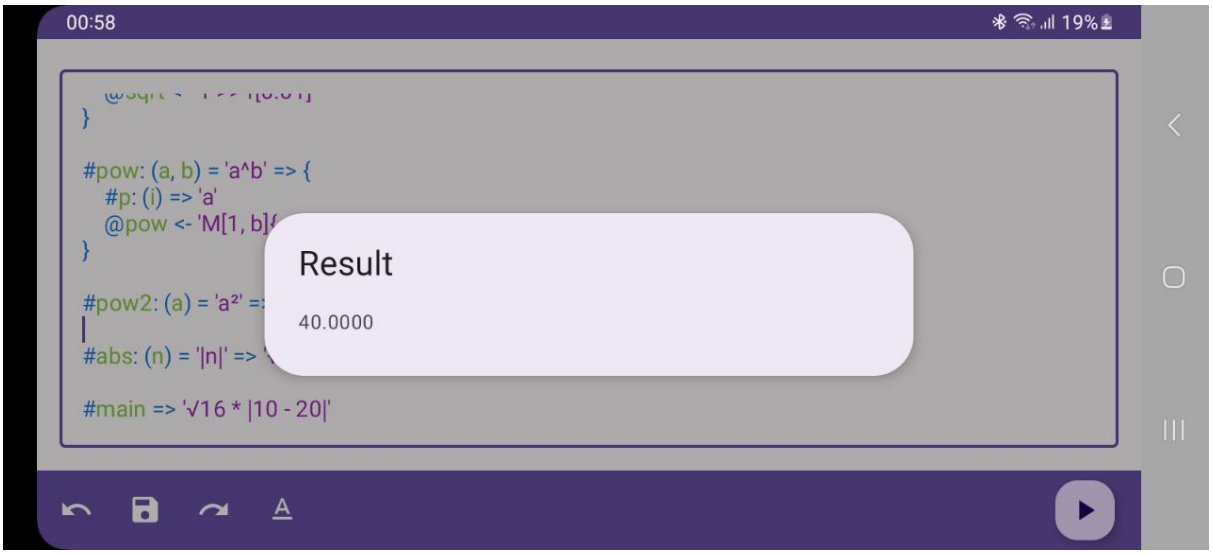

Рисунок 3.28 – Результат виконання програмного коду

Розроблена мова програмування дозволяє вдало керувати синтаксисом виклику функцій, що дозволяє реалізувати розрахунок похідної з загальноприйнятим виглядом звернення до оголошеної функції.

Лістинг 3.5 Приклад реалізації похідної (рис. 3.29) і обчислення виразу

```
#add: (a, b) = 'a + b' => builtIn { }
#subtract: (a, b) = 'a - b' => builtIn { }
#multiply: (a, b) = 'a * b' => builtIn { }
#divide: (a, b) = 'a / b' => builtIn { }

#brackets: (expression) = '(expression)' => 'expression'

#multiplication: (i, n, lambda(i)) = 'M[i, n]{ lambda }' => builtIn { }
#approximation: (n, f(n), epsilon) = 'n >> f[epsilon]' => builtIn { }

#sqrt: (a) = '√a' => {
   #f: (n) => 'n - (n² - a) / (2 * n)'
   @sqrt <- '1 >> f[0.01]'
}
```

```
#pow: (a, b) = 'a^b' => {
    #p: (i) => 'a'
    @pow <- 'M[1, b]{ p }'
}

#pow2: (a) = 'a²' => 'a^2'

#derivative: (f(x), x) = 'f`(x)' => {
    @derivative <- '(f(x + 0.001) - f(x - 0.001)) / 0.002'
}

#main => {
    #func: (x) => 'x² + √x'
    @main <- 'func`(10)'
}
```

Рисунок 3.29 – Результат виконання програмного коду

ВИСНОВКИ

В межах роботи було проаналізовано предметну область і об’єкт роботи з огляду на існуючі напрацювання в областях інформатики та інженерії програмного забезпечення, а також пройдено ряд етапів для досягнення мети роботи:

– спроєктовано мову програмування, її лексику, синтаксис і семантику;

– розроблено та спроєктовано інструменти лексичного аналізу;

– розробено та спроєктовано інструменти синтаксичного аналізу;

– розроблено та спроєктовано інструменти семантичного аналізу;

– розроблено та спроєктовано віртуальну машину;

– розроблено та спроєктовано інструменти компіляції;

– розроблено та спроєктовано інтегроване середовище розробки;

– програмно реалізовано інструменти лексичного аналізу;

– програмно реалізовано інструменти синтаксичного аналізу;

– програмно реалізовано інструменти семантичного аналізу;

– програмно реалізовано віртуальну машину;

– програмно реалізовано компілятор;

– програмно реалізовано інтегроване середовище розробки;

– проведено тестування реалізованих інструментів;

– проведено тестування інтегрованого середовища розробки.

Результати кваліфікаційної роботи є вдалими, адже спроєктовані і реалізовані програмно підходи, алгоритми та інструменти дозволяють розширити можливості мов програмування. Розроблена мова програмування дозволяє використовувати складний нетиповий синтаксис виклику створених користувачем математичних функцій, що дозволяє записувати математичні вирази у звичному для математичних робіт вигляді і вирішує поставлену задачу.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Rojas, R., & Hashagen, U. (Eds.). (2002). The first computers: History and architectures. MIT press.

2. Ceruzzi, P. E. (2003). A history of modern computing. MIT press.

3. Kinoshenko, D., Mashtalir, V., & Yegorova, E. (2006). CLUSTERING METHOD FOR FAST CONTENTBASED IMAGE RETRIEVAL. In Computer Vision and Graphics: International Conference, ICCVG 2004, Warsaw, Poland, September 2004, Proceedings (pp. 946-952). Springer Netherlands.

4. Kinoshenko, D., Mashtalir, V., Shlyakhov, V., & Yegorova, E. (2012). nested Partitions Properties for spatial content Image retrieval. In Multimedia Storage and Retrieval Innovations for Digital Library Systems (pp. 240-269). IGI Global.

5. Kinoshenko, D., Mashtalir, V., Yegorova, E., & Vinarsky, V. (2005). Hierarchical partitions for content image retrieval from large-scale database. In Machine Learning and Data Mining in Pattern Recognition: 4th International Conference, MLDM 2005, Leipzig, Germany, July 9-11, 2005. Proceedings 4 (pp. 445-455). Springer Berlin Heidelberg.

6. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1993). Design patterns: Abstraction and reuse of object-oriented design. In ECOOP'93—Object-Oriented Programming: 7th European Conference Kaiserslautern, Germany, July 26–30, 1993 Proceedings 7 (pp. 406-431). Springer Berlin Heidelberg.

7. Martin, R. C. (2009). Clean code: a handbook of agile software craftsmanship. Pearson Education.

8. Martin, R. C. (2017). Clean architecture.

9. Kinoshenko, D., Mashtalir, V., & Shlyakhov, V. (2007). A partition metric for clustering features analysis.

10. Bodyanskiy, Y., Kinoshenko, D., Mashtalir, S., & Mikhnova, O. (2012). On-line video segmentation using methods of fault detection in multidimensional time sequences. International Journal of Electronic Commerce Studies, 3(1), 1-20.

11. Chupikov, A., Kinoshenko, D., Mashtalir, V., & Shcherbinin, K. (2007). Image retrieval with segmentation-based query. In Adaptive Multimedia Retrieval: User, Context, and Feedback: 4th International Workshop, AMR 2006, Geneva, Switzerland, July 27-28, 2006, Revised Selected Papers 4 (pp. 207-221). Springer Berlin Heidelberg.

12. Kinoshenko, D., Mashtalir, V., Shlyakhov, V., & Yegorova, E. (2011). Metrical properties of nested partitions for image retrieval. In Machine Learning Techniques for Adaptive Multimedia Retrieval: Technologies Applications and Perspectives (pp. 18-49). IGI Global.

13. Mashtalir, S., Mikhnova, O., & Stolbovyi, M. (2018, August). Sequence matching for content-based video retrieval. In 2018 IEEE Second International Conference on Data Stream Mining & Processing (DSMP) (pp. 549-553). IEEE.

14. Oleg, K., Sergii, M., & Mykhailo, S. (2017, October). Video clustering via multidimensional time-series analysis. In Proceedings of the 9th International Conference on Information Management and Engineering (pp. 60-63).

15. Ye, B., Shafronenko, A., & Mashtalir, S. (2020). Online robust fuzzy clustering of data with omissions using similarity measure of special type-Lecture Notes in Computational clutolligence and Decision Waking-Cham.

16. Keith, D., & COOPER, T. (2022). ENGINEERING A COMPILER. MORGAN KAUFMANN PUBLISHER.

17. Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2020). Compilers: principles, techniques and tools.

18. Sedgewick, R., Wayne, K. (2011). Algorithms, 4th Edition.. Addison-Wesley.

19. Bragilevsky, V. (2021). Haskell in Depth. Simon and Schuster.

20. Pierce, B. C. (2002). Types and programming languages. MIT press.

21. Haggarty, R. (2001). Discrete mathematics for computing. Addison-Wesley Longman Publishing Co., Inc..

22. Abelson, H., & Sussman, G. J. (1996). Structure and interpretation of computer programs (p. 688). The MIT Press.

23. Roughgarden, T. (2017). Algorithms illuminated. Soundlikeyourself publishing.

24. Jones, S. P., Hall, C., Hammond, K., Partain, W., & Wadler, P. (1993, July). The Glasgow Haskell compiler: a technical overview. In Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference (Vol. 93).

25. Watt, D. A., & Brown, D. F. (2000). Programming language processors in Java: compilers and interpreters. Pearson Education.

26. Lindholm, T., Yellin, F., Bracha, G., & Buckley, A. (2014). The Java virtual machine specification. Pearson Education.

27. Skeen, J., & Greenhalgh, D. (2018). Kotlin Programming: The Big Nerd Ranch Guide. Pearson Technology Group.

28. Moskala, M., & Wojda, I. (2017). Android Development with Kotlin. Packt Publishing Ltd.

29. Cheng, Y., & Domínguez, A. O. (2019). Advanced android app architecture: real-world app architecture in Kotlin 1.3. Razeware LLC.

30. Hardy, B., & Phillips, B. (2013). Android programming: the big nerd ranch guide. Addison-Wesley Professional.