Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
«КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра мультимедійних систем факультету інформатики

Побудова інтерпретатора з використанням скінченних автоматів на основі стеку

Текстова частина до курсової роботи

Виконав студент 4 курсу
БП «Інженерія програмного забезпечення» Романюк Олександр Андрійович

Керівник курсової роботи
асистент Корнійчук
Максим Анатолійович

Київ – 2023

КАЛЕНДАРНИЙ ПЛАН ВИКОНАННЯ

ДИПЛОМНОЇ РОБОТИ

Тема: Побудова інтерпретатора на основі скінченного автомату

Календарний план виконання роботи:

№ п/п	Назва етапу курсового проекту (роботи)	Термін виконання етапу	Примітка
2.1.	Отримання завдання на курсову роботу.	30.06.2022	
2.2.	Ознайомлення з існуючою інформацією по темі	05.07.2022	
2.3.	Ознайомлення з існуючими системами-аналогами роботи	30.07.2022	
2.4.	Початок створення практичної частини	15.08.2022	
2.5.	Початок написання теоретичної частини	15.10.2022	
2.6.	Подання проміжної версії практичної частини		
2.7.	Аналіз практичної частини; її коригування		
2.8.	Остаточне завершення написання теоретичної частини роботи та розробки практичної частини; коригування		
2.9.	Створення презентації		
2.10	. Захист курсової роботи		

Студент Романюк О.А.

Керівник Борозенний С.О.

"_____" _________________

ЗМІСТ

КАЛЕНДАРНИЙ ПЛАН ВИКОНАННЯ ДИПЛОМНОЇ РОБОТИ………………2

ЗМІСТ 3

Перелік термінів та умовних позначень 4

ВСТУП.. 5

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ. ПОСТАНОВКА ЗАВДАННЯ ДИПЛОМНОЇ РОБОТИ 6

1.1 Аналіз сучасного стану питання 6

1.2 Постановка задачі 8

1.3 Використане програмне забезпечення 9

РОЗДІЛ 2. ТЕОРЕТИЧНІ ВІДОМОСТІ 10

2.1 Основні підходи до запуску програм 10

2.1 Поняття інтерпретатора 11

2.2 Метод токенізації у інтерпретаторах.. 13

2.4 FSM як варіант покращення алгоритму токенізації 14

РОЗДІЛ 3. ОПИС РЕАЛІЗАЦІЇ ПРОГРАМНОГО ПРОДУКТУ 16

3.1 Опис синтаксису мови програмування 16

3.2 Створення базових концепцій 16

3.3 Створення першої FSM моделі 17

3.4 Масштабування архітектури із застосування патернів програмування 19

3.5 Особливості роботи з пам’яттю 21

3.6 Тестування ситеми 21

ВИСНОВОК 23

ВИКОРИСТАНІ ДЖЕРЕЛА... 25

Додатки 26

Перелік термінів та умовних позначень

FSM (Finite state machine)— різновид автомата-абстракції, що використовується для опису шляху зміни стану об'єкта в залежності від поточного стану та інформації отриманої ззовні.

Стан — це опис статусу системи, яка очікує на виконання переходу.

Перехід — це набір дій, які мають бути виконані, коли виконується умова або коли надходить подія.

Оптимізація FSM – пошук машини з мінімальною кількістю станів, яка виконує ту саму функцію.

ВСТУП

Головною тенденцією в розвитку інформаційних технологій на сьогодні є розвиток Інтернету, вдосконалення його можливостей задля задоволення потреб користувачів, зокрема шляхом створення великої кількості різноманітних користувацьких веб-застосунків. Широкого застосування на сьогоднішній день набули системи онлайн компіляторів, перекладачів коду з однієї мови програмування на інші та систем які покликані полегшити процес розробки.

Наразі високою популярністю користується зокрема сервіс Chat GPT, особливо серед розробників. Зокрема через його можливості синтаксичного аналізу програмного коду. Зазвичай це необхідно, щоб переконатись, що код написаний правильно. Це може включати перевірку відкриття та закриття дужок, лапок та інших роздільників, перевірку наявності правильних ключових слів та операцій. Даний сервіс, як і більшість аналогів у сфері аналізу програмного коду, використовують алгоритм токенізації для розбиття вхідного тексту на лексичні токени. Такий метод вважається досить ефективним та широко використовується у галузі аналізу програмного коду через його простоту та швидкість. Проте у даній роботі спробуємо продемонструвати алгоритм , оснований на скінченних автоматах для синтаксичного аналізу програмного коду.

Мета: Створити зручний інтерпретатор за допомогою скінченних автоматів для синтаксичного аналізу вхідного тексту та можливостей його виконати.

Завдання: Розробити систему, здатну розпізнавати вхідних набір символів на наявність у ній програмного коду чи математичного виразу. У випадку, якщо програма не виявить у тексті помилок – спробувати виконати вхідний програмний код чи обрахувати математичний вираз у ньому.

Практичне значення: Інтерпретатор, що базується на FSM, може бути корисним у різних областях, включаючи освіту, де такий інструмент може допомогти студентам вивчити основи програмування, в промисловості, де він може бути використаний для автоматичної обробки та аналізу коду; у наукових

та інженерних застосуваннях, де потрібно обробляти складні математичні вирази.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ. ПОСТАНОВКА ЗАВДАННЯ ДИПЛОМНОЇ РОБОТИ

1.1 Аналіз сучасного стану питання

Інтерпретатори на основі кінцевих автоматів (FSM) були областю досліджень в інформатиці протягом кількох десятиліть. Конструкція інтерпретаторів з використанням FSM була широко вивчена завдяки їх здатності обробляти послідовні дані високоефективним способом. Використання інтерпретаторів на основі FSM знайшло своє застосування у різних областях, таких як обробка природної мови, мови програмування та обробка сигналів.

Принцип дії автоматів можна спостерігати в багатьох пристроях сучасного суспільства, які виконують заздалегідь визначену послідовність дій в залежності від послідовності подій, з якими вони пов'язані. Простими прикладами є торгові автомати, які видають продукти, коли в них опускається відповідна комбінація монет; ліфти, послідовність зупинок яких визначається поверхами, на які їдуть пасажири; світлофори, які змінюють послідовність сигналів, коли на них чекають автомобілі; кодові замки, які вимагають введення послідовності чисел у відповідному порядку.

Скінченний автомат має меншу обчислювальну потужність, ніж деякі інші моделі обчислень, такі як машина Тюрінга. Різниця в обчислювальній потужності означає, що існують обчислювальні завдання, які машина Тюрінга може виконати, але не може FSM. Це пов'язано з тим, що пам'ять FSM обмежена кількістю станів, які вона має. Скінченний автомат має таку ж обчислювальну потужність, як і машина Тюрінга, яка обмежена таким чином, що її голова може виконувати лише операції "читання" і завжди повинна рухатися зліва направо. FSM вивчаються в більш загальній області теорії автоматів.

Створення та оптимізація FSM може бути викликом, особливо для великих та складних систем.

1. Процес проектування FSM включає визначення станів, переходів та дій, що мають відбуватися при переході між станами. Це вимагає глибокого розуміння системи та може бути складним для великих систем.
2. Оптимальна FSM має мінімальну кількість станів та переходів, які достатньо для моделювання системи. Алгоритми мінімізації FSM шукають способи зменшити кількість станів та переходів, не втрачаючи при цьому потрібної функціональності.
3. Залежно від вимог до продуктивності та ресурсів, FSM може бути виконана у вигляді програмного коду, апаратного забезпечення або комбінації обох. Вибір найкращого варіанту виконання може бути непростим.
4. Перевірка того, що FSM правильно відображає очікуване поведінку системи, є важливою частиною процесу розробки. Це може включати формальну верифікацію, тестування або інші методи.
5. Як система еволюціонує або змінюється, FSM може потребувати оновлення або модифікації. Це може включати додавання нових станів або переходів, зміну дій при переходах між станами або перегляд умов переходів.

Одним із викликів у створенні інтерпретаторів на основі FSM є розробка алгоритмів, які можуть ефективно створювати та оптимізовувати FSM-моделі. Це пов’язано із сучасною тенденцією розвитку мов програмування, які передбачають постійні оновлення і додавання до них нових структур і засобів розробки. Для розв’язання цього виклику було запропоновано різноманітні методи створення інтерпретаторів на основі FSM, такі як метод підходу “live-to-dead approach” та методи побудови з використанням регулярних виразів.

Метод "live-to-dead approach" є одним із підходів до побудови інтерпретаторів на основі кінцевих автоматів (FSM). Цей підхід базується на ідентифікації "живих" і "мертвих" станів у FSM. "Живий" стан визначається як стан, з якого можна досягнути кінцевого стану. Навпаки, "мертвий" стан - це стан, з якого кінцевий стан недосяжний. Суть цього підходу полягає в тому, щоб

спочатку визначити всі можливі "мертві" стани, а потім побудувати FSM, уникаючи цих станів. Такий підхід може полегшити процес побудови FSM і зменшити кількість станів, що спрощує розробку та аналіз.

Регулярні вирази – це потужний інструмент для представлення та впорядкування даних, які можуть бути використані для побудови FSM. Суть цього методу полягає в тому, що кожен регулярний вираз може бути представлений як FSM. Кожен символ або група символів у регулярному виразі відповідає стану в FSM, а переходи між станами визначаються операторами регулярних виразів (наприклад, конкатенація, альтернація). Цей підхід дозволяє автоматизувати процес побудови FSM і значно спрощує його для великих або складних систем. Більше того, регулярні вирази дозволяють виконувати складні операції з пошуку і відповідності, що можуть бути корисні в багатьох областях застосування FSM.

Однак, незважаючи на це, питання ефективності та оптимізації інтерпретаторів на основі FSM залишається актуальним і досі.

Таким чином дослідження конструкції інтерпретаторів на основі FSM є актуальним та важливим завданням в галузі інформатики, яке вимагає подальшого розвитку та вдосконалення методів створення та оптимізації FSM-моделей.

1.2 Постановка задачі

Головною метою даної роботи є розробка швидкого та ефективного інтерпретатора, який здатен обробляти вхідний текст і перетворювати його на послідовність команд для виконання на конкретній мові програмування. Для досягнення цієї мети, планується проведення досліджень щодо застосування FSM (Finite State Machine) в інтерпретаторах.

Архітектура інтерпретатора має велике значення для його ефективності та масштабованості. Вона повинна бути гнучкою, щоб дозволити додавання нових структур мови програмування без необхідності революційних змін в коді. Це може бути досягнуто шляхом використання модульної архітектури, де різні частини інтерпретатора (наприклад, синтаксичний

аналізатор, семантичний аналізатор, генератор коду) розробляються та підтримуються окремо. Такий підхід є особливо корисним для FSM-інтерпретаторів, оскільки він дозволяє легко додавати, видаляти або модифікувати стани та переходи в автоматі, не впливаючи на інші частини системи. Крім того, він забезпечує гнучкість для оптимізації FSM, таких як мінімізація кількості станів або переходів.

Більш того, належне проектування архітектури інтерпретатора може сприяти покращенню продуктивності. Наприклад, архітектура, що оптимізує використання ресурсів (наприклад, пам'ять або процесорний час) для часто використовуваних операцій, може значно покращити загальну продуктивність системи.

1.3 Використане програмне забезпечення

Основним засобом для розробки було обрано мову Java. Дана мова програмування на мою думку є оптимальним підходом до розв’язання поставленої задачі через її особливості. Перш за все це її об'єктно-орієнтована парадигма, що дозволяє моделювати складні структури даних та алгоритми. А також той факт, що Java підтримує вбудовані засоби для роботи зі збіркою сміття, що зменшує ризик виникнення пам'яткових помилок. Також Java обрана як основний інструмент для розробки інтерпретатора на основі FSM з наступних причин:

1. **Платформна незалежність**: Java відома своєю "записуйте один раз, запускайте будь-де" філософією. Код, написаний на Java, може бути виконаний на будь-якій платформі, що має встановлену Java Virtual Machine (JVM). Це означає, що інтерпретатор, розроблений на Java, буде масштабованим і зручним для використання на різних платформах.
2. **Об'єктно-орієнтована мова**: Java є об'єктно-орієнтованою мовою, що дозволяє використовувати принципи наслідування, інкапсуляції та поліморфізму. Це допомагає створити чистий та легко розширюваний код.

3. **Стандартні бібліотеки**: Java має широкий вибір стандартних бібліотек, які можуть спростити розробку інтерпретатора. Наприклад, бібліотеки для роботи з рядками, колекціями, потоками даних тощо.
4. **Розробка багатопоточних додатків**: Java має вбудовані засоби для багатопоточного програмування. Це може допомогти покращити продуктивність інтерпретатора за рахунок паралельного виконання завдань.
5. **Безпека:** Java розроблена з урахуванням безпеки, що включає в себе сильну типізацію, автоматичне управління пам'яттю, захист від переповнення буфера та інших типів атак. Система управління пам'яттю Java також зменшує ймовірність виникнення помилок, пов'язаних з пам'яттю.
6. **Спільнота розробників:** Java має одну з найбільших спільнот розробників у світі, що означає широку підтримку, велику кількість ресурсів для навчання та безліч відкритого програмного забезпечення. Це може бути корисним при розробці та налагодженні інтерпретатора.
7. **Підтримка з боку великих компаній:** Java активно підтримується і розвивається такими гігантами технологічної галузі, як Oracle та Google, що гарантує її стабільність та надійність.

Для відображення кінцевого результату та можливості користувачів спробувати розроблену систему буде використано клієнт серверну архітектуру на основі Spark-Java. Spark Java є легким та простим у використанні фреймворком для створення веб-додатків, що дозволяє швидко розробляти веб-застосунки на мові Java та запускати їх на вбудованому веб-сервері без складних налаштувань.

РОЗДІЛ 2. ТЕОРЕТИЧНІ ВІДОМОСТІ

2.1 Основні підходи до запуску програм

Коли мова йде про програмування, одне з найбільш поширених питань полягає в тому, який з підходів до перетворення вихідного коду програми в

машинний код є кращим. Існують два основних підходи: інтерпретатор та компілятор.

Інтерпретатор читає вихідний код програми рядок за рядком та виконує його відразу. Виконання програми відбувається в той же час, коли код інтерпретується. Це означає, що якщо в програмі є помилки, вони зазвичай виявляються під час виконання програми. Інтерпретатор не створює окремий файл машинного коду, тому що код перетворюється в машинний код прямо під час виконання. Інтерпретатори зазвичай повільніше за компілятори, оскільки потребують більше часу на обробку коду при кожному запуску програми.

З іншого боку, компілятор аналізує весь вихідний код програми і перетворює його в машинний код в одному з двох випадків: на етапі компіляції або під час виконання спеціальної програми – динамічної бібліотеки (JIT-компіляція). Якщо в програмі є помилки, вони зазвичай виявляються на етапі компіляції. Компілятори створюють окремий файл машинного коду, який можна використовувати для запуску програми без необхідності перетворення вихідного коду знову. Компілятори зазвичай швидше за інтерпретатори, оскільки потребують більше часу на обробку коду при одноразовій компіляції, але дозволяють більш швидке виконання програми в подальшому.

2.1 Поняття інтерпретатора

Як уже згадувалось раніше – інтерпретатор є програмою, яка здатна обробляти вхідний текст і перетворювати його на послідовність команд для виконання на конкретній мові програмування. Основна мета інтерпретатора полягає у забезпеченні швидкого та ефективного виконання програми без необхідності її компіляції. Зазвичай розробка інтерпретатора починається з визначення мови програмування, для якої буде розроблятися інтерпретатор. Для цього необхідно вивчити синтаксис та семантику мови, а також зрозуміти основні принципи її роботи.

Зазвичай, в структурі інтерпретатора виділяють дві основні компоненти: лексичний аналізатор та синтаксичний аналізатор. Лексичний аналізатор служить для розбиття вхідного тексту на лексеми (слова, числа, знаки операцій

тощо). Синтаксичний аналізатор використовує отримані лексеми та перевіряє правильність синтаксичної конструкції вихідної програми. Важливо зазначити, що структура інтерпретатора може бути різною залежно від мови програмування та її вимог.

Після визначення структури інтерпретатора, розроблюється алгоритм інтерпретації, який складається з послідовності команд, що виконуються при обробці кожної лексеми вхідного тексту. Цей алгоритм повинен бути оптимальним з точки зору швидкості та ефективності роботи інтерпретатора.

Часто в таких випадках, виділяють різні інтерпретатори відповідно до їх будови:

1) Інтерпретатор на основі складових частин. Даний підхід полягає в тому, що програма розбивається на окремі функціональні частини, які можуть бути інтерпретовані окремо. Ці функціональні частини можуть бути написані на мовах програмування вищого рівня або ж на мовах низького рівня, таких як асемблер.

2) Інтерпретатор на основі таблиці символів. Даний підхід полягає в тому, що інтерпретатор містить таблицю символів, яка містить інформацію про змінні, функції та інші символи, які використовуються в програмі. Цей підхід є більш ефективним, ніж попередній, оскільки зменшує кількість потрібної пам'яті для зберігання програми.

3) Інтерпретатор на основі стеку. Цей підхід полягає в тому, що інтерпретатор використовує стек для зберігання інструкцій та даних, необхідних для виконання програми. Кожен раз, коли інтерпретатор зустрічає нову інструкцію, він додає її до стеку. Коли інструкція виконується, інтерпретатор видаляє її зі стеку.

Кожен з цих способів має свої переваги та недоліки і вибір конкретного методу залежить від потреб користувача та вимог до програм. Варто зазначити, що також існують гібридні підходи до реалізації інтерпретаторів, які комбінують різні методи для досягнення більшої ефективності та гнучкості в обробці програм.

2.2 Метод токенізації у інтерпретаторах

Метод токенізації – це процес розбиття вхідного тексту на окремі частини, які називають токенами. Токени можуть бути словами, числами, знаками операцій та іншими елементами мови програмування, які необхідно обробляти в процесі виконання програми. У інтерпретаторах, метод токенізації використовується для зчитування вхідного тексту програми та його розбиття на окремі токени. Цей процес зазвичай відбувається перед виконанням програми, коли вхідний текст перетворюється на послідовність токенів, які зберігаються в пам'яті. Токенізація може бути реалізована різними способами. Наприклад, в інтерпретаторах мов програмування, які мають чітку граматику, токенізація може бути здійснена за допомогою лексичного аналізатора, який використовує регулярні вирази для визначення токенів. У інших випадках, коли мова програмування має більш складну граматику, можуть використовуватися більш складні методи токенізації, такі як алгоритм рекурсивного спуску.

Даний алгоритм знайшов широке застосування, хоча не позбавлений деяких недоліків, таких як:

1. Проблема зі змістом. В деяких випадках, даний метод може не враховувати контекст або зміст тексту в якому він зараз перебуває. Це може призвести до неправильного розуміння тексту і, як наслідок, до помилок у виконанні команд.
2. Проблема зі синтаксисом. Інтерпретатор може мати внутрішні правила синтаксису, які можуть порушитися при токенізації. Дана проблема може виникнути, якщо наприклад у випадку, коли роздільник між параметрами функції не буде визначено – інтерпретатор може не змогти правильно інтерпретувати цю функцію.
3. Проблеми з ефективністю. Якщо текст містить багато символів, то токенізація може стати дуже обтяжливою для системи. Це може призвести до збоїв в системі або до великих затримок в часі обробки тексту.
4. Проблеми з точністю. Якщо токенізація не враховує всі можливі варіанти синтаксису, то це може призвести до помилок у виконанні команд.

Наприклад, якщо токенізатор не визначає, що деякі символи можуть бути як знаками операцій, так і знаками пунктуації, то це може призвести до помилкового розуміння тексту.

В інтерпретаторах, основаних на кінцевих автоматах (FSM), процес токенізації може бути реалізований як FSM, де стани відповідають різним "режимам" розпізнавання (наприклад, всередині числа, ідентифікатора, коментаря тощо), а переходи визначаються на основі вхідних символів. Коли FSM виявляє, що токен завершено (наприклад, коли він бачить пробіл після ідентифікатора), він генерує відповідний токен для подальшої обробки. Токенізація є критично важливою для успішного аналізу та інтерпретації вхідного тексту, оскільки вона визначає, як вхідний текст буде розподілений на окремі частини, які можуть бути оброблені на наступних етапах інтерпретації.

2.4 FSM як варіант покращення алгоритму токенізації

FSM (fimite state machine) може розглядатись як один із варіантів для покращення алгоритму токенізації. Застосування скінченного автомату може зменшити кількість помилок і покращити здатність інтерпретатора до правильного розуміння тексту. Оскільки даний підхід, це певна математична модель, яка може зобразити систему, що працює в різних станах, залежно від вхідних даних, то його застосування має бути покликаним на покращення розуміння контексту інтерпретатора, що дозволяє більш точно розуміти токени.

У даному підході, планується відмовитись від регулярних виразів, а застосовувати наперед вбудований синтаксис. Це дозволить інтерпретатору більш точно розуміти токени, а також дозволить використовувати менше ресурсів на обрахунки і як наслідок збільшити ефективність системи. Передбачається, що такий підхід дозволить ефективно розпізнавати неправильні команди і повідомляти про них користувача.

Таким чином введення FSM в логіку роботи інтерпретатора може бути корисним для покращення токенізації, що дозволяє збільшити його точність та продуктивність.

Finite State Machine (FSM), або скінчений автомат, може використовуватися для покращення алгоритму токенізації. Це може зменшити кількість помилок та покращити здатність інтерпретатора правильно інтерпретувати текст.

FSM – це математична модель, яка може зобразити систему, яка працює в різних станах залежно від вхідних даних. Кінцеві автомати (FSM) можуть допомогти покращити процес токенізації за допомогою більш структурованого та систематичного підходу до розпізнавання лексичних одиниць. Ось декілька причин, чому FSM можуть бути корисними при токенізації:

1. **Ясність та структура**: FSM дозволяє чітко визначити можливі стани та переходи між ними, що дозволяє легко моделювати процес розпізнавання різних типів токенів.
2. **Ефективність**: Машини станів зазвичай дуже ефективні з точки зору часу виконання, оскільки вони працюють за принципом одного проходу (один символ обробляється за раз) і не вимагають складного аналізу або пошуку вглибину.
3. **Надійність**: FSM може допомогти виявляти та обробляти помилки на ранніх стадіях, оскільки неправильні символи або послідовності символів можуть призвести до невдалих станів або переходів.
4. **Масштабованість та гнучкість**: Із зростанням складності мови програмування можна легко додавати нові стани та переходи в FSM, що дозволяє легко адаптувати процес токенізації до нових вимог.

Для покращення токенізації планується відмовитися від використання регулярних виразів та застосовувати наперед вбудований синтаксис. Це може допомогти інтерпретатору більш точно розуміти токени та використовувати менше ресурсів на обчислення, що в свою чергу покращить ефективність системи.

Введення FSM в логіку роботи інтерпретатора може бути корисним для покращення токенізації, оскільки це дозволить ефективно розпізнавати неправильні команди та повідомляти про них користувача. Більш того,

використання FSM дозволить збільшити точність інтерпретації токенів і зменшити кількість помилок, що може забезпечити більш високу якість взаємодії між користувачем та системою.

Узагалі, використання FSM може бути корисним у випадках, коли велика кількість різних станів потрібна для точного розуміння вхідних даних. Це може забезпечити більш точну інтерпретацію.

РОЗДІЛ 3. ОПИС РЕАЛІЗАЦІЇ ПРОГРАМНОГО ПРОДУКТУ

3.1 Опис синтаксису мови програмування

На початковому етапі, слід визначити синтаксис мови програмування, яку буде розуміти наш інтерпретатор. Дана мова програмування передбачала наявність:

- Чисел
- Булевих операцій
- Математичних функцій
- Змінних/ ініціалізації змінних
- Циклів
- Типів даних
- Тернарних операторів
- Структур даних

У більшості мов програмування є певний набір зарезервованих слів, які мають спеціальне значення і не можуть використовуватись як ідентифікатори (назви змінних, функцій тощо). Відповідно до синтаксисису нашої мови програмування яку ми спробуємо реалізувати , будуть зарезервовані наступні слова: “print” , “for” , “while” , “min” , “max” , “min” , “switch” . У разі якщо користувач спробує використати їх як назви змінних то отримує помилку.

3.2 Створення базових концепцій

Оскільки наш інтерпретатор буде основаним на принципах роботи FSM , потрібно визначити базові концепції. Згідно із класичним визначенням, нам

необхідно визначити початковий стан (startState), кінцевий стан (finisState), набір тимчасових станів (temporaryStates), список можливих переходів(transitions). Додавши до цього набір базових методів, таких як отримати початковий стан (getStartState), отримати кінцевий стан (getFinishState), отримати перелік можливих переходів (getPossibleTransitions), і чи є даний стан проміжним (isTemporaryState), ми отримаємо опис базової концепції, яка відповідатиме за створення матриці переходів (MatrixBuilder).

Наступним етапом стає створення певного перемикача (Transducer), який буде здійснювати перехід і виконувати певні дії в новому стані. У нашому випадку, дана концепція буде перевіряти символ і парсити його.

Дані дві концепції, забезпечують головну логіку роботи нашої програми, які реалізовані у класі FiniteStateMachine, де на основі матриці переходів і самих переходах наша програма може здійснювати аналіз вхідного набору даних. Варто зазначити, що сам набір даних, ми зберігаємо у вигляді масиву символів, і по мірі руху по ньому зберігаємо лише порядковий номер місця, в якому програма зараз відпрацьовує. Це необхідно для того, щоб повернути користувачу помилку у разі вводу невірних даних.

3.3 Створення першої FSM моделі

У даному розділі, спробуємо описати базову концепцію того, як здійснюється аналіз вхідних даних. Першою задачею, з якою ми зіштовхнулись – стало аналіз на наявності серед вхідних даних числа. На рисунку 1 зображено графічна інтерпретація такого автомату для числа.

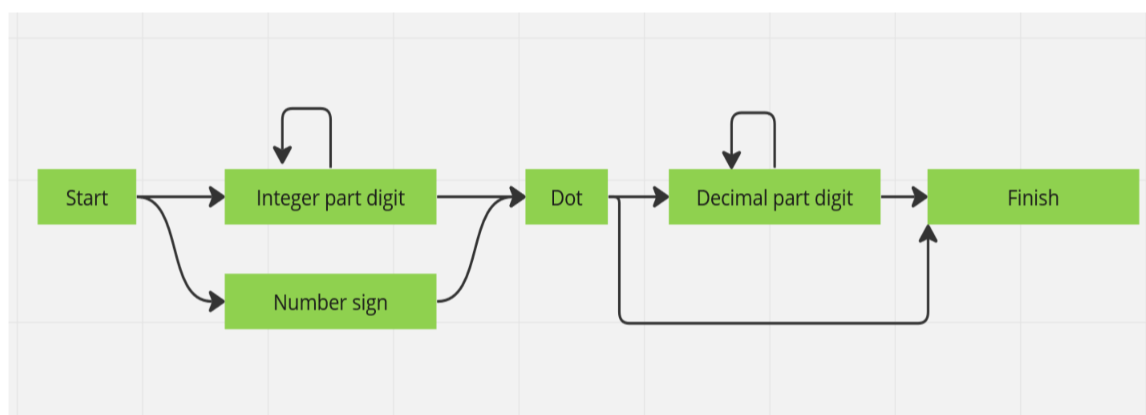

Рисунок 1. FSM – Number

У нас є початковий стан (Start) і кінцевий (Finish). Також наведені можливі стани, в які може переходити FSM. Вони побудовані на очевидній закономірності, що будь яке число може мати знак, цифри для вираження цілої частини, крапку для розділення цілої і дробової частини числа, а також цифри на вираження дробової частини. На зображені продемонстровано набір можливих переходів, які забезпечують вірне сприйняття чи є набір символів числом. У разі, якщо у нас трапиться символ, що не підпадає під жоден із станів, потрібно буде сигналізувати про помилку.

Створення першої моделі кінцевого автомата (FSM) вимагає кількох основних кроків. Давайте розглянемо їх на прикладі простого автомата для розпізнавання чисел у вхідному потоці символів.

1. **Визначення станів**: Першим кроком у створенні FSM є визначення можливих станів. У нашому випадку ми можемо мати два стани: "Всередині числа" та "Поза числом".
2. **Визначення переходів**: Наступним кроком є визначення переходів між станами. У нашому випадку:
   - Якщо ми знаходимося "Поза числом" і зустрічаємо цифру, ми переходимо в стан "Всередині числа".
   - Якщо ми знаходимося "Всередині числа" і зустрічаємо нецифровий символ, ми переходимо в стан "Поза числом".
3. **Визначення дій**: Для кожного переходу ми визначаємо дію, що виконується при цьому переході. Наприклад, коли ми переходимо від "Всередині числа" до "Поза числом", ми можемо згенерувати токен числа.
4. **Початковий та кінцеві стани**: Визначаємо початковий стан (в нашому випадку, "Поза числом") і кінцеві стани, якщо вони є.
5. **Реалізація**: За допомогою обраної мови програмування, у нашому випадку Java, ми можемо реалізувати цей FSM.

Декілька додаткових моделей FSM , для аналізу певних структур , розміщено у Додатках (Рис.5 - Рис.8).

3.4 Масштабування архітектури із застосування патернів програмування

Приклад з числом демонструє загальний підхід до того, як відбувається аналіз складних конструкцій. Будь яка нова структура, яку ми будемо вводити так чи інакше буде використовувати інші. Для прикладу для введення дужок, ми будемо використовувати попередню FSM Expression і додавати до неї новий елемент – дужки. FSM для функцій і процедур в свою чергу буде використовувати попередні структури такі як дужки і вирази. Відповідно актуальним стає питання побудови чіткої ієрархії, яка дозволить нашій програмі легко переходити між можливими станами.

При масштабуванні архітектури програмного забезпечення з використанням кінцевих автоматів (FSM), є кілька патернів програмування, які можуть виявитися корисними. (**State**): Цей патерн дозволяє об'єкту змінювати свою поведінку при зміні його внутрішнього стану. Це відповідає роботі FSM, де кожен стан може мати різні переходи та дії в залежності від вхідних даних. (**Strategy**): Цей патерн може бути корисним, коли FSM має багато різних типів переходів або дій, які повинні виконуватися в залежності від поточного стану та вхідних даних. Кожна стратегія відповідає окремому набору переходів або дій. (**Observer**): Цей патерн дозволяє об'єктам підписуватися на оновлення стану інших об'єктів. Це може бути корисно, якщо є багато FSM, які повинні взаємодіяти один з одним або якщо деякі дії повинні виконуватися тільки при певних змінах стану. (**Singleton**): Якщо FSM є центральною частиною системи і її повинна бути тільки одна, цей патерн може забезпечити, що буде створено тільки один екземпляр FSM. (**Command**): Цей патерн дозволяє інкапсулювати запити в як об'єкти, що дозволяє робити запити на основі параметрів і ставити запити в чергу. Це може бути корисно, коли FSM повинен виконувати складні дії або коли дії повинні виконуватися асинхронно

Таким чином, наша архітектура буде передбачати, що при загальній інтерпретації коду, в певний момент ми можемо рухатись через набір певних структур. В свої чергу, дана структура може містити інші і т. д. В такому випадку,

потрібна архітектура, яка дозволить в потрібний момент переходити в іншу структуру, пробувати аналізувати її . У разі якщо такий вхід був невірним, то намагатися відкотитися на один рівень назад і спробувати знайти структуру , яка буде відповідати нашим потребам. Якщо ж такого не буде знайдено, то вочевидь потрібно буде повертати помилку, оскільки автор надав код, який не може розпізнати система.

Фундаментом для нашої структури стане певна модель, основана на породжувальному патерні – абстрактної фабрики. Даний патерн дає змогу створювати сімейства пов’язаних об’єктів, не прив’язуючись до конкретних класів створюваних об’єктів. У звичайному випадку, коли ми створюємо об'єкти, ми зазвичай прив'язуємося до конкретного класу цих об'єктів. Однак, в даному випадку, такий підхід може бути неефективним або навіть неможливим. Імплементувавши даний патерн в нашу програму, ми можемо створювати об’єкти даних структур без прив’язки до класів цих об’єктів. Таким чином, ми виділяємо програмний код в в одне місце, що спрощуючи підтримку коду, а також спрощує можливість додавання нових структур до програми. Окрім того, ми ще й дотримуємось одного із важливих принципів Solid – принцип відкритості/закритості.

Аналогічним чином, буде реалізована і логіка на нижньому рівні. Так як у нас є структури з різною кількість можливих імплементацій – наприклад функції. Так як у програмі реалізовані різні функції, такі як sum, min, max та інші то впровадження аналогічної реалізації але порядком нижче дозволить нам покращити нашу реалізацію.

Наступним але не менш важливим моментом стає типізація змінних. На даному етапі реалізації програмного продукту, не реалізовувались ключові слова як наприклад int/double та інші. В нашому випадку ми маємо змінну і в ній може міститись будь яка інформація. Будь то це число, символ, рядок, чи булеве значення. Таким чином, після оголошення змінної ми не знаємо якого саме типу вона буде. На даному моменті, був застосований поведінковий патерн

відвідувач. В нашому випадку в залежності від того яке значення для нас приходить для змінної, ми можемо починати її обробляти.

3.5 Особливості роботи з пам’яттю

На перших етапах побудови інтерпретатора, ми намагались реалізувати аналізатор для математичних виразів. У цьому випадку, нам не потрібно було виділяти значних об’ємів пам’яті. На даному етапі, доцільно було скористатись алгоритмом Shunting Yard. Це алгоритм, що використовується для перетворення арифметичних виразів з інфіксної форми в постфіксну форму (зворотній польський запис). Інфіксна форма – це традиційний спосіб запису арифметичних виразів, в якому оператори розташовуються між операндами. Наприклад, вираз "3 + 4 * 2" – це інфіксна форма. У постфіксній формі операнди розташовані перед операторами. Наприклад, вираз "3 4 2 * +" – це його постфіксна форма. Постфіксна форма зручна для обчислення значення виразу за допомогою стеку.

Алгоритм Shunting Yard працює наступним чином:

1. Створити порожній стек для операторів і порожній вихідний список для виразу в постфіксній формі.
2. Розглянути кожен символ в інфіксному виразі зліва направо.
3. Якщо символ є операндом - додати його до вихідного списку.
4. Якщо символ є відкриваючою дужкою, додати її до стеку.
5. Якщо символ є закриваючою дужкою, видаляти оператори зі стеку та додавати їх до вихідного списку, доки не зустрінеться відкриваюча дужка. Відкриваючу дужку потрібно видалити зі стеку, але не додавати до вихідного списку.
6. Якщо символ є оператором, додати його до стеку. Перед додаванням оператора перевірити, чи вищий пріоритет має оператор в стеці. Якщо так, видалити його зі стеку та додати до вихідного списку.
7. Повторювати кроки 3-6, доки не буде розглянуто всі символи в вхідному виразі

Таким чином, використовуючи даний алгоритм, було забезпечено правильно роботу нашого інтерпретатора при роботі з математичними виразами.

Дещо схожий підхід згодом був використаний під час зберігання частин програм для подальшої інтерпретації. Але на цей раз, було використано об'єкт типу Deque, який зберігатиме об'єкти класу ShuntingYard. Таким чином, програма зможе зберігати частини коду, які наприклад знаходять в середині циклів. Що до змінних , то вони зберігаються в окремій структурі за принципом “ім’я – значення”.

3.5 Тестування системи

Для перевірки коректності результатів було створено ряд тестів , які включали у себе перевірку коректності роботи певних структур і Їх різних комбінацій. Були створені різні тестові сценарії: сценарії , які давали позитивний результат (positiveCases) , та програми які мали б впасти у конкретний момент появи у коді помилки(negativeCases). Для кожного із тестів було описано повідомлення , яке у разі не успішоного проходження тесту , допоможе розробнику швидко усвідомити причину падіння. Зокрема дані тестові випадки для тестування ініціалізації змінних наведені на Рис.2 та Рис 3.

```
public class InitVarProgramTest extends AbstractProgramTest {

    no usages  Oleksandr
    static Stream<Arguments> positiveCases() {
        return Stream.of(
                of( ...arguments: "a = 5;b = 4; print(a+b); print(a);", "[9.0][5.0]", "Two variables initialization and simple " +
                        "sum action inside print procedure test has failed."),

                of( ...arguments: "a=(2*(7+3/(2+3)^4)); print(a);", "[14.0096]", "Variable initialization with complex expression " +
                        "and procedure print test has failed."),

                of( ...arguments: "a = 5 > 2; print(5>2);", "[true]", "Boolean variable initialization test has failed"),

                of( ...arguments: "a=10;print(a);", "[10.0]", "Variable initialization and procedure print test has failed."),

                of( ...arguments: "a = 5; b = 7 + 7; print(a*b);", "[70.0]", "Two variables initialization and multiplication of them t

                of( ...arguments: "a = max(3,min(5,10)); b = 3 + 7; print(a*b);", "[50.0]", "Execution of code with nesting math functi

                of( ...arguments: "a=-7; b=4+7; c=7*2 ;print(5*(b+3)+c);", "[84.0]",
                        "3 variables initialization and calculation of them test has failed")
        );
    }
```

Рис. 2 Опис позитивних тестових випадків

```
no usages  Oleksandr
static Stream<Arguments> negativeCases() {
    return Stream.of(
            of( ...arguments: "a = 6; 7; print(a);", 7, "Number without variable test has not throw exception"),

            of( ...arguments: "a = 0,3;  print(0);", 5, "Number without wrong floating dot"),

            of( ...arguments: "print(a);", 7, "Not initialized variable test has not throw exception"),

            of( ...arguments: "a = a; print(a);", 5, "Wrong initialization of variable test has not throw exception"),

            of( ...arguments: "a; = 6 print(a);", 1, "Separator in wrong place test has not throw exception"),

            of( ...arguments: "a == 6; print(a);", 3, "Not allowed double assign test has not throw exception"),

            of( ...arguments: "a = 6 print(a);", 6, "Code without separators test has not throw exception"),

            of( ...arguments: "a = 6; 7; print(a);", 7, "Number without variable test has not throw exception")
    );
}
```

Рис. 3 Опис негативних тестових випадків

Окрім цього , було створено окремий веб інтерфейс за допомогою Spark Java , для того щоб кожен користувач системою міг скористатись даною системою. Результати роботи наведені на Рис. 4

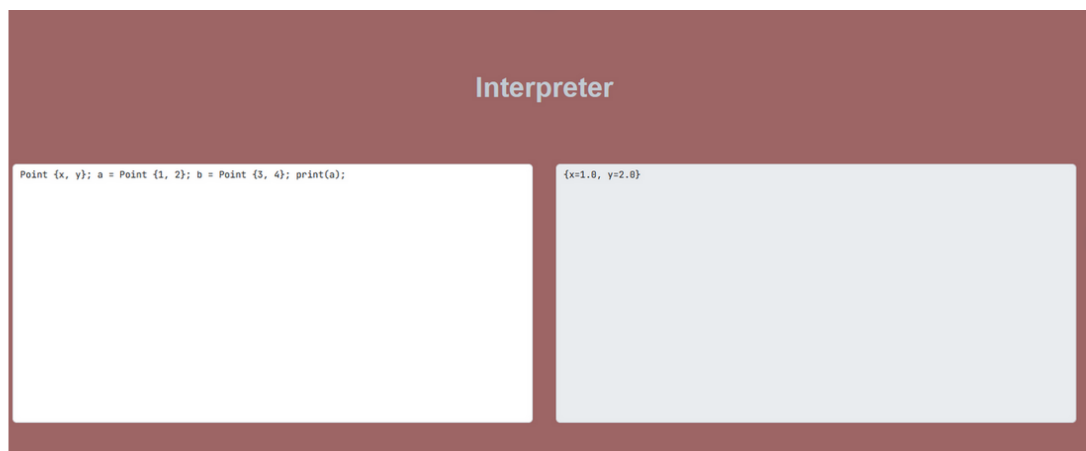

Рис.4 Результати виконання програми

ВИСНОВОК

Дана дипломна робота присвячена розробці інтерпретатора для програмного коду. Мета роботи полягала в створенні програмного забезпечення, яке може зчитувати дані, аналізувати їх та повертати результат користувачу. Для досягнення цієї мети, було проведено дослідження різних алгоритмів та методів розробки інтерпретаторів, а також створено власний алгоритм, що відповідає потребам даної роботи.

Було розглянуто основні поняття та техніки, необхідні для розуміння і реалізації інтерпретатора, описано структуру інтерпретатора та його компонентів, таких як: лексичний аналізатор, синтаксичний аналізатор та виконавча машина. Також було проаналізовано і порівняно різні алгоритми та методи розробки інтерпретаторів. У розділі про розробку було описано процес створення програмного забезпечення, починаючи від проектування архітектури інтерпретатора, написання коду та його тестування. Було розглянуто різні етапи розробки та взаємодію компонентів інтерпретатора між собою.

У результаті роботи було створено інтерпретатор, який може приймати дані, аналізувати та виводити результат. Було використано алгоритм Shunting, що забезпечило зручний спосіб обчислення та збереження даних за допомогою стеку. Під час розробки було враховано принципи чистого коду та SOLID, а також використання різного роду патернів програмування, що дозволило створити програмне забезпечення, яке є ефективним, підтримуваним та розширюваним.

Тестування програмного забезпечення дозволило виявити та виправити помилки та недоліки, та забезпечило коректну та стабільну роботу інтерпретатора. Враховуючи результати тестування, можна стверджувати, що розроблений інтерпретатор математичних виразів є надійним та функціональним засобом.

Таким чином, дана робота демонструє успішне застосування різних принципів та методів розробки програмного забезпечення для створення інтерпретатора програмного коду. Результатом роботи є ефективний та функціональний продукт, який може бути використаний для аналізу даних користувача.

ВИКОРИСТАНІ ДЖЕРЕЛА

1) "Java and Natural Language Processing" - Richard Reese

2) "Designing Finite State Machines for Real-World Applications" - Arie Avnur

3) "Using Finite State Machines in Natural Language Processing" - Ronald M. Kaplan and Martin Kay

4) Поведінкові патерни на ресурсі https://refactoring.guru/uk/design-patterns/state

5) "Design Patterns: Elements of Reusable Object-Oriented Software" авторів Erich Gamma, Richard Helm, Ralph Johnson, та John Vlissides.

6) "Head First Design Patterns" авторів Elisabeth Freeman, Eric Freeman, Bert Bates, та Kathy Sierra

7) "Introduction to Finite Automata". Geeks for Geeks. Retrieved 2021, from https://www.geeksforgeeks.org/introduction-of-finite-automata/

8) "Java Design Patterns". Tutorials Point. Retrieved 2021, from https://www.tutorialspoint.com/design_pattern/index.htm

9) Smock, J. (2020). Mastering Regular Expressions: Powerful Techniques for Perl and Other Tools (3rd ed.). O'Reilly Media.

Додатки

У даному додатку наведено декілька схем за якими будувались FSM для інтерпретації певних структур.

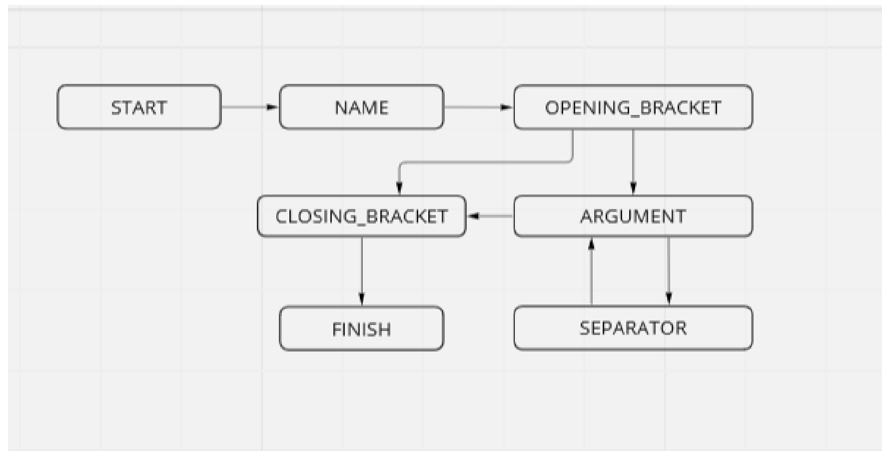

Рис.5 FSM Function and Procedure

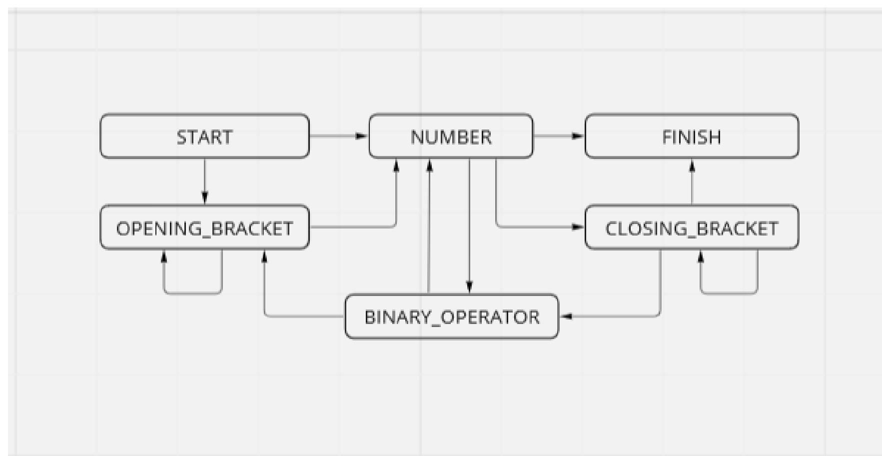

Рис.6 FSM Expression

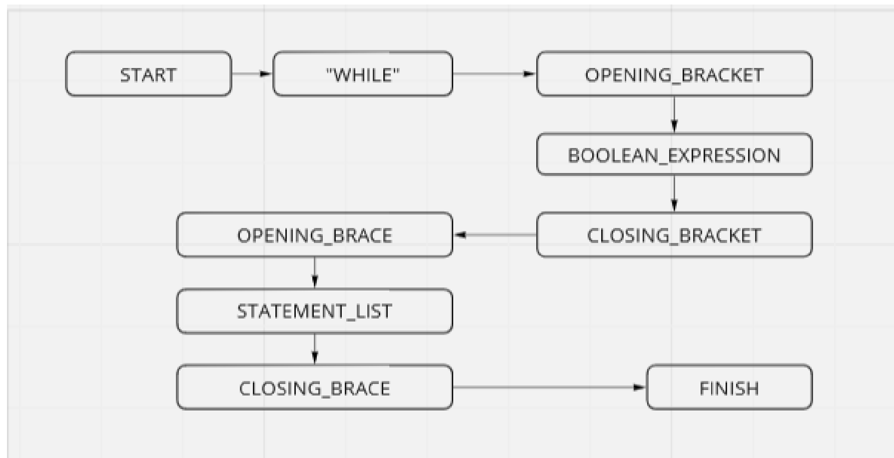

Рис.7 FSM While Operator

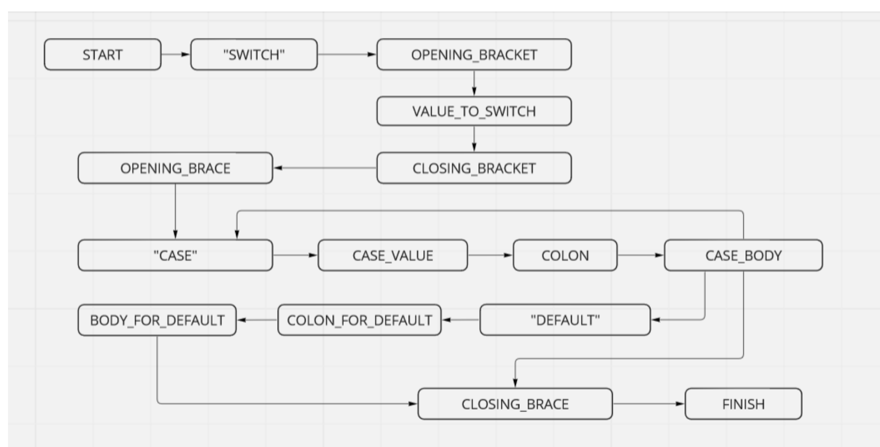

Рис.8A Switch Operator