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

Факультет прикладної математики

Кафедра програмного забезпечення комп'ютерних систем

«До захисту допущено»

Завідувач кафедри

_____ Євгенія СУЛЕМА

«___» _____ 2025 р.

Дипломний проєкт

на здобуття ступеня бакалавра

за освітньо-професійною програмою «Інженерія програмного забезпечення мультимедійних та інформаційно-пошукових систем» спеціальності 121 Інженерія програмного забезпечення

на тему: «Інтегроване середовище розроблення програмного забезпечення мовою ASAMPL 2.0. Компілятор»

Виконав:
студент IV курсу, групи КП-13
Міщенко Владислав Романович _____

Керівник:
завідувач кафедри ПЗКС, д.т.н., доцент,
Сулема Євгенія Станіславівна _____

Консультант з нормоконтролю:
доцент кафедри ПЗКС, к.т.н., доцент,
Онай Микола Володимирович _____

Рецензент:
старший викладач кафедри ПМА,
Темнікова Олена Леонідівна _____

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

Київ – 2025 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет прикладної математики
Кафедра програмного забезпечення комп'ютерних систем

Рівень вищої освіти – перший (бакалаврський)

Спеціальність – 121 «Інженерія програмного забезпечення»

Освітньо-професійна програма «Інженерія програмного забезпечення мультимедійних та інформаційно-пошукових систем»

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Євгенія СУЛЕМА
«___»_____2024 р.

ЗАВДАННЯ

на дипломний проєкт студенту

Міщенко Владиславу Романовичу

1. Тема проєкту «Інтегроване середовище розроблення програмного забезпечення мовою ASAMPL 2.0. Компілятор», керівник проєкту Сулема Євгенія Станіславівна, завідувач кафедри ПЗКС, д.т.н., доцент, затверджені наказом по університету №1808-С від «29» травня 2025 р.

2. Термін подання студентом проєкту «13» червня 2025 р.

3. Вихідні дані до проєкту: див. Технічне завдання.

4. Зміст пояснювальної записки:
   - аналіз технологій розроблення компіляторів та мов програмування;
   - розробка інтерпретатору;
   - опис розроблених алгоритмів
   - аналіз розробленого застосунку;

5. Перелік обов'язкового графічного матеріалу:
   - схема роботи модуля інтерпретації (креслення);
   - схема роботи модуля синтаксичного аналізу (креслення);
   - робота інтерпретатора (плакат);
   - розбиття коду на вузли абстрактного синтаксичного дерева (плакат).

6. Консультанти розділів проєкту

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Онай М.В., доцент		

7. Дата видачі завдання «31» жовтня 2024 р.

Календарний план

№ з/п	Назва етапів виконання дипломного проєкту	Термін виконання етапів проєкту	Примітка
1.	Вивчення літератури за тематикою проєкту	13.11.2024	
2.	Розроблення та узгодження технічного завдання	22.11.2024	
3.	Проектування архітектури інтерпретатора	15.12.2024	
4.	Підготовка матеріалів першого розділу дипломного проєкту	28.12.2024	
5.	Розроблення алгоритмів для компонентів застосунку	01.02.2025	
6.	Підготовка матеріалів другого розділу дипломного проєкту	19.02.2025	
7.	Програмна реалізація інтерпретатора	11.03.2025	
8.	Тестування програмного застосунку	18.03.2025	
9.	Підготовка матеріалів третього розділу дипломного проєкту	26.03.2025	
10.	Підготовка матеріалів четвертого розділу дипломного проєкту	14.04.2025	
11.	Підготовка графічної частини дипломного проєкту	24.04.2025	
12.	Оформлення документації дипломного проєкту	27.05.2025	

Студент Владислав МІЩЕНКО

Керівник проєкту Євгенія СУЛЕМА

АНОТАЦІЯ

У дипломному проєкті розглянуто процес розробки інтерпретатора мови програмування ASAMPL 2.0, який є консольним застосунком, написаним мовою Java. Здійснено аналіз сучасних технологій створення трансляторів, зокрема лексичних і синтаксичних аналізаторів, а також розглянуто порівняльні характеристики мов програмування для реалізації трансляторів.

У ході реалізації проєкту створено основні компоненти інтерпретатора: лексер, парсер, модуль інтерпретації та структура абстрактного синтаксичного дерева (AST). Інтерпретатор підтримує базові типи даних, умовні оператори, цикли, функції та вбудовані методи для роботи з кортежами.

Також проведено інтегроване тестування розробленого програмного забезпечення, зосереджене на перевірці коректності збереження стану пам’яті та результатів обчислень. У підсумку сформульовано рекомендації щодо подальшого розвитку проєкту, включаючи розширення функціональності, оптимізацію обчислень і реалізацію графічного інтерфейсу.

ABSTRACT

This diploma project presents the development of an interpreter for the ASAMPL 2.0 programming language, implemented as a console application written in Java. The work includes an analysis of modern compiler construction technologies, particularly lexical and syntactic analyzers, as well as a comparison of programming languages suitable for building interpreters.

During the development, key components of the interpreter were implemented: the lexer, parser, interpretation module, and the structure of the abstract syntax tree (AST). The interpreter supports basic data types, conditional statements, loops, functions, and built-in methods for tuple manipulation.

Integrated testing of the software was conducted, focusing on the correctness of memory state handling and the accuracy of computations. As a result, recommendations for further development were formulated, including functionality expansion, optimization of expression evaluation, and the implementation of a graphical user interface.

ДП.045300-01-90 Інтегроване середовище розроблення програмного забезпечення мовою ASAMPL 2.0. Компілятор. Відомість проєкту

Позначення	Найменування	Кіл-ть	Примітка
	Документація проєкту		
ДП.045300-02-91	Інтегроване середовище розроблення	4	
	програмного забезпечення		
	забезпечення мовою ASAMPL 2.0.		
	Компілятор. Технічне завдання		
ДП.045300-03-81	Інтегроване середовище розроблення	54	
	програмного забезпечення		
	мовою ASAMPL 2.0. Компілятор.		
	Пояснювальна записка		
ДП.045300-04-51	Інтегроване середовище розроблення	4	
	програмного забезпечення		
	мовою ASAMPL 2.0. Компілятор.		
	Програма та методика		
	тестування		
ДП.045300-05-34	Інтегроване середовище розроблення	4	
	програмного забезпечення		
	мовою ASAMPL 2.0. Компілятор.		
	Керівництво користувача		
ДП.045300-06-99	Інтегроване середовище розроблення	1	
	програмного забезпечення		
	мовою ASAMPL 2.0. Компілятор.		
	Схема роботи модуля інтерпретації.		
	Діаграма діяльності		

Позначення	Найменування	Кіл-ть	Примітка
ДП.045300-07-99	Інтегроване середовище розроблення	1	
	програмного забезпечення		
	мовою ASAMPL 2.0. Компілятор.		
	Схема роботи модуля синтаксичного		
	аналізу. Діаграма діяльності		
ДП.045300-08-98	Інтегроване середовище розроблення	1	
	програмного забезпечення		
	мовою ASAMPL 2.0. Компілятор.		
	Компакт-диск		

Факультет прикладної математики

Кафедра програмного забезпечення комп'ютерних систем

"ЗАТВЕРДЖЕНО"

Завідувач кафедри

_____ Євгенія СУЛЕМА

"___" _____ 2024 р.

ІНТЕГРОВАНЕ СЕРЕДОВИЩЕ РОЗРОБЛЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ МОВОЮ ASAMPL 2.0. КОМПІЛЯТОР

Технічне завдання

ДП.045300-02-91

"ПОГОДЖЕНО"

Керівник проєкту:

_____ Євгенія СУЛЕМА

Нормоконтроль:

_____ Микола ОНАЙ

Виконавець:

_____ Владислав МІЩЕНКО

2024

ЗМІСТ

1. НАЙМЕНУВАННЯ ТА ГАЛУЗЬ ЗАСТОСУВАННЯ

Назва розробки: інтегроване середовище розроблення програмного забезпеченя мовою ASAMPL 2.0. Компілятор.

Галузь застосування: інформаційні технології.

2. ПІДСТАВА ДЛЯ РОЗРОБЛЕННЯ

Підставою для розроблення є завдання на дипломне проєктування, затверджене кафедрою програмного забезпечення комп'ютерних систем Національного технічного університету України «Київський політехнічний інститут імені Ігоря Сікорського» (КПІ ім. Ігоря Сікорського).

3. ПРИЗНАЧЕННЯ РОЗРОБКИ

Розробка призначена для використання в якості програмного інструменту, що забезпечує виконання коду мовою програмування ASAMPL 2.0. Розроблений застосунок може бути використаний для освітніх цілей, тестування концепцій мови та подальшого розширення синтаксису.

4. ВИМОГИ ДО ПРОГРАМНОГО ПРОДУКТУ

Компілятор повинен забезпечувати такі основні функції:

1) відтворення програм, написаних згідно з формальним описом синтаксису мови програмування ASAMPL 2.0;
2) підтримка вбудованих методів для виконання операцій в алгебраїчній системі агрегатів;
3) можливість відтворити програми у покроковому режимі дебагу, з моніторингом пам'яті і операцій.

5. ВИМОГИ ДО ПРОЄКТНОЇ ДОКУМЕНТАЦІЇ

У процесі виконання проєкту повинна бути розроблена наступна документація:

1) пояснювальна записка;
2) програма та методика тестування;
3) керівництво користувача;
4) креслення:
   – «Схема роботи модуля інтерпретації. Діаграма діяльності»;
   – «Схема роботи модуля синтаксичного аналізу. Діаграма діяльності».

6. ЕТАПИ ПРОЄКТУВАННЯ

Вивчення літератури за тематикою роботи..... 13.11.2024
Розроблення та узгодження технічного завдання 22.11.2024
Проектування архітектури інтерпретатора 15.12.2024
Розроблення алгоритмів для компонентів застосунку..... 01.02.2025
Програмна реалізація інтерпретатора..... 11.03.2025
Тестування програмного застосунку 18.03.2025
Підготовка матеріалів текстової частини проєкту 14.04.2025
Підготовка матеріалів графічної частини проєкту 24.04.2025
Оформлення технічної документації проєкту..... 27.05.2025

7. ПОРЯДОК ТЕСТУВАННЯ РОЗРОБКИ

Тестування розробленого програмного продукту виконується відповідно до “Програми та методики тестування”.

Факультет прикладної математики

Кафедра програмного забезпечення комп'ютерних систем

"ЗАТВЕРДЖЕНО"

Завідувач кафедри

_____ Євгенія СУЛЕМА

"___" _____ 2025 р.

ІНТЕГРОВАНЕ СЕРЕДОВИЩЕ РОЗРОБЛЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ МОВОЮ ASAMPL 2.0. КОМПІЛЯТОР

Пояснювальна записка

ДП.045300-03-81

"ПОГОДЖЕНО"

Керівник проєкту:

_____ Євгенія СУЛЕМА

Нормоконтроль:

_____ Микола ОНАЙ

Виконавець:

_____ Владислав МІЩЕНКО

2025

ЗМІСТ

СПИСОК ТЕРМІНІВ, СКОРОЧЕНЬ ТА ПОЗНАЧЕНЬ

Дебагінг – процес пошуку та усунення помилок у програмному коді. Покрокове контрольоване виконання коду програми.

Компілятор – програма, яка перетворює вихідний код на машинний.

Лексер – лексичний аналізатор, який перетворює вхідний текст на послідовність токенів.

Лог – журнал подій, повідомлення, які фіксують виконання програми.

Парсер – синтаксичний аналізатор, який будує синтаксичне дерево на основі токенів.

Репозиторій – місце зберігання програмного коду, його версій, документації та супровідних файлів.

Токен – слово, яке лексер виокремлює з вихідного коду.

Інтерпретатор – вид транслятора, який здійснює по-операторну (по-командну, порядкову) обробку.

AST (*Abstract Syntax Tree*) – Абстрактне синтаксичне дерево, що відображає синтаксичну організацію вихідного коду.

ASAMPL – (*Algebraic System of Aggregates and Mulsemedia data Processing Language*) – мова програмування, що зосереджена на працю із алгебрагічною системою агрегатів та мульсимедійними даними.

DSL (*Domain-Specific Language*) – предметно-орієнтована мова програмування, створена для вирішення завдань у певній галузі.

IDE (*Integrated Development Environment*) – інтегроване середовище розробки, що включає в собі редактор коду, інструменти для компіляції, дебагінгу та тестування.

Maven – система автоматизації збирання Java-проєктів і управління залежностями.

PyPI (*Python Package Index*) – центральний репозиторій програмного забезпечення для мови Python, де розміщуються бібліотеки.

ВСТУП

На нинішньому етапі розвитку інформаційних технологій важливе місце, не дивлячись на стрімкий розвиток технологій нейромереж, все ще займають мови програмування і прикладне написання коду. Саме це забезпечує подальший розвиток і створення ефективних, масштабованих та зручних у підтримці програмних систем. У науковій діяльності та навчальному процесі виникає необхідність у використанні спеціалізованих мов, які будуть адаптовані під конкретні завдання, будуть зручнішими. Однією із таких мов є ASAMPL 2.0 – мова алгебраїчної системи агрегатів та обробки мульсемедійних даних.

Незважаючи на існування специфікації вищезгаданої мови, на момент початку розроблення не існувало компілятора або інтерпретатора, здатного виконувати програми, написані останньою версією цієї мови.

Саме тому, метою даного дипломного проєкту є розробка модулю для інтегрованого середовища розроблення програмного забезпечення мовою програмування ASAMPL 2.0, а саме компілятора, із функцією дебагінгу. Цей модуль забезпечить підтримку мови, реалізує базові функції існуючого синтаксису, а також дозволить проводити трасування коду (покрокове виконання) для зручності налагодження програм.

У межах дипломного проєкту буде досліджено принципи побудови компіляторів, розроблено архітектуру інтерпретатора, реалізовано механізм поетапного виконання коду та здійснено тестування програмного засобу на прикладових задачах.

Отримані результати у вигляді початкового ядра мови програмування ASAMPL 2.0 можуть бути використані, як основа для подальшого розроблення повноцінного IDE.

1. АНАЛІЗ ТЕХНОЛОГІЙ РОЗРОБЛЕННЯ КОМПІЛЯТОРІВ ТА МОВ ПРОГРАМУВАННЯ

В цьому розділі буде розглянуто перелік технологій, які можуть бути доцільними у контексті розроблення даного програмного забезпечення, а саме компілятора. На даному етапі буде проведено аналіз готових рішень, які можуть спростити подальшу працю, обрано мову програмування, якою, власне, і буде написаний програмний застосунок, а також, після проведеного порівняння, буде відібрано потрібний для розроблення стек технологій.

1.1. Будова та види трансляторів

Почати варто із визначення, що таке компілятор, а також дослідження того, як вони працюють. Компілятор – програма-транслятор, яка перетворює вихідний код програми, написаний певною мовою, на машинний код, зрозумілий комп’ютеру. Попри єдину базову мету, трансляцію вихідного коду мовою програмування у форму, придатну до виконання, інструменти відрізняються за своїм призначенням і функціями. Умовно, їх можна розділити на такі два види, як власне компілятор та інтерпретатор [1].

Компілятор працює із програмою в цілому, перетворюючи її на виконуваний комп’ютерний код, двійковий файл з інструкціями для комп’ютера. Тобто головним завданням компілятора є перетворення вихідного коду мовою програмування високого рівня на мову нижчого рівня.

Інтерпретатор – це транслятор, який виконує набір інструкцій із програмного коду високого рівня, попередньо не компілюючи їх в машинний код. Це явище називається процесом інтерпретації – аналізу і виконання вихідного коду порядково, команда за командою.

Тобто, порівнюючи ці види трансляторів, можна побачити, що вони відрізняються, як за принципом роботи, так і за сценаріями використання. Нижче, у табл. 1.1 наведено порівняння цих видів програмних засобів.

Таблиця 1.1

Порівняння видів трансляторів

	Інтерпретатор	Компілятор
Визначення	Програмний засіб, що виконує вихідний код мовою високого рівня	Транслятор, який перетворює вихідний код мовою високого рівня на машинну мову
Приклади мов	Python, JavaScript, PHP	C++, Rust, Pascal
Використання	Проекти, де потрібна швидка розробка. Часто мови, які використовуються в інтерактивному середовищі.	Проекти, що націлені на високу продуктивність.
Принцип дії	Читання і виконання кожного рядку коду почергово	Читання всього коду програми і створення виконуваного файлу
Швидкодія	Інтерпретований код відтворюється повільніше за скомпільований	Відтворення виконуваних файлів швидше за інтерпретацію
Помилки	Вивід помилок по мірі надходження у покроковому виконанні	Про помилки повідомляється після компілювання

Отже, як можемо побачити, кожен із видів трансляторів має свої переваги і недоліки. Вибір типу транслятору залежить від поставленої задачі. У цьому проєкті було вирішено робити саме інтерпретатор, адже саме з ним вдасться реалізувати функцію дебагінгу – покроковому

виконанні коду із логуванням даних, змінних, які знаходяться у пам’ятті робочої машини, і як вона змінюється.

Перейдемо до структури інтерпретаторів. Вони так само бувають різними за будовою, але загально, у процесі перетворення вихідного коду у логічну послідовність дій беруть участь саме такі компоненти:

1. Лексичний аналізатор або лексер, виділяє із текстового файлу токени, згідно із описом синтаксису інтерпретованої мови. Простими словами, розбиває рядок на текстові елементи, слова, які мають свій контекстний опис, будь то назва змінної, типу даних, або структурної одиниці у коді.
2. Парсер – компонент, задачою якого є, використовуючи токени сформовані на етапі лексичного аналізу, сформувати так зване дерево, в якому ієрархія зв’язків між вузлами відображає логічну структуру коду. Таке дерево називається AST (Abstract Syntax Tree), або ж абстрактне синтаксичне дерево. Логіка зв’язків між нодами цього дерева залежить від синтаксичних правил цільової мови програмування. Наприклад, після розпізнавання токену типу даних, парсер очікує декларацію змінної або функції, а після декларації може очікувати кінець рядку, або ж операції присвоєння, математичного обчислення тощо. Тобто в парсері відбувається також синтаксичний аналіз, процес, під час якого відбувається перевірка на коректність послідовності токенів.
3. Останнім модулем є безпосередньо інтерпретатор. Задача цього компонента – покроково обходити абстрактне синтаксичне дерево, згори до низу, і виконувати операції, зазначені у вузлах цього дерева і у разі помилок, відображати логи про ці помилки у консолі.

Отже, тепер, знаючи принцип побудови інтерпретатора, можна продовжити аналізом готових рішень або технологій, які будуть необхідні при розробці цього програмного засобу.

1.2. Аналіз готових рішень

1.2.1. ANTRL

ANTLR (Another Tool For Language Recognition) – сучасний генератор парсерів, який дозволяє автоматично створювати лексер і парсер, тобто лесичний і синтаксичний аналізатори коду [2].

Це потужний інструмент, який налаштований на читання, обробку або переклад структурованого тексту або бінарних файлів. Перевагами ANTLR є підтримка дерев синтаксичного розбору, можливість створення власних правил обробки вузлів створених парсером вузлів дерева, а також наявність документації до інструменту, що полегшить розробку модулів розроблюваного компілятору [3].

Розглядаючи такий інструмент для розроблення модулів компілятору, можна зазначити, що, по-перше, ANTLR не підходить для задачі через обмежену підтримку, шаблонність вузлів синтаксичного дерева. Це значить, що у подальшій розробці буде складно додавати нові типи нод (вузлів) , які матимуть іншу логіку зв’язків, наприклад, іншу кількість зв’язків або ж рекурсивну логіку інтерпретування. Також, мінусом є факт того, що цей інструмент, як і переважна більшість генераторів коду, створює велику кількість проміжного коду, що негативно впливає на швидкодію створюваного компілятору.

1.2.2. JavaCC

JavaCC (Java Compiler Compiler) – це генератор синтаксичних аналізаторів для мови програмування Java. Цей інструмент дозволяє описати граматику безпосередньо у вигляді правил та коментарів, після чого генерує код мовою Java для парсингу [4].

Цей генератор стане у нагоді при написанні компіляторів для простих за складністю мов, якщо вся реалізація виконується саме Java мовою. Він дозволить швидко створити базову структуру мови [5].

Із недоліків, є те, що це рішення вважається застарілим, його можливості поступаються вищезгаданрму ANTLR у процесі обробки помилок або побудови AST. Також цей інстумент генерує надто великі обсяги шаблонного коду, факт наявності якого ускладнює ручне втручання у код, підтримку і масштабування проекту розробником.

1.2.3. PLY

PLY (Python Lex-Yacc) – бібліотека, написана для мови програмування Python, що підримує написання правил лексичного і синтаксичного аналізу у вигляді Python-функцій зі спеціальними іменами. Розбір елементів коду виконується безпосередньо у пам’яті, без генерації окремих файлів [6].

PLY підходить для побудови невеликих інтерпретаторів мов або DSL (domain-specific language) – мов програмування, спеціально розроблених для вирішення певного переліку прикладних задач [7].

Незважаючи на простоту використання, PLY побудований мовою програмування Python, що свідчить про недостатню строгість типізації, а також занизьку швидкодію, що необхідна для масштабованого інтерпретатора. Також, сам Python менш зручний для побудови складної архітектури із класами абстрактного синтаксичного дерева і обробкою їх у інтерпретаторі.

1.2.4. Порівняння інструментів

На цьому етапі буде проведено порівняння досліджених інструментів, і прийнято рішення із їх використання у розробці компілятора. Порівняння буде проведено із розглядом таких особливостей, як мова реалізації, підтримка абстрактного синтаксичного дерева, складність освоєння, гнучкість, а також після цього будуть вказані основні недоліки кожного із способів вирішення проблеми. Також до порівняння буде включено варіант розроблення модулів компілятора з нуля, власноруч. Суб’єктивну

характеристику, результати порівняння, проведеного автором роботи, можна переглянути у табл. 1.2.

Таблиця 1.2

Порівняльна характеристика інструментів для розроблення компілятора

Інструмент	Мова реалізації	Підтримка AST	Складність освоєння	Гнучкість
ANTLR	Java	Так	Середня	Низька (Обмежена шаблонами)
JavaCC	Java	Часткова	Висока	Середня
PLY	Python	Ні	Низька	Середня
Ручна реалізація	Буде визначено далі	Власна структура	Середня	Висока

Під час розбору технологій було визначено, що найголовнішим недоліком ANTLR є генерація складного коду, над яким непросто виконувати дебагінг, інструмент JavaCC є застарілим і незручним для інтеграції, PLY – займає більше часу на відтворення коду, та має погану підтримку строго визначених типів даних. Головним недоліком ручної реалізації є те, що весь компілятор доведеться писати самостійно, углиблюватися у логіку процесів всередині, досліджувати взаємодію модулів.

Отже, після проведення аналізу, перевагу було віддано ручній реалізації модулів компілятора, адже саме таким шляхом можна максимально досягти потрібної гнучкості, простоти реалізації, контролю над виконанням коду, а також розуміння структури написаного інтерпретатору.

Наступним кроком буде аналіз та визначення основної мови програмування, якою буде реалізовано проєкт.

1.3. Аналіз мов програмування

Далі будуть описані найліпші кандидати з мов програмування, які можуть бути використані під час написання компілятора. Після виокремлення їхніх переваг та недоліків, буде обрано мову програмування разом із набором бібліотек для реалізації проєкту.

1.3.1. C++

Першою мовою програмування, яка спадає на думку у контексті вибору мови для написання компілятора є C++. Це мова програмування загального призначення, яка забезпечує повний контроль над пам’яттю, ефективну роботу з ресурсами а тому, і високу продуктивність.

У списку програмного забезпечення, що було написано цією мовою програмування, можна побачити такі інструменти, як IDE Visual Studio, Microsoft Office, Adobe Photoshop, Illustrator, Premiere Pro [8].

Таким чином, можна зазначити, що ця мова гарно підходить до написання комплексних інструментних застосунків, через її високу швидкодію та повний контроль над ресурсами системи.

Із недоліків C++ можна виокремити високий поріг входу, адже управління пам’яттю вручну через відсутність автоматичного збору сміття та не типовий для інших мов синтаксис може ускладнити та затягнути розробку модулів компілятора.

1.3.2. Python

Наступною мовою для аналізу є Python. Це високорівнева мова програмування. Її зазвичай використовують для розроблення вебзастосунків, машиного навчання, роботи з великим обсягом даних. Основними принципами мови є простота, читабельність коду та мінімалізм.

Із технічних особливостей Python є динамічна типізація, у якій тип змінної визначається під час виконання, що пришвидшує процес розроблення, але впливає на ризик помилок у масштабних програмах.

Також Python має дуже розвинену екосистему. Репозиторій PyPI (Python Package Index) містить велику кількість бібліотек, націлених на виконання максимально різних задач [9].

Із його недоліків варто виокремити низбку продуктивність, тобто швидкість інтерпретованого коду поступається конкурентам. Також відсутність суворої типізації ускладнює написання великих, структурованих проектів, де важливими є чітка визначенність, цілісність даних.

Провівши дослідження, можна зазначити, що ця мова програмування пропонує користувачу простий синтаксис із швидким стартом і високою продуктивністю розроблення, разом із великою кількістю доступних бібліотек. Але не є оптимальним вибором через те, що не забезпечує користувача достатньою стабільністю та керованістю.

1.3.3. Java

Java – це мова загальнопризначена мова програмування із суворою типізацією, що орієнтована на об’єктно-орієнтоване програмування. Ця мова широко використовується для розроблення веб-застосунків, android рішень, серверного ПЗ. Одна із головних переваг цієї мови програмування міститься у її гаслі – “Write once, run anywhere”. Це забезпечується через архітектуру, побудовану на платформі Java Virtual Machine. Тобто, якщо для певної платформи існує створена для неї JVM, то будь який код, написаний мовою Java і перетворений на байт-код, може бути відтвореним на цій платформі [10].

Із технічних особливостей мови виділяється типобезпечність, автоматичне керування пам’яттю, та високий рівень абстракції. Великою перевагою також є дуже розвинена стандартна бібліотека мови, яка містить у собі перелік різноманітних класів колекції, Input/Output класів, а також інструментів для роботи із виключеннями.

Аналізуючи Java і порівнюючі її із вищезгаданими мовами програмування, було знайдено такі переваги, як простота синтаксису і

структури коду, у порівнянні із C++ і в той самий час сувора типізація і вищій рівень абстракції у порівнянні із Python. Із недоліків було визначено необхідність написання великої кількості шаблонного коду, навіть для простих задач, а також дещо повільніший запуск у порівнянні з кодом C++, через час, який займає завантаження віртуальної машини.

1.3.4. Порівняння мов програмування

Для того, щоб визначитися із вибором мови програмування для проєкту, потрібно виокремити критерії відбору, які є важливими у контексті написання даного проєкту, компілятору мови ASAMPL 2.0.

Такими критеріями є продуктивність мови, тобто швидкодія самої програми, простота синтаксису і швидкість розроблення, типізація, а також контроль над виконанням коду, можливість дебагінгу та опрацювання виключень. Результати порівняння можна побачити у табл. 1.3.

Таблиця 1.3

Порівняльна характеристика мов програмування

Критерій	C++	Python	Java
Продуктивність	Компіляція у машинний код забезпечує максимальну швидкодію.	Інтерпретована мова, що сповільнює виконання.	JVM витрачає більше часу на запуск, але JIT-компіляція покращує швидкодію
Простота синтаксису	Складний синтаксис і ручне керування пам’яттю	Лаконічний, інтуїтивно зрозумілий синтаксис	Суворий, але зрозумілий синтаксис із деякою багатослівністю

Продовження табл. 1.3

Контроль над виконанням	Є повний доступ до пам’яті, низькорівневих операцій	Автоматичне керування пам’яттю, спрощена модель виконання	Контроль за потоками, винятками, ресурсами
Типізація	Статична	Динамічна	Статична
Швидкість розроблення	Висока складність синтаксису уповільнює процес.	Швидка розробка завдяки простоті синтаксису та бібліотек	Синтаксис більш формальний, громіздкий, але простий для розуміння і швидкої адаптації розробника

Отже, порівнявши ці три мови програмування, вибір було зроблено у сторону відносно легшого синтаксису, але при наявному балансі між швидкодією написаного застосунку і прийнятним рівнем контролю над виконанням. Було обрано мову програмування Java.

1.4. Обгрунтування обраних технологій

Після проведеного аналізу готових рішень і мов програмування, на цьому етапі буде визначено, які ж технології буде використано у написанні інтерпретатору. Для того, щоб зробити вибір, треба визначитися з вимогами до програмного продукту.

Застосунок має забезпечувати користувача такими можливостями, як:

1. Відтворення коду, написаного згідно з формальним описом синтаксису мови ASAMPL 2.0.
2. Повідомлення користувача про виниклі виключення, під час відтворення коду програми.
3. Відлагодження, тобто дебагінг коду, покрокове виконання програми із відображенням інформації про стан пам’ятті програми.

Таким чином, було вирішено створити транслятор-інтерпретатор, для того, щоб проєкт виконував вимогу, що стосується покрокового відлагодження коду. Мову програмування для написання інтерпретатору було обрано саме Java, адже цей вибір – компроміс між швидкодією та суворою типізацією, що неймовірно важливо під час написання цього інструменту.

Для тестування програмного продукту буде використано бібліотеку Java JUnit – бібліотеку мови Java, яка дозволить написати і відтворити ізольовано, як модульні, так інтегровані тести компонентів написаного застосунку.

Для підключення бібліотеки буде використано фреймворк для автоматизації зборки проектів – Apache Maven.

2. РОЗРОБКА ІНТЕРПРЕТАТОРУ

На цьому етапі буде спроектовано архітектуру майбутнього інтерпретатору. Проектування відбуватиметься згідно з даними вимогами до дипломного проєкту, а також принципами побудови трансляторів. Після того, як архітектуру буде визначено, наступним кроком буде написання самого застосунку із подальшим написанням автоматизованого тестування продукту.

2.1. Аналіз вимог до проєкту

Створений інструмент повинен підтримувати такі функції:

1. Відтворення файлів з кодом, написаним мовою ASAMPL 2.0.
2. Відтворення програми у режимі дебагінгу, покрокове виконання із моніторингом пам’яті програми.
3. Логування помилок у зчитуваному коді, під час виконання програми.

Тобто, застосунок має бути повноцінним функціонуючим транслятором, який читає файл з кодом ASAMPL 2.0 і перекладає його на мову, зрозумілу комп’ютеру. Для зручності розроблення застосунків вищезазначеною мовою, інтерпретатор повинен мати передбачений сценарій дебагінгу програми. Під час виключень, що виникають під час відтворення коду, користувач має бути проінформованим про ці виключення.

Із додаткових вимог, застосунок повинен бути протестованим, щоб кожен сценарій поведінки, який передбачений кодом, мав перебачуваний результат.

2.2. Архітектура інтерпретатора

Як було визначено раніше, обраний тип транслятора, який відповідає поставленим вимогам – інтерпретатор. Перейдімо до будови інтерпретатора. Найголовнішими, найбільшими компонентами системи є лексичний аналізатор, він же лексер, синтаксичний аналізатор (парсер), і, власне, механізм виконання коду – інтерпретатор. Важливу роль роботи такої системи відіграють об’єкти, які передають інформацію із одного модулю в інший. Такими об’єктами є токени із лексера і абстрактне синтаксичне дерево із парсера. Принцип переходу даних в такій системі можна розглянути на рис. 2.1.

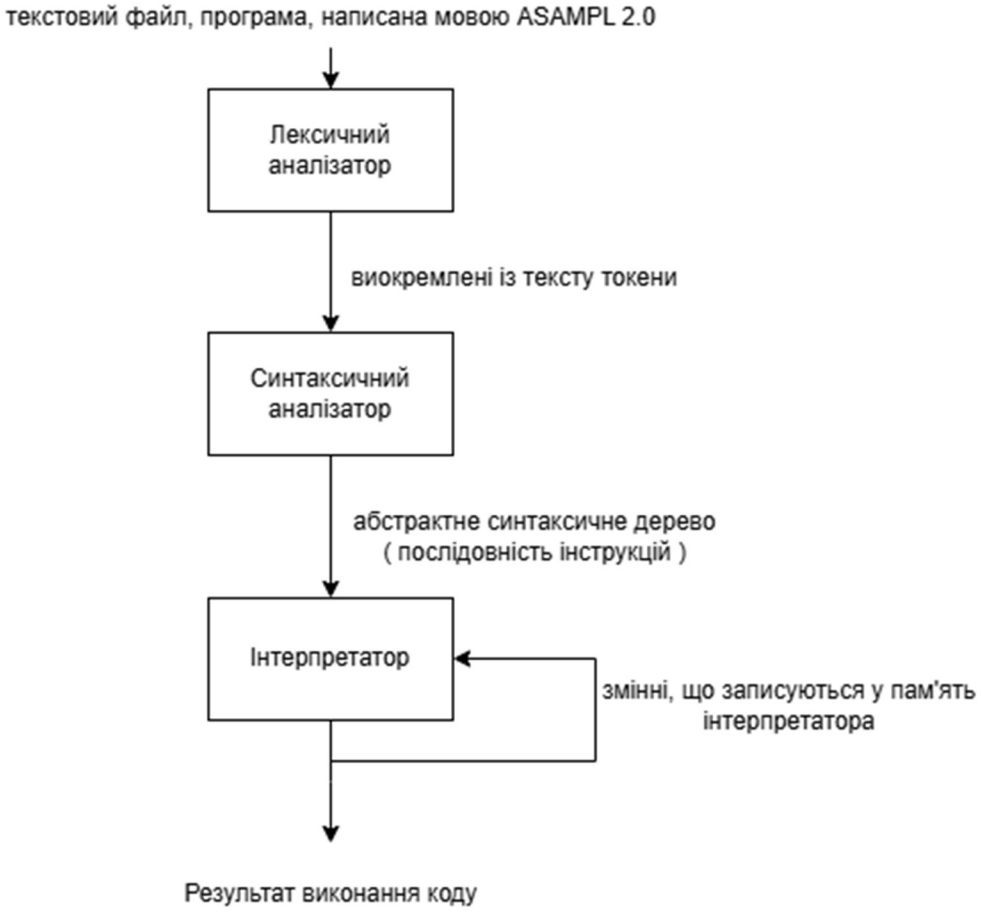


Рис. 2.1. Умовна схема роботи інтерпретатору

Як можна побачити, у лексері і парсері відбувається лише початкова підготовка операцій, зазначених у вхідному коді. Всі інструкції, що

стосуються саме виконання коду, відбуваються у модулі інтерпретації. Саме в цьому модулі відбувається запис, зміна, видалення змінних, логічні операції та математичні обчислення. Прицнип дії кожного із модулей буде окремо розглянуто у наступних розділах.

2.3. Лексер

Лексер – модуль інтерпретатору, який перетворює певний масив тексту у набір токенів. Якщо перейти до аналогій, то будь яке правильно побудоване речення можна розбити на слова та розділові знаки. Так само лексер розбиває код на менші структурні одиниці, токени. Об’єкти токенів містять у собі текстове представлення слова, що цей самий токен формує, а також значення типу токена, будь це тип даних, назва змінної, числове значення тощо.

Перш ніж проектувати лексичний аналізатор, якій працює з визначеним переліком типів токенів, потрібно визначитися із можливими конструкціями, які містяться у формальному описі синтаксису мови ASAMPL 2.0, щоб сформувати список можливих токенів лексеру.

2.3.1. Граматика мови ASAMPL 2.0

Нововведенням, яке з’явилося у другій версії мови, порівняно з першою стала імперативність. Імперативність мови значить те, що програма із себе становить послідовність команд, які змінюють стан. В більшості таких мов, як і в ASAMPL 2.0 синтаксис є Сі-подібним. Сі-подібний синтаксис має такі характерні риси, як:

1. Блоки коду у фігурних дужках.
2. Інструкції розділяються крапкою з комою.
3. Мова має конструкції, наприклад if, while, for, switch, return.

Таким чином, перейдемо до ключових слів вищезгаданої мови ASAMPL 2.0, розпізнавання яких буде реалізовано у модулі лексичного аналізу. Види конструкцій можна переглянути у табл. 2.1.

Таблиця 2.1

Синтаксичні елементи мови програмування ASAMPL 2.0

Конструкція або елемент	Види	Приклади
Числове значення	Ціле число, десятковий дріб	0, 1, 1.1, 5.3
Назва змінної	Назва змінної, назва функції	a, b, sum, printA
Булеве значення	Істина, фальш	true, false
Часове значення	Відносний час (hh:mm:ss:ms)	23:30:15:96
Логічні оператори	Більше, менше, дорівнює, більше-дорівнює, менше-дорівнює, або, та, ні, не дорівнює	>, <, <=, >=, ==, !=, !, ||, &&
Математичні оператори	Присвоєння, плюс, мінус, множення, ділення, приведення до степеня	=, +, –, , *, /, ^
Дужки	Круглі, квадратні, фігурні	(,), {, }, [,]
Тип даних	Ціле число, дробове, текст, булеве значення, час, кортеж, агрегат, пустота	int, float, string, boolean, time, tuple, aggregate, void
Декларація логічного блоку	Якщо, інакше якщо, інакше	if, elif, else
Декларація блоку циклу	Конструкція з умовою, конструкція з ітератором	while, for
Оператор повернення	Повернення із функції	return
Вбудовані функції мови	Функції для роботи з кортежами, функція виводу змінної	print, uni, sec, dif, sdif, xsec, ord, asort, dsort, singl, extr, ins, get
Текстове значення	Текстове значення	“text”, “example”
Плаваюча крапка	Крапка, для позначення дестякового дробу	.

Тепер, знаючи, які конструкції і принципи побудови команд очікувати у вхідному коді програми, можна переходити до планування побудови лексичного аналізатору.

2.3.2. Принцип дії лексичного аналізатора

Перейдемо до алгоритму, завдяки якому можна буде визначати токени, і прокручувати їх один за одним. Для цього треба покроково, по одному символу розглядати код і виокремлювати існуючі, очікувані конструкції. Так як нам важливо зберігати позицію токена, який ми знайдемо, було вирішено зробити лексер за ітеративним шаблоном. Припустимо, ми маємо метод, який будемо викликати кожен раз, коли нам знадобиться наступний токен, то його можна використати у циклі парсера, постійно запрошуючи нову лексему, допоки не дійдемо до кінця файлу. Для того, щоб дізнатися, чи дійшов процес до кінця коду, буде створено метод перевірки Принцип взаємодії такого лексера із парсером можна побачити на рис. 2.2.

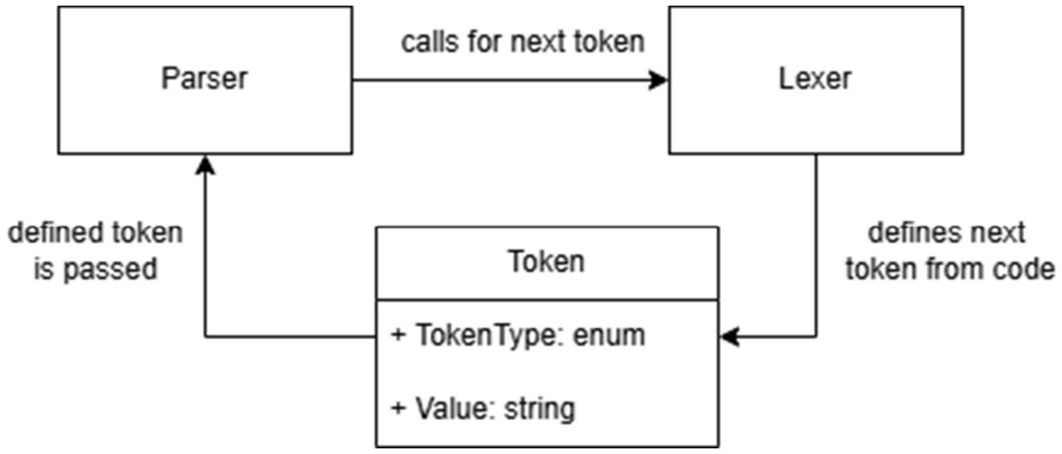


Рис. 2.2. Умовна схема взаємодії лексера і парсера

Проблему відокремлення різних типів токенів буде вирішено таким чином – спочатку, при кожному виклику умовного методу *nextToken* варто пропустити всі пробіли і переноси рядку, аж поки ми не досягнемо символа. Далі треба виконати перевірку на токени, які містять в собі лише один

символ. Такими лексемами можуть бути математичні операції, крапка з комою, логічні операції тощо. Якщо алгоритм знайшов відповідність – повертає новостворений токен певного типу, інакше далі йде перевірка на змінну або число.

Перевірка на змінну читає всі наступні символи після першого спрацювання, записуючи їх у буфер, допоки не дійде до символу, що не може міститися у назві, крапки з комою, пробілу чи переносу рядку. Далі прочитане слово порівнюється із ключовими синтаксисними словами, щоб перевірити, чи є ця лексема декларацією типу даних, конструкції циклу, перевірки тощо. Якщо ні, то алгоритм повертає токен назви змінної, інакше – токен ключового слова, яке визначила перевірка.

Останньою дією в методі має бути перевірка на числове значення. Працюватиме вона так само, як із назвами змінних, але перевірятимуться лише символи цифр. Також всередині визначення чисельного значення варто очікувати зустріти крапку, таким чином, маючи змогу розпізнати десятковий дріб і повернути його у вигляді токена.

Якщо жодна із перевірок не поверне результата, а ітерації не дійшли до кінця файлу, варто повернути виключення, адже лексер не зміг розпізнати токен, що є неочікуваною поведінкою для компонента. Це означатиме, що користувач ввів непередбачену конструкцію або символ.

2.4. Парсер

Задача модуля синтаксичного аналіза, або ж парсера, полягає в тому, щоб із послідовності отриманих з лексера токенів створити абстрактне синтаксичне дерево.

AST дерево міститиме у собі перелік інструкцій, які повинен буде виконати інтерпретатор. Такі інструкції, що йдуть одна за одною, називаються вузлами абстрактного синтаксичного дерева. Після того, як парсер розпізнає певну граматичну конструкцію, він створює відповідний вузол, який відтворює цю конструкцію. Цей процес відбувається

рекурсивно, тобто вузол може мати підвузли, і в залежності від типу інструкції, їх може бути різна кількість. Приклад умовних типів вузлів можна побачити в табл. 2.2.

Таблиця 2.2

Умовні типи вузлів абстрактного синтаксичного дерева

Вузол	Вміст
Бінарна операція	Ліва частина, оператор, права частина
Операція присвоєння	Змінна, вираз для присвоєння
Конструкція з умовою (if)	Вираз умови, тіло конструкції (список інструкцій)
Виклик функції	Ім’я функції, список аргументів

Дерево містить в собі певну кількість визначених вузлів, які можуть мати посилання на один, два, три вузли. Також, існують вирази-блоки, які мають містити в собі список внутрішніх інструкцій, які виконуються відповідно з внутрішньою умовою виконання блоку. Парсинг дерева відбувається рекурсивно, що дозволяє мати нескінченну ієрархію вузлів дерева, а це дозволяє парсеру описати вихідний код програми будь якої складності і відобразити його в AST Щоб полегшити сприйняття такого дерева, далі буде наведено графічний приклад результату процеса формування абстрактного синтаксичного дерева певного рядку коду. Розбиття рядку на необхідну кількість у послідовності інструкцій з рекурсивним розбиттям аргументів цих інструкцій. На рис. 2.3 показано, як умовно відбувається формування абстрактного синтаксичного дерева для певної операції присвоєння.

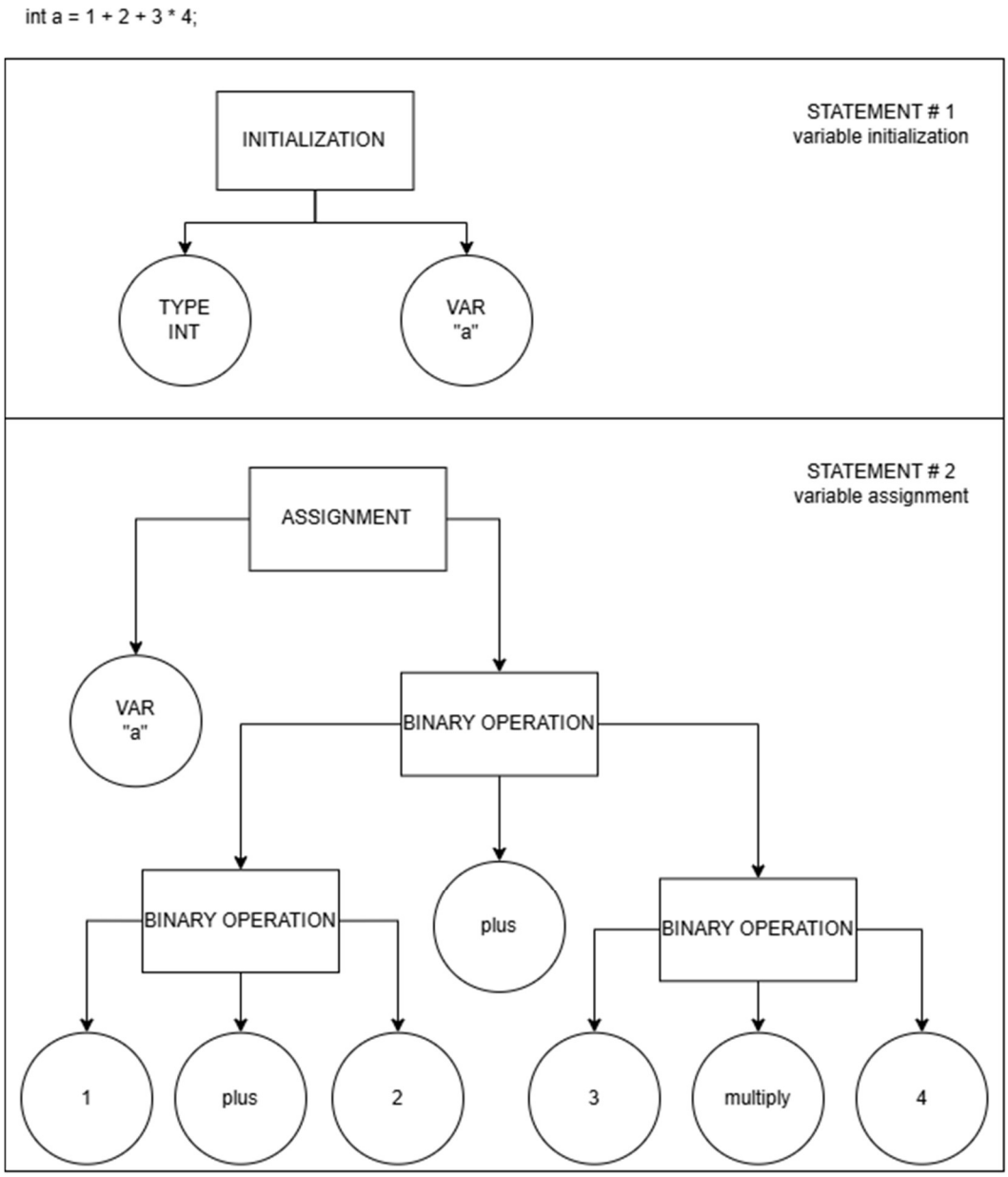


Рис. 2.3. Умовна схема формування абстрактного синтаксичного дерева

Як можна побачити, рядок коду після обробки за логікою синтаксичного аналізатора, був розбитий на два твердження:

1. Операцію ініціалізації змінної цілочисельного типу у пам’ять, де лівою гілкою є тип змінної, а правою – назва.
2. Операцію присвоєння значення, де назва змінної ліворуч, а праворуч вираз бінарної операції, що рекурсивно містить в собі ще два вирази бінарних операцій.

Варто зазначити, що обчислення кожного бінарного твердження відбувається рекурсивно знизу догори, якщо мова йде про бінарні операції, таким чином операції будуть відповідні пріорітетам. Тобто, наприклад, першим чином будуть відбуватися мультиплікативні, а потім вже адитивні операції у виразах.

Формування вузлів контейнерів у парсері повинно відбуватися рекурсивно, тому як блок з умовою, або ж блоки циклів мають містити в собі всередині одну або більше інструкцій, які так само можуть бути контейнерами.

2.5. Інтерпретатор

Сформоване AST дерево потрапляє у інтерпретатор – модуль, задача якого ітеративно пройти кожне твердження дерева програми, виконуючи інструкцію за інструкцією.

Цей компонент також повинен містити у собі реалізацію пам’яті програми. Це можна досягти, створивши дві колекції, що міститимуть в собі дані про збережені змінні і функції. Приклад умовної одиниці кожної з колекцій можна переглянути у табл. 2.3.

Таблиця 2.3

Умовні одиниці колекцій пам’яті інтерпретатора

Тип	Дані, що містяться в об’єкті			
Програмна змінна	Назва	Тип змінної	Значення	
Функція	Назва	Тип повернення	Список інструкцій	Список аргументів

Також, при розробці модулю інтерпретації, увагу слід приділити області видимості змінних значень. Змінна, що оголошена всередині блоку, конструкції, не повинна бути досяжної із більш відкритих ділянок коду. Так само назви змінних, що знаходяться всередині оголошеної функції, не

повинні асоціюватися інтерпретатором із попередньо оголошеними змінними, або змінними, що будуть ініціалізовані після.

Таким чином, маючи коректно працюючу пам’ять інтерпретатору, можна приступати до реалізації зчитування і обходу абстрактного дерева, інструкції якого будуть вносити зміни у пам’ять, змінюючи стан програми.

Важливою частиною модуля інтерпретації є перевірка доступності вказаних операцій у дереві. Для цього слід перевіряти кожну операцію на сумісність типів. Саме таким способом можна досягти суворої типізації мови, що зменшить кількість помилок, які отримує користувач-розробник, під час написання коду.

Інтерпретатор – найбільший за обсягом модуль транслятора, тому правильна обробка і логування виключень відіграє велику роль у побудові цього компонента.

Також, саме цей модуль потребує ретельного тестування на правильність відпрацювання функцій, передбачених мовою ASAMPL 2.0, бо саме він є основним логічним модулем виконання інструкцій.

3. ОПИС РОЗРОБЛЕНИХ АЛГОРИТМІВ

У цьому розділі будуть описані найважливіші алгоритми, які були створені під час розроблення проєкту. Вищезгадані алгоритми є центральною частиною програмної реалізації модулів інтерпретатора та визначають його поведінку під час обробки файлів із вихідним кодом мовою ASAMPL 2.0.

Також, особливу увагу приділено формуванню абстрактного синтаксичного дерева, а також механізму обробки і виконання команд на основі цього дерева.

Крім того, розділ міститиме опис проблем, які виникали під час розроблення, та шляхи, якими їх було вирішено.

3.1. Виокремлення токенів із файлу з кодом

Першим етапом, при запуску процесу інтерпретації, свою роботу розпочинає компонент лексичного аналізу. На вхід він отримує змінну текстового типу, попередньо сформовану із наданого файлу, що містить у собі повний код програми.

Вилучення токенів відбувається покроково, один за одним, за допомогою головного методу nextToken. Це зроблено для того, щоб завжди мати в доступі порядкову позицію вилученого токена, розбираючи код поступово, при виклику вищезазначенного методу у модулі синтаксичного аналізу.

Для того, щоб дізнатися, чи дійшов процес до кінця файлу, використано метод isEndOfCode. Він звіряє порядковий номер останнього розглянутого токена із кількістю символів у коді і повертає відповідний результат.

Маючи ці два публічні методи, nextToken та isEndOfCode, компонент парсер не матиме проблем з тим, щоб циклом прогортати весь список

вилучених лексем і на основі їх типу та порядкового індекса сформувати вузли синтаксичного дерева.

Далі детальніше розберемо логіку розпізнавання токенів у текстовому файлі. Список створених типів токенів можна побачити на рис. 3.1.

```
1  package ua.kp13.mishchenko;
2
3  public enum TokenType {
4      NUMBER, VARIABLE, STRING, FALSE, TRUE, TIME_VALUE,
5      OPER_AND, OPER_OR, OPER_NOT, OPER_IS, OPER_LESS, OPER_MORE,
6      OPER_LESS_EQUALS, OPER_MORE_EQUALS, OPER_IS_NOT,
7      OPER_EQUALS, OPER_PLUS, OPER_MINUS, OPER_MULTIPLY, OPER_DIVISION, OPER_POWER,
8      TYPE_INT, TYPE_STRING, TYPE_FLOAT, TYPE_BOOLEAN, TYPE_TIME, NEXT_LINE,
9      TYPE_TUPLE, TYPE_AGREGATE,
10     FLOATING_POINT, DOUBLE_QUOTES,
11     BRACKET_CURLY_OPENED, BRACKET_CURLY_CLOSED, BRACKET_OPENED, BRACKET_CLOSED,
12     IF, ELIF, ELSE,
13     WHILE, FOR,
14     RETURN, FUNCTION, VOID, COMMA, TWO_DOT,
15     BRACKET_SQUARE_OPENED, BRACKET_SQUARE_CLOSED
16 }
17
```

Рис. 3.1. Список існуючих типів токенів інтерпретатора мови ASAMPL 2.0

Першочергово відбувається перевірка текста на односимвольні типи токенів, такі як оператори OPER_PLUS, OPER_MINUS, скибки тощо. При неоднозначності конструкції, наприклад, якщо ми маємо OPER_MORE та OPER_MORE_EQUALS, то згідно з алгоритмом, перевіряється наступний символ, так можна точно визначити, на який тип токена посилатися.

Після перевірки текста на вищезазначені конструкції, лексер перевіряє чи не є поточний символ літерою. Якщо так, то запускається процес визначення назви змінної. Лексер записує у буфер всі літери і цифри допоки не наштовхнеться на пробіл, перенос рядку, або ж інший спеціальний символ. Щоб остаточно впевнитися, що було визначено саме назву змінної, відбуваються перевірки конструкції у буфері із ключовими фразами синтаксису мови. У разі співпадіння токену назначається відповідний тип і значення, інакше токен є назвою змінної.

Передостаннім випадком того, чим може бути розглянутий токен – є числове значення. Якщо поточний символ є цифрою, то викликається метод

визначення числа. Принцип дії цього методу є аналогічним читанню назви слова, але дещо відрізняється. Окрім цифр у числі також передбачено появу плаваючої крапки, для відображення дробу. Також в цьому методі працює механізм розпізнавання шаблону часового проміжку. Тобто, якщо конструкція містить у собі двокрапки, і відповідає паттерну часового типу значення, то такий токен буде розпізнано як час.

І останній варіант розвитку подій – жодна з перевірок не повернула позитивний результат. У такому випадку лексер повертає виключення, адже не зміг визначити і виокремити поточний токен. Слід зауважити, що при такому сценарії застосунок зупинить свою роботу, адже не зможе провести синтаксичний аналіз коду і інтерпретацію відповідно.

3.2. Створення AST при синтаксичному аналізі коду

Як вже було зазначено раніше, синтаксичний аналіз коду формує логічну послідовність дій зрозумілу інтерпретатору, абстрактне синтаксичне дерево із набору лексем.

Цей процес можна порівняти із створенням сенсу певній комбінації слів, побудову правильного речення, маючи на меті донести якусь думку сказаним. Як в побудові речень людськими мовами, у мовах програмування є теж свої правила і принципи послідовності слів (лексем) у реченні. Саме так, передбачаючи всі можливі наступні фрази, можна побудувати парсер для інтерпретатору, який буде надавати цим наборам токенів контекстуальне забарвлення, певну значимість.

Так як мова ASAMPL 2.0 є імперативною, тобто код являє собою послідовність інструкцій, які виконуються одна за одною, то AST на нижніх рівнях виглядатиме як колекція непов’язаних між собою але послідовно пронумерованих інструкцій, а чим вище по цим інструкціям буде проходити інтерпретатор, тим більше вони будуть рекурсивно розгалуджуватися на окремі прості операції. Сформовані типи вузлів дерева можна розглянути у табл. 3.1.

Таблиця 3.1

Типи вузлів абстрактного синтаксичного дерева

Тип вузлу	Внутрішні дані, посилання	Опис
InitializationNode	String variableName Token variableType Node expression	Операція ініціалізації змінної із внутрішнім виразом
AssignmentNode	String variableName Node expression	Операція присвоєння значення змінній із внутрішнім виразом
BinaryOperationNode	Node left Node right Token operator	Бінарна операція з двома аргументами та оператором
LogicalOperationNode	Node left Node right Token operator	Логічна операція з двома аргументами і оператором
WhileLoopNode	Node runCondition List<Node> statementsList	Конструкція циклу із умовою виконання та списком внутрішніх інструкцій
VariableNode	Token name	Посилання на змінну
TupleNode	TokenType type List<Node> values	Колекція, кортеж, що має тип та масив внутрішніх значень
TimeNode	Token value	Часовий вираз
StringNode	Token value	Текстовий вираз
ReturnNode	Node expression	Операція повернення значення із блоку

Продовження табл. 3.1

ProgramNode	List<Node> statements	Початковий вузол абстрактного дерева. Містить посилання на всі внутрішні інструкції
NumberNode	Token value	Числовий вираз
FunctionNode	VariableNode name Token returnType List<VariableEntry> arguments List<Node> innerStatements	Вираз ініціалізації функції. Блок. Має тип повернення, назву, перелік аргументів та список внутрішніх операцій
FunctionCallNode	VariableNode name List<Node> arguments	Операція виклику функції. Має посилання на список аргументів і назву
ForLoopNode	Node runCondition Node step Node counter List<Node> statementsList	Конструкція циклу із умовою виконання, ітератором, кроком і списком внутрішніх інструкцій
ConditionNode	TokenType type Node expression List<Node> innerStatements	Конструкція умови. Має тип (IF, ELSE IF, ELSE) і список внутрішніх інструкцій
BooleanNode	Token value	Вузол булевого значення
AgregateNode	List<Tuple> values	Вузол значення агрегату. Має масив кортежів всередині

Певні із визначених вузлів, як можна побачити, мають посилання на інші вузли всередині себе. Завдяки цьому і формується зрозуміла

інтерпретатору послідовність інструкцій, адже кожен вузол має свій власний тип, в залежності від якого, будуть виконуватися відповідні операції.

Перейдемо до формування дерева. Першим об’єктом парсер створює контейнер ProgramNode. Цей вузол всередині себе вміщатиме інструкції написаної програми.

Наступним кроком буде аналіз порядку токенів з коду, попередньо розібраного лексером. Логіка огляду токенів парсера дещо відрізняється від формування їх лексером. Лексичний аналізатор запам’ятовує теперішню і попередню лексему, тоді як парсеру важливо знати про поточний і наступний токени. Таким чином синтаксичний аналізатор зможе перевіряти пари лексем, і робити висновки з приводу того, що за інструкція може вміщати собі цю пару ключових слів у визначеному порядку.

Вузол програми може містити в собі такі операції, як:

1. Ініціалізація (декларація) змінної.
2. Присвоєння значення змінній.
3. Декларація функції.
4. Виклик функції.
5. Конструкції циклу.
6. Конструкції умови.
7. Повернення значення.

Слідуючи з цього, першим чином відбувається перевірка послідовності токенів на співпадіння із синтаксичними правилами цих конструкцій.

Наприклад, щоб парсер розпізнав ініціалізацію змінної, першим токеном послідовності повинен бути тип даних. Після типу даних має бути назва змінної, потім, наступним токеном може бути як символ присвоєння значення і вираз після, так і крапка з комою. Тому тут відбувається розгалудження і передбачаються всі можливі випадки формування твердження. Якщо ж твердження не вдалося сформувати по очікуваним

правилам, то викликається виключення, яке зазначає, що відбулася помилка під час парсингу токенів програмного коду.

Варіацій правильних розташувань токенів, які формуватимуть логічне, синтаксично правильне твердження – безліч, і глибина вкладоності цих вузлів залежить від самого масштабності коду програми. Тому синтаксичний розбір необхідно проводити рекурсивно.

Із рекурсивно опрацьовуваних вузлів-операцій можна виділити вирази з бінарними та логічними операціями. Принцип їх дії заключається в постійній рекурсії і розділенні математичного або логічного виразу на менші частинки, які міститимуть в собі посилання на наступні операції, із збереженням пріорітету операцій, допоки цей процес рекурсії не дійде до констант, або ж аргументів виразу.

Описуючи алгоритм обробки виразів, треба зазначити, що він, як раніше було зазначено, реалізує рекурсивний парсинг арифметичних і логічних виразів мови. Алгоритм обробки арифметичних виразів будує дерево за таким сценарієм:

1. Головний метод parseExpression запускає процес побудови дерева виразів. Він викликає parsePriorOper для обробки операцій вищого пріорітету (множення, ділення, піднесення до степеня). Потім у циклі обробляє операції нижчого пріорітету (додавання, віднімання). На кожному кроці створюючи новий вузол BinaryOperationNode, де поточний вираз стає лівим піддеревом, а новий операнд – правим.
2. Метод parsePriorOper виконує розбір операцій вищого пріорітету. Спочатку викликаючи parseArgs для обробки елементарних частин виразу (число, змінна, вираз у дужках). І у циклі обробляє оператори вищого пріорітету, створюючі наступні BinaryOperationNode.
3. Метод parseArgs виконує обробку елементарних виразів. Якщо токен є константою, то створюється відповідний вузол значення

(StringNode, NumberNode), якщо токен є назвою змінної, то повертається VariableNode. Якщо замість очікуваного аргументу було отримано відкриваючу круглу дужку, то рекурсивно викликається метод із першого пункту і після отриманого результату очікується наявність закриваючої дужки. Якщо ж токен неочікуваний, то повертається виключення парсеру.

Приклад результату роботи вищеописаного алгоритму по розбору послідовності токенів математичного виразу і відповідну побудову AST можна побачити на рис. 3.2.

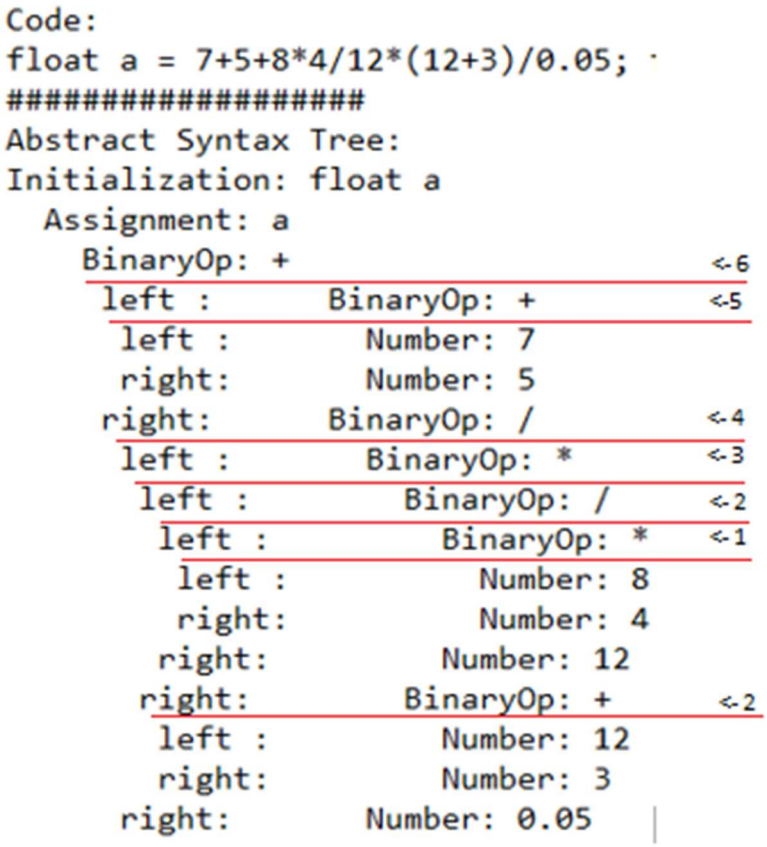


Рис. 3.2. Абстрактне синтаксичне дерево сформоване, як результат обробки парсером математичного виразу

На малюнку рівень табуляції визначає рівень вкладеності операції. Пріорітет математичних операцій, виконуваних інтерпретатором при обході

даного вузла йде із найвкладенішої до початкової. Числами праворуч вказано порядок обрахування операцій.

Аналогічно працює парсинг логічних виразів, рекурсивно обробляючи логічні вирази і формуючи для них AST дерево.

Інші ж інструкції мають більш просту механіку парсингу. Замість математичного виразу у операції присвоєння може бути просте значення, або ж посилання на іншу змінну.

Лише у випадку, коли всі передбаченні формальним описом мови варіанти послідовності були перевірені на співпадіння із поточним порядком токенів, і як результат, парсер не знайшов відповідності жодній з конструкцій, то повертається виключення, що повідомляє про неочікуваний токен у коді, або відсутність наступного взагалі.

3.3. Опрацювання інструкцій модулем інтерпретації

Модуль інтерпретації відіграє ключову роль у виконанні програмного коду мовою програмування ASAMPL 2.0. Його основна задача – покрокове опрацювання абстрактного синтаксичного дерева, яке формується на етапі парсингу. Інтерпретатор виконує команди, інструкції програми, відповідно до структури дерева, послідовно обробляючи кожен вузол.

Опрацювання інструкцій відбувається рекурсивно. Деякі вузли дерева, наприклад конструкція з умовою, цикли або виклики функцій можуть містити вкладені послідовності команд із взаємодію з пам’яттю, які потребують окремого виконання, із ізольованим контекстом програми. Кожна команда відбувається в рамках контексту виконання. Контекст – це набір змінних, які є доступними на даному рівні вкладенності. Так, наприклад, локальна змінна, яку було створено у блоці умови, не буде доступною після закінчення даного блоку. Це є можливим через змінну-індикатор в основному методі модулю, яка відображає, чи є перелік інструкцій вкладеними в якийсь певний блок. Під час виконання інструкцій в такому режимі, інтерпретатор запам’ятовує змінні, які були створені у

вкладених конструкціях, і відповідно видалить ці тимчасові змінні, після виконанання всіх внутрішніх операцій.

Процес інтерпретації починається обробкою початкових інструкцій з верхнього рівня. Як було визначено раніше иакими операціями є ініціалізація змінних та функцій, присвоєння значення, умовні конструкції, цикли та виклик створених функцій. Для кожного із типів інструкції передбачено відповідну поведінку інтерпретатора, наприклад, при декларації змінної у пам’яті виділяється місце під певною вказаною назвою змінної даного типу.

Далі, в залежності від вхідного коду, ці початкові інструкції можуть доповнюватися внутрішніми операціями. Так, після ініціалізації, змінній можна присвоїти значення певного арифметичного виразу, обрахування якого вже відбуватиметься рекурсивно, все більше занурюючись у вкладені твердження AST.

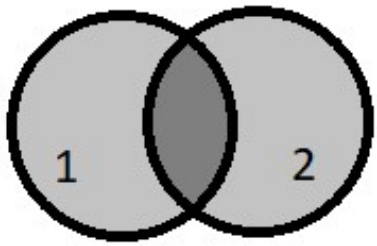
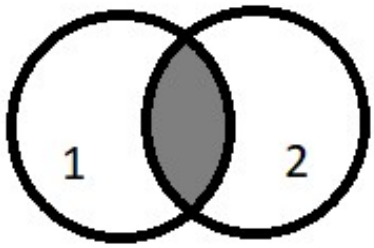
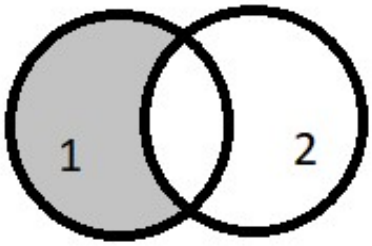
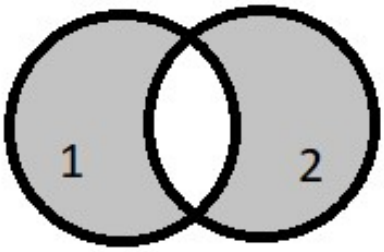
Цей компонент всього застосунку є найбільшим, адже саме тут реалізовано механізм перевірки на сумісність типів даних у аргументів. У інтерпретаторі також передбачено більшість помилок, які будуть викликані неправильно написаним кодом. Із видів помилок, які можливо отримати у інтерпретаторі є виключення сумісності типів даних, ініціалізація вже оголошених змінних, звернення до неіснуючих змінних, функцій.

Також важливою функцією розробленого інтерпретатора є запуск програми у режимі дебагінгу. Даний режим дозволяє користувачеві зупинятися на кожному кроці виконання інструкцій, щоб моніторити стан пам’яті, коректність відпрацювання написаної програми, а також рефакторити код за потреби.

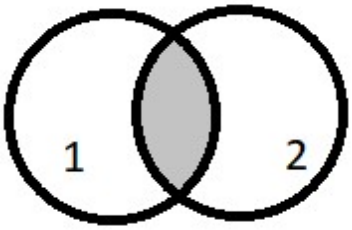
Важливим етапом інтерпретації є ініціалізація вбудованих методів мови програмування ASAMPL 2.0. Список вбудованих методів і принцип їх дії можна переглянути у табл. 3.2.

Таблиця 3.2

Вбудовані методи мови програмування ASAMPL 2.0

Назва методу	Представлення	Опис
Tuple uni (tuple1, tuple2)		Об’єднання двох кортежів. Повертає кортеж, що містить усі елементи із першого і другого
Tuple sec (tuple1, tuple2)		Перетин кортежів. Повертає кортеж, що містить лише ті елементи, які є одночасно в першому і другому з повторами
Tuple dif (tuple1, tuple2)		Різниця. Повертає кортеж з елементами, які належать першому, але не належать другому
Tuple sdif (tuple1, tuple2)		Симетрична різниця. Повертає кортеж з елементами, які є в одному з кортежів, але не в обох

Продовження табл. 3.2

Tuple xsec (tuple1, tuple2)		Розширений перетин. Повертає кортеж із елементами, які є одночасно в першому і другому, без повторів
Tuple dsort (tuple)	[5,4,3,2,1]	Повертає кортеж із відсортованими числовими значеннями за спаданням
Tuple asort (tuple)	[1,2,3,4,5]	Повертає кортеж із відсортованими числовими значеннями за зростанням
Tuple ord (String tuple)	[a,b,c,d,e]	Повертає кортеж із відсортованими по алфавітному порядку текстовими значеннями
Tuple singl (tuple)	–	Повертає кортеж із унікальними елементами
Type extr (tuple, index)	–	Видаляє елемент під певним індексом у кортежі. Повертає видалений елемент
Type ins (tuple, index, variable)	–	Вставляє елемент у кортеж у певну індексну позицію

Продовження табл. 3.2

Type get (tuple, index)	–	Повертає елемент з певного індексу кортежу
Void print (variable)	–	Друкує у консоль значення, змінну

Вище, серед вбудованих функцій, можна було побачити операції між кортежами, які відбуваються згідно з логікою операцій між множинами. Окрім цього, також варто звернути увагу на вбудовані методи, які призначені саме для виконання операцій в алгебраїчній системі агрегатів. Хоч вони і мають схожі назви із операціями між кортежами, але логіка їх реалізації і відпрацювання значно відрізняється. Опрацьовуючи агрегати, ми беремо до уваги те, що всі операції порівняння і результуючі перетворення відбуваються всередині кортежів, що є частиної цих агрегатів.

Реалізовано такі операції над агрегатами, як: об’єднання, переріз, різниця, симетрична різниця, виключний переріз, проріджування, видалення та вставлення. На рис. 3.3-3.10 відображено принцип їх дії [11]. На зазначених рисунках колір кола відповідає належності елементу певному кортежу. Якщо у кіл обрамлення однакового кольору, то ці елементи є однаковими за значенням.

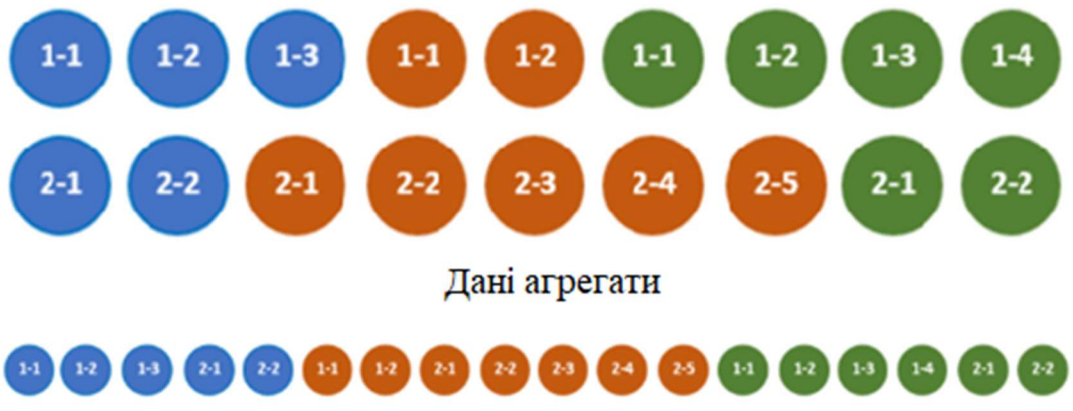


Рис. 3.3. Операція об’єднання агрегатів

Рис. 3.4. Операція перерізу агрегатів

Рис. 3.5. Операція виключного перерізу агрегатів

Рис. 3.6. Операція різниці агрегатів

Рис. 3.7. Операція симетричної різниці агрегатів

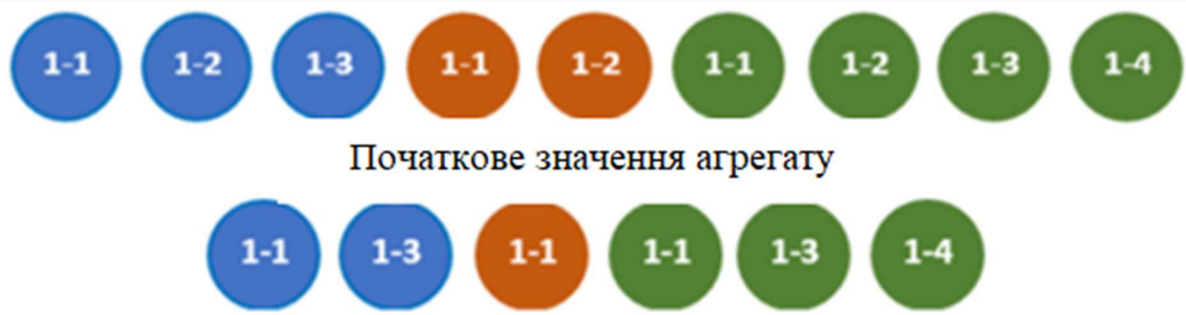


Рис. 3.8. Операція видалення елементів другої позиції із агрегату

Рис. 3.9. Операція вставки елементу *x* в агрегат

Рис. 3.10. Операція проріджування агрегату

У інтерпретаторі ці функції викликаються відповідними методами:

1) agUni (aggregate1, aggregate2);

2) agSec (aggregate1, aggregate2);

3) agXsec (aggregate1, aggregate2);

4) agDif (aggregate1, aggregate2);

5) agSdif (aggregate1, aggregate2);

6) agExtr (aggregate, index);

7) agIns (aggregate, element, index);

8) agSingl (aggregate).

Перейдемо до конкретизованого опису принципу роботи модуля інтерпретації. Для наглядності, на рис. 3.11 можна розглянути назви методів і полів, що містить в собі клас інтерпретатор.

Рис. 3.11. Відображення методів і полів класу Interpreter

Найголовнішими методами даного компонента є run та debug. Саме з них починається виконання інструкцій, зазначених в AST. Відрізняються вони лише передачою індикатора запуску програми у режимі дебагінгу. Всі інші операції в них виконуються аналогічно.

Вищезазначені методи спочатку додають до глобальної пам’яті інтерпретатора вбудовані методи мови ASAMPL 2.0 для роботи з кортежами, і викликають метод runStatements в стандартному режимі, і режимі дебагінгу відповідно.

Після цього модуль обходить, набір інструкцій, сформований в дерево, першочергово перевіряючи на валідність і виконуючи зазначені операції. Як можна побачити, методи, що націлені на виконання операцій містять у сигнатурі аргумент з об’єктом пам’яті програми, що відтворюється. Це зроблено саме для збереження контексту вкладених в блоки операцій.

Також, як було зазначено раніше, у пам’яті зберігаються не лише змінні, а й функції. Логіка збереження функцій дещо схожа. Але замість конкретного значення, у пам’яті зберігається набір очікуваних аргументів і послідовність інструкцій, виокремлений з дерева вузол.

Окремого пояснення вартий механізм відтворення програми у режимі дебагінгу. Цей режим дозволяє розробнику покроково виконати інструкції програми. Дебагінг дозволяє переглянути відображення абстрактного синтаксичного дерева, і кожної поточної операції, що виконується в інтерпретаторі. Під час кожної паузи між кроками, користувачеві виводиться інформація про стан пам’яті змінних і функцій. Це дозволить у реальному часі відстежувати, як змінюються значення змінних і що на це впливає. Перехід на виконання наступної інструкції відбувається через натискання клавіші Enter. Різницю між відтворенням програми у стандартному режимі та режимі дебагінгу можна побачити на рис. 3.12 та рис. 3.13.

```
###################
Code:
int a = 0;
print(a );
a = 4;
print(a);


###################
Abstract Syntax Tree:
Initialization: int a
  Assignment: a
    Number: 0
Function call: print
 Variable: a
Assignment: a
  Number: 4
Function call: print
 Variable: a
statements amount: 4
####################
RUNNING STATEMENT #1 [ ua.kp13.mishchenko.ast.InitializationNode@47089e5f ]
 Initialization: int a
   Assignment: a
     Number: 0
VARIABLES MEMORY:  empty
FUNCTIONS MEMORY:  empty

####################
RUNNING STATEMENT #2 [ ua.kp13.mishchenko.ast.FunctionCallNode@2ef9b8bc ]
 Function call: print
  Variable: a
VARIABLES MEMORY:
VariableEntry [type=TYPE_INT, value=0, name=a]
FUNCTIONS MEMORY:  empty

0
```

Рис. 3.12. Результат виконання коду у режимі дебагінгу

Рис. 3.13. Результат виконання коду у стандартному режимі

4. АНАЛІЗ РОЗРОБЛЕНОГО ЗАСТОСУНКУ

У цьому розділі буде описано результат проекту, розроблений інтерпретатор мови програмування ASAMPL 2.0.

4.1. Особливості реалізації інтерпретатора

Інтерпретатор мови програмування ASAMPL 2.0 реалізовано у вигляді консольного застосунку, який поширюється, як JAR-файл. Це дає змогу зручно запускати програму з будь якого середовища, де встановлено Java Virtual Machine.

Такий формат дозволяє легко інтегрувати інтерпретатор у зовнішні системи, скрипти або редактори коду.

Запуск інтерпретатора відбувається з командного рядка з передачою двох аргументів:

```
java -jar asampl _2_0_compiler.jar <path> <isDebug>
```

Команда містить в собі два параметра:

1. *path* – абсолютний шлях до текстового файлу, що містить програму, написану мовою ASAMPL 2.0.
2. *isDebug* – булеве значення (true або false), параметр, який вмикає або вимикає режим дебагінгу. Необов’язковий параметр. За замовчуванням програма запускається у стандартному режимі.

Запустивши код, все, що користувач гіпотетично може побачити в консолі це значення, що були виведені методом print, або ж виключення, якщо є лексична або синтаксична помилка у наданому програмному коді мовою ASAMPL 2.0. В режимі ж дебагінгу користувач має можливість переглянути стан пам’яті, синтаксичне дерево коду і виконувані операції на кожному кроці.

Приклад запуску застосунку у консолі можна розглянути на рис. 4.1.

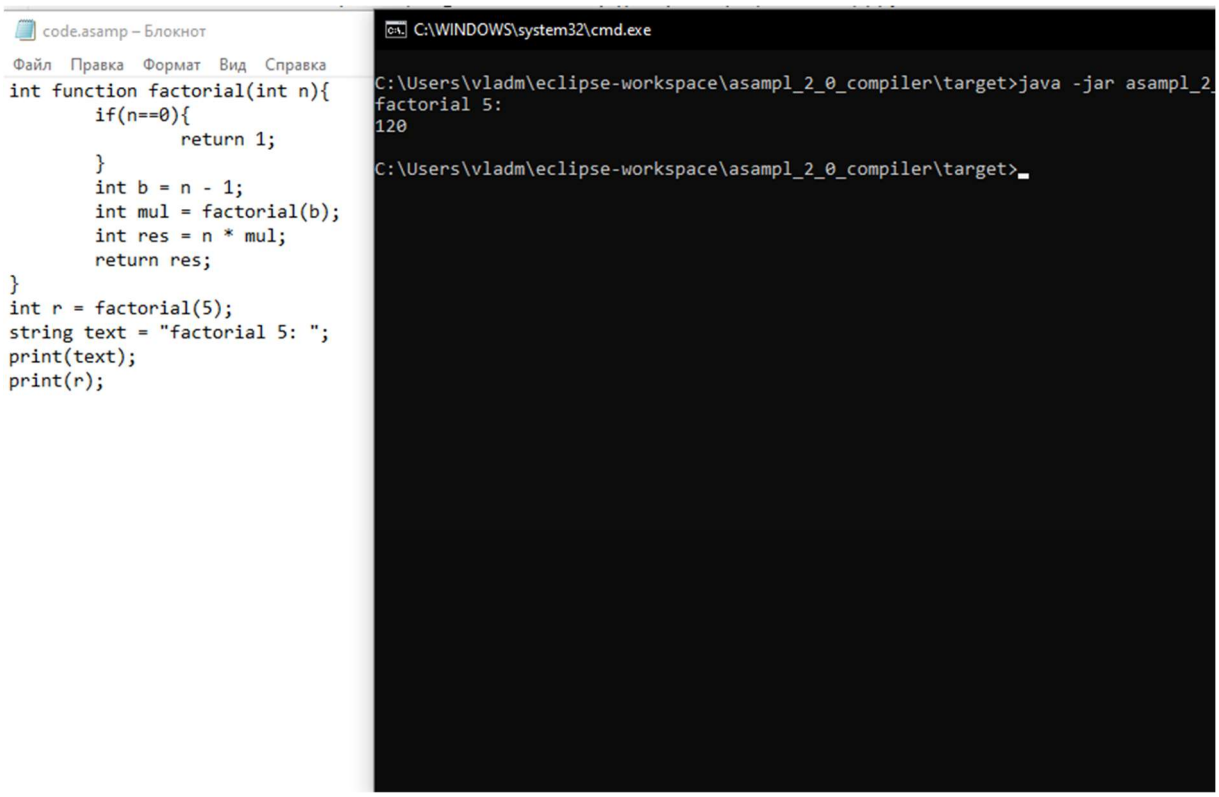


Рис. 4.1. Приклад запуску консольного застосунку інтерпретатору мови ASAMPL 2.0

4.2. Інтегроване тестування компонентів

Після реалізації проєкту, як набору основних компонентів, що взаємодіють між собою – лексичного, синтаксичного аналізаторів та інтерпретатора – виникають думки з приводу перевірки надійності такої системи. Є необхідність перевірити їхньої взаємодії в межах програмного циклу. Із цією метою було написано інтегроване автоматизоване тестування, яке дозволить виявити помилки як в певних окремих модулях, так і в їхній взаємодії, бо правильність виконання цільового коду для інтерпретатора – критична необхідність.

Інтегроване тестування охоплює всі реалізовані в проєкті групи операцій. Для кожної з них були сформовані тестові скрипти мовою ASAMPL 2.0, під час виконання яких автоматизовано перевірялися значення змінних у пам’яті інтерпретатора, порівнювалися із очікуваними значеннями відпрацювання скрипта. Таким чином було забезпечено

впевненість у правильній роботі ядра інтерпретатора, а також у правильній взаємодії і відпрацюванні кожного з модулів аналізу коду.

Тести були написані за допомогою бібліотеки JUnit. Принцип роботи тестів полягав в тому, щоб зберігати стан пам’яті інтерпретатора після обробки інструкцій, і порівнювати його із очікуваними, заздалегідь прорахованими результатами.

Це значно полегшить подальше вдосконалення ядра, адже не доведеться перевіряти працю кожного з модулів і кожної операції окремо вручну.

Все, що треба зробити для перевірки проєкту після нововведень, це запустити всі тести одним кліком і отримати результат. В разі помилки або неспівпадіння у консоль виводиться причина такого результату.

Приклад виконання всіх написаних інтегрованих тестів інтерпретатора можна побачити на рис. 4.2.

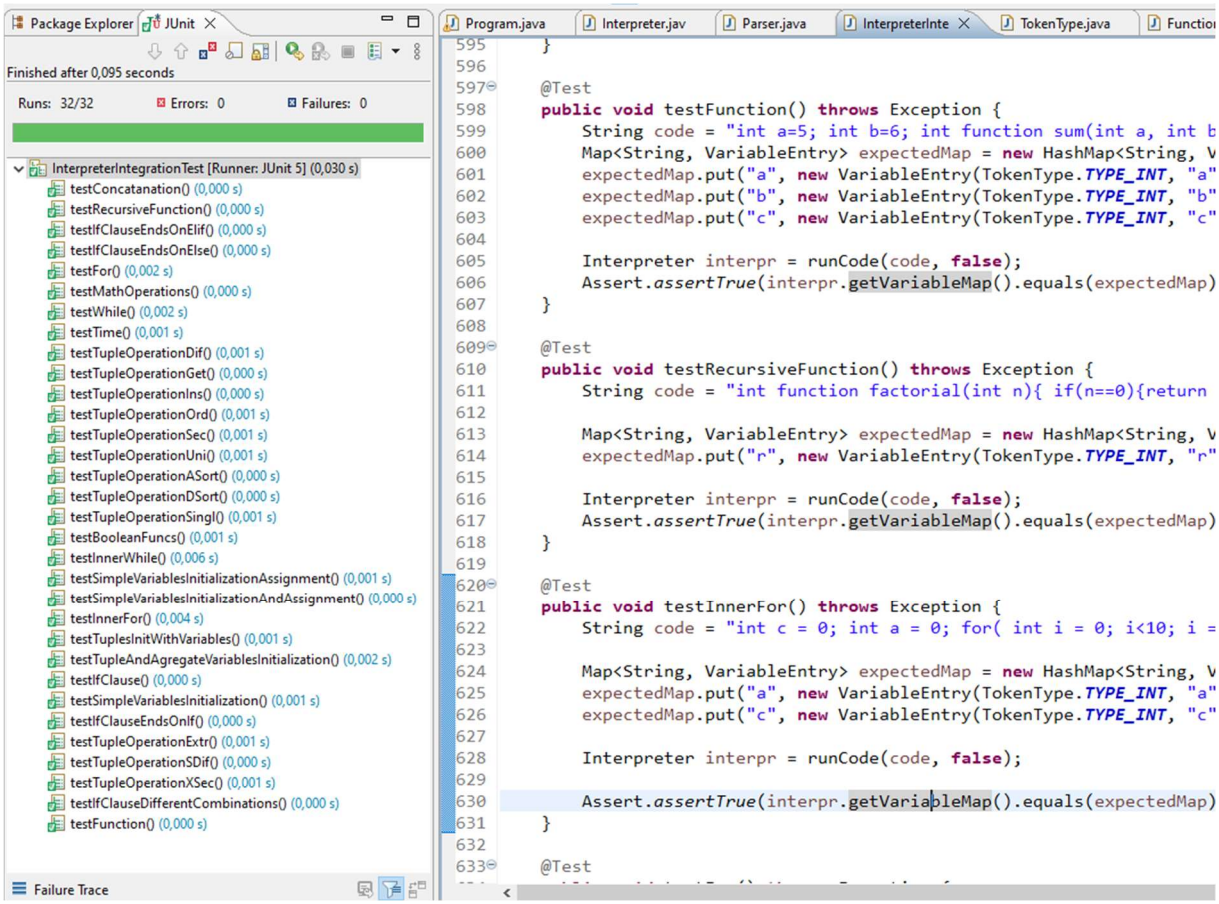


Рис. 4.2. Результат інтегрованого тестування готового інтерпретатору

Як можна побачити, таке тестування дозволяє досить швидко зрозуміти, яка функція перестала працювати, після того, як розробник вніс зміни до вихідного коду ядра. Маючи такі знання, можна легше визначити, в якому місці знаходиться баг.

Результати тестування підтвердили працездатність усіх розроблених шляхів використання, а також стабільність роботи інтерпретатора при послідовному виконанні складних переліків інструкцій. Усі знайдені помилки були усунені, що дозволило отримати надійну поведінку системи при роботі з програмами різної складності.

4.3. Рекомендації щодо подальшого вдосконалення

Розроблений застосунок, інтерпретатор мови програмування ASAMPL 2.0 має багато напрямків для вдосконалення.

По-перше, головним напрямом для вдосконалення і доопрацювання ядра є масштабування його можливостей. Це пов’язано з тим, що у розробленій версії транслятора не підтримуються певні функції, можливості, що передбачені формальним описом мови програмування ASAMPL 2.0. Такими нереалізованими можливостями є, наприклад, робота у багатопотоковому режимі, або ж певні типи даних, колекції, які ще не були додані у цю версію інтерпретатора. Такою самою функцією є підключення сторонніх бібліотек і імпортів, передбачених вищезгаданим описом мови.

В другу чергу, на думку автора, варто відокремити від модулю інтерпретації компоненти перевірки змінних на тип, перевірки на сумісність типів змінних, функцій, їх аргументів. Це зробить подальшу розробку і масштабування проекту набагато чистішими і зрозумілішими.

Ще на думку спадає модифікація розробленого ядра шляхом оптимізації механізму виконання інструкцій, за рахунок використання JIT-компіляції. Метод “just in time” компіляції передбачає не повну компіляцію програми до машинного коду перед виконанням, а змішаний алгоритм трансляціїї інструкцій. Частина коду перетворюється з інтерпретованого в

машинний код і кешується для подальшого використання [12]. Цей механізм може значно покращити продуктивність ядра, оскільки частини коду, що виконуються найчастіше, можуть бути оптимізованими на льоту.

ВИСНОВКИ

У результаті виконання дипломної роботи, було досягнуто поставлену мету – спроектовано та реалізовано інтерпретатор мови програмування ASAMPL 2.0, орієнтованої на роботу з алгебраїчною системою агрегатів та обробкк мульсемедійних даних. Основну увагу було зосереджено на розробці модулів лексичного та синтаксичного аналізу, побудові абстрактного синтаксичного дерева, обробки інструкцій, що формують це дерево, а також забезпеченні підтримки ключових конструкцій, що містяться у формальному описі вищезазначеної мови.

У першому розділі було проведено аналіз існуючих технологій створення трансляторів. Розглянуто їх класифікацію (компілятори, інтерпретатори), проаналізовано найпопулярніші генератори парсерів, здійснено порівняння їхніх можливостей. Також було розглянуто особливості популярних мов програмування, з концентрацію на їх придатність до реалізації компіляторів. На основі аналізу було обрано написати ядро власноруч мовою програмування Java.

У другому розділі було описано архітектуру інтерпретатора, а також основні його модулі: лексер, парсер, і сам інтерпретатор. Досліджено принципи побудови абстрактногого синтаксичного дерева. Було описано найголовніші засади нового синтаксису мови програмування ASAMPL 2.0.

Третій розділ містить опис і реалізацію основних алгоритмів системи: виділення токенів, проведення синтаксичного аналізу і виконання інструкцій інтерпретатором. Було продемонстровано вбудовані функції роботи з кортежами.

У четвертому розділі було описано особливості реалізації готового програмного продукту. Описано і продемонстровано роботу ядра. Також у цьому розділі міститься опис способу перевірки якості розробленого інтерпретатору, перевірено коректність відпрацьовування передбачених синтаксисом мови операцій. Наприкінці четвертого розділу були

запропоновані способи подальшого вдосконалення інтерпретатору ASAMPL 2.0, шляхом оптимізації, доповнення, масштабування.

Таким чином, під час дипломного проєктування було використано принципи побудови трансляторів на практиці, реалізовано ядро виконання програм другою версією мови алгебраїчної системи агрегатів та обробки мульсемедійних даних.

СПИСОК ВИКОРИСТАНИХ ЛІТЕРАТУРНИХ ДЖЕРЕЛ

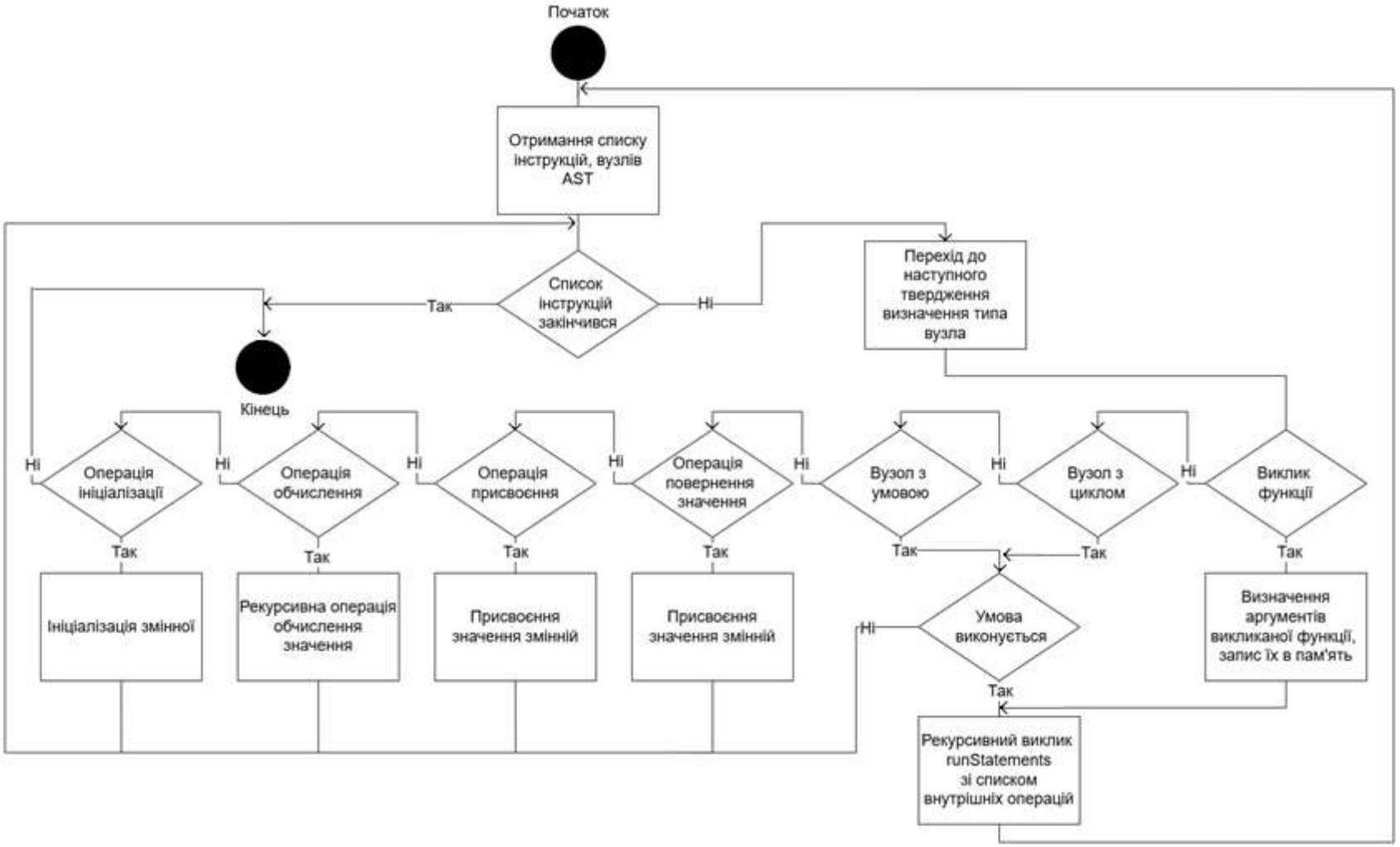
1. Компілятор та інтерпретатор: розуміння основ. [Електронний ресурс] – Режим доступу: https://foxminded.ua/kompiliator-ta-interpretator/ – Дата доступу: 29.05.2025 р.
2. ANTLR [Електронний ресурс] – Режим доступу: https://uk.wikipedia.org/wiki/ANTLR – Дата доступу: 29.05.2025 р.
3. What is ANTLR? [Електронний ресурс] – Режим доступу: https://www.antlr.org/ – Дата доступу: 29.05.2025 р.
4. JavaCC [Електронний ресурс] – Режим доступу: https://uk.wikipedia.org/wiki/JavaCC – Дата доступу: 29.05.2025 р.
5. JavaCC 21 Parser Generator [Електронний ресурс] – Режим доступу: https://habr.com/ru/articles/521664/ – Дата доступу: 29.05.2025 р.
6. PLY (Python lex-Yacc) - An Introduction [Електронний ресурс] – Режим доступу: https://www.geeksforgeeks.org/ply-python-lex-yacc-an-introduction/?ref=ml_lbp – Дата доступу: 29.05.2025 р.
7. Введення в DSL. Частина 1 [Електронний ресурс] – Режим доступу: https://habr.com/ru/articles/94259/ – Дата доступу: 29.05.2025 р.
8. Мова програмування C++ [Електронний ресурс] – Режим доступу: https://acode.com.ua/uroki-po-cpp/ – Дата доступу: 29.05.2025 р.
9. Python Package Index [Електронний ресурс] – Режим доступу: https://en.wikipedia.org/wiki/Python_Package_Index – Дата доступу: 29.05.2025 р.
10. What is Java technology and why do I need it? [Електронний ресурс] – Режим доступу: https://www.java.com/en/download/help/whatis_java.html – Дата доступу: 29.05.2025 р.
11. Andreas Pester, Yevgeniya Sulema, Ivan Dychka, Olga Sulema. Temporal Multimodal Data Processing Algorithms Based on Algebraic System of Aggregates. Journal “Algorithms”. 2023, 16(4), 186. [Електронний ресурс]

– Режим доступу: https://doi.org/10.3390/a16040186 – Дата доступу: 29.05.2025 р.

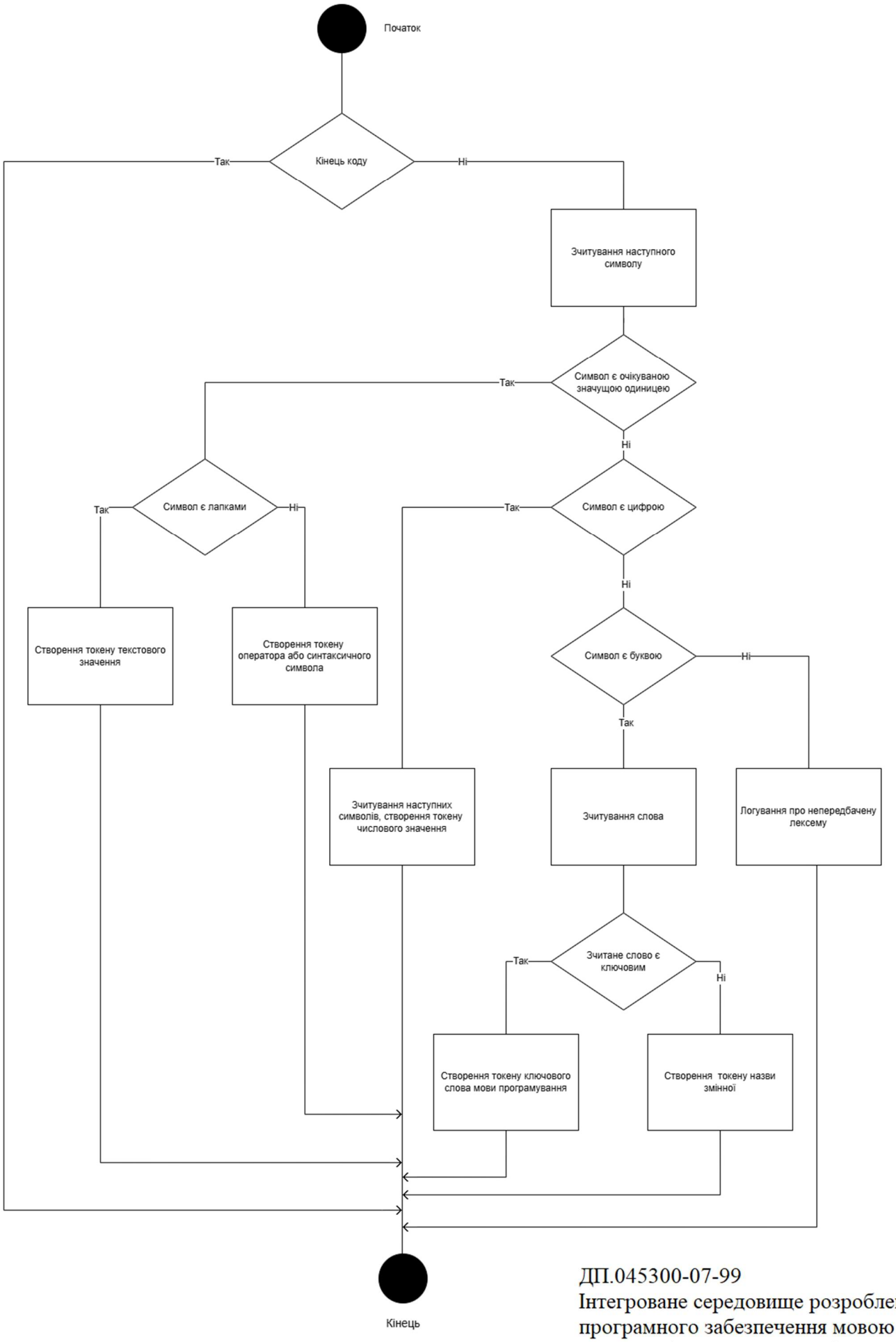
12. Компіляція і інтерпретація в сучасному JIT. [Електронний ресурс] – Режим доступу: https://dou.ua/forums/topic/35506/ – Дата доступу: 29.05.2025 р.

ДОДАТКИ

Додаток 1
Копії графічних матеріалів



ДП.045300-06-99
Інтегроване середовище розроблення програмного забезпечення мовою ASAMPL 2.0. Компілятор. Схема роботи модуля інтерпретатації. Діаграма діяльності



ДП.045300-07-99
Інтегроване середовище розроблення програмного забезпечення мовою ASAMPL 2.0. Компілятор. Схема роботи модуля синтаксичного аналізу. Діаграма діяльності

code.asamp – Блокнот

Файл Правка Формат Вид Справка

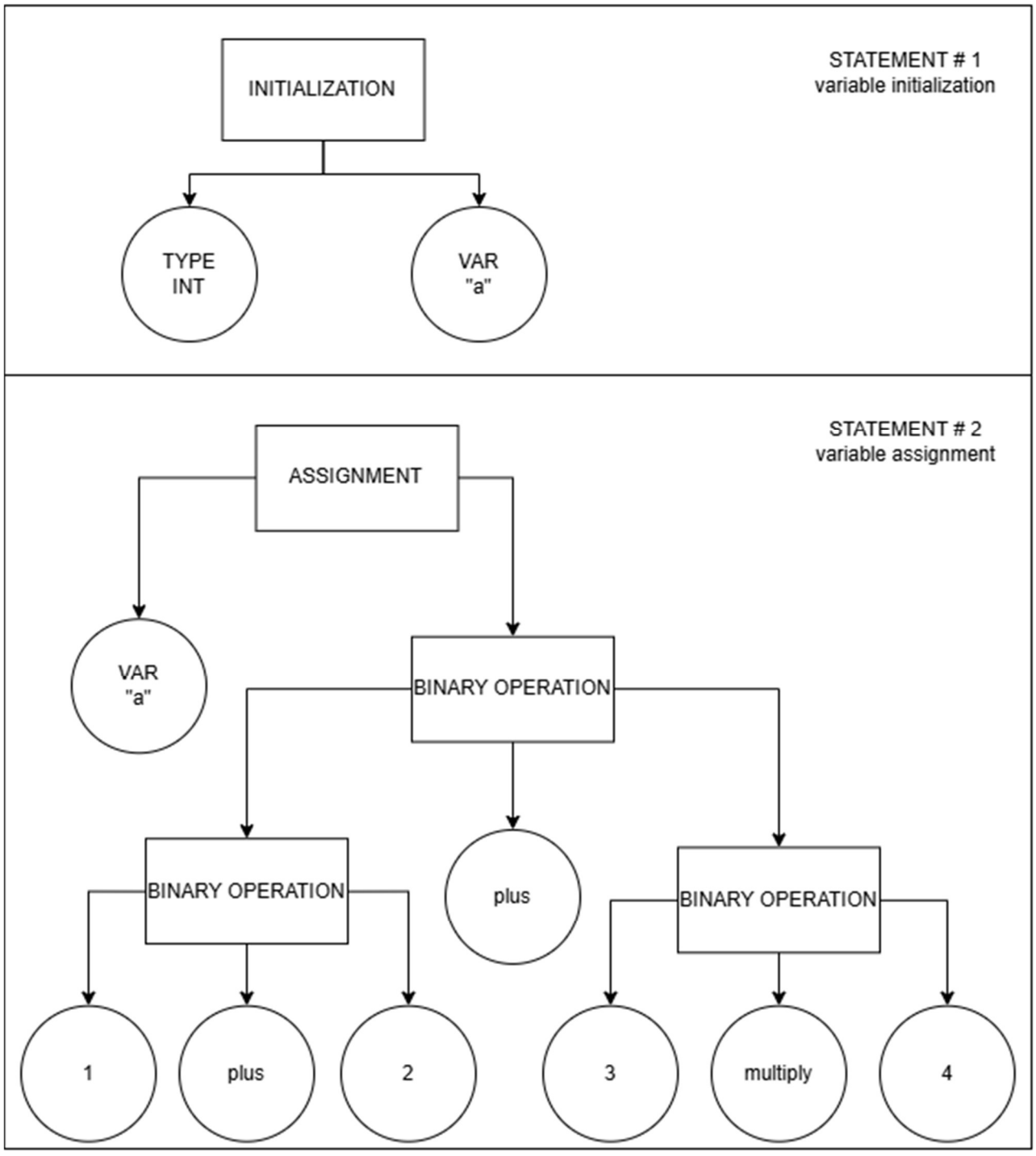
```
int function factorial(int n){
        if(n==0){
                return 1;
        }
        int b = n - 1;
        int mul = factorial(b);
        int res = n * mul;
        return res;
}
int r = factorial(5);
string text = "factorial 5: ";
print(text);
print(r);
```

C:\WINDOWS\system32\cmd.exe

```
C:\Users\vladm\eclipse-workspace\asampl_2_0_compiler\target>java -jar asampl_2_
factorial 5:
120

C:\Users\vladm\eclipse-workspace\asampl_2_0_compiler\target>
```

Приклад роботи інтерпретатора
Група КП-13, Міщенко В.Р.

int a = 1 + 2 + 3 * 4;



Умовна схема розбиття коду на вузли
абстрактного синтаксичного дерева
Група КП-13, Міщенко В.Р.

Додаток 2
Лістинг програми

Лістинг 1. Вхідний метод застосунку

```
	public static void main(String[] args) throws IOException,
URISyntaxException {
		if(args.length < 1) {
			System.out.println("ERROR. PROGRAM EXECUTION MAY INCLUDE TWO
PARAMS: fileUrl[\"C:\\example.asamp\"], isDebug[true/false](optional)");
			return;
		} else if(args.length == 2) {
			isDebug = Boolean.parseBoolean(args[1]);
		}
		codeFilePath = args[0];

		String code = new
String(Files.readAllBytes(Paths.get(codeFilePath)));
		Lexer lexer = new Lexer(code);
		Parser parser = new Parser(lexer);

		Node ast = null;

		try {
			ast = parser.parse();
		} catch (Exception e) {
			e.printStackTrace();
		}
		Interpreter interpr = new Interpreter(ast);

		if (isDebug) {
			System.out.println("####################");
			System.out.println("Code: ");
			System.out.println(code);
			System.out.println("####################");
			System.out.println("Abstract Syntax Tree:");

			ast.printAST("");
			System.out.println("statements amount: " + ((ProgramNode)
ast).getStatements().size());

			try {
				interpr.debug();
			} catch (InterpreterException e) {
				e.printStackTrace();
			}

			System.out.println("####################");
			System.out.println("program end.");
			interpr.printVariablesMap(interpr.getVariableMap());
			interpr.printFunctionsMap();
		} else {
			try {
				interpr.run();
			} catch (InterpreterException e) {
				e.printStackTrace();
			}
		}

	}
```

Лістинг 2. Основний метод модуля лексичного аналізу

```
public boolean nextToken() throws LexerException {


            while (!isEndOfCode() && !isExceptionThrown) {

                  previousToken = currentToken;

                  char currentChar = code.charAt(currentIndex);

                  if (Arrays.asList(' ', '\r', '\t',
'\n').contains(currentChar)) {
                        skipWhiteSpace();
                        continue;
                  }

                  else if (currentChar == ';') {
                        currentToken = new Token(TokenType.NEXT_LINE, ";");
                        moveIndex();
                  }

                  else if (currentChar == '=') {
                        if(code.charAt(currentIndex + 1) != '=') {
                              currentToken = new Token(TokenType.OPER_EQUALS,
"=");
                              moveIndex();
                        } else { //////////BOOL
                              currentToken = new Token(TokenType.OPER_IS,
"==");
                              currentIndex +=2;
                        }
                  }

                  else if (currentChar == '+') {
                        currentToken = new Token(TokenType.OPER_PLUS, "+");
                        moveIndex();
                  }

                  else if (currentChar == '-') {
                        currentToken = new Token(TokenType.OPER_MINUS, "-");
                        moveIndex();
                  }

                  else if (currentChar == '*') {
                        currentToken = new Token(TokenType.OPER_MULTIPLY, "*");
                        moveIndex();
                  }

                  else if (currentChar == '/') {
                        currentToken = new Token(TokenType.OPER_DIVISION, "/");
                        moveIndex();
                  }

                  else if (currentChar == '^') {
                        currentToken = new Token(TokenType.OPER_POWER, "^");
                        moveIndex();
                  }

                  else if (currentChar == '.') {
```

```
                        currentToken = new Token(TokenType.FLOATING_POINT, ".");
                        moveIndex();
                }

                else if (currentChar == '(') {
                        currentToken = new Token(TokenType.BRACKET_OPENED, "(");
                        moveIndex();
                }

                else if (currentChar == ')') {
                        currentToken = new Token(TokenType.BRACKET_CLOSED, ")");
                        moveIndex();
                }

                else if (currentChar == ',') {
                        currentToken = new Token(TokenType.COMMA, ",");
                        moveIndex();
                }

                else if (currentChar == '[') {
                        currentToken = new
Token(TokenType.BRACKET_SQUARE_OPENED, "[");
                        moveIndex();
                        currentChar = code.charAt(currentIndex);
                        if(currentChar == ']') {
                                currentToken = new Token(TokenType.TYPE_TUPLE,
"tuple []");
                                moveIndex();
                        }
                }
                else if (currentChar == ']') {
                        currentToken = new
Token(TokenType.BRACKET_SQUARE_CLOSED, "]");
                        moveIndex();
                }

                else if (currentChar == '{') {
                        currentToken = new Token(TokenType.BRACKET_CURLY_OPENED,
"{");
                        moveIndex();
                }

                else if (currentChar == '}') {
                        currentToken = new Token(TokenType.BRACKET_CURLY_CLOSED,
"}");
                        moveIndex();
                }
                //////////BOOL
                else if (currentChar == '!') {
                        if(code.charAt(currentIndex + 1) == '=') {
                                currentToken = new Token(TokenType.OPER_IS_NOT,
"!=");
                                currentIndex +=2;
                        } else {
                                currentToken = new Token(TokenType.OPER_NOT,
"!");
                                moveIndex();
                        }
                }
                else if (currentChar == '>') {
```

```
                        if(code.charAt(currentIndex + 1) == '=') {
                                currentToken = new
Token(TokenType.OPER_MORE_EQUALS, ">=");
                                currentIndex +=2;
                        } else {
                                currentToken = new Token(TokenType.OPER_MORE,
">");
                                moveIndex();
                        }
                }
                else if (currentChar == '<') {
                        if(code.charAt(currentIndex + 1) == '=') {
                                currentToken = new
Token(TokenType.OPER_LESS_EQUALS, "<=");
                                currentIndex +=2;
                        } else {
                                currentToken = new Token(TokenType.OPER_LESS,
"<");
                                moveIndex();
                        }
                }
                else if (currentChar == '|') {
                        if(code.charAt(currentIndex + 1) == '|') {
                                currentToken = new Token(TokenType.OPER_OR,
"||");
                                currentIndex +=2;
                        } else {
                                throw new LexerException("Token not defined.");
                        }
                }
                else if (currentChar == '&') {
                        if(code.charAt(currentIndex + 1) == '&') {
                                currentToken = new Token(TokenType.OPER_AND,
"&&");
                                currentIndex +=2;
                        } else {
                                throw new LexerException("Token not defined.");
                        }
                }
                else if (currentChar == '}') {
                        currentToken = new Token(TokenType.BRACKET_CURLY_CLOSED,
"}");
                        moveIndex();
                }
                ////////// READ STRING TEXT
                else if (currentChar == '\"') {
                        moveIndex();

                        boolean anotherQuotesDetected = false;
                        for (int i = (currentIndex); i < code.length(); i++) {
                                if (code.charAt(i) == '\"') {
                                        anotherQuotesDetected = true;
                                }
                        }

                        if (!anotherQuotesDetected) {
                                throw new LexerException("closing \" required");
                        }
                        anotherQuotesDetected = false;
                        StringBuilder sb = new StringBuilder();
                        currentChar = code.charAt(currentIndex);
```

```
                        while (currentChar != '\"' && !isEndOfCode()) {

                                sb.append(currentChar);

                                currentChar = code.charAt(currentIndex);

                                moveIndex();

                                if (isEndOfCode())
                                        break;

                                currentChar = code.charAt(currentIndex);
                        }

                        currentToken = new Token(TokenType.STRING,
sb.toString());

                        moveIndex();
                }

                ////////// NUMBERS, VARIABLES AND KEY-WORDS
                else if (Character.isDigit(currentChar)) {
                        String number = readNumber();
                        String time = "";
                        currentChar = code.charAt(currentIndex);
                        if(number.length() == 2 && currentChar == ':') {
                                time += number;
                                moveIndex();
                                currentChar = code.charAt(currentIndex);
                                for(int i = 0; i<3; i++) {
                                        String result = readNumber();
                                        if(result.length() == 2) {
                                                time += ":" + result;
                                        } else {
                                                throw new LexerException("lexer
error occured while parsing time construction");
                                        }
                                        moveIndex();
                                        currentChar = code.charAt(currentIndex);
                                }
                                currentToken = new Token(TokenType.TIME_VALUE,
time);
                        } else {
                                currentToken = new Token(TokenType.NUMBER,
number);
                        }
                }
                else if (Character.isLetter(currentChar)) {
                        String variableName = readVariable();

                        if (variableName.equalsIgnoreCase("int")) {
                                currentToken = new Token(TokenType.TYPE_INT,
"int");
                        } else if (variableName.equalsIgnoreCase("string")) {
                                currentToken = new Token(TokenType.TYPE_STRING,
"string");
                        } else if (variableName.equalsIgnoreCase("agregate")) {
                                currentToken = new Token(TokenType.TYPE_AGREGATE,
"agregate");
                        } else if (variableName.equalsIgnoreCase("float")) {
```

```
                                    currentToken = new Token(TokenType.TYPE_FLOAT,
"float");
                              } else if (variableName.equalsIgnoreCase("boolean")) {
                                    currentToken = new Token(TokenType.TYPE_BOOLEAN,
"boolean");
                              } else if (variableName.equalsIgnoreCase("time")) {
                                    currentToken = new Token(TokenType.TYPE_TIME,
"time");
                              } else if (variableName.equalsIgnoreCase("true")) {
                                    currentToken = new Token(TokenType.TRUE, "true");
                              } else if (variableName.equalsIgnoreCase("false")) {
                                    currentToken = new Token(TokenType.FALSE,
"false");
                              } else if (variableName.equalsIgnoreCase("if")) {
                                    currentToken = new Token(TokenType.IF, "if");
                              } else if (variableName.equalsIgnoreCase("elif")) {
                                    currentToken = new Token(TokenType.ELIF, "else
if");
                              } else if (variableName.equalsIgnoreCase("else")) {
                                    currentToken = new Token(TokenType.ELSE, "else");
                              } else if (variableName.equalsIgnoreCase("while")) {
                                    currentToken = new Token(TokenType.WHILE,
"while");
                              } else if (variableName.equalsIgnoreCase("for")) {
                                    currentToken = new Token(TokenType.FOR, "for");
                              } else if (variableName.equalsIgnoreCase("void")) {
                                    currentToken = new Token(TokenType.VOID, "void");
                              } else if (variableName.equalsIgnoreCase("function")) {
                                    currentToken = new Token(TokenType.FUNCTION,
"function");
                              } else if (variableName.equalsIgnoreCase("return")) {
                                    currentToken = new Token(TokenType.RETURN,
"return");
                              } else {
                                    currentToken = new Token(TokenType.VARIABLE,
variableName);
                              }
                        }
                        else {
                              isExceptionThrown = true;
                              throw new LexerException("Token not defined. Got: \"" +
currentChar + "\" line: " + line + " , pos: " + pos);
                        }
                        return true;
                  }
                  return false;
            }
```

Лістинг 3. Три ключових рекурсивних методи модулю синтаксичного аналізу

```
private Node parseExpression() throws ParserException {

            Node left = parsePriorOper();

            while (currentToken != null
                        && (currentToken.getType() == TokenType.OPER_PLUS ||
currentToken.getType() == TokenType.OPER_MINUS)) {

                  Token operator = currentToken;

                  move();

                  Node right = parsePriorOper();

                  left = new BinaryOperationNode(left, operator, right,
lexer.getLine());
            }

            while (currentToken != null
                        && (currentToken.getType() == TokenType.OPER_AND ||
currentToken.getType() == TokenType.OPER_OR)) {
                  Token operator = currentToken;

                  move();

                  Node right = parsePriorOper();

                  left = new LogicalOperationNode(left, operator, right,
lexer.getLine());
            }
            // }

            return left;
      }
private Node parsePriorOper() throws ParserException {
            Node left = parseArgs();

            while (currentToken != null && (currentToken.getType() ==
TokenType.OPER_MULTIPLY
                        || currentToken.getType() == TokenType.OPER_DIVISION
                        || currentToken.getType() == TokenType.OPER_POWER)) {
                  Token operator = currentToken;
                  move();

                  Node right = parseArgs();
                  left = new BinaryOperationNode(left, operator, right,
lexer.getLine());
            }

            while (currentToken != null && (currentToken.getType() ==
TokenType.OPER_IS
                        || currentToken.getType() == TokenType.OPER_MORE ||
currentToken.getType() == TokenType.OPER_LESS
                        || currentToken.getType() == TokenType.OPER_IS_NOT
                        || currentToken.getType() == TokenType.OPER_MORE_EQUALS
                        || currentToken.getType() ==
TokenType.OPER_LESS_EQUALS)) {
```

```
                    Token operator = currentToken;
                    move();

                    Node right = parseArgs();

                    left = new LogicalOperationNode(left, operator, right,
lexer.getLine());
             }
             return left;
      }
private Node parseArgs() throws ParserException {
             Token token = currentToken;

             if (token.getType() == TokenType.STRING) {
                    move();
                    return new StringNode(token, lexer.getLine());
             } else if (token.getType() == TokenType.NUMBER) {
                    move();
                    return new NumberNode(token, lexer.getLine());
             } else if (token.getType() == TokenType.TIME_VALUE) {
                    move();
                    return new TimeNode(token, lexer.getLine());
             } else if (token.getType() == TokenType.FALSE || token.getType() ==
TokenType.TRUE) {
                    move();
                    return new BooleanNode(token, lexer.getLine());
             } else if (token.getType() == TokenType.OPER_NOT) {
                    move();
                    Node expression = parsePriorOper();
                    return new LogicalOperationNode(expression, token, null,
lexer.getLine());
             } else if (token.getType() == TokenType.VARIABLE) {
                    move();
                    return new VariableNode(token, lexer.getLine());
             } else if (token.getType() == TokenType.BRACKET_OPENED) {
                    move();
                    Node expression = parseExpression();
                    if (currentToken.getType() != TokenType.BRACKET_CLOSED &&
!parsingForExpressions) {
                           throw new ParserException("Expected closing bracket."+ "
line: " + lexer.getLine() + " , pos: " + lexer.getPos());
                    }
                    move();
                    return expression;
             } else if (token.getType() == TokenType.NEXT_LINE || token.getType()
== TokenType.COMMA
                           || lookAheadToken != null) {
                    move();
                    if (lookAheadToken != null) {
                           return parseStatement();
                    } else {
                           return null;
                    }
             } else {
                    throw new ParserException("Unexpected token: " +
token.getValue() + "." + " line: " + lexer.getLine() + " , pos: " +
lexer.getPos());
             }
      }
```

Лістинг 4. Обробка інструкцій модулем інтерпретації

```
private VariableEntry runStatements(List<Node> statements, boolean isInner,
                    Map<String, VariableEntry> variableMap) throws
InterpreterException {
             List<String> clearList = new ArrayList<String>();

             for (int i = 0; i < statements.size(); i++) {

                    Node statement = statements.get(i);
                    if(isDebug) {
                           System.out.println("#####################");
                           if(isInner) {
                                  System.out.println("RUNNING INNER STATEMENT #" +
(i + 1) + " [ " + statement.toString() + " ]");
                           } else {
                                  System.out.println("RUNNING STATEMENT #" + (i +
1) + " [ " + statement.toString() + " ]");
                           }
                           statement.printAST(" ");
                           printVariablesMap(variableMap);
                           printFunctionsMap();

                           Scanner sc = new Scanner(System.in);
                           try{
                                  sc.nextLine();
                           } catch(Exception ex) {
                                  sc.close();
                           }

                    }

                    /*if(statement == null) {
                           return null;
                    }*/

                    // INITIALIZATION
                    if (statement.getClass() == InitializationNode.class) {
                           runInitializationOperation((InitializationNode)
statement, variableMap);
                           if (isInner) {
                                  clearList.add(((InitializationNode)
statement).getVariableName());
                           }
                    }
                    // ASSIGNMENT
                    else if (statement.getClass() == AssignmentNode.class) {

                           runAssignmentOperation((AssignmentNode) statement,
variableMap);

                    }
                    // BINARY OPERATION
                    else if (statement.getClass() == BinaryOperationNode.class) {

      System.out.println(runBinaryOperation((BinaryOperationNode) statement,
variableMap));

                    }
                    // CONDITIONS SINGLE IF OR IF-ELIF-ELSE BLOCK
```

```
                    else if (statement.getClass() == ConditionNode.class) {
                          List<ConditionNode> ifClauseConstruction = new
ArrayList<ConditionNode>();


                          boolean flag = false;
                          while (flag == false && statements.get(i).getClass() ==
ConditionNode.class) {
                                if (((ConditionNode) statements.get(i)).getType()
== TokenType.IF
                                            && ifClauseConstruction.size() == 0)
{
                                      ifClauseConstruction.add((ConditionNode)
statements.get(i));

                                      i++;
                                } else if (((ConditionNode)
statements.get(i)).getType() == TokenType.ELIF) {
                                      ifClauseConstruction.add((ConditionNode)
statements.get(i));

                                      i++;
                                } else if (((ConditionNode)
statements.get(i)).getType() == TokenType.ELSE) {
                                      ifClauseConstruction.add((ConditionNode)
statements.get(i));

                                      i++;
                                } else {
                                      i--;
                                      flag = true;
                                }
                                if (i > statements.size() - 1) {
                                      i--;
                                      break;
                                }
                          }
                          if (statements.get(i).getClass() != ConditionNode.class)
{
                                i--;
                          }



                          boolean passed = false;
                          for (int j = 0; j < ifClauseConstruction.size(); j++) {
                                if (passed) {
                                      break;
                                }
                                ConditionNode condition =
ifClauseConstruction.get(j);
                                passed = runConditionStatements(condition,
variableMap);
                          }
                    }
                    // WHILE LOOP
                    else if (statement.getClass() == WhileLoopNode.class) {
                          WhileLoopNode node = (WhileLoopNode) statement;
                          Node expression = node.getRunCondition();
```

```
                        List<Node> innerStatements = node.getStatementsList();

                        while (checkConditionExpression(expression,
variableMap)) {
                              runStatements(innerStatements, true,
variableMap);
                        }

                  }
                  // FOR LOOP
                  else if (statement.getClass() == ForLoopNode.class) {
                        ForLoopNode node = (ForLoopNode) statement;
                        Node initCounter = node.getCounter();
                        Node expression = node.getRunCondition();
                        Node step = node.getStep();
                        List<Node> innerStatements = node.getStatementsList();

                        if (initCounter.getClass() == InitializationNode.class)
{
                              runInitializationOperation((InitializationNode)
initCounter, variableMap);
                              clearList.add(((InitializationNode)
initCounter).getVariableName());
                        } else {
                              throw new InterpreterException("1st argument of
'FOR' loop construction must be initialization." + " line: " +
initCounter.getLine() );
                        }

                        while (checkConditionExpression(expression,
variableMap)) {

                              runStatements(innerStatements, true,
variableMap);
                              if (step.getClass() == AssignmentNode.class) {
                                    runAssignmentOperation((AssignmentNode)
step, variableMap);
                              } else {
                                    throw new InterpreterException("3rd
argument of 'FOR' loop construction must be assignment." + " line: " +
step.getLine());
                              }
                        }
                  }
                  // FUNCTION INITIALIZATION
                  else if (statement.getClass() == FunctionNode.class) {
                        FunctionNode node = (FunctionNode) statement;
                        String name = node.getName().getToken().getValue();
                        functionsMap.put(name, node);
                  }

                  // FUNCTION CALL
                  else if (statement.getClass() == FunctionCallNode.class) {
                        FunctionCallNode funcCall = (FunctionCallNode)
statement;

                        runFunctionCall(funcCall, variableMap);

                  } else if (statement.getClass() == ReturnNode.class) {
                        isReturnCalled = true;
```

```
                        VariableEntry res = runReturn((ReturnNode) statement,
variableMap);
                        return res;

                  }



                  if (isReturnCalled) {
                        isReturnCalled = false;
                        break;
                  }
            }

            // CLEARING MEMORY FROM LOCAL INNER VARIABLES
            for (String varName : clearList) {
                  variableMap.remove(varName);
            }
            clearList.clear();
            return null;
      }
```

Додаток 3
Копія презентації

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

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ
КАФЕДРА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ КОМП’ЮТЕРНИХ СИСТЕМ

ІНТЕГРОВАНЕ СЕРЕДОВИЩЕ РОЗРОБЛЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ МОВОЮ ASAMPL 2.0. КОМПІЛЯТОР

Виконав: Міщенко Владислав Романович

Керівник: завідувач кафедри, д.т.н. доцент Сулема Євгенія Станіславівна

Київ – 2025

ПОСТАНОВКА ЗАДАЧІ

Мета проекту: розроблення компілятору із функцією налагодження коду, для підримки мови програмування ASAMPL 2.0.

ЗАВДАННЯ

1. Провести аналіз технологій розроблення компіляторів.
2. Спроектувати архітектуру інтерпретатора, продумати логіку взаємодії між модулями.
3. Розробити алгоритми для компонентів, програмний продукт.
4. Проаналізувати і протестувати готовий застосунок.

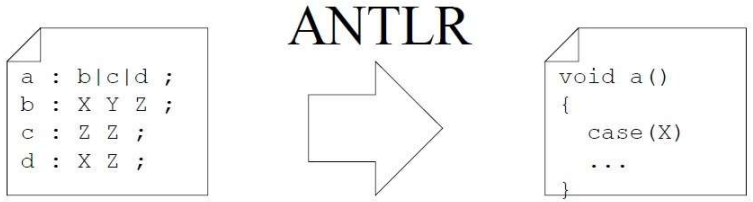
ОБРАНІ ЗАСОБИ РОЗРОБЛЕННЯ

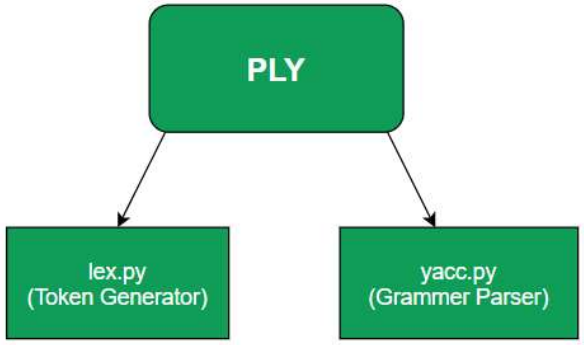
1. **Із проаналізованих мов (C++, Python, Java) було обрано мову програмування Java.**
2. **Для забезпечення автоматизованого тестування було обрано бібліотеку JUnit.**
3. **Також було підключено Apache Maven – систему зборки проектів.**

ГОТОВІ РІШЕННЯ

JavaCC™

The Java Parser Generator

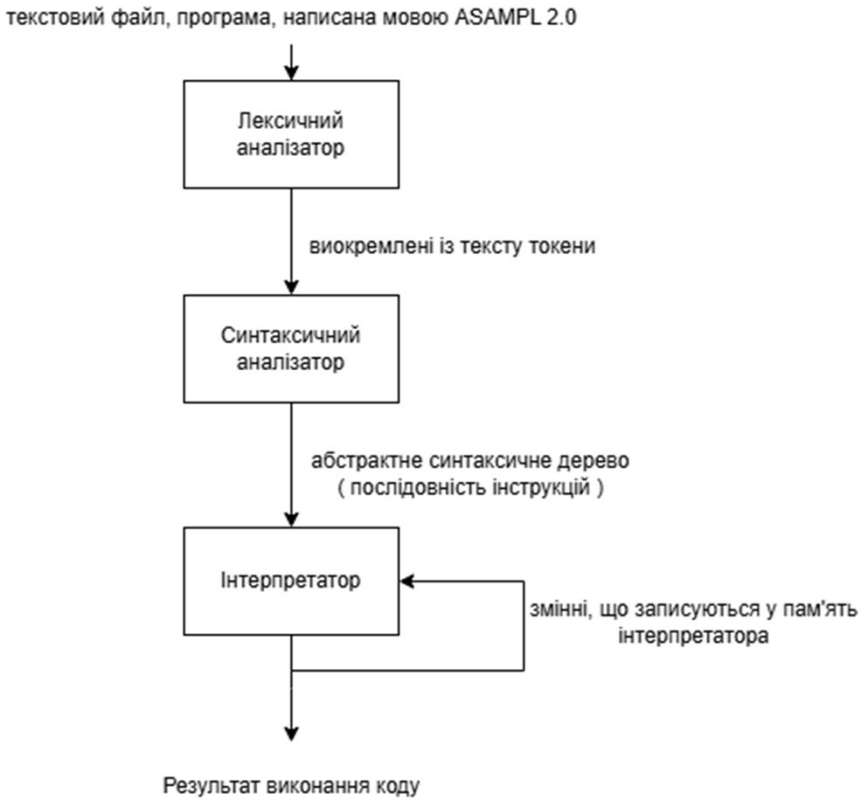




АРХІТЕКТУРА ІНТЕРПРЕТАТОРА

Основні компоненти:

1. Лексер
2. Парсер
3. Модуль інтерпретації



ЛЕКСЕР (МОДУЛЬ ЛЕКСИЧНОГО АНАЛІЗУ)

Задача цього компонента полягає в тому, щоб розбити текстовий файл з кодом на відомі, передбачені формальним синтаксисом, конструкції.

Модуль взаємодіє з парсером, передаючи йому один за одним токени (лексеми), які знаходить у коді

ПАРСЕР (МОДУЛЬ СИНТАКСИЧНОГО АНАЛІЗУ)

Парсер формує інструкції із чіткої послідовності лексем.

Перевіряє порядок розташування списку токенів, задля того, щоб знайти відповідність між порядком і існуючими синтаксичними конструкціями.

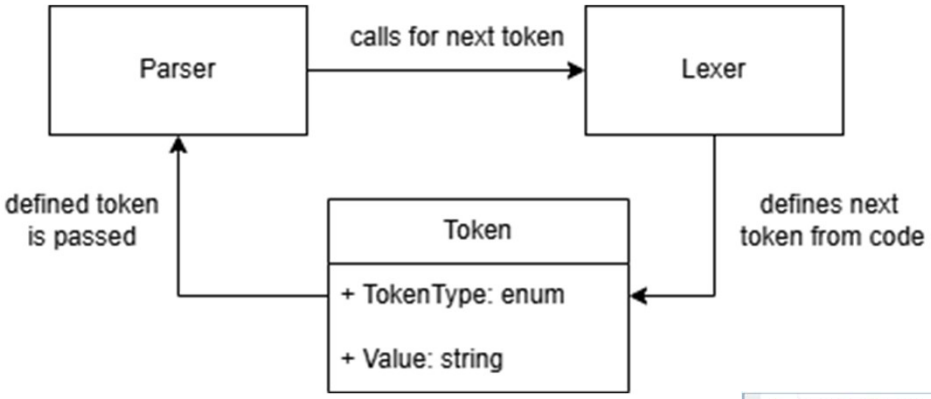
Передає у компонент інтерпретації абстрактне синтаксичне дерево.

ІНТЕРПРЕТАТОР (КОМПОНЕНТ, ЩО ВИКОНУЄ ГОТОВІ ІНСТРУКЦІЇ)

На основі абстрактного синтаксичного дерева (ієрархії команд) виконує певні, вказані в ньому дії, проводить обчислення, запис в пам’ять.

Найважливіший і найбільший модуль застосунку.

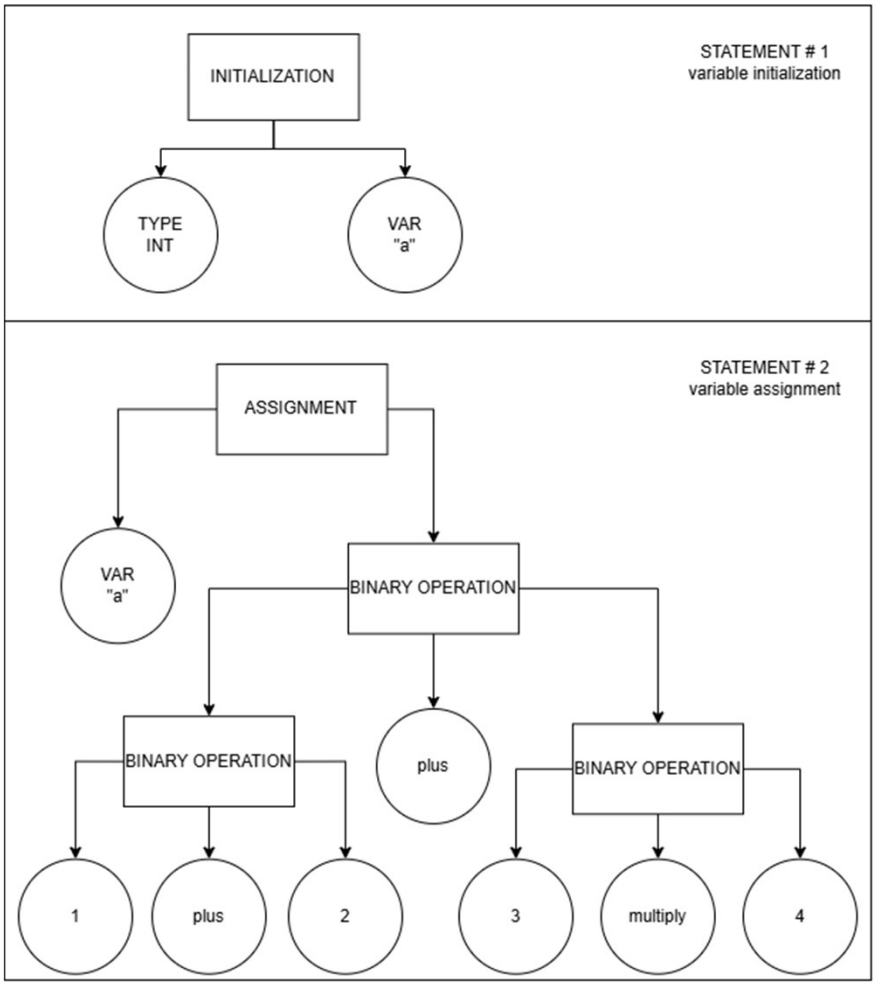
Взаємодія лексера і парсера



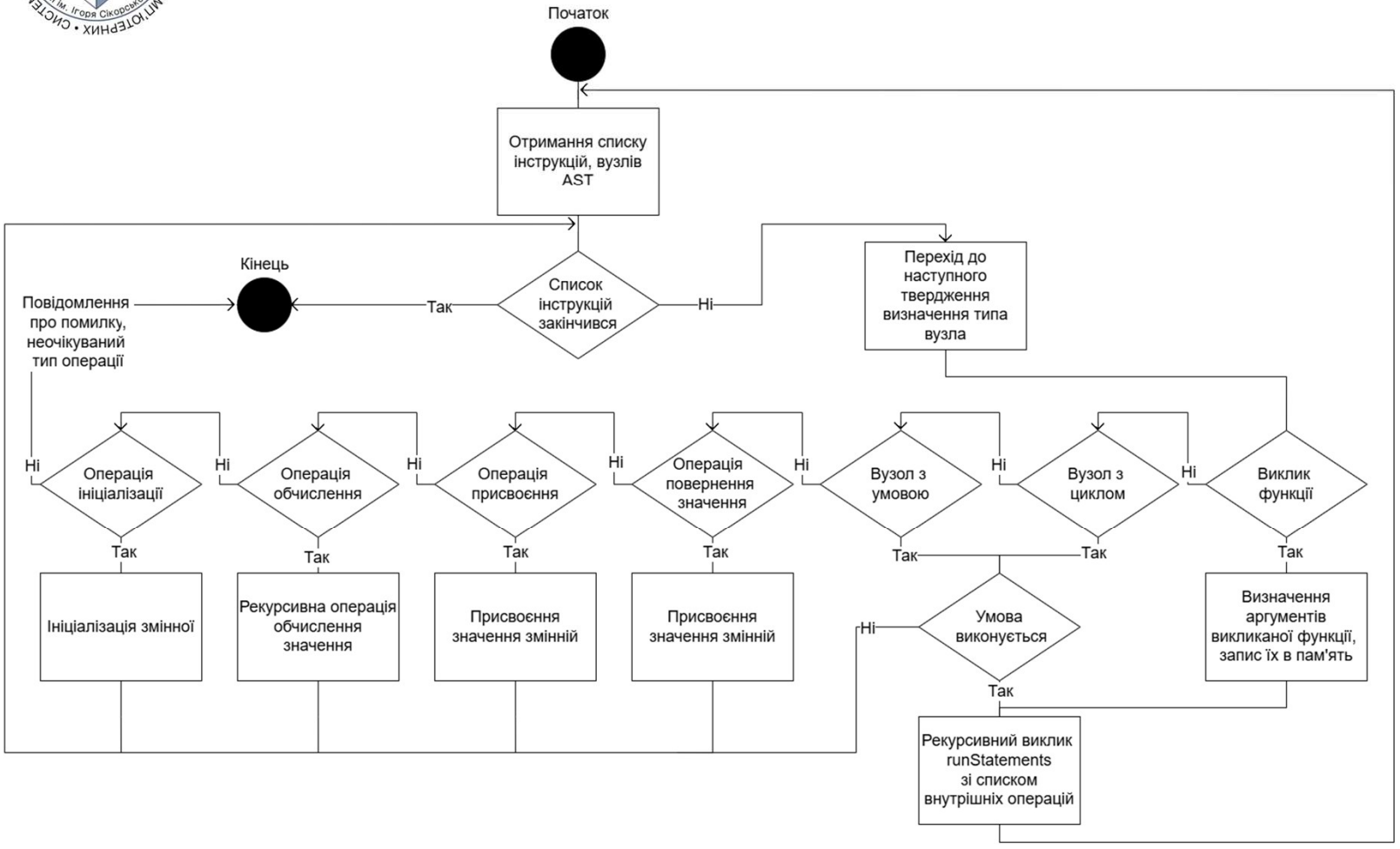

```java
package ua.kp13.mishchenko;

public enum TokenType {
    NUMBER, VARIABLE, STRING, FALSE, TRUE, TIME_VALUE,
    OPER_AND, OPER_OR, OPER_NOT, OPER_IS, OPER_LESS, OPER_MORE,
    OPER_LESS_EQUALS, OPER_MORE_EQUALS, OPER_IS_NOT,
    OPER_EQUALS, OPER_PLUS, OPER_MINUS, OPER_MULTIPLY, OPER_DIVISION, OPER_POWER,
    TYPE_INT, TYPE_STRING, TYPE_FLOAT, TYPE_BOOLEAN, TYPE_TIME, NEXT_LINE,
    TYPE_TUPLE, TYPE_AGREGATE,
    FLOATING_POINT, DOUBLE_QUOTES,
    BRACKET_CURLY_OPENED, BRACKET_CURLY_CLOSED, BRACKET_OPENED, BRACKET_CLOSED,
    IF, ELIF, ELSE,
    WHILE, FOR,
    RETURN, FUNCTION, VOID, COMMA, TWO_DOT,
    BRACKET_SQUARE_OPENED, BRACKET_SQUARE_CLOSED
}
```

Опис формування абстрактного синтаксичного дерева

int a = 1 + 2 + 3 * 4;

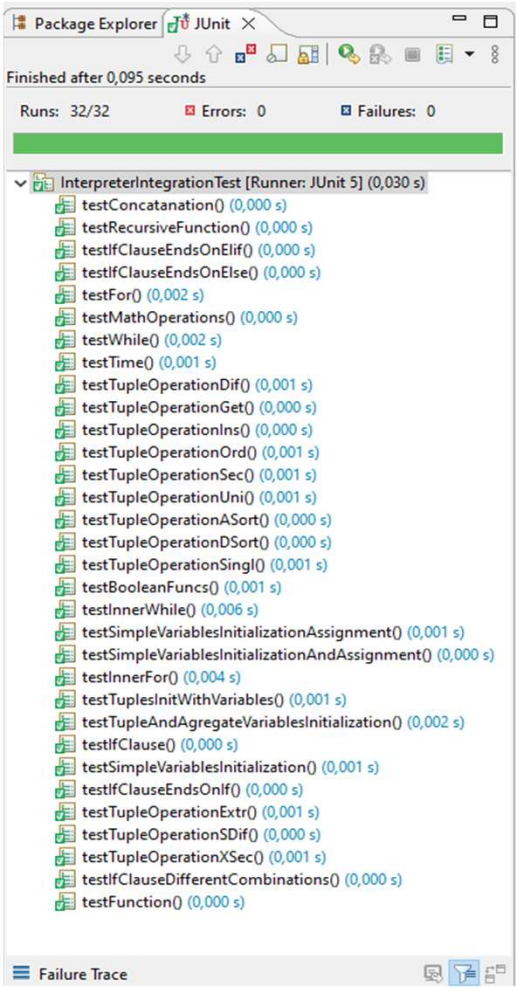


Опис обходу абстрактного синтаксичного дерева інтерпретатором




```
Code:
float a = 7+5+8*4/12*(12+3)/0.05;
###################
Abstract Syntax Tree:
Initialization: float a
  Assignment: a
    BinaryOp: +                                  <-6
     left :      BinaryOp: +                     <-5
      left :       Number: 7
      right:       Number: 5
     right:      BinaryOp: /                     <-4
      left :      BinaryOp: *                     <-3
       left :       BinaryOp: /                  <-2
        left :        BinaryOp: *                <-1
         left :         Number: 8
         right:         Number: 4
        right:        Number: 12
       right:       BinaryOp: +                  <-2
        left :        Number: 12
        right:        Number: 3
      right:      Number: 0.05
```

Логіка інтегрованого тестування інтерпретатору




```java
@Test
public void testFunction() throws Exception {
    String code = "int a=5; int b=6; int function sum(int a, int b){return a+b;}; int c = sum(a,b);";
    Map<String, VariableEntry> expectedMap = new HashMap<String, VariableEntry>();
    expectedMap.put("a", new VariableEntry(TokenType.TYPE_INT, "a", 5));
    expectedMap.put("b", new VariableEntry(TokenType.TYPE_INT, "b", 6));
    expectedMap.put("c", new VariableEntry(TokenType.TYPE_INT, "c", 11));

    Interpreter interpr = runCode(code, false);
    Assert.assertTrue(interpr.getVariableMap().equals(expectedMap));
}
```

ДЕМОНСТРАЦІЯ РОБОТИ ІНТЕРПРЕТАТОРА

code.asamp – Блокнот

Файл Правка Формат Вид Справка

```
int function factorial(int n){
        if(n==0){
                return 1;
        }
        int b = n - 1;
        int mul = factorial(b);
        int res = n * mul;
        return res;
}
int r = factorial(5);
string text = "factorial 5: ";
print(text);
print(r);
```

C:\WINDOWS\system32\cmd.exe

```
C:\Users\vladm\eclipse-workspace\asampl_2_0_compiler\target>java -jar asampl_2
factorial 5:
120

C:\Users\vladm\eclipse-workspace\asampl_2_0_compiler\target>
```

C:\WINDOWS\system32\cmd.exe - java -jar asampl_2_0_compiler.jar "C:\Users\vladm\Desktop\...

```
C:\Users\vladm\Desktop\asampl_2_0_env\asampl_2_0_compiler\target>java -jar asampl_2
_0_compiler.jar "C:\Users\vladm\Desktop\code.txt" true
###################
Code:
int function factorial(int n){
        if(n==0){
                return 1;
        }
        int b = n - 1;
        int mul = factorial(b);
        int res = n * mul;
        return res;
}

int r = factorial(5);
string text = "factorial 5: ";
print(text);
print(r);
###################
Abstract Syntax Tree:
Function : factorial
 Return type: int
 Arguments:
  VariableEntry [type=TYPE_INT, value=null, name=n]
 Inner statements (5):
   IF clause:
    expression:
     LogicalOp: ==
     left :        Variable: n
```

ВИСНОВКИ

- Після аналізу технологій і готових рішень, було спроектовано архітектуру інтерпретатора.
- Далі було розроблено сам програмний продукт, який містить у собі три компоненти.
- Було розроблено алгоритм виконання програми у режимі дебагінгу
- Інтерпретатор було протестовано шляхом покриття компонентів інтегрованими тестами.

Дякую за увагу!

Факультет прикладної математики

Кафедра програмного забезпечення комп'ютерних систем

"ЗАТВЕРДЖЕНО"

Завідувач кафедри

_____ Євгенія СУЛЕМА

"___" _____ 2024 р.

ІНТЕГРОВАНЕ СЕРЕДОВИЩЕ РОЗРОБЛЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ МОВОЮ ASAMPL 2.0. КОМПІЛЯТОР

Програма та методика тестування

ДП.045300-04-51

"ПОГОДЖЕНО"

Керівник проєкту:

_____ Євгенія СУЛЕМА

Нормоконтроль:

_____ Микола ОНАЙ

Виконавець:

_____ Владислав МІЩЕНКО

2024

ЗМІСТ

1. ОБ’ЄКТ ВИПРОБУВАНЬ

Інтерпретатор, що відтворює застосунки, написані за формальним описом мови програмування ASAMPL 2.0, написаний мовою Java.

2. МЕТА ТЕСТУВАННЯ

У процесі тестування має бути перевірено наступне:

1) функціональна працездатність компонентів інтерпретатора;
2) коректність роботи інструменту у режимі дебагінгу;
3) забезпечення правильного відпрацювання вбудованих функцій мови ASAMPL 2.0;
4) відповідність вимогам Технічного завдання.

3. МЕТОДИ ТЕСТУВАННЯ

Тестування виконується методом Gray Box Testing. Перевіряється як код, так і безпосередньо програмний продукт на відповідність вимогам.

Використовуються наступні методи:

1) функціональне тестування, зокрема на рівні Critical path test (базове тестування);
2) тестування продуктивності програмного забезпечення, зокрема Stability testing (тестування стабільності) ;
3) тестування взаємодії компонентів, інтегроване тестування;
4) димове тестування.

4. ЗАСОБИ ТА ПОРЯДОК ТЕСТУВАННЯ

Працездатність інтерпретатора перевіряється шляхом:

1) динамічного ручного тестування – введенням граничних та недопустимих значень у якості різних комбінацій інструкцій, що описані у програмі;

2) динамічного ручного тестування на відповідність функціональним вимогам;
3) статичного тестування коду, перегляд коду;
4) тестування компілятору на різних комп’ютерах;
5) тестування стабільності роботи при різних умовах;
6) тестування шляхом покриття компонентів автоматизованими тестами;
7) тестування побудованого виконуваного файлу.

Факультет прикладної математики

Кафедра програмного забезпечення комп'ютерних систем

"ЗАТВЕРДЖЕНО"

Завідувач кафедри

_____ Євгенія СУЛЕМА

"___" _____ 2025 р.

ІНТЕГРОВАНЕ СЕРЕДОВИЩЕ РОЗРОБЛЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ МОВОЮ ASAMPL 2.0. КОМПІЛЯТОР

Керівництво користувача

ДП.045440-05-34

"ПОГОДЖЕНО"

Керівник проєкту:

_____ Євгенія СУЛЕМА

Нормоконтроль:

_____ Микола ОНАЙ

Виконавець:

_____ Владислав МІЩЕНКО

2025

ЗМІСТ

1. Програмні вимоги для запуску інтерпретатора

Для того, щоб використовувати компілятор, на комп'ютері повинен бути встановлений Java Runtime Environment 1.8+ версії. Це дозволить запускати застосунки, що були написані мовою програмування Java. На рис. 1 можна побачити приклад перевірки на наявність встановленої JRE.

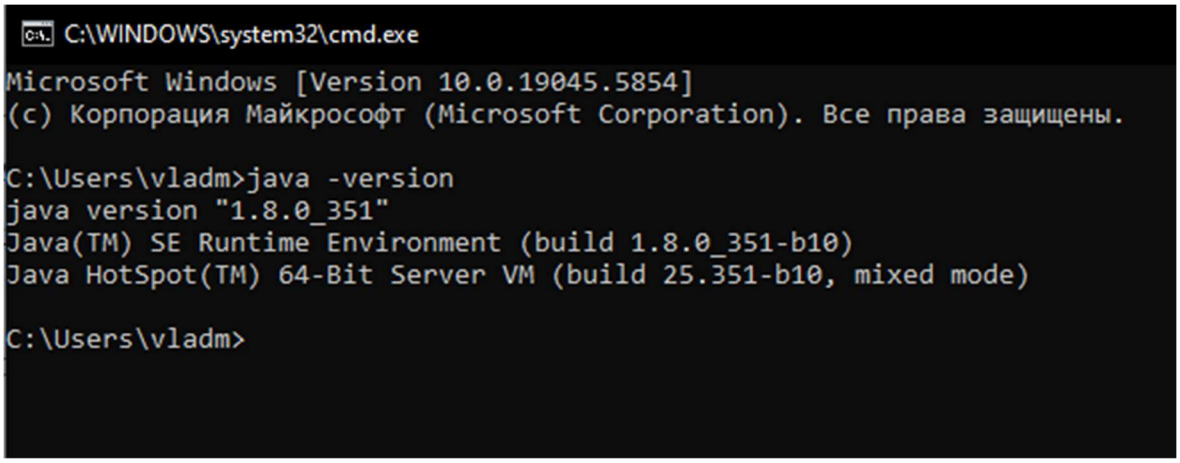


Рис. 1. Перевірка на наявність встановленого Java Runtime Environment

Після того, як JRE буде встановлено, можна користуватися інструментом. Варто зазначити, що розроблений інтерпретатор працюватиме на усіх популярних операційних системах, куди можливо встановити середовище виконання Java (Windows, Linux, MacOS).

2. Опис послідовності дій для запуску програми мовою ASAMPL 2.0

Для виконання програми, необхідно створити файл текстового формату, у якому міститиметься код, після чого, у командному рядку відтворити наступну команду:

```
java -jar asampl_2_0_compiler.jar «path» «isDebug»
```

Де параметрами виступають:

1. *path* – абсолютний шлях до файлу з кодом;

2. *isDebug* – булевий (true/false) параметр, що відповідає за запуск інтерпретатору у режимі дебагінгу;

Приклад відтворення програми (рис. 2) у звичайному режимі і у режимі дебагінгу можна побачити на рис. 3-4.

code.txt – Блокнот

Файл Правка Формат Вид Справка

```
int function factorial(int n){
        if(n==0){
                return 1;
        }
        int b = n - 1;
        int mul = factorial(b);
        int res = n * mul;
        return res;
}

int r = factorial(5);
string text = "factorial 5: ";
print(text);
print(r);
```

Рис. 2. Програма, написана мовою ASAMPL 2.0

Рис. 3. Демонстрація роботи інтерпретатора у звичаному режимі

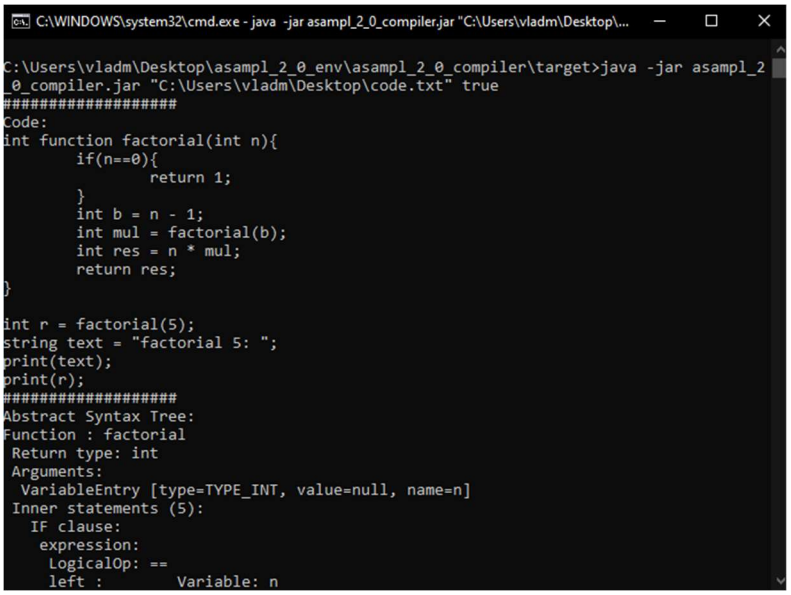


Рис. 4. Демонстрація роботи інтерпретатора у режимі дебагінгу