КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА
Факультет комп'ютерних наук та кібернетики
Кафедра інтелектуальних програмних систем

Кваліфікаційна робота

на здобуття ступеня бакалавра

за спеціальністю 121 Програмна інженерія

на тему:

РОЗРОБКА ІНСТРУМЕНТАЛЬНОГО ЗАСОБУ ДЛЯ ВІДЛАГОДЖЕННЯ ПРОГРАМ МОВОЮ HASKELL ШЛЯХОМ ПОКРОКОВОГО ОБЧИСЛЕННЯ ПРОМІЖНИХ ВИРАЗІВ

Виконав студент 4-го курсу
Денис ЩЕРБАК _____ (підпис)

Науковий керівник:
асистент, кандидат фіз.-мат. наук
Костянтин ЖЕРЕБ _____ (підпис)

Засвідчую, що в цій роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент _____(підпис)

Роботу розглянуто й допущено до захисту на засіданні кафедри інтелектуальних програмних систем

«25» травня 2022 р.,
протокол № 10
Завідувач кафедри
Олександр ПРОВОТАР _____ (підпис)

Київ – 2022

РЕФЕРАТ

Обсяг роботи: 43 сторінки, кількість ілюстрацій: 15, використані джерела: 31

АБСТРАКТНІ СИНТАКСИЧНІ ДЕРЕВА, ВІДЛАГОДЖЕННЯ ПРОГРАМ, ЗНЕВАДЖЕННЯ ПРОГРАМ, НАВЧАЛЬНИЙ ІНСТРУМЕНТ, ПОКРОКОВІ ОБЧИСЛЕННЯ, HASKELL, HASKELL STEP-BY-STEP EVALUATOR.

Мета роботи: створення програмного забезпечення, що буде виконувати покрокове обчислення виразів, написаних мовою Haskell, реалізуючи можливість пошуку помилок та несправностей в кодах програм, а також граючи роль навчально-демонстраційного інструмента для людей, що починають своє знайомство з мовою.

Методи розробки: лексичний аналіз, синтаксичний аналіз, побудова абстрактних синтаксичних дерев.

Об’єкт роботи: програмне забезпечення, створене для покрокового обчислення виразів, написаних мовою Haskell.

Інструменти розробки: Середовище розробки CLion, мова програмування C++20, бібліотека Boost.Spirit v.1.79

Результати роботи: виконано загальний огляд технологій та підходів, що вже реалізовані з метою вирішення даної задачі. На основі переваг та недоліків розглянутих засобів було сформульована стратегія реалізації та був розроблений програмний продукт, що виконує покроковий обрахунок значень виразів мовою Haskell.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧЕННЯ

AST – Abstract Syntax Tree, абстрактне синтаксичне дерево;

GHC – Glasgow Haskell Compiler, Компілятор для мови Haskell з Глазго, стандартний компілятор вищевказаної мови програмування;

GHCi – Glasgow Haskell Compiler-Interpreter – інтерактивна програма для інтерпретації виразів мови Haskell в реальному часі;

GHCD – GHC Debugger – засіб відлагодження компілятора GHC;

IDE – Integrated Development Environment, інтегроване середовище розробки;

HSSE – Haskell Step-by-Step Evaluator, робоча назва програмного забезпечення, описаного в роботі;

HAST – Haskell Abstract Syntax Tree, структура, що використовується в проєкті для представлення програм мовою Haskell. В рамках цієї роботи є синонімічним терміном до AST.

ООП – об’єктно-орієнтоване програмування;

ВСТУП

Оцінка сучасного стану об’єкта дослідження або розробки.

Впродовж багатьох років мови об’єктно-орієнтованої парадигми займали домінуючу позицію серед розробників програмного забезпечення: ширше застосування ООП призводило до більшої популярності серед розробників-початківців. Більший попит серед розробників призводив до створення більшої кількості нового та актуального програмного забезпечення, в той час як нестача інструментів для функційного програмування (ФП) призводила до нестачі нових спеціалістів, які могли б реалізувати ці інструменти.

Втім, попри суттєві відмінності від канонічного ООП-підходу, ФП має свої беззаперечні переваги, які все ширше і ширше застосовуються в мовах зі змішаними парадигмами. Для грамотного використання таких засобів необхідно створення інструментів для ознайомлення розробників з принципами роботи мов ФП, а також покрокового відлагодження відповідних програм, і нестача таких програм є неабияким фактором стримування темпів розвитку індустрії.

Актуальність роботи та підстави для її виконання.

Відсутність готового і простого у використанні інструменту для відлагодження програм суттєво ускладнює розробку. Водночас, більшість існуючих інструментів зачасту не зручні для практичного застосування при відлагодженні (використовуються як навчальний інструмент з обмеженим функціоналом), або ж потребують доброго розуміння ФП для використання, тобто не є підходящим вибором для початківців.

Мета й завдання роботи.

Метою кваліфікаційної роботи є створення програмного засобу для покрокового виконання програм, написаних мовою Haskell. Для досягнення поставленої мети сформульовано наступні завдання:

- Дослідити існуючі реалізації аналогічних інструментів;

- Сформулювати вимоги до продукту, базуючись на недоліках наявних інструментів;
- Реалізувати ядро програмного продукту;
- Додати до продукту зручний інтерфейс для подальшої інтеграції з графічними засобами.

Об’єкт, методи й засоби дослідження та розробки.

Об’єктом розроблення програмного забезпечення є процес відлагодження програм мовою Haskell шляхом їх покрокового виконання.

В якості інструментів реалізації програмного засобу було обрано CLion 2021.3.3 – інтегроване середовище розробки (IDE) для мови програмування C++, що надає ліцензований доступ до своєї продукції для студентів вищих навчальних закладів всього світу.

Реалізація здійснена мовою C++ стандарту 2020 року (C++20), з використанням засобів стандартної бібліотеки та сімейства бібліотек Boost v.1.79.

Можливі сфери застосування.

Створений програмний продукт може бути широко застосований як у навчальних цілях – з метою освоїти основні принципи роботи ФП – так і для відлагодження продуктів реалізованих мовою Haskell, в процесі їх розробки.

РОЗДІЛ 1. КОНТЕКСТ ПОСТАВЛЕНОЇ ЗАДАЧІ

1.1 Розвиток мов програмування. Причини і наслідки низької популярності функційних мов.

Історію розвитку мов та технологій програмування можна розглядати починаючи з першої половини XX століття. Однак, про розповсюдженість мов варто говорити, починаючи з середини 1980-х років, оскільки тоді все більше людей почало опановувати професію розробника програмного забезпечення[1]. На той час найбільш популярними мовами були C (та її нащадок, “C with classes”, який згодом стане базисом для мови програмування C++), Pascal, Ada, Lisp та Fortran. З-поміж цих мов лише Lisp є мовою, що напряму підтримує функційну парадигму, хоча і в ньому з часом з’явилося багато засобів, що притаманні імперативним мовам, як-от цикли. Популярність решти мов можна пояснити тим фактом, що їх підхід – імперативний, покроковий, з чітким виділенням та звільненням пам’яті – був подібний до того, з чим уже було знайома велика кількість програмістів: асемблерні мови, Fortran тощо. Відповідно використання цих мов давало можливість програмувати сучасні системи і не потребувало від програміста великих зусиль в зміні парадигми. Нащадками таких мов також стануть Java, C#, Python та ін., і їх розвиток на багато років закріпив парадигму ООП як основну.

1.2 Переваги використання функційних мов.

Попри широку розповсюдженість ООП, функційні мови високого рівня розроблялися і надалі. Їх аналіз[2,3] показав наступні переваги у використанні:

1. Використання чистих функцій без збереження стану дозволяє спростити процедуру **написання надійнішого коду**. Написання функцій, що не є чистими, покладає на програміста багато додаткового навантаження, зокрема потреба завжди враховувати, чи не був змінений стан чи значення об’єктів, які використовує функція;

2. Функційні мови сильно спрощують **написання багатопоточного коду.** Недоліки імперативних мов проявляють себе найбільш сильно в багатопоточному і паралельному програмуванні, створюючи ризики появи проблем, що вже стали класичними: стан гонитви, взаємного блокування тощо. Без потреби змінювати стани об’єктів, потенціал до появи таких проблеми у функційних мовах значно обмежений;
3. **Прискорення** за рахунок використання **лінивої стратегії обчислень** (стратегії виклику за потребою) – вирази у мові Haskell не будуть обчислені до того моменту, поки не будуть безпосередньо задіяні у функції. Таким чином, обчислення, що вимагають великих витрат обчислювальних ресурсів, котрі були б повністю обраховані в класичній, строгій стратегії обчислень, але могли б бути невикористаними, будуть просто проігноровані у разі, якщо така ситуація виникне у функційній мові.
4. **Оптимізація при компіляції** – з причин, описаних в попередньому пункті, також прямим наслідком є той факт, що незалежні між собою обчислення (чисті функції) компілятор може запускати паралельно, і цей факт часто використовується зокрема в компіляторах мови Haskell.
5. **Можливість повного покриття тестами** – оскільки кожна чиста функція повертає результат незалежно від інших частин програми, написання тестів для кожної функції (в купі з перевіркою програми на коректність типів компілятором) дає дуже високу надійність роботи програм на Haskell за умови їх компіляції.

1.3 Можливі недоліки функційних програм

До основних недоліків мов ФП можна віднести:

1. **Відсутність чіткого контролю виділення пам’яті**, що присутнє в деяких імперативних мовах. Як наслідок, такі мови вимагають автоматизованого контролю пам’яті, так званих систем “збору

сміття”, що може негативно впливати на швидкодію програм. Варто, втім, відзначити, що більшість високорівневих мов програмування практикують застосування таких підходів до контролю пам’яті, а також той факт, що особливості виділення пам’яті у функційних мовах (відсутність глобальних змінних і станів) дозволяють використовувати ефективні методи контролю пам’яті.

2. Неабиякий виграш у швидкодії за рахунок обчислень за потребою може вартувати великих додаткових витрат пам’яті, необхідних для того, аби динамічно зберігати усі накопичені обчислення. Для боротьби з цим недоліком у Haskell введені спеціальні оператори, які вимагають від компілятора повністю обчислити значення аргументів перш, ніж передавати їх у функцію – однак їх вдале використання потребує якісного розуміння принципів роботи мови.

1.4 Чисті функційні мови. Проблеми їх відлагодження

Окремим недоліком у використанні мов ФП є ускладнена процедура їх відлагодження. Цей процес у імперативних мовах здійснюється звичним для розробників шляхом: покроковим виконанням інструкцій та моніторингом значень і станів об’єктів після кожної інструкції. Спрощує цей момент також те, що кількість інструкцій в межах одного рядка, як правило, дуже мала, і навіть, якщо якийсь рядок містить кілька команд, його можна розбити на групу рядків, в кожному з яких буде послідовно виконана тільки одну інструкцію, і прослідкувати хід роботи програми під час кожної з них окремо.

При роботі з функційними мовами такий підхід до зневадження унеможливлюється. Навіть, якщо семантичні правила мови дозволяють різні підходи до відступів, кожне визначення функції представляє собою одну конструкцію, що може бути скомпонована з довільної кількості інших конструкцій, таких як функції чи конструктори типів даних. Вказання рядка, на якому відбулася зупинка виконання програми, втрачає будь-який зміст. По-перше, навіть у разі, якщо компілятор окрім рядка давав би номер

символу в рядку, на якому знаходиться потрібна функція, відсутність нумерації стовпчиків зробила б процес пошуку помилки неефективним. По-друге, той факт що Haskell використовує стратегію обчислень за потребою, призводить до того, що виконання функцій буде відбуватися у порядку, відмінного від порядку їх запису у файлі програми. По-третє, немає можливості отримати значення чи стан змінних за їх відсутності, лише значення, що передані в аргументи поточної функції.

В наступному розділі буде розглянуто існуючі методи зневадження програм мовою Haskell та їх підходи до боротьби з проблемами, описаними вище.

1.5 Покрокове обчислення виразів як навчальний інструмент

Попри те, що функційні мови не користуються такою популярністю, як мови ООП, останніми роками сформувався потужний запас книг та курсів, які навчають мові Haskell. Більшість з них[4-6] в якості вправи для початківців рекомендують саме покрокове обчислення виразів за правилами мови, як вправу, що допомагає краще зрозуміти принципи ФП. В той час, як для простих програм це можна виконати вручну, інструмент, який міг би розписувати та демонструвати наочно ці кроки, з можливістю пропустити кроки, присвячені роботі уже освоєних функцій, і сфокусуватися на принципах роботи нової функції, міг би суттєво спростити процес навчання і пошуку помилок, допущених при написанні коду.

РОЗДІЛ 2. ІСНУЮЧІ ПІДХОДИ ДО РОЗВ'ЯЗКУ ПРОБЛЕМИ

2.1 Glasgow Haskell Compiler та його засоби відлагодження

Мова Haskell, як вже було зазначено, має досить довгу історію, однак значний розвиток цієї мови розпочався лише останніми роками. У зв'язку з цим не всі програми та засоби, що застосовуються розробниками, були адаптовані чи реалізовані для цієї мови. Це ж можна сказати і про засоби відлагодження (debug), які, хоч і присутні в стандартному наборі засобів для Haskell, не відповідають усім стандартам сучасного програмного забезпечення.

Перше обмеження, що накладається на GHC – він консольний, і його інструменти відлагодження не інтегровані з розповсюдженими IDE (відсутність єдиної прийнятої IDE для розробки є загальним недоліком мови). Розглянемо можливості, які надає GHCD. Доступні команди[7]:

- *:break* – Залежно від специфікації, встановлює точку зупинки виконання на потрібному рядку (можливо також вказання потрібного стовпчика) або ж за назвою функції вищого рівня (тобто функції, областю видимості якої є весь файл);
- *:step* – Продовжує виконання програми у покроковому режимі;
- *:continue* – Продовжує виконання програми, до наступної точки зупинки;
- *:list* – Демонструє визначення функції, обчислення якої було призупинено зупинкою програми. Зневаджувач має режим у якому ця команда виконується після кожної зупинки програми
- *:print* – Виводить значення певного виразу, який є частиною програми. У випадку, якщо він ще не було обчислений, демонструється лише тип виразу;
- *:force* – Виводить значення заданого виразу, обчислюючи його, якщо воно не було обчислене. Використання цієї команди може призвести до того, що робота програми, що налагоджується, буде відмінною від роботи цієї ж програми при стандартному запуску.

- *:back* – Повернення програми до попередньої точки зупинки;
- *:delete* – видалити задану точку зупинки.

Також можливий запуск обчислення виразів у покроковому режимі, поки поточна програма зупинена. В такому разі поточне виконання програми кладеться на стек, і GHCD переходить до виконання заданого в останній команді виразу. При завершенні його обчислення GHCD повертається до останньої точки зупинки на стеку. Для роботи в такому режимі передбачені команди:

- *:show* – Продемонструвати контекст, тобто якому місці в програмі перебуває поточне обчислення, а також останнє обчислення на стеку;
- *:abandon* – Перервати поточне обчислення та повернутися до останнього обчислення на стеку.

Ще один режим, в якому можливий запуск GHCD – трасування, в ході якого буде зафіксовано послідовність обчислення виразів в ході роботи програми. При його використанні використовуються наступні команди:

- *:trace* – Обчислення виразу в режимі трасування;
- *:hist* – Демонстрація номерів рядків та стовпців, що містять вирази, обчисленні в ході виконання програми, в хронологічній послідовності.

Загалом, GHCD дає досить широкий спектр можливостей, і може бути використаний при відлагодженні програм. Втім, спільнота розробників відзначає кілька його недоліків[8], через які цей засіб не користується значною популярністю:

1. При використанні GHCD дуже ускладнена можливість по-різному ставитися до викликів одної і тої ж функції в різному контексті. Для цього необхідний правильний підхід до розташування точок зупинки поза функцією, а також динамічне керування точками зупинки всередині функції, у випадку, якщо її потрібно пропустити або, навпаки, дослідити. Інструменти для відлагодження в імперативних мовах дозволяють як пропустити деталі виконання функції

(Step-over) так і заглибитися в її покрокову реалізацію (Step-in), не вимагаючи при цьому активації та деактивації точок зупинки.

2. Процес активації та деактивації точок зупинки прив’язаний до назв функцій (в такому разі точка прив’язується до всього тіла функції) або номерів рядків у файлі. Такий підхід може бути дуже незручним при відлагодженні великих програм, або таких, що складаються з великих і комплексних функцій.
3. Цей інструмент не розрахований на те, щоб опрацьовувати заздалегідь скомпільовані програми[7], тому з ним можливо лише дослідження функцій, написаних користувачем, робота вбудованих функцій завжди виконується в один крок.

2.2 Бібліотечні засоби тестування та відслідковування значень виразів

2.2.1 Бібліотека Debug.Trace

Ця бібліотека надає кілька функцій для роботи з даними, зокрема такі як Debug.Trace.trace та Debug.Trace.traceShow, які можуть бути вбудовані в програму і виводять на екран значення своїх аргументів перед тим, як передати їх у наступну функцію. Цей метод може бути дуже корисним при відслідковуванні значень виразів. Втім, при використанні таких засобів для відлагодження кількох функцій вивід програми може стати дуже перевантаженим і складним для сприйняття. Додатковим ускладненням слугує той факт, що лінива стратегія обчислень погано узгоджується з виводом, який вимагає атомарності та чіткої послідовності виконання команд, і уникнення ситуацій, де вивід команди trace буде посимвольно змішаний з рештою виводу програми, буде вимагати додаткових зусиль та витрат часу від розробника.

Також варто відзначити, що загальний підхід пошуку помилок в програмі через консольне виведення значень виразів в процесі її виконання є поганою практикою в розробці за згодою багатьох представників спільноти, тому якісне відлагодження потребує інших підходів.

2.2.2 Бібліотеки QuickCheck та Haskell hedgehog

Основним напрямком використання цих бібліотек є безпосереднє тестування коду. QuickCheck[9] є розповсюдженою та загальновживаною бібліотекою, найпоширенішою функцією в якій є однойменна quickCheck, що приймає лямбда-вираз, який має повертати значення булевого типу, і перевіряє коректність такого предиката на великій кількості випадково згенерованих аргументів, тип яких визначається типом параметрів переданого лямбда-виразу.

Бібліотека hedgehog[10] є зручним доповненням для такого інструменту, що дозволяє зручно задавати власні діапазони для тестування та генерації випадкових значень.

Окрім безпосереднього застосування при тестуванні, функції з описаних бібліотек можуть бути застосовані в якості аналогів виразів-припущень (assertion) з імперативних мов, якщо обчислення якогось тестового виразу буде здійснено перед його передачею в якості аргумента в наступну функцію.

2.2.3 Використання бібліотек в якості інструментів відлагодження

Окремою негативною рисою підходів, описаних в цьому пункті є те, що вони вимагають внесення змін у код та застосування в ньому власних функцій для того, щоб проводити його аналіз та відлагодження. І якщо у випадку з написанням тестів цей процес є виправданим, за умови що тести використовуються як гарант коректності написаного коду, то використання описаних засобів як постійних інструментів відлагодження, особливо у великих проектах, буде мати суттєвий негативний вплив, як на якість написаного коду, так і на швидкість, з якою розробники зможуть його реалізувати та підготувати до повноцінної роботи. У зв’язку з цим існує необхідність незалежного програмного рішення, яке буде мати здатність працювати з кодом і досліджувати його без внесення змін у текст програми.

2.3 Альтернативні версії програм для покрокових обчислень

Також при дослідженні теми, мною було знайдено кілька програмних рішень, що виконують подібну функцію, однак мають обмежений функціонал, або ж не є зручними у використанні. Нижче я наводжу приклади найкращих альтернатив.

2.3.1 GHC-VIS

Описаний в роботі [11] застосунок GHC-VIS є дуже якісно реалізованим аналізатором програм мовою Haskell. Втім, його основною ідеєю була демонстрація принципів спільного володіння виділеною пам’яттю та лінивих обчислень. Концентрація на цих двох темах призвела до того, що графічне представлення програм, які аналізує GHC-VIS дуже швидко стає надзвичайно громіздким і складним для сприйняття, що можна побачити на ілюстраціях.

Також, робота, що описує цей інструмент, була опублікована в 2013 році, веб-сторінка, що описує правила взаємодії з інструментом, містить посилання на застарілі та видалені ресурси, а команди, які вказані на ній, не дають можливості встановити продукт через проблеми з залежностями. Мною була знайдена сторінка проєкту на сервісі GitHub, однак інструкції зі встановлення і компіляції там також наштовхнулися на проблеми із залежностями. Ці факти дозволяють зробити висновок, що проект[12] не підтримується до сучасних версій мови і не є істотним варіантом для людей, які хочуть займатися відлагодженням програм.

2.3.2 Haskell Expression Evaluator

Даний інструмент, описаний в роботі [13], за своєю ідеєю дуже подібний до моєї роботи, що має доступний візуальний веб-інтерфейс.

Проте, необхідно відзначити кілька мінусів цього засобу. Приклад його застосування можна побачити на рисунку 2.1.

По-перше, засіб є орієнтованим на навчальний процес і повну демонстрацію всіх кроків обчислення програми. Дуже багато кроків

демонструються в надмірних деталях, і засіб не пропонує жодного сценарію, який би дозволив пропустити розписування зайвих кроків.

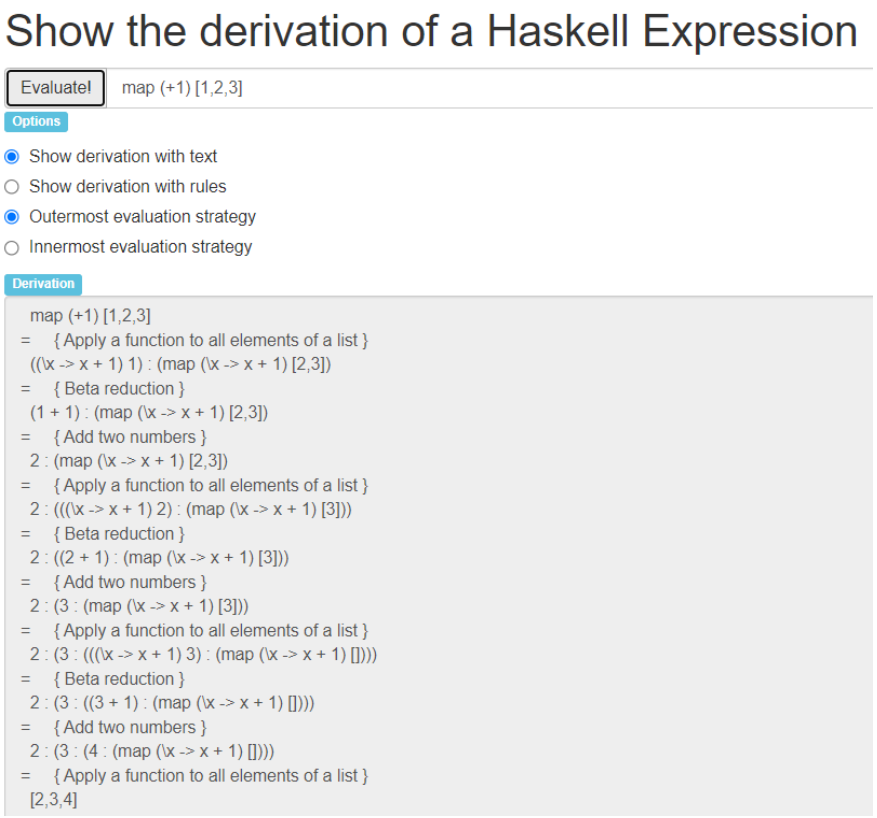

Рис. 2.1 Приклад використання
Haskell Expression Evaluator

По-друге, ця програма дозволяє працювати тільки з дуже обмеженим набором функцій, визначених у стандартній бібліотеці, і не підтримує можливості завантажити власний код і відтворити його покрокове обчислення.

По-третє, деякі приклади (навіть запропоновані авторами роботи) можуть давати некоректний результат в деяких режимах роботи програми, приклад можна побачити на рисунку 2.2.

2.4 Формулювання вимог, базуючись на недоліках знайдених зразків

Провівши аналіз інструментів, які можуть вирішувати подібні задачі, були сформульовані наступні вимоги до програми, що буде розроблятися:

Зручне консольне представлення, а також бути спроектованою з можливістю інтегрувати графічну складову;

- Доступ до коду стандартних бібліотеки мови Haskell, і, відповідно, мати можливість аналізувати і покроково обчислювати вбудовані функції, з заглибленням на довільну кількість ітерацій;
- Можливість обробляти надані користувачем файли з кодом, та покроково виконувати функції з них;
- Гнучка систему керування, і мати можливість як зануритися у визначення кожної конкретної функції, так і пропустити її виконання;
- Застосунок має постійно візуалізувати, на якому етапі він перебуває через вираз, що обчислюється, без прив'язки до номерів рядків чи стовпчиків;
- Програма має мати можливість проводити аналіз та покрокове обчислення програм, отримуючи файли з кодом чи текст в якості аргументу, без потреби якось модифікувати їх для роботи.

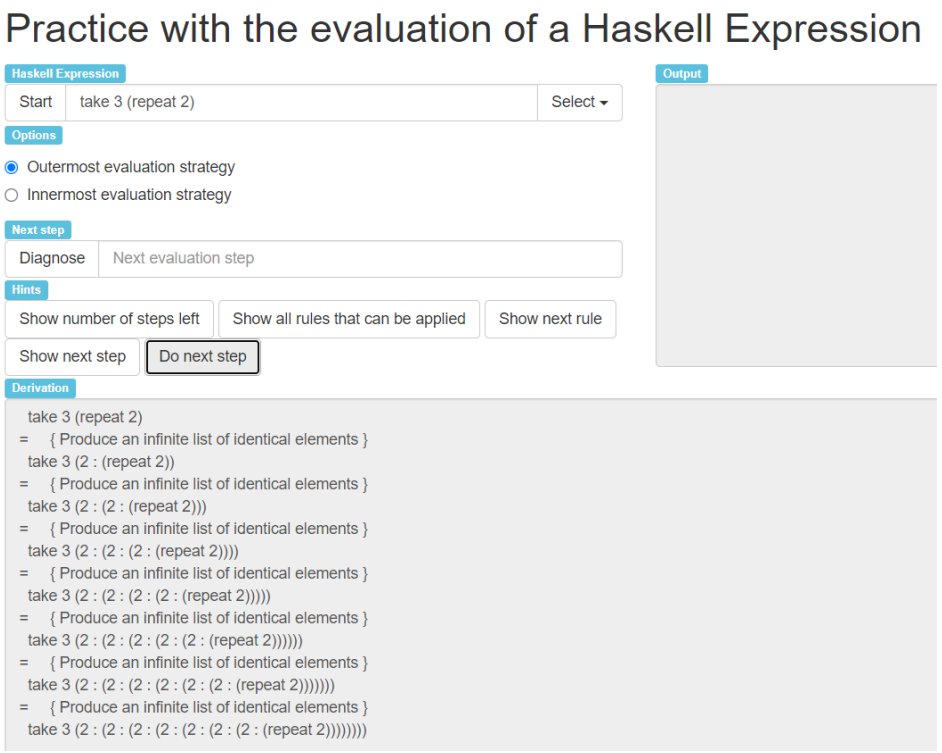

Рис. 2.2 Приклад некоректної поведінки
Haskell Expression Evaluator

У наступному розділі я розгляну структуру будови цієї програми, та підходи які я застосував для її написання.

РОЗДІЛ 3. РЕАЛІЗАЦІЯ HASKELL-STEP-BY-STEP-EVALUATOR

3.1 Реалізація “ядра” обчислень

Однією з потенційних вимог до продукту було створення зручного графічного інтерфейсу. Втім, варто відзначити, що це завдання досить широке, і таке що може мати багато різних інтерпретацій – як створення веб-сторінки, що відправляє відповідні запити на серверну частину додатку, і отримує у відповідь необхідну для візуалізації інформацію, так і додаток для персонального комп’ютера з аналогічним функціоналом. Але основним моїм завданням була реалізація саме програмної логіки, тобто тої частини програми, яка зможе провести аналіз наданих файлів з програмами мовою Haskell, згенерувати потрібні структури даних, і здійснити ітерацію по цим структурам, повертаючи в якості результату інформацію про поточні кроки обчислення програмного виразу. Отримавши описане “ядро”, що приймає в якості аргументів лише програмний код та набір інструкцій по пересуванню ним, і на кожному кроці повертає у зручному вигляді поточний стан виразу та інформацію про можливі подальші кроки, при подальшому розвитку проєкту довкола такого ядра можна буде побудувати програму з довільним інтерфейсом користувача. Саме тому написання вищеописаного “ядра” є ключовим для вирішення поставленої задачі.

3.2. Представлення функцій у вигляді HAST

Для представлення функцій в рамках програми використано абстрактні синтаксичні дерева (AST), зокрема при виконанні роботи була розроблена їх модифікація, що отримала робочу назву HAST (Haskell Abstract Syntax Tree). Зазначу, що HAST є незначною модифікацією, тому впродовж роботи терміни AST та HAST є взаємозамінними. Основні відмінності цієї структури від класичних AST:

- Динамічна обудова HAST
- Представлення вершин, що відповідають виклику певної функції:
- Представлення списків, кортежів та користувацьких структур даних.

При побудові HAST вершина-функція не будує подальшого піддерева, що відображає її тіло, аж поки обчислення цієї вершини не буде безпосередньо викликано користувачем. В такий спосіб втілюється принцип відкладених обчислень, дотримання якого дуже важливе для коректного обчислення програм мовою Haskell.

Зокрема, до безпосереднього виклику не обчислюється тіло вершин-функцій. Також, кожна вершина-функція зберігає список дерев-шаблонів, кожному з яких відповідає своє тіло функції. Це дозволяє реалізувати ще один принцип мови – порівняння зі зразком, що полягає в тому, що функція може мати кілька різних реалізацій, залежно від структури і значень своїх аргументів.

Списки реалізовані у вигляді піддерева, де корінь містить посилання на перший елемент списку та піддерево, що рекурсивно відображає решту списку. Це представлення віддзеркалює те, як списки реалізовані в мові Haskell [14], і дозволяє працювати з відкладеними обчисленнями, функціями, що генерують потенційно нескінченні списки, та ін. Аналогічний підхід, з міркувань симетрії та поліморфізму, застосовано для представлення кортежів та користувацьких структур даних. Впродовж цього розділу будуть наведені приклади та рисунки, що наочно проілюструють описані тут структури даних.

3.3 Загальна головна схема роботи програми

Загальна схема роботи HSSE та архітектура застосунку представлені на рисунках 3.1 та 3.2.

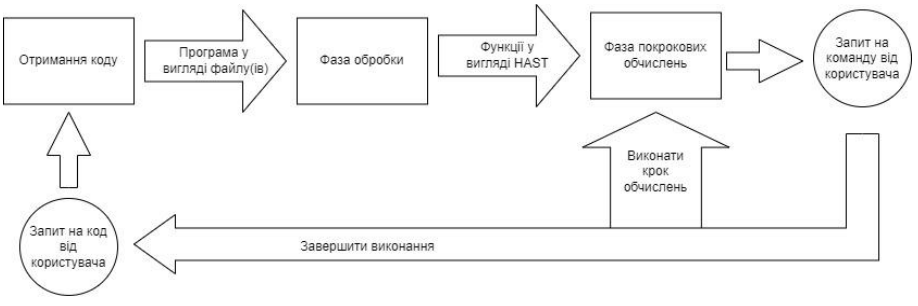

Рис. 3.1 Загальна схема роботи
Haskell Step-by-Step Evaluator

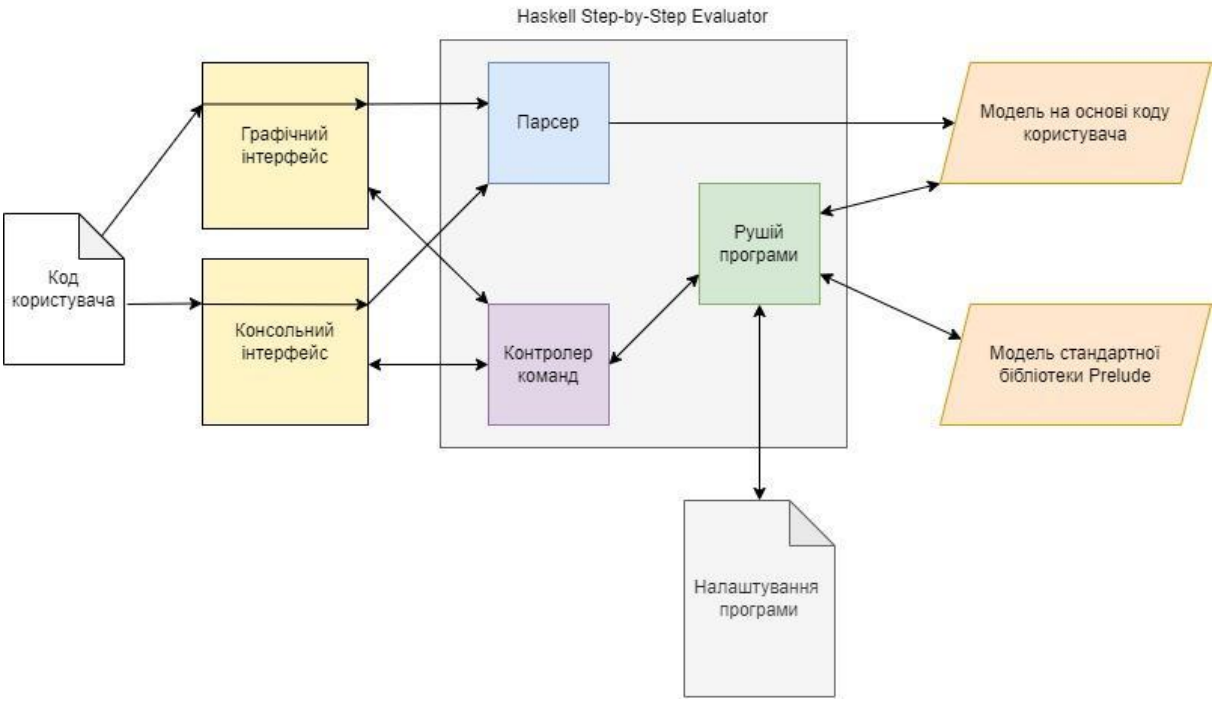

Рис. 3.2 Архітектура застосунку

Розглянемо дві основні фази роботи програми – підготовку та фазу покрокових обчислень.

3.3.1 Фаза підготовки

Попередня обробка файлу є першим кроком у підготовчій фазі. Основна причина, з якої є потреба в попередній обробці є те, що мова Haskell дозволяє використовувати у своїх програмах два підходи до визначення вкладеності і області видимості функцій та виразів: використовуючи символи фігурних дужок та крапок з комою або, альтернативно, використовуючи відступи. При цьому в одному програмному файлі і навіть в межах однієї функції дозволяється використовувати обидва підходи. Цей факт ускладнює обробку функцій в чистому вигляді, оскільки вимагає сильно більш складних правил граматики мови, на якій буде працювати лексичний аналізатор. Втім, таку особливість мови можна використати на користь, що і було реалізовано у фазі попередньої обробки. Всі відношення вкладеності функцій виражені за допомогою синтаксичних відступів можна замінити на аналогічні вирази, що використовуватимуть дужки і крапки з комою[15]. Після такої

заміни кожен вираз, що описує поведінку функції на певних аргументах можна перенести в один рядок, і лексичний аналіз такого виразу буде відчутно спрощеним.

Приклад вигляду функції в результаті попередньої обробки можна побачити на схемі, зображеній нижче:

```
functionWithHardWhere :: Num a => a -> a
functionWithHardWhere x = let y = z + 1 where z = q - u where q = w + t where w = e - 1
                                                                                e = x + 1
                                                                      t = 1
                                                            uu = 4
                                    u = 1
                            in y + o where{o = 1}

simplified :: Num a => a -> a
simplified x = let{y=z+1 where{z=q-u where{q=w+t where{w=e-1;e=x+1};t=1};u=4};u=1} in y + o
where{o = 1}
```

Можна проспостерігати, що вирази, які знаходилися на однаковому відступі в початковій функції (*functionWithHardWhere*), після обробки (функція *simplified*) знаходяться на однаковій “глибині” відносно фігурних дужок і розділені крапками з комами. Попри те, що запис функції *simplified* на схемі є менш читабельним з точки зору розробника, він набагато простіше обробляється лексичним аналізатором, до того ж дозволяючи йому працювати з одним рядком за один раз, не потребуючи постійної підтримки інформації про попередні чи наступні рядки.

Лексичний аналіз є наступним кроком у обробці файлу програми. Після нього кожен рядок програми представляється у вигляді набору так званих “токенів”, тобто спеціальних структур, які зберігають інформацію про початковий вигляд певної частини виразу, і про його тип. Для виконання цієї задачі було використано бібліотеку Boost.Spirit з сімейства бібліотек Boost.

Spirit – це бібліотека мови C++, що дозволяє генерувати парсери за правилами, що подібні до розширеної форми Бекуса-Наура[16]. Бібліотека Spirit використовує схожу нотацію, адаптуючи її під перегружені стандартні оператори мови C++, що дозволяє вбудовувати правила безпосередньо в код програми і одразу застосовувати, без потреби в

додатковому кроці трансляції з будь-якого формату в код. Ця зручність та інтегрованість з мовою розробки стали основною причиною вибору саме цієї бібліотеки як допоміжного засобу лексичного аналізу.

Засоби Boost.Spirit, окрім безпосередньо лексичного аналізу, надають широкі можливості для побудови абстрактного синтаксичного дерева, тому початковим наміром було використання бібліотеки як інструменту повноцінного перетворення програми у набір дерев, що репрезентують відповідні функції. Таким чином за допомогою однієї технології можна було б вирішити проблеми лексичного та синтаксичного аналізу. Втілити цей задум не вдалося через те, що бібліотека Spirit на своєму нинішньому етапі розвитку, версії Boost v.1.79, не надає інструментів для динамічного створення і комбінування правил. Відповідно, усі правила граматики мають бути описані статично у коді програми. Такий підхід не дає реалізувати задуманий інструмент в повній мірі, оскільки мова Haskell дозволяє, зокрема, задання власних операторів із вказанням їх рівня пріоритетності. Розглянемо приклад:

```
infixl 5 *+*
(*+*) :: Num a => a -> a -> a
(*+*) x y = x * x + y * y

infixl 7 ***
(***) :: Num a => a -> a -> a
(***) x y = x * x + y * y

addSumSquares ((x1,y1),(x2,y2)) = x1 * y1 + x2 * y2
addSumSquares' ((x1,y1),(x2,y2)) = x1 *+* (y1 + x2) *+* y2
addSumSquares'' ((x1,y1),(x2,y2)) =(x1 *** y1) + (x2 *** y2)
```

В прикладі оголошено два оператори, що визначені користувачем: (*+*) та (***). Різниця між цими операторами полягає виключно в їх пріоритетності. Таким чином, якщо розглянути функцію *addSumSquares*, замінивши в ній оператор множення на один з вищевказаних операторів, не додаючи ніяких дужок, то отримаємо функції, що еквівалентні функціям позначеним штрихом та двома штрихами відповідно. При побудові AST це

необхідно враховувати, аби задати правильну послідовність виконання команд у дереві. Це можна було б реалізувати шляхом динамічної генерації правил граматики на етапі попередньої обробки файлу – у випадку, якщо в файлі задаються нові оператори, додати їх до правила відповідної пріоритетності, враховуючи, що пріоритетність обмежена десятьма рівнями. Оскільки такий варіант не може бути реалізований через Boost.Spirit, був обраний інший підхід: Boost Spirit генерує максимально спрощений вигляд AST – набір токенів, що можуть містити вкладені токени, у випадку, якщо під час обробки один з виразів був вкладений у дужки, при тому що таке рекурсивне вкладення може мати довільну глибину. Отримане дерево вкладених токенів передається на опрацювання синтаксичному аналізатору, який, в свою чергу, повертає представлення кожної функції у вигляді множини AST.

Наступним кроком є **синтаксичний аналіз і побудова HAST.** Отримане дерево токенів буде перетворене у HAST у кілька ітерацій. Спершу відсіюються коментарі, інформація про назву модуля та підключені модулі. Кожен підключений модуль кладеться на стек і запускається його обробка за аналогічною схемою, для того щоб отримати з інших модулів інформацію про запозичені звідти функції. На другому кроці отримується інформація про користувацькі типи даних, оператори та класи типів, що оголошені в модулі, їхні назви додаються до переліку ключових слів, які можуть бути використані в модулі, який перебуває під опрацюванням. Після цього аналогічно скануються назви і сигнатури функцій модуля. В наступному кроці кожне визначення функції, зокрема функцій, визначених в класах типів, перетворюється в повноцінне AST, і кожен його токен транслюється в один з можливих видів вершин AST:

- Літерал (рядок, символ, дробове або ціле число)
- Назва функції, що може примінятись до аргументів
- Назва конструктора типу даних, що може примінятись до аргументів

- Назва оператора, що може примінятись до аргументів (як у інфіксній, так і в префіксній формі)
- Ідентифікатор (у тілі функції позначає один з аргументів чи результат його деконструкції, який передається у тіло як результат співставлення зі зразком, про що розповідається в наступному підрозділі.

3.3.2 Фаза покрокового обчислення виразів

Після завершення опрацювання усіх файлів, програма отримує набір функцій, кожна з яких представлена відповідним HAST. Після цього програма переходить у режим, в якому може приймати вирази в якості вхідних даних. Кожен такий вираз також перетворюється у HAST, однак це представлення не надається кінцевому користувачу. Він отримує на екран лише поточний вираз, підказку про те, яка частина цього виразу буде обрахована наступною, і інформацію про свої можливі подальші кроки.

- Порахувати наступний крок (Step-forward) – наступна за пріоритетністю дія буде виконана, її результат буде підставлений в поточний вираз. Приклад:

```
2 ^ 3 + 3 ^ 4
8 + 3 ^ 4
```

- Обрахувати наступний крок поступово (Step-in) – наступна дія буде обчислена окремо від контексту всього виразу, при цьому можна дослідити тіло функції чи оператора, прослідкувати який результат дадуть різні співставлення зі зразками тощо;
- Обрахувати весь поточний вираз (Step-out) – весь вираз в поточному контексті буде обрахований повністю (до слабкої нормальної форми, або ж до нормальної форми, якщо користувач обрав відповідний режим роботи)[4]. При застосуванні цієї команди, якщо попередньо була застосована команда Step-in, виконання цієї команди поверне виконання у той контекст, з якого користувач вийшов раніше, інакше вираз обчислюється до кінця і повертає результат;

- Повернутися до попереднього кроку (Step-back) – скасовує останній виконаний крок;

Після того, як буде виконано останній крок, програма сповіщає про завершення виконання.

3.4 Реалізація особливостей мови програмування через HAST

Представлення функції у вигляді, описаному у попередньому підрозділі має багато переваг, що стосуються безпосередньо представлення мови Haskell.

3.4.1 Порівняння зі зразком

Порівняння зі зразком є дуже важливим і поширеним інструментом в мові Haskell, що дозволяє легко і лаконічно замінити безліч перевірок з класичних мов програмування. Розглянемо це на прикладі, що зображений нижче:

```
templateExample :: (Int, Int, [Char]) -> Char
templateExample (0, _, _ ) = '0'
templateExample (a, b, []) = if a < b then '1' else '2'
templateExample (_, _, x:xs) = x
```

Функція *templateExample* отримує на вхід кортеж із трьох елементів – двох цілих чисел, та списку Unicode-символів, і повертає символ в якості результату. З отриманим кортежем послідовно виконуються дві перевірки. У випадку, коли перше число в кортежі дорівнює нулю, функція одразу повертає значення хиби. В іншому випадку, відбувається перевірка, чи масив, який є третім елементом, є пустим. Якщо така перевірка проходить успішно, то повертається символ ‘a’ або ‘b’, залежно від того, яке з чисел в кортежі буде більше. За умови, якщо друга перевірка також не буде успішною – це буде означати, що список символів не порожній – повертається перший символ зі списку. Варто також відзначити, що оператор ‘:’ використовується як конструктор списку. Його лівий аргумент – це перший елемент списку, а правий – посилання на решту списку. Відповідно ‘x’ – це перший елемент списку, має тип Char, тобто символ, в той час як ‘xs’ має тип списку символів.

Оскільки тіла функції в алгоритмі перетворюються в AST, ми можемо легко застосувати аналогічний підхід до зразків у оголошенні функції, отримавши AST-зразки. Таким чином, вибір підходящого тіла функції здійснюється за рахунок порівняння дерева-зразка та дерева аргумента. І на цьому етапі результат порівняння можна визначити за структурою дерева, часто така перевірка не буде навіть вимагати порівняння значень всередині дерев. Розглянемо на прикладі наведеної вище функції *templateExample,* яка отримує в якості аргументу кортеж (3, 4, [‘x’,’y’,’z’]). Представлення такого кортежу можемо побачити на рисунку 3.3.

Варто відмітити специфічну структуру побудови дерева. Такі структури, як кортежі, списки і іменовані користувацькі структури даних представлені у вигляді вершини, що має в першій дочірній вершині значення, яке збережено в структурі (перший елемент списку, кортежу, чи перше значення в користувацькій структурі даних). Друга дочірня вершина – або відсутня, що свідчить про закінчення кортежу чи структури даних, або містить посилання на продовження структури, що має такий же формат – решту списку, кортежу і т.і. Такий підхід запозичений із самої мови Haskell, в якій реалізація будь-яких списків здійснюється через структуру зв’язного списку. Переваги такого підходу стануть зрозумілими після дослідження кількох прикладів в цьому та наступному розділах, зокрема присвячених роботі зі списками, шаблонами аргументів функцій та зв’язування імені аргументу зі значеннями. Отримане дерево буде порівняно із першим деревом-зразком, який можна побачити на рисунку 3.4.

AST-зразок, отримавши AST в якості аргументу має перевірити чи таке AST підходить під описаний зразок. Вузли “ANY”, які відображають символ “_” (такий символ використовується на позначення зразка, що приймає будь-який вираз), будуть повертати значення істина, якщо отримують довільне AST-дерево на вхід. Вузли, що мають вузли-нащадки,

окрім власної перевірки мають також дочекатися позитивної перевірки від усіх своїх нащадків, щоб повернути позитивний результат.

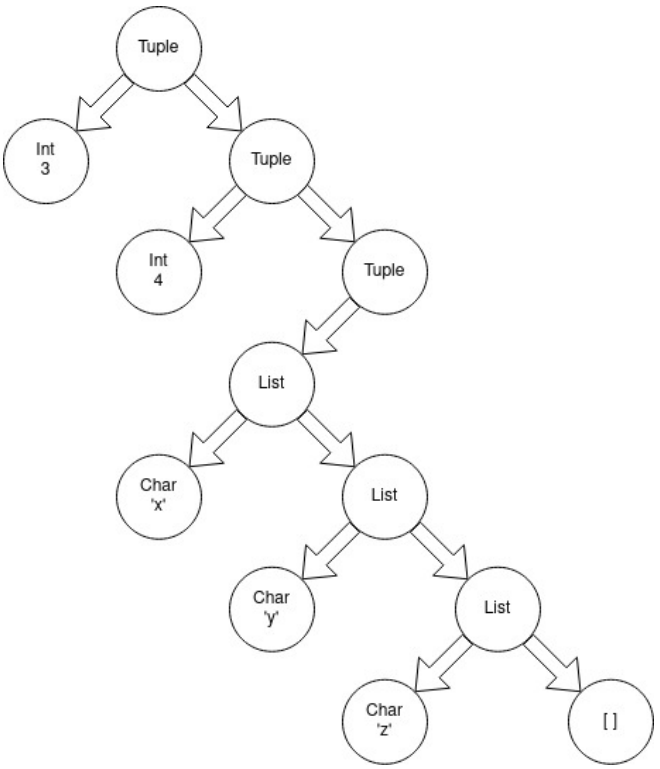

Рис. 3.3 Представлення кортежу (3, 4, [‘x’,’y’,’z’])
у вигляді абстрактного синтаксичного дерева

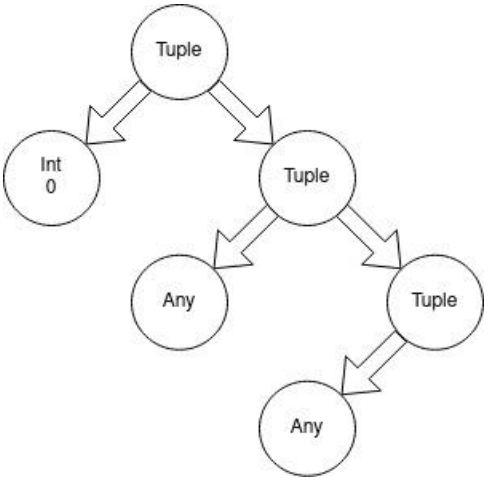

Рис 3.4 Зображення першого шаблону аргументів
функції templateExample у вигляді AST

Для даного дерева результат порівняння буде негативним, оскільки вузол, що містить літерал (в даному випадку числове значення нуля) в

якості перевірки на істинність повертає результат порівняння свого значення зі значенням у вузлі, що перевіряється, або, в даному випадку, `0 == 3`, що є хибним твердженням, і, відповідно, робить негативним результат перевірки всього дерева.

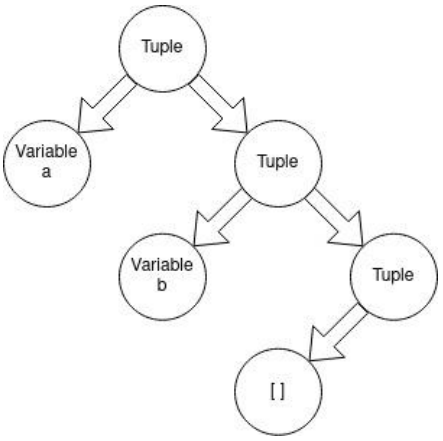

Рис 3.5 AST, що представляє другий шаблон аргументів функції templateExample

Наступною відбудеться перевірка з деревом на рисунку 3.5. Ця перевірка не пройде успішно, оскільки структура дерева-аргумента не підходить під структуру дерева-шаблону.

Остання перевірка, зображена на рисунку 3.6, працює коректно. Вузли “Variable” аналогічно вузлам “Any” дають позитивний результат перевірки для довільного піддерева, однак, окрім цього, прив’язують піддерево-аргумент до тимчасового імені, яке може бути використане в тілі функції.

3.4.2 Списки (потоки) невизначеної довжини

Мова Haskell не містить реалізації масивів. Натомість, усі подібні структури реалізовані через зв’язні списки, тобто структури, що або є порожнім списком, або містять елемент і посилання на зв’язний список.

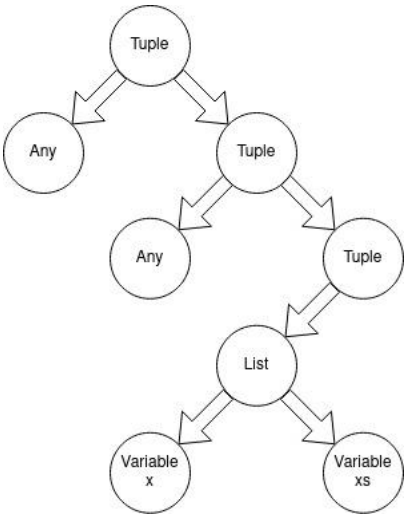

Рис. 3.6 AST, що представляє третій шаблон аргументів функції templateExample

Такий підхід дозволяє мові в рамках використання стандартних списків реалізовувати роботу з потоками даних, тобто функція, що повертає масив, може повертати потенційно нескінченний масив, і решта функцій, що працюють з її результатом можуть опрацьовувати його елементи, не враховуючи його довжини. Також така особливість зручна при роботі з функціями, що породжують нескінченний список, до якого можна застосувати довільну умову чи фільтр, що зробить його скінченним. Наступний приклад має продемонструвати, як це відбувається, а також показати, чому представлення виразів у вигляді AST робить обчислення таких виразів зручним. Розглянемо наступну функцію *fibs*, що не приймає аргументів, тобто є сталим значенням:

```
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

ЇЇ представлення можна побачити на рисунку 3.7.

Звернемо увагу на вершину з назвою zipWith на рисунку 3.7. Це – функційна вершина, і вона не має нащадків, натомість зберігає масив дерев, які вона приймає в якості аргументів, а також масив шаблонів для цих аргументів, і відповідне кожному шаблону тіло функції, що використовує змінні, отримані з цих шаблонів. На рисунку 3.7 представлений спрощений вигляд такої вершини, і дерев, що є аргументами цієї функції.

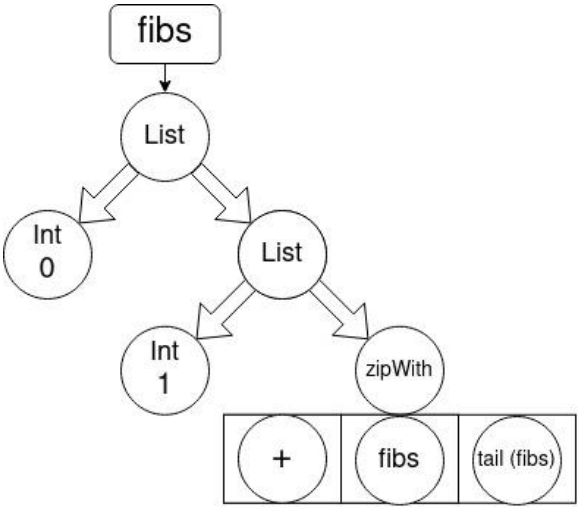

Рис 3.7 Представлення fibs у вигляді AST

Реалізація функції fibs не є надто важливою для нашого прикладу, приймемо, що вона повертає послідовність Фібоначчі: 0,1,1,2,3,5…, де $F_n = F_{n-1} + F_{n-2}$.

Застосуємо до неї функцію *take*:

```
take 0 _ = []
take n (x:xs) = x : take (n-1) xs
```

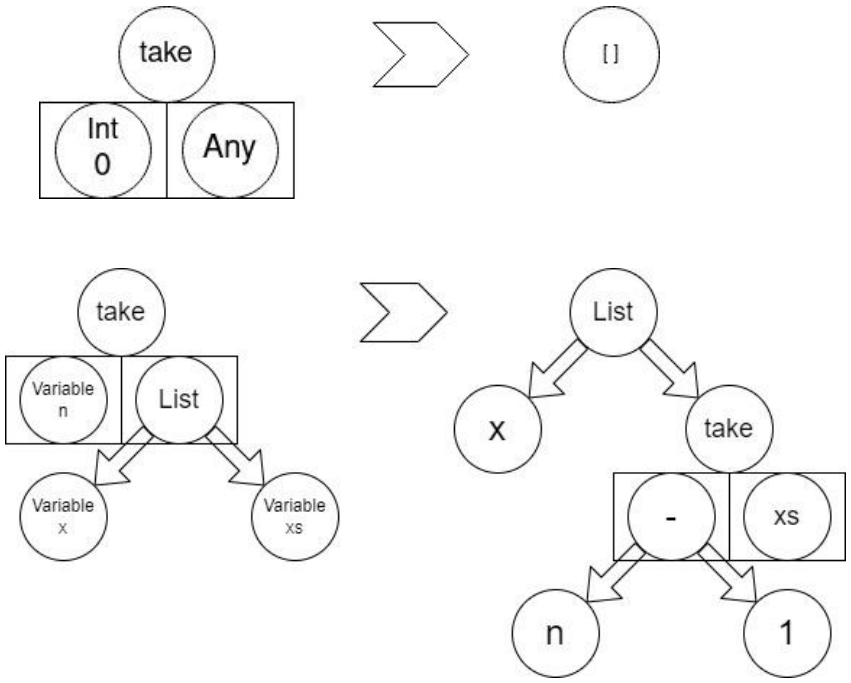

Рис. 3.8 Два можливих представлення шаблонів аргументів функції take, та відповідні їм AST тіла функції

Попри те, що список *fibs* генерується необмежено, наявність у функції take умови виходу з рекурсії результат обчислення виразу

```
take 3 fibs
```

буде скінченним, і буде обчислений за кроками, які можна побачити на рисунках).

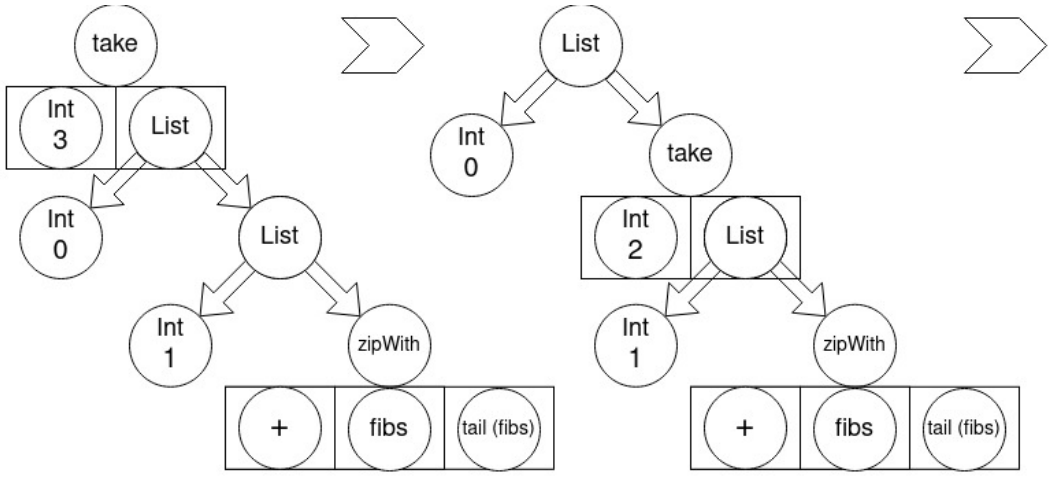

Рис. 3.9 Перший крок виконання команди take 3 fibs, що отримує вираз у слабкій нормальній формі

Після виконання першого ж кроку процедура виконання завершується, адже отриманий вираз знаходиться у слабкій нормальній формі, тобто будь-які виклики функцій перебувають всередині конструкторів типів даних або анонімних функцій. На цьому етапі у користувача буде можливість продовжити виконання, поки вираз не буде доведений до вигляду, в якому його можна вивести на екран, тобто повного застосування всіх функцій до їх аргументів. В такому випадку будуть виконані подальші кроки, зображені на рисунках 3.10 та 3.11. Текстовий варіант такого обрахунку буде виглядати наступним чином:

```
take 3 (fibs)
0 : take 2 (1 : zipWith (+) fibs (tail fibs))
0 : 1 : take 1 (zipWith (+) fibs (tail fibs))
0 : 1 : take 1 ((0 + 1) : zipWith (+) (1:...) ((0+1):...))
0 : 1 : (0 + 1) : take 0 (zipWith (+) (1:...) ((0+1):...))
0 : 1 : 1 : take 0 (zipWith (+) (1:...) ((0+1):...))
0 : 1 : 1 : []
```

```
[0,1,1]
```

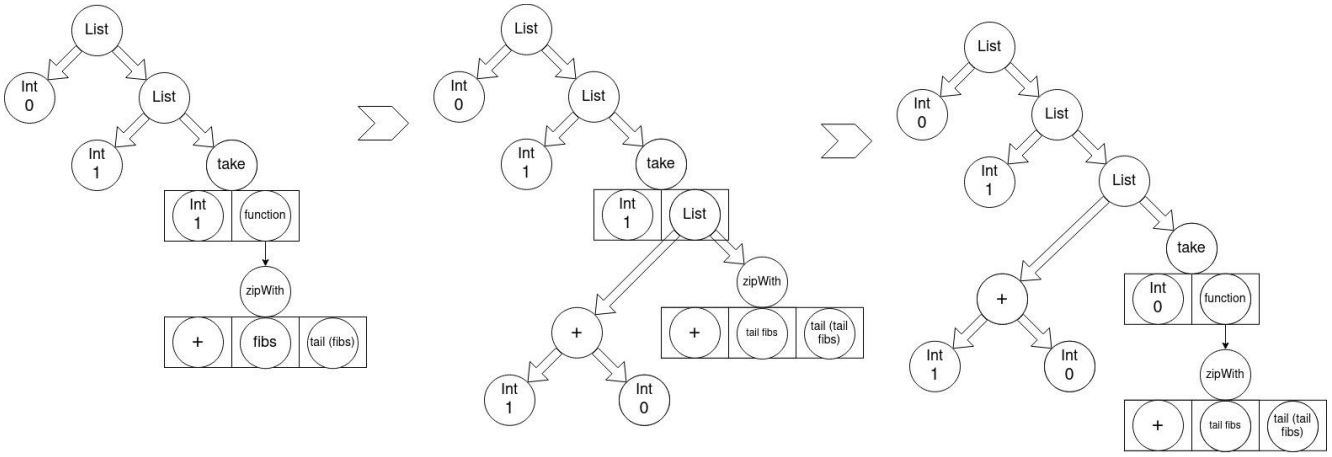

Рис 3.10 Проміжні кроки виконання команди take 3 fibs

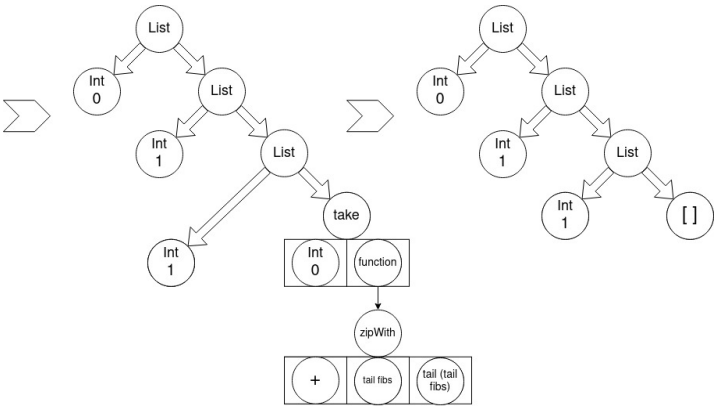

Рис 3.11 Фінальні кроки обчислення значення виразу take 3 fibs.

3.4.3 Відкладені обчислення

У попередньому прикладі також можна побачити те, як використання AST, і, зокрема, вершин-функцій, дозволяє реалізувати ліниву стратегію обчислень. На кожному наступному кроці виконується тільки функція, що знаходиться найвище у AST, а функції, що є проміжними вершинами, будуть обчислені тільки тоді, коли значення їх результату знадобиться для порівняння зі зразком.

В наступному розділі буде продемонстровано, як використання Haskell Step-by-Step Evaluator може бути використаним в якості засобу вивчення мови Haskell, а також як засіб її відлагодження.

РОЗДІЛ 4. ПРИКЛАДИ ЗАСТОСУВАННЯ ДОДАТКУ

4.1 Використання додатку для дослідження функції map

Функції вищого порядку – це функції, котрі в якості аргументу та результату можуть повертати інші функції. Ця концепція є надзвичайно важливою для ФП, оскільки дозволяє будувати і описувати функції шляхом комбінування створених раніше функцій[17]. Функція *map* – надзвичайно важливий елемент у такій побудові. Будучи одним з найпростіших представників функцій вищого порядку, *map* часто виступає фундаментом при побудові більш складних функцій для роботи з масивами. Також, з цих причин, ця функція часто виступає прикладом при демонстрації роботи функцій вищих порядків. Її робота описується наступним чином – вона приймає в якості першого аргумента деяку унарну функцію *f* – тобто функцію від одного аргумента, а в якості другого – масив елементів, що можуть бути аргументами для функції *f*. Результатом виконання функції map буде новий список, кожен елемент якого утворений в результаті застосування функції *f* до відповідного елемента масиву-аргумента. Таким чином, функція *map* – це спосіб застосувати деяку функцію одночасно до всіх елементів списку.

Потенціал цієї функції також збільшує той факт, що всі функції в мові Haskell за замовчуванням є каррованими[18,19]. Це означає, що кожна функція приймає один аргумент, застосовує його до свого тіла, і повертає функцію, що має арність на одиницю меншу. Це робить кожну функцію унарною функцією вищого порядку, таку властивість мова отримала від лямбда-числення[20], оскільки саме така нотація стала базисом для формування мови[21]. Таким чином, можна передати функції map в якості першого аргумента і функцію від трьох аргументів. В такому випадку результатом буде масив функцій, в кожної з яких буде зафіксовано значення першого аргументу, і, відповідно, всі функції в отриманому масиві будуть бінарними. Потреба застосувати функцію до всіх елементів списку постає в

усіх мовах програмування, тому подібна функція реалізована в більшості сучасних мов програмування – JavaScript[22], Java[23], C++[24] та ін.

Часто, демонстрацію принципів роботи цієї функції здійснюють за допомогою демонстрації багатьох прикладів її застосування. Такий підхід можна побачити на рис. 4.1, що зображає приклади, що надаються в книзі [22].

```
GHCi> map reverse ["dog","cat", "moose"]
["god","tac","esoom"]
GHCi> map head ["dog","cat", "moose"]
"dcm"
GHCi> map (take 4) ["pumpkin","pie","peanut butter"]
["pump","pie","pean"]
GHCi> map ("a "++) ["train","plane","boat"]
["a train","a plane","a boat"]
GHCi> map (^2) [1,2,3]
[1,4,9]
```

Рис. 4.1 Демонстрація властивостей функції map шляхом демонстрації численних прикладів

How do we write out what `map f` does? Note, this order of evaluation doesn't represent the proper non-strict evaluation order, but it does give an idea of what's going on:

```
map (+1) [1, 2, 3]
```

If we desugar the list syntax:

```
map (+1) (1 : (2 : (3 : [])))
```

`:` is defined as `infixr 5`, so the parentheses associate to the right. We aren't passing an empty list to `map`, so the second pattern match fires:

```
(+1) 1 :
  map (+1)
    (2 : (3 : []))
```

Apply `(+1)` to the next value, cons onto the result of mapping over the rest:

```
(+1) 1 :
  ((+1) 2 :
    (map (+1)
      (3 : [])))
```

This is the last time we'll trigger the second case of `map`:

```
(+1) 1 :
  ((+1) 2 :
    ((+1) 3 :
      (map (+1) [])))
```

Triggering the base case that handles the empty list and returns the empty list:

```
(+1) 1 :
  ((+1) 2 :
    ((+1) 3 : []))
```

Finishing the reduction of the expression:

```
2 : ((+1) 2 : ((+1) 3 : []))
2 : 3 : (+1) 3 : []
2 : 3 : 4 : [] == [2, 3, 4]
```

Using the syntactic sugar of list, here's an approximation of what `map` is doing for us:

```
map f [1, 2, 3] == [f 1, f 2, f 3]

map (+1) [1, 2, 3]
        [(+1) 1, (+1) 2, (+1) 3]
        [2, 3, 4]
```

Рис. 4.2 Демонстрація роботи функції map шляхом покрокового обчислення прикладу[25]

Альтернативний підхід до дослідження властивостей функції map розглянутий у книзі [25], продемонстрований на рис. 4.2. У цьому підході, окрім демонстрації прикладів застосування, автори розписують результат покрокового виконання функції, зокрема демонструючи процес деконструкції списку-аргументу та конструкції списку-результату. В такий спосіб набагато краще демонструються основні принципи роботи мови, а також дозволяє розвинути інтуїцію з приводу того, які кроки буде виконувати компілятор при обчисленні тих чи інших функцій.

Демонстрація навіть такого простого прикладу вимагала виконання всіх обчислень вручну, що вимагало постійного контролю над порядком виконання дій та створювало потенціал для помилки в обчисленнях. Саме з метою уникнення таких помилок було розроблено Haskell Step-by-Step Evaluator, і у прикладі нижче можна побачити, як буде виглядати обчислення аналогічного прикладу у розробленому додатку:

```
map (+1) [1,2,3]
((+1) 1 : map (+1) [2,3])
```

Після виконання першої операції, отриманий вираз знаходиться в слабкій нормальній формі. Обчислення зупиняється, якщо тільки користувач не вкаже про потребу обрахувати значення виразу до нормальної форми – в якій всі функції застосовані для своїх аргументів. В такому випадку будуть виконані наступні кроки:

```
(2: map (+1) [2,3])
(2: (+1) 2 : map (+1) [3])
(2: 3 : map (+1) [3])
(2: 3 : (+1) 3 : map (+1) [])
(2: 3 : 4 : map (+1) [])
(2: 3 : 4 : [])
[2,3,4]
```

Важливо відзначити – як і у випадку з розрахунками на рис. 4.2, порядок виконання дій може відрізнятися від порядку, який обере компілятор. Однак, це не має створити ніяких додаткових ускладнень при відлагодженні або навчальному процесі, з кількох причин. По-перше, порядок дій, яким керується HSSE, цілком відповідає всім правилам

лінивих обчислень. По-друге, в силу того що всі функції в мові є чистими, порядок обчислень не буде ніяким чином впливати на результат. Таким чином, навіть у випадку, якщо компілятор мови буде здійснювати іншу послідовність кроків для обрахування результату, шлях, обраний HSSE буде завжди приводити до аналогічної відповіді, за умови коректних аргументів, або ж призводити до такої ж помилки, за умови що аргументи не відповідають вимогам функції.

4.2 Дослідження принципів роботи функцій foldl та foldr

Функція map не завжди демонструється шляхом покрокового виконання виразів, оскільки є знайомою багатьом програмістам і вважається досить тривіальною. В той час, функції згортки, такі як foldr та foldl, мають дуже схожу логіку, але виконують дії в різному порядку. Для демонстрації цієї відмінності більшість досліджених навчальних посібників[26-30] вдаються до описаного методу покрокового виконання. Розглянемо, як виглядатиме застосування цих функцій до кількох простих прикладів із застосуванням HSSE:

```
foldr (-) 0 [1,2,3]
1 - foldr (-) 0 [2,3]
1 - (2 - foldr (-) 0 [3])
1 - (2 - (3 - foldr (-) 0 []))
1 - (2 - (3 - 0))
1 - (2 - 3)
1 - (-1)
2

foldl (-) 0 [1,2,3]
foldl (0-1) [2,3]
foldl ((0-1)-2) [3]
foldl (((0-1)-2)-3) []
(((0-1)-2)-3)
(((-1)-2)-3)
((-3)-3)
-6
```

Описані приклади наочно демонструють послідовність виконання команд, дозволяють прослідкувати за тим, як змінюються значення виразів і в якому порядку у виразах накопичуються аргументи. Це дозволяє легко продемонструвати відмінності між роботами лівої та правої згортки, навіть для людини, що не знайома з їх визначеннями.

4.3 Застосування додатку для відлагодження програм

Розглянемо застосування HSSE в якості інструменту відлагодження на простому прикладі. Нехай задана наступна функція *nationalGrade*, що перетворює оцінку студента, виставлену за 100-бальною шкалою, у оцінку за національною шкалою:

```
nationalGrade grade
   | grade >= 90 = "Відмінно"
   | grade >= 75 = "Добре"
   | otherwise   = "Незадовільно"
   | grade >= 60 = "Задовільно"
```

При внесенні змін у цю функцію (скажімо, додавання нового варіанту оцінки), у ній була допущена помилка. Не вдаючись в деталі синтаксису мови, зазначимо, що функція використовує вирази-вартові[31], кожен з яких є умовою, і кожній такій умові відповідає своє тіло функції: певний аналог операторів розгалуження (if) або перемикача (switch) з імперативних мов програмування. Помилка, допущена при побудові функції, полягає в тому, що порядок написання таких виразів грає роль – в якості тіла функції буде використаний оператор, що відповідає першому з виразів, що матиме значення істини. Ключове слово otherwise позначає вираз, що завжди має значення істини, таким чином, якщо умови, записані вище, не будуть виконуватися, функція поверне тіло, відмічене цим ключовим словом. Відповідно, всі умови, записані нижче умови `otherwise` будуть проігноровані, і така функція на жодних вхідних даних не буде повертати результат `"Задовільно"`. Розглянемо виконання такої функції на вхідному значенні `65`, щоб продемонструвати, як використання HSSE дозволяє виправити допущену помилку. Обчислення значення виразу:

```
nationalGrade 65
"Незадовільно"
```

Поверне результат `"Незадовільно"`. Для того, щоб дослідити некоректну поведінку функції, застосуємо одну з можливостей HSSE – розглянути покрокове обчислення функції без контексту (Step-in). При застосуванні цієї команди до функції *nationalGrade* можна прослідкувати порядок обчислення виразів-вартових, та побачити, який із них буде обрано і чому:

```
nationalGrade 65
   | grade >= 90 = "Відмінно"
   | grade >= 75 = "Добре"
   | otherwise   = "Незадовільно"
   | grade >= 60 = "Задовільно"
------
nationalGrade 65
   | grade >= 75 = "Добре"
   | otherwise   = "Незадовільно"
   | grade >= 60 = "Задовільно"
------
nationalGrade 65
   | otherwise   = "Незадовільно"
   | grade >= 60 = "Задовільно"
------
"Незадовільно"
```

В даному прикладі умова, яка обчислюється на кожному кроці, виділена жирним шрифтом, а кожен наступний крок відділений рисками. Можливі також інші підходи, що будуть зумовлені реалізаціями графічного чи консольного інтерфейсу, що буде використовувати HSSE.

Проаналізувавши кроки, продемонстровані HSSE, можна внести правки до функції *nationalGrade*:

```
nationalGrade grade
   | grade >= 90 = "Відмінно"
   | grade >= 75 = "Добре"
   | grade >= 60 = "Задовільно"
   | otherwise   = "Незадовільно"
```

Виправлена версія функції буде повертати вірний результат для довільної коректної оцінки за 100-бальною шкалою.

ВИСНОВКИ

В рамках виконання роботи було розроблено програмний продукт, який дозволяє аналізувати коди програм, написаних мовою Haskell, та реалізовувати їх покрокове виконання з метою відлагодження або ж дослідження основних принципів роботи. Для цього був проведений аналіз ключових конкурентних програмних засобів, що використовуються в аналогічних цілях, сформовані списки їхніх переваг та недоліків. На базі отриманих характеристик сформовано технічні вимоги до програмного забезпечення.

Розроблено методику представлення функцій мови Haskell у вигляді модифікації абстрактних синтаксичних дерев з урахуванням особливостей мови, продемонстровано переваги такого підходу – можливість представлення утвореною структурою даних ключових особливостей мови, таких як відкладені обчислення, порівняння з шаблоном тощо.

Спроєктована архітектура додатку, у форматі, що передбачає спрощення реалізації різнопланових графічних інтерфейсів для зручної взаємодії з програмним забезпеченням. Також запропонована архітектура надає можливість для додатку працювати одночасно і з користувацьким програмним кодом, і з функціями, визначеними у стандартних бібліотеках мови Haskell.

Згідно з утвореними вимогами розроблено програмне забезпечення. Продемонстровано потенціал застосування розробленого додатку у навчально-демонстраційних цілях, а також з метою відлагодження програм.

Розроблений програмний продукт може бути покращений шляхом створення різноманітних графічних інтерфейсів користувача, а також за рахунок імплементації підтримки нових можливостей мови Haskell.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Звіт бюро статистики ринку праці США [Електронний ресурс] – Режим доступу: https://www.bls.gov/oes/estimates_88_95.htm

2. Hudak P. Conception, evolution, and application of functional programming languages / P. Hudak // ACM Computing Surveys. - 1989 - Вип. 3 - Розд. 21 - с. 359 −411

3. Hughes J. Why Functional Programming Matters / J. Hughes // Research Topics in Functional Programming / D. A. Turner – Сполучені Штати Америки, Addison-Wesley Publishing Company, 2007. – с.18 - 19, 40-41.

4. Allen C. Haskell from first principles / C. Allen, J. Moronuki – Сполучені Штати Америки, Gumroad Inc., 2015 – с.29-35 – ISBN 9781945388002

5. Mena A. S. Practical Haskell / A. S. Mena – Сполучені Штати Америки, Springer Science + Business Media Finance Inc., 2019 – с.36-37 – ISBN 978-1-4842-4480-7

6. Lambert K. A. A Gentle Introduction to Functional Programming in Haskell / K. A. Lambert – Сполучені Штати Америки, Kenneth Lambert, 2016– с.36-37 – ISBN 978-0-9859567-1-4

7. Документація технології GHCD [Електронний ресурс] – Режим доступу: http://downloads.haskell.org/~ghc/7.4.1/docs/html/users_guide/ghci-debugger.html (переглянуто 05.06.2022).

8. Обговорення можливостей відлагодження програм, написаних мовою Haskell розробниками на онлайн-форумі reddit.com [Електронний ресурс] – Режим доступу: https://www.reddit.com/r/haskell/comments/qb9t3f/why_is_the_debugger_so_bad_in_haskell_or_is_it/ (переглянуто 05.06.2022)

9. Документація бібліотеки QuickCheck [Електронний ресурс] – Режим доступу: https://hackage.haskell.org/package/QuickCheck (переглянуто 05.06.2022)

10. Документація бібліотеки Haskell.HedgeHog [Електронний ресурс] – Режим доступу: https://hackage.haskell.org/package/hedgehog (переглянуто 05.06.2022)

11. Felsing D. Visualization of Lazy Evaluation and Sharing : Квал. роб. бак. фіз.-мат. наук : 27.09.12 / Felsing D. – Німеччина, 2012. – 363 с.

12. Інструкція та приклади використання інструменту GHC-VIS [Електронний ресурс] – Режим доступу: http://felsin9.de/nnis/ghc-vis/#introduction (переглянуто 05.06.2022)

13. Olmer T. Evaluating Haskell expressions in a tutoring environment / T. Olmer, B. Heeren, J. Jeuring [Електронний ресурс] - Режим доступу: https://www.bls.gov/oes/estimates_88_95.htm (переглянуто 05.06.2022)

14. Allen C. Haskell from first principles / C. Allen, J. Moronuki – Сполучені Штати Америки, Gumroad Inc., 2015 – с.300-301 – ISBN 9781945388002

15. Haskell Report 98 [Електронний ресурс] – Режим доступу: https://www.haskell.org/onlinereport/haskell2010/ (переглянуто 05.06.2022)

16. Документація бібліотеки Boost.Spirit [Електронний ресурс] – Режим доступу: https://www.boost.org/doc/libs/1_79_0/libs/spirit/doc/html/spirit/preface.html (переглянуто 05.06.2022)

17. Hughes J. Why Functional Programming Matters / J. Hughes // Research Topics in Functional Programming / D. A. Turner – Сполучені Штати Америки, Addison-Wesley Publishing Company, 2007. – с. 20-25

18. Hudak P. Conception, evolution, and application of functional programming languages / P. Hudak // ACM Computing Surveys. - 1989 - Вип. 3 - Розд. 21 - с.382

19. O'Sullivan B. Real world Haskell / B. O'Sullivan, J. Goerzen, D. Stewart – Сполучені Штати Америки, O'Reilly Media, 2009 – с.100-103 – ISBN 978-0-596-51498-3

20. Hudak P. Conception, evolution, and application of functional programming languages / P. Hudak // ACM Computing Surveys. - 1989 - Вип. 3 - Розд. 21 - с. 363-368

21. Hutton G. Programming in Haskell / G. Hutton – Великобританія, Cambridge University Press, 2016 – с.12, 21 – ISBN 978-1-316-62622-1

22. Will K. Get programming with Haskell / K. Will – Сполучені Штати Америки, Manning Publications Co., 2018 – с.84-85 – ISBN 9781617293764

23. Документація мови Java [Електронний ресурс] – Режим доступу: https://vertex-academy.com/tutorials/ru/java-8-stream-map/ (переглянуто 05.06.2022)

24. Документація мови C++ (стандарт 2020 року) [Електронний ресурс] – Режим доступу: https://vertex-academy.com/tutorials/ru/java-8-stream-map/ (переглянуто 05.06.2022)

https://en.cppreference.com/w/cpp/algorithm/for_each

25. Allen C. Haskell from first principles / C. Allen, J. Moronuki – Сполучені Штати Америки, Gumroad Inc., 2015 – с.329-330 – ISBN 9781945388002

26. O'Sullivan B. Real world Haskell / B. O'Sullivan, J. Goerzen, D. Stewart – Сполучені Штати Америки, O'Reilly Media, 2009 – с.93 – ISBN 978-0-596-51498-3

27. Allen C. Haskell from first principles / C. Allen, J. Moronuki – Сполучені Штати Америки, Gumroad Inc., 2015 – с.88 – ISBN 9781945388002

28. Онлайн-курс з вільним доступом для вивчення мови Haskell [Електронний ресурс] – Режим доступу: http://learnyouahaskell.com/higher-order-functions#folds (переглянуто 05.06.2022)

29. Will K. Get programming with Haskell / K. Will – Сполучені Штати Америки, Manning Publications Co., 2018 – с.88 – ISBN 9781617293764

30. Mena A. S. Practical Haskell / A. S. Mena – Сполучені Штати Америки, Springer Science + Business Media Finance Inc., 2019 – с.83 – ISBN 978-1-4842-4480-7

31. O'Sullivan B. Real world Haskell / B. O'Sullivan, J. Goerzen, D. Stewart – Сполучені Штати Америки, O'Reilly Media, 2009 – с.68-69 – ISBN 978-0-596-51498-3