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

Кваліфікаційна робота

Освітній ступінь – магістр

На тему «**РЕАЛІЗАЦІЯ ЛЯМБДА-ЧИСЛЕННЯ В HASKELL**»

Виконав: студент 2-го року навчання
освітньої програми «Комп'ютерні науки»,
спеціальності 122 Комп'ютерні науки
Соболєв Владислав Олександрович
Керівник: Проценко Володимир Семенович
доктор фіз-мат наук, доцент.
Кваліфікаційна робота захищена
з оцінкою ______________________
Секретар ЕК ____________________
«____» __________________ 2025 р.

Київ – 2025

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ
Зав. кафедри інформатики
кандидат фіз-мат наук, доцент
Гороховський Семен Самуїлович

____
(підпис)
“_____” ___________2025 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на кваліфікаційну роботу

студенту 2 р.н. магістерської програми Комп’ютерні науки
Соболєву Владиславу Олександровичу
розробити реалізацію лямбда-числення в Haskell.

Зміст текстової частини кваліфікаційної роботи:

Зміст
Анотація
Ключові слова
Вступ
1. Теоретичні основи
2. Haskell як мова реалізації
3. Проектування та реалізація
4. Функціональність та архітектурні рішення
5. Тестування та валідація
6. Порівняння з аналогічними роботами
7. Застосування в освіті
Висновки
Список використаної літератури
Додатки

Дата видачі “_____” __________ 2025 р.

Керівник
В.С.Проценко, доктор фіз-мат наук, доцент ____________________
(підпис)

Завдання отримав
В.О. Соболєв ____________________
(підпис)

Тема: Реалізація лямбда-числення в Haskell

Календарний план виконання роботи:

	Назва етапу магістерської роботи	Термін виконання етапу	Примітка
1	Отримання завдання на дипломну роботу.	23.11.2024	
2.	Огляд технічної літератури за темою роботи.	11.12.2024	
3.	Розробка плану та структури роботи.	2.02.2025	
4.	Аналіз сучасних методів реалізації лямбда-числення в Haskell	13.02.2025	
5.	Розробка застосунку.	15.04.2025	
6.	Тестування застосунку.	22.04.2025	
7.	Виправлення помилок і доопрацювання недоліків.	21.05.2025	
8.	Створення слайдів для доповіді та написання доповіді.	25.05.2025	
9.	Попередній захист магістерської роботи.	30.05.2025	
10.	Корегування роботи за результатами попереднього захисту.	2.06.2025	
11.	Завершення написання текстової частини.	6.06.2025	
12.	Написання документації застосунку.	7.06.2025	
13.	Остаточне оформлення пояснювальної роботи та слайдів.	9.06.2025	
14.	Захист магістерської роботи.	12.06.2025	

Студент: Соболєв Владислав Олександрович

Керівник: Проценко Володимир Семенович

“______”______________ 2025 р.

Зміст

Анотація 5

Ключові слова 5

Вступ 6

1. Теоретичні основи 8

1.1. Історія лямбда-числення 8

1.2. Синтаксис та семантика 8

1.3. Стратегії обчислення 10

1.4. Кодування Черча 11

1.5. Зв'язок із Haskell 12

2. Haskell як мова реалізації 13

2.1. Переваги принципів Haskell 13

2.2. Зіставлення зі зразком та алгебраїчні типи даних 13

2.3. Використання Parsec 14

2.4. Підтримка Haskell рекурсії та підстановки 15

3. Проектування та Реалізація 16

3.1. Розробка абстрактного синтаксичного дерева (AST) 16

3.2. Розбір лямбда-виразів 16

3.3. Гарний друк та відображення виразів 18

3.4. Роздільна здатність середовища та змінних 19

3.5. Підстановка та перезапис без захоплення 20

3.6. Механізм обчислення 21

4. Функціональність та архітектурні рішення 24

4.1. Підтримка зовнішніх середовищ 24

4.2. Інтеграція з графічним інтерфейсом 25

5. Тестування та Валідація 27

6. Порівняння з аналогічними роботами 29

6.1. Інші лямбда-інтерпретатори на Haskell 29

6.2. Відмінності з типізованим лямбда-численням та інтерпретаторами 29

6.3. Компроміси в розробці реалізації 30

Висновки 31

Список використаної літератури 33

Додатки 34

Анотація

Ця робота присвячена розробці та реалізації інтерактивного інтерпретатора нетипізованого лямбда-числення мовою Haskell. Проєкт сфокусований на створенні освітнього інструменту, що дозволяє користувачам зрозуміти концепції лямбда-обчислень та функціонального програмування.

Ключові аспекти реалізації включають розробку абстрактного синтаксичного дерева (AST), надійного парсера за допомогою Parsec, а також механізму обчислення, що використовує повний нормальний порядок редукції з коректною підстановкою та α-перетворенням. Інтерпретатор підтримує кодування Черча для арифметики та булевої логіки. Реалізація включає інтерактивний графічний інтерфейс користувача (GUI) на базі GTK+, який забезпечує інтуїтивне введення декількох виразів та покрокову візуалізацію β-редукції. Також є можливість збереження користувацьких визначень у базі даних SQLite для створення власних бібліотек функцій.

Ключові слова

Лямбда-числення, Haskell, Функціональне програмування, Інтерпретатор, Абстрактне синтаксичне дерево (AST), Парсинг, Бета-редукція, Нормальний порядок, GTK+, SQLite, Покрокова візуалізація.

Вступ

Лямбда-числення, запропоноване Алонзо Чорчем у 1930-х роках [1], є формальною системою, призначеною для вираження обчислень через абстракцію функцій та їх застосування. Його мінімалістична природа — наявність лише однієї операції (підстановки) та однієї схеми визначення функції — робить його найпростішою універсальною мовою програмування.

Розробка лямбда-числення була мотивована необхідністю формалізації поняття ефективного обчислення та подолання обмежень традиційних мов і математичних нотацій, де функції переважно іменуються [2]. Це ускладнює роботу з функціями вищого порядку. Лямбда-нотація (наприклад, $\lambda x.x^2$), на відміну від класичного підходу, дає змогу визначати функції без необхідності іменування, що підвищує гнучкість і точність у вираженні обчислювальних процесів.

Символ λ був обраний з міркувань зручності, а не через глибокі семантичні причини. Незважаючи на це, лямбда-числення стало фундаментом функціонального програмування, забезпечуючи формальну основу, еквівалентну машині Тюрінга за обчислювальною потужністю. Його зосередженість на абстрактних правилах редукції, а не на фізичному виконанні, підкреслює зв'язок із програмним забезпеченням як моделлю обчислень [2].

Мінімалізм і обчислювальна повнота лямбда-числення роблять його універсальним інструментом для формального аналізу мов програмування та моделей обчислень. Як Тюрінг-повна система, λ-числення дозволяє моделювати будь-який алгоритм, що надає йому центральне місце в теоретичній інформатиці [2].

Особливої ваги λ-числення набуває у контексті функціонального програмування, де воно слугує математичним підґрунтям для таких ключових концепцій, як функції вищого порядку, каррінг і застосування функцій. Його формальна строгість забезпечує узгоджену модель обробки абстракцій, що лежать в основі мов Haskell, ML, Lisp та інших.

На практиці принципи λ-числення застосовуються в оптимізації програм, зокрема через перетворення λ-виразів у комбінаторну форму. Теорема про повноту комбінаторів демонструє, що будь-який λ-терм може бути представлений виключно через базові комбінатори (наприклад, S, K, I), що спрощує виконання функціональних програм [3].

Реалізація лямбда-числення, попри його абстрактний характер, має вагоме практичне значення, зокрема в контексті функціонального програмування. Haskell як чисто функціональна мова має наступні переваги у парадигмі лямбда-числення:

- Відсутність побічних ефектів та незмінність даних в Haskell відповідають математичній строгості λ-числення, де кожне вираження — це детермінована функція.

- Розвинуті механізми абстракції в Haskell дають змогу чітко й ефективно описувати структуру та правила редукції λ-виразів.
- Використання Haskell як мови, що еволюціонувала з λ-числення, демонструє фундаментальні принципи обчислень та створює основу для подальших досліджень у сферах компіляції, формальної верифікації та дизайну мов програмування [1].

Метою даної роботи є розробка інтерактивного програмного засобу з графічним інтерфейсом для побудови, збереження та покрокової інтерпретації виразів у лямбда-численні. Основна увага приділяється створенню зручного середовища для експериментального дослідження обчислювальних процесів, притаманних цій формальній системі.

1. Теоретичні основи

1.1. Історія лямбда-числення

Розробка лямбда-числення (λ-числення), заснованого американським математиком Алонзо Чорчем у 1930-х роках, була спрямована на формалізацію поняття "обчислюваності" в рамках проблеми розв'язності (Entscheidungsproblem), сформульованої Давидом Гільбертом [3]. Ця проблема вимагала пошуку універсального алгоритму для визначення істинності математичних тверджень.

Паралельно з роботою Чорча, Алан Тюрінг розробив концепцію машин Тюрінга. Незважаючи на різні підходи, λ-числення Чорча та машини Тюрінга виявилися еквівалентними за своєю обчислювальною потужністю. Це відкриття відоме як теза Чорча-Тюрінга і встановлює формальні межі того, що може бути обчислено. [3]

Чорч запровадив λ-нотацію для представлення функцій без необхідності їх іменування, виражаючи їх як абстрактні математичні об'єкти. Символ λ був обраний для позначення функціональної абстракції [4].

Початково λ-числення Чорча розглядалося як універсальний формалізм для математики. Однак, ранні нетипізовані версії зіткнулися з парадоксами (подібними до парадоксу Рассела) через можливість самозастосування функцій. Для вирішення цих проблем Чорч пізніше розробив типізоване λ-числення, додавши поняття типів для запобігання суперечностям [3].

1.2. Синтаксис та семантика

Лямбда-числення (λ-числення) є мінімалістичною формальною системою, що визначає обчислення за допомогою кількох базових елементів та правил.

В основі λ-числення лежать три фундаментальні конструкції [1]:

- Змінна (variable): Базовий ідентифікатор, що позначає значення.
- Лямбда-абстракція (lambda abstraction): Визначає функцію. Має вигляд λid.expr, де id — аргумент функції, а expr — її тіло.
- Застосування (application): Застосовує функцію до аргументу. Має вигляд expr1expr2, де expr1 — функція, а expr2 — аргумент. Застосування є лівоасоціативним: expr1expr2expr3 інтерпретується як ((expr1expr2)expr3).

Поняття вільної та зв'язаної змінної [1]:

- Зв'язана змінна: Змінна x є зв'язаною, якщо її входження знаходиться в області дії λx. Тобто, вона виступає аргументом певної лямбда-абстракції.

- Вільна змінна: Змінна x є вільною у виразі, якщо вона не зв'язана жодною лямбда-абстракцією в цьому виразі.
   - Приклад: У виразі (λx.xy): x є зв'язаною змінною, y — вільною. У виразі (λx.xy)(λy.y): перша y (у лівому підвиразі) є вільною, друга y (у правому підвиразі) — зв'язаною.

Лямбда-числення визначає три основні правила перетворення (редукції), які зберігають семантичне значення виразу [1]:

- Альфа-перетворення (α-редукція): Перейменування зв'язаних змінних без зміни функціональності виразу.
   - λx.expr≡λy.expr[y/x], де expr[y/x] означає заміну всіх вільних входжень x на y у виразі expr.
   - Наприклад: λx.x≡λz.z.
- Бета-перетворення (β-редукція): Імітація застосування функції до аргументу шляхом підстановки аргументу в тіло функції.
   - (λx.expr1)expr2→expr1[expr2/x], де expr1[expr2/x] означає підстановку expr2 замість усіх вільних входжень змінної x у expr1.
   - Перед підстановкою необхідно переконатися, що жодна вільна змінна з expr2 не стане зв'язаною в expr1 за допомогою попередньої α- редукції.
- Ета-перетворення (η-редукція): Встановлення функції λx.(fx) як такої що еквівалентна f, якщо x не є вільною змінною в f.
   - λx.(fx)≡f, якщо x∈/FV(f), де FV(f) — множина вільних змінних в f.
   - Наприклад: λx.(idx)≡id.

Додаткові визначення [1]:

- Редекс (redex): Частина лямбда-виразу, до якої можна застосувати правило редукції.
   - Приклад: (λx.x)y.
- Редукція: Процес послідовного застосування правил перетворення до лямбда-виразу з метою його спрощення.
- Нормальна форма (normal form): Лямбда-вираз, який більше не містить жодних редексів.
   - Хоча не кожен лямбда-вираз має нормальну форму (приклад: Ω=(λx.xx)(λx.xx) – редукується нескінченно), але якщо вона існує, вона є унікальною (Теорема Чорча-Россера).
- Обчислення: процес редукції виразу до нормальної форми.
   - Вибір стратегії редукції впливає на порядок обчислень, що буде детальніше розглянуто в наступному розділі.

1.3. Стратегії обчислення

Вибір порядку виконання редукцій, коли існує кілька редексів, визначається стратегією обчислення (або стратегією редукції). Цей вибір визначає ефективність та завершуваність обчислень.

Дві основні стратегії визначення пріоритетності редукції редексів [4]:

- Нормальний порядок (Normal Order): Завжди редукується найлівіший, найбільш зовнішній редекс. Аргументи редукуються лише тоді, коли їхнє значення є необхідним для обчислення тіла функції.
  - Переваги: Гарантовано знаходить нормальну форму, якщо вона існує. Дозволяє обробляти потенційно нескінченні структури даних.
  - Недоліки: Аргументи обчислюються кілька разів, якщо вони багаторазово використовуються в тілі функції.
- Аплікативний порядок (Applicative Order): Спочатку редукуються всі аргументи до їх нормальної форми (якщо можливо), а потім застосовується функція. Це відповідає виклику за значенням (call-by-value) у багатьох традиційних мовах програмування.
  - Переваги: Кожен аргумент обчислюється лише один раз.
  - Недоліки: Якщо аргумент не має нормальної форми, обчислення може не завершитися, навіть якщо його значення не було б використане.

Стратегії також розрізняються за глибиною редукції виразу [4]:

- Слабке обчислення (Weak Head Normal Form - WHNF): Редукує вираз лише до його слабкої головної нормальної форми. Тобто редукція припиняється, коли зовнішній рівень виразу стає лямбда-абстракцією або змінною. Внутрішні редекси, які не є частиною "головної" структури, залишаються нередукованими.
  - Приклад: Вираз (λx.(λy.(+xy)))(2+3) у WHNF буде λy.(+(2+3)y). Вираз (2+3) у тілі функції не обчислюється.
  - WHNF часто використовується для ефективності, оскільки дозволяє швидко отримати форму результату без повного обчислення всіх його частин.
- Сильне обчислення (Normal Form - NF): Редукує вираз повністю, доки він не досягне своєї нормальної форми. У кінцевому виразі не залишиться жодного редексу, включно з внутрішніми.
  - Сильне обчислення дає повністю редукований результат, але може бути менш ефективним або не завершитися для виразів без нормальної форми.

У даній роботі використовуються стратегії нормального порядку та сильного обчислення, що оптимально для демонстраційного інтерпретатора лямбда-числення. Такий підхід гарантує знаходження нормальної форми виразу і дозволяє коректно обробляти вирази, які можуть не мати нормальної форми, забезпечуючи їхню повну редукцію до заданого ліміту кроків. Сильне обчислення, у свою чергу, забезпечує

повну редукцію всіх внутрішніх редексів, що є необхідним для покрокової візуалізації трансформацій виразу на кожному етапі.

1.4. Кодування Черча

Одна з фундаментальних властивостей чистого лямбда-числення (λ-числення) — це його здатність представляти як обчислення, так і дані (логічні значення, натуральні числа, пари) виключно за допомогою функцій.

Логічні значення [1]

Логічні значення true (істина) та false (хибність) кодуються як функції, що приймають два аргументи і повертають перший або другий з них відповідно:

true ≡ λxy.x

false ≡ λxy.y

На основі цих визначень реалізуються логічні операції:

AND ≡ λab.ab(λuv.v) (або λab.abfalse)

OR ≡ λab.a(λuv.u)b (або λab.atrue b)

NOT ≡ λa.afalse true

Натуральні числа (числа Черча) [1]

Натуральні числа (від 0) кодуються як функції, що приймають функцію-наступник s та початкове значення z, а потім застосовують s до z n разів:

0 ≡ λsz.z

1 ≡ λsz.s(z)

n ≡ λsz.s^n(z)

На основі цих визначень будуються арифметичні операції [1]:

SUCC (наступник) ≡ λnsz.s(nsz)

ADD (додавання) ≡ λab.aSUCC b (або λabsz.as(bsz))

MULT (множення) ≡ λab.a(bs)z (або λab.a(bs))

Реалізація функції PRED (попередник) вимагає використання допоміжних структур, як пари Черча.

PRED (попередник) ≡ λn.FST (n(λp.PAIR (SND p)(SUCC (SND p))) (PAIR zero zero))

Структури даних (Пари Черча) [2]

Пари (кортежі) також можуть бути представлені як функції:

PAIR ≡ λxyf.fxy (при застосуванні до функції f передає x та y як її аргументи)

FST (перший елемент) ≡ λp.ptrue

SND (другий елемент) ≡ λp.pfalse

1.5. Зв'язок із Haskell

λ-числення не є просто теоретичною основою; його принципи втілені в синтаксисі та семантиці функціональних мов. Одними з перших прямих втілень ідей λ-числення були мови сімейства Lisp, де функції розглядалися як першокласні об'єкти, а основною формою визначення функції став лямбда-вираз. Scheme, діалект Lisp, ще більше акцентував на мінімалізмі та чистій інтерпретації λ-числення, підкреслюючи лексичну область видимості [3].

Haskell найповніше і найпослідовніше реалізує принципи λ-числення, адже це чиста, лінива, функціональна мова. У Haskell визначення функцій є безпосередніми лямбда-абстракціями.

Ключові особливості Haskell, що походять з лямбда-числення, включають [3]:

- Функції вищого порядку: Можливість передавати функції як аргументи та повертати їх як результати.
- Ліниві обчислення (lazy evaluation): Прямий наслідок нормального порядку редукції в лямбда-численні. Дозволяє працювати з потенційно нескінченними структурами даних.
- Чистота: Функції в Haskell не мають побічних ефектів, що відповідає математичній моделі лямбда-функцій.
- Каррірування: Функції в Haskell є каррірованими за замовчуванням, що дозволяє застосовувати їх до аргументів по одному.

Каррірування (currying) — техніка перетворення функції, яка приймає множину аргументів, на послідовність функцій, кожна з яких приймає лише один аргумент [5].

2. Haskell як мова реалізації

Вибір мови програмування Haskell для реалізації інтерпретатора лямбда-числення є обґрунтованим її архітектурними особливостями та філософією. Haskell, як чиста функціональна мова, надає природне та ефективне середовище для моделювання функціональних обчислень, що значно спрощує та підвищує елегантність процесу імплементації.

2.1. Переваги принципів Haskell

Haskell — це чиста функціональна мова програмування, де обчислення виражаються виключно як застосування функцій. Така архітектура ідеально відповідає математичній суті лямбда-числення, де функції є детермінованими перетвореннями.

Додатково, Haskell є статично типізованою та типобезпечною мовою, що забезпечує надійність коду та запобігає помилкам типів, невизначеності, арності функцій, і рефакторингу.

2.2. Зіставлення зі зразком та алгебраїчні типи даних

Однією з ключових переваг Haskell для реалізації систем, подібних до інтерпретатора лямбда-числення, є використання алгебраїчних типів даних (Algebraic Data Types – ADT) [2] у поєднанні з зіставленням зі зразком (pattern matching).

Для представлення абстрактного синтаксичного дерева (AST) [2] лямбда-виразу визначено тип даних Expr як ADT:

```haskell
-- Lambda calculus expression AST
data Expr
  = Var String          -- variable, e.g., "x"
  | Lam String Expr     -- lambda abstraction, e.g., \x. expr
  | App Expr Expr       -- application, e.g., e1 e2
  deriving (Eq, Show)
```

Рисунок 2.1 – визначення типу даних Expr

Це класичний приклад ADT, де Expr може набувати однієї з трьох форм: Var (змінна), Lam (лямбда-абстракція) або App (застосування – два вирази типу Expr).

Зіставлення зі зразком дозволяє обробляти різні варіанти конструктора Expr. Функція step, призначена для покрокової редукції, реалізована з використанням окремих гілок для кожного типу редексу. Тут кожна гілка функції step прямо відповідає конкретному типу оброблюваного редексу.

```haskell
-- One step of normal-order reduction
step :: Expr -> Maybe Expr
step (App (Lam x e1) e2) = Just (subst x e2 e1)
step (App e1 e2) =
  case step e1 of
    Just e1' -> Just (App e1' e2)
    Nothing ->
      case step e2 of
        Just e2' -> Just (App e1 e2')
        Nothing -> Nothing
step (Lam x e) = fmap (Lam x) (step e)
step _ = Nothing
```

Рисунок 2.2. – Визначення функції step

2.3. Використання Parsec

Для перетворення вхідних рядків коду лямбда-числення у його внутрішнє представлення за допомогою AST типу Expr використовується бібліотека Parsec.

На відміну від генерації парсерів з окремих граматик (як у Yacc чи ANTLR), Parsec дозволяє визначати правила граматики безпосередньо в коді Haskell. Це забезпечує наступні переваги:

- Модульність: Кожне синтаксичне правило визначається як окремий парсер, що сприяє його легкій комбінації з іншими.
- Читабельність: Визначення парсерів у Parsec тісно відображають структуру граматики, що підвищує їхню зрозумілість та зручність для підтримки.
- Гнучкість: Спрощується обробка помилок парсингу та інтеграція нових синтаксичних конструкцій.

Вибір даної бібліотеки має також і недоліки в порівнянні з альтернативами:

- Продуктивність: Parsec реалізує рекурсивний спуск, який може призводити до бектрекінгу при неоднозначних граматиках.
- Повідомлення про помилки: Стандартні повідомлення про помилки в Parsec відображають технічні деталі процесу парсингу, а не логічні синтаксичні помилки високого рівня.

Проблема продуктивності не є пріоритетом у даній роботі, а повідомлення про помилки у програмі покращені порівняно з базовими повідомленнями Parsec.

2.4. Підтримка Haskell рекурсії та підстановки

Haskell як чиста функціональна мова природно підтримує рекурсію як основний механізм ітерації. Багато ключових функцій інтерпретатора, такі як визначення вільних змінних (freeVars), виконання підстановки (subst) та обчислення (eval), є рекурсивними, що забезпечує пряме відображення структури λ-виразів.

Імплементація функції subst прямо відповідає математичному визначенню підстановки, включаючи механізми обробки захоплення змінних шляхом генерації унікальних імен (через α-перетворення).

```haskell
-- Substitute a variable in an expression
subst :: String -> Expr -> Expr -> Expr
subst var val expr@(Var x)
  | x == var  = val
  | otherwise = expr
subst var val (App e1 e2) =
  App (subst var val e1) (subst var val e2)
subst var val lam@(Lam x e)
  | x == var = lam  -- variable is bound, skip
  | x `S.member` freeVars val =
    let used = freeVars val `S.union` freeVars e
        x' = freshVar used x
        e' = subst x (Var x') e
    in Lam x' (subst var val e')
  | otherwise = Lam x (subst var val e)
```

Рисунок 2.5 - Визначення функції subst

3. Проектування та Реалізація

3.1. Розробка абстрактного синтаксичного дерева (AST)

Центральним елементом будь-якого інтерпретатора мови є його внутрішнє представлення програми. У даній роботі, таким представленням виступає абстрактне синтаксичне дерево (AST).

Однією з ключових переваг лямбда-числення є його мінімалізм. Тут використовуються лише три базових поняття: змінні, абстракції та застосування (виклики функцій). Цей мінімалістичний підхід використано при розробці AST, що також зменшує ймовірність помилок та спрощує подальший аналіз та обчислення виразів.

У мові Haskell, що сприяє моделюванню таких структур завдяки її підтримці ADT, ми визначаємо AST для лямбда-виразів за допомогою типу даних Expr:

```
-- Lambda calculus expression AST
data Expr
  = Var String           -- variable, e.g., "x"
  | Lam String Expr      -- lambda abstraction, e.g., \x. expr
  | App Expr Expr        -- application, e.g., e1 e2
  deriving (Eq, Show)
```

Рисунок 3.1 – визначення типу даних Expr

3.2. Розбір лямбда-виразів

Парсер відповідає за перетворення синтаксичного тексту, написаного користувачем, у відповідну структуру даних Expr. У даній роботі для цієї мети використовується бібліотека Parsec.

Розробка парсера за допомогою Parsec складається з визначення граматики лямбда-числення у вигляді комбінації спеціалізованих парсерів. Основні синтаксичні конструкції правила для лямбда-виразів, які необхідно розпізнавати, є змінна та лямбда-абстракція, застосування, дужки.

У коді це відображається через послідовне визначення парсерів:

```haskell
lambdaExpr :: Parser Expr
lambdaExpr = do
  char '\\' <|> char 'λ'
  spaces
  vars <- many1 (many1 letter <* spaces)  -- parse one or more variables
  char '.'
  spaces
  body <- expr
  return $ foldr Lam body vars            -- nest them: \x y . E  => Lam x (Lam y E)
```

Рисунок 3.2. - визначення парсерів змінної, дужок, абстракції та застосування

Парсер lambdaExpr дозволяє користувачу використовувати як символ «\», так і «λ» для позначення абстракції. Також парсер враховує пробіли між елементами синтаксису і дозволяє користувачам писати вирази в читабельному форматі без жорсткого форматування.

Конструкція many1 (many1 letter <* spaces) дозволяє парсеру розпізнавати послідовність змінних у лямбда-абстракції. Після розбору, foldr Lam body vars автоматично перетворює це на вкладені абстракції для зручності.

Рекурсивна природа визначення expr дозволяє парсеру обробляти довільно вкладені лямбда-вирази. Наприклад, тіло абстракції може бути іншою абстракцією, або застосування може містити вкладені функції та аргументи.

Парсер parens забезпечує правильне групування виразів. Оскільки операція застосування є лівоасоціативною, використання дужок є необхідним для зміни стандартного порядку операцій. Далі парсер appExpr використовує foldl1 App для правильної лівоасоціативності при застосуванні.

```haskell
parens :: Parser Expr
parens = between (char '(' >> spaces) (spaces >> char ')') expr

appExpr :: Parser Expr
appExpr = do
  es <- many1 (spaces *> (parens <|> varExpr <|> lambdaExpr) <* spaces)
  return $ foldl1 App es

expr :: Parser Expr
expr = appExpr <|> varExpr <|> lambdaExpr <|> parens

parseExpr :: String -> Either ParseError Expr
parseExpr = parse (spaces *> expr <* spaces <* eof) ""
```

Рисунок 3.3. - визначення парсерів – дужок, лівоасоціативного, основного та парсеру повного виразу

Порядок комбінування парсерів у expr має значення для уникнення неоднозначностей. Більш складні парсери зазвичай розміщуються раніше і у даній роботі appExpr є найскладнішим, оскільки може містити інші типи виразів, тому стоїть першим.

Розроблений парсер перетворює текстове представлення лямбда-виразів у структуроване AST для подальшої обробки та обчислення.

3.3. Гарний друк та відображення виразів

Після успішного розбору лямбда-виразу в його внутрішнє представлення (AST Expr) та перед виконанням обчислень необхідно відобразити цей вираз у читабельному та коректному форматі. У даному інтерпретаторі ця функціональність забезпечується механізмом "гарного друку" (pretty-printing).

Метою гарного друку є перетворення внутрішнього представлення Expr назад у зрозумілий рядок, який точно відображає його структуру та значення. Це вимагає обережного підходу до розстановки дужок.

У даній роботі це досягається завдяки реалізації функції showExpr, яка рекурсивно проходить по структурі Expr та формує відповідний рядок:

```
-- Printing Expr with parentheses
showExpr :: Expr -> String
showExpr (Var x) = x
showExpr (Lam x e) = "λ" ++ x ++ "." ++ showExpr e
showExpr (App e1 e2) = "(" ++ showExpr e1 ++ " " ++ showExpr e2 ++ ")"
```

Рисунок 3.4. - визначення функції showExpr

Тут найцікавішим для розгляду є showExpr (App e1 e2) з точки зору друку дужок. Оскільки застосування є лівоасоціативним і має вищий пріоритет зв'язування, ніж абстракція, застосування завжди оточено дужками. Це гарантує, що вирази, такі як f (g x) або (λx.x) y, відображаються коректно, з уникненням неоднозначностей. Наприклад, App (Var "f") (App (Var "g") (Var "x")) стає (f (g x)). Це відповідає інтуїтивному запису цих виразів. Без дужок, App (Var "f") (App (Var "g") (Var "x")) міг би бути відображений як "f g x", що при зворотному парсингу інтерпретувалося б як ((f g) x) через лівоасоціативність, змінюючи його семантику.

На додаток до базової функції showExpr, Haskell надає зручний механізм через клас типів Show та похідну реалізацію. Проте, стандартна реалізація Show часто додає надмірну кількість дужок (відображаючи структуру конструкторів AST), що робить вирази менш читабельними. Тому фокус реалізації функція showExpr тут полягає у мінімізації надлишкових дужок при збереженні однозначності синтаксису.

3.4. Роздільна здатність середовища та змінних

У лямбда-численні, як і в будь-якій мові програмування, змінні відіграють ключову роль. Але необхідно визначити, як ідентифікуються та розв'язуються ці змінні. Цей процес називається роздільною здатністю змінних

Хоча базове лямбда-числення дозволяє створювати вирази без явних імен (наприклад, λx.x), на практиці програмування вимагає можливості давати назви функціям та значенням. Це значно підвищує читабельність, модульність та можливість повторного використання коду. У нашому інтерпретаторі ця можливість реалізована через концепцію іменованих виразів або визначень. Користувач може присвоїти лямбда-виразу певне ім'я, яке потім може бути використане в інших виразах. Ці іменовані вирази зберігаються в середовищі — відображенні імен на відповідні їм лямбда-вирази.

Користувачі можуть зберігати свої власні лямбда-визначення в базі даних SQLite. При завантаженні програми ці записи зчитуються та інтегруються в загальне середовище за допомогою функції buildEnvFromRecords. Для використання їх у подальших сесіях.

```
-- Extract name-body map from DB records
buildEnvFromRecords :: [Record] -> Map String Expr
buildEnvFromRecords records = Map.fromList [ (name, parseUnsafe body)
                                           | Record _ _ _ name body _ <- records ]
```

Рисунок 3.5. - визначення функції buildEnvFromRecords

Існують дві основні філософії щодо того, як розв'язуються змінні, якщо ім'я зустрічається в кількох областях:

- Статична роздільна здатність (лексична область видимості - Lexical Scoping):

У цьому підході змінна розв'язується на основі того місця, де вона була визначена у вихідному коді, а не там, де функція викликається. Підхід робить програми більш передбачуваними та легкими для розуміння, оскільки поведінку функції можна визначити, просто прочитавши її код та код зовнішніх функцій, які її охоплюють.

- Динамічна роздільна здатність (Dynamic Scoping):

Цей підхід, який був популярний у деяких ранніх мовах (наприклад, ранні версії Lisp), розв'язує змінну на основі її значення в поточному контексті виконання або в "ланцюжку викликів". Це може призвести до непередбачуваної поведінки та ускладнити розуміння програм, оскільки значення змінної може залежати від неочевидних "віддалених" викликів функцій.

Лямбда-числення за своєю природою має статичну область видимості. Це означає, що при β-редукції вільні змінні зберігають своє початкове значення, незалежно від зовнішнього контексту, в якому відбувається застосування. Це досягається завдяки

застосуванню α-перетворень під час підстановки, про що детальніше йдеться в наступному підрозділі.

3.5. Підстановка та перезапис без захоплення

Основою інтерпретатора лямбда-числення є коректна реалізація підстановки. Ця операція є фундаментальною для виконання β-редукції, оскільки імітує процес застосування функції до її аргументу шляхом заміни формального параметра на фактичний аргумент у тілі функції. Але для запобігання частим помилкам необхідно враховувати проблему захоплення змінних.

Підстановка (у даній роботі реалізована функцією subst) перевіряє, чи ім'я змінної, що зв'язується (наприклад, x у Lam x e), не співпадає з будь-якою вільною змінною в значенні, яке підставляється. Якщо таке співпадіння відбувається, зв'язана змінна в Lam має бути перейменована на нове ім'я, яке не конфліктує ні з вільними змінними, ні з іншими змінними в e.

```
-- Substitute a variable in an expression
subst :: String -> Expr -> Expr -> Expr
subst var val expr@(Var x)
  | x == var  = val
  | otherwise = expr
subst var val (App e1 e2) =
  App (subst var val e1) (subst var val e2)
subst var val lam@(Lam x e)
  | x == var = lam  -- variable is bound, skip
  | x `S.member` freeVars val =
      let used = freeVars val `S.union` freeVars e
          x' = freshVar used x
          e' = subst x (Var x') e
      in Lam x' (subst var val e')
  | otherwise = Lam x (subst var val e)
```

Рисунок 3.6 - Визначення функції subst

Для коректного виконання α-перетворення нам потрібен надійний спосіб генерації нових унікальних імен змінних, які гарантовано не конфліктуватимуть з жодними існуючими змінними у виразі. Для цього була створена функція freshVar:

```haskell
-- Free variables in an expression
freeVars :: Expr -> S.Set String
freeVars (Var x)     = S.singleton x
freeVars (App e1 e2) = freeVars e1 `S.union` freeVars e2
freeVars (Lam x e)   = S.delete x (freeVars e)

-- Generate a fresh variable name not in the set of used variables
freshVar :: S.Set String -> String -> String
freshVar used base = head $ filter (`S.notMember` used) candidates
  where candidates = [base ++ show i | i <- [1..]]
```

Рисунок 3.6 - Визначення функції freshVar

Алгоритм freshVar приймає множину used всіх вільних змінних, які вже присутні у виразі, та інших змінних, які могли б спричинити конфлікт і базове ім'я base.

Він генерує послідовність кандидатів на нові імена, додаючи до базового імені числа: [base ++ "1", base ++ "2", base ++ "3", ...].

Потім він фільтрує цю послідовність, залишаючи лише ті імена, які не містяться у множині used.

Нарешті, head бере перше таке знайдене ім'я.

Цей метод гарантує, що згенероване ім'я є унікальним для поточного контексту і не призведе до захоплення змінних, забезпечуючи коректність та детермінованість обчислень.

3.6. Механізм обчислення

Після того, як лямбда-вираз був успішно розібраний в AST і розв'язані всі іменовані змінні, центральним етапом роботи інтерпретатора є механізм обчислення. Цей механізм відповідає за застосування правил редукції до виразу, щоб привести його до нормальної форми.

У даному інтерпретаторі використовується стратегія редукції за нормальним порядком. Як було описано вище, ця стратегія полягає в тому, що завжди редукується найлівіший редекс.

Ця реалізація використовує саме "повний" нормальний порядок, тобто редукція відбувається доки весь вираз не досягне своєї нормальної форми, включаючи редукцію всередині тіл функцій. Це відрізняється від слабкої нормальної форми (WHNF), яку часто використовують мови з лінивими обчисленнями (як Haskell за замовчуванням) для швидкого отримання зовнішньої форми виразу. Повний нормальний порядок гарантує виконання всіх можливих обчислень і максимальне спрощення виразу.

Основу механізму обчислення складає функція step, яка виконує один крок редукції:

```haskell
-- One step of normal-order reduction
step :: Expr -> Maybe Expr
step (App (Lam x e1) e2) = Just (subst x e2 e1)
step (App e1 e2) =
  case step e1 of
    Just e1' -> Just (App e1' e2)
    Nothing ->
      case step e2 of
        Just e2' -> Just (App e1 e2')
        Nothing -> Nothing
step (Lam x e) = fmap (Lam x) (step e)
step _ = Nothing
```

Рисунок 3.7 - Визначення функції step

Логіка step віддзеркалює правила нормального порядку:

Якщо наявне застосування (App) лямбда-абстракції (Lam) до аргументу, виконується β-редукція за допомогою функції subst.

Якщо дано застосування, але лівий операнд (функція) ще не є лямбда-абстракцією, ми рекурсивно редукується функція (step e1).

Якщо функція вже в нормальній формі, але правий операнд (аргумент) є редексом, редукується аргумент (step e2).

Якщо дана лямбда-абстракція, рекурсивно редукується тіло абстракції (step e). Це відповідає "повному" нормальному порядку, оскільки навіть вирази всередині тіл функцій редукуються до кінця.

Функція step повертає Maybe Expr, оскільки Nothing вказує на те, що подальші редукції неможливі, тобто вираз досяг своєї нормальної форми.

Для кращого розуміння, даний інтерпретатор надає можливість трасування обчислення, тобто відображення всіх проміжних кроків редукції виразу.

Це досягається за допомогою функції evalStepsLimited:

```haskell
-- Evaluate expression with a maximum step limit (returns only computed steps)
evalStepsLimited :: Int -> Expr -> [Expr]
evalStepsLimited maxSteps = go 0 []
  where
    go n acc e
      | n >= maxSteps = acc ++ [e]  -- return partial result up to maxSteps
      | otherwise = case step e of
          Just e' -> go (n + 1) (acc ++ [e]) e'
          Nothing -> acc ++ [e]     -- normal form reached
```

Рисунок 3.7 - Визначення функції evalStepsLimited

Функція evalStepsLimited починає з початкового виразу e і потім рекурсивно застосовує step до результату попереднього кроку, збираючи всі проміжні вирази в список, доки step не поверне Nothing (що вказує на досягнення нормальної форми). Цей список кроків потім виводиться в інтерфейсі користувача, надаючи детальну картину процесу обчислення.

Нормальний порядок гарантує знаходження нормальної форми, якщо вона існує, та деякі лямбда-вирази можуть ніколи не досягти нормальної форми (наприклад, (λx.xx)(λx.xx) – комбінатор Ω, який редукується сам у себе).

Щоб уникнути нескінченних циклів обчислення, в межах ліміту в 1000 кроків, обчислення припиняється. Це забезпечує стабільність програми та надає користувачеві зворотний зв'язок про потенційно нескінченні обчислення, не дозволяючи програмі зависнути.

4. Функціональність та архітектурні рішення

4.1. Підтримка зовнішніх середовищ

Для будь-якого практичного інтерпретатора критично важлива здатність до розширення його функціоналу за допомогою користувацьких визначень. У даному лямбда-інтерпретаторі ця можливість реалізована через підтримку зовнішніх середовищ. Це дозволяє зберігати та завантажувати іменовані лямбда-вирази, розширюючи базовий набір можливостей інтерпретатора.

Програма використовує базу даних SQLite як сховище для користувацьких визначень лямбда-виразів. Кожне визначення зберігається як запис у таблиці eLambda, що містить такі поля:

idModel, id, num: Числові ідентифікатори, що допомагають організувати записи.

name: Ім'я лямбда-виразу. Це ім'я буде використовуватись для посилання на вираз у коді.

txBody: Тіло лямбда-виразу у вигляді текстового рядка.

txComm: Коментар до визначення, що дозволяє користувачам додавати пояснення.

Функція loadDatabase відповідає за зчитування цих записів з вказаного файлу .db. Вона встановлює з'єднання з базою даних, виконує SQL-запит для вибірки всіх записів з таблиці eLambda та перетворює їх на список об'єктів Record у Haskell. Ці записи потім використовуються для побудови середовища.

```
-- Load the database and print records
loadDatabase :: FilePath -> IO [Record]
loadDatabase dbPath = do
  conn <- open dbPath
  results <- query_ conn "SELECT idModel, id, num, name, txBody, txComm FROM eLambda" :: IO [Record]
  close conn
  return results
```

Рисунок 4.1 - Визначення функції loadDatabase

Користувацький інтерфейс (GUI) надає кнопку "Load .db File", яка викликає діалогове вікно вибору файлу. Після вибору файлу, loadDatabase зчитує дані, оновлює таблицю відображення записів у GUI та зберігає їх у пам'яті програми (IORef records).

Після завантаження з бази даних, функція buildEnvFromRecords перетворює список Record у Map String Expr. Кожен запис з бази даних, що містить ім'я (name) та тіло (txBody) лямбда-виразу, додається до цього відображення.

```haskell
-- ----------------------------
-- Environment handling
-- ----------------------------

-- Resolve named variables
resolveEnv :: Map String Expr -> Expr -> Expr
resolveEnv env (Var x) = Map.findWithDefault (Var x) x env
resolveEnv env (App e1 e2) = App (resolveEnv env e1) (resolveEnv env e2)
resolveEnv env (Lam x e) = Lam x (resolveEnv env e)

-- Extract name-body map from DB records
buildEnvFromRecords :: [Record] -> Map String Expr
buildEnvFromRecords records = Map.fromList [ (name, parseUnsafe body)
                                           | Record _ _ _ name body _ <- records ]
```

Рисунок 4.2 - Визначення функцій resolveEnv, buildEnvFromRecords

```haskell
-- Evaluate a program with an environment
evalProgram :: Env -> [Statement] -> Either String ([Expr], Expr)
evalProgram env stmts =
  case splitAtAssignments stmts of
    (defs, ExprOnly e) ->
      let newEnv = foldl insertDef env defs
          resolved = resolveEnv newEnv e
          steps = evalStepsLimited 1000 resolved
      in Right (steps, last steps)
    _ -> Left "No expression to evaluate."
  where
    insertDef acc (Assign name expr) = Map.insert name expr acc
    insertDef acc _ = acc

    splitAtAssignments xs = (init xs, last xs)
```

Рисунок 4.3 - Визначення функції evalProgram

У наведеній evalProgram, env становить базове середовище. До нього потім додаються визначення, зроблені безпосередньо в полі вводу користувача (Assign String Expr). Потім застосовується функція resolveEnv, яка замінює всі іменовані змінні в кінцевому виразі на їхні відповідні лямбда-вирази з цього об'єднаного середовища.

4.2. Інтеграція з графічним інтерфейсом

Для інтерактивності та доступності у даному інтерпретаторі лямбда-числення реалізовано графічний інтерфейс користувача, що дозволяє безпосередньо

взаємодіяти з лямбда-виразами, спостерігати за редукцією та керувати визначеннями. Для побудови GUI використовується бібліотека Haskell GTK.

Ключова інтерактивна функціональність інтерпретатора зосереджена на дозволі користувачеві вводити лямбда-вирази та спостерігати за їхнім обчисленням:

- Введення лямбда-виразу: Користувач вводить бажаний лямбда-вираз (можливо, з використанням раніше визначених імен) у багаторядкове текстове поле (lambdaInputView).
- Запуск обчислення: Натискання кнопки "Evaluate" ініціює процес обчислення. Функція-обробник onEval читає текст з lambdaInputView.
- Парсинг та підготовка середовища: Введений текст передається парсеру (parseProgram), який перетворює його на список операторів (Statement). Потім, система об'єднує вбудовані визначення (preludeEnv) та визначення, завантажені з бази даних, формуючи повне середовище (Env).
- Покрокова візуалізація оцінки: Це одна з найцінніших функцій інтерфейсу. Замість того, щоб просто відобразити кінцевий результат, інтерпретатор використовує функцію evalStepsLimited, яка генерує повний список проміжних кроків редукції. Кожен крок показує поточний стан лямбда-виразу. Цей список, відформатований за допомогою showExpr для гарного друку, виводиться в окреме текстове поле (textView).

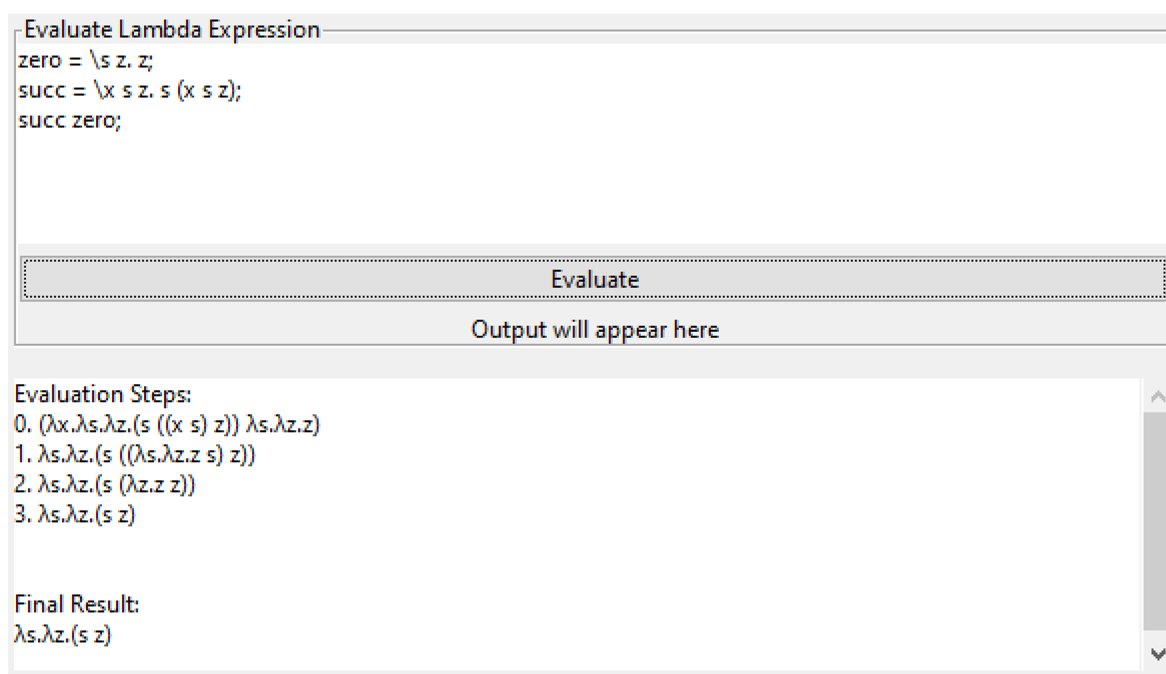

Рисунок 4.4 – Приклад обчислення програмою виразу «Наступне число від 0»

Крім цього, GUI також надає функціональність для збереження нових визначень у базу даних (через поля введення та кнопку "Save to .db File") та можливість відображення вже збережених записів у TreeView, що створює цілісну систему для керування та експериментів із лямбда-виразами.

5. Тестування та Валідація

Розробка інтерпретатора лямбда-числення вимагає ретельного тестування та валідації. Тестування даної роботи зосереджується на ключових аспектах: парсингу, підстановці та механізмі оцінки.

1. Тести для парсингу:
   - Мета: Переконатися, що парсер правильно перетворює вхідні рядки лямбда-виразів у відповідне AST Expr.
   - Випадки:
   - Базові вирази: Перевірка розбору окремих змінних (x), простих абстракцій (\x.x) та застосувань (f x).
   - Складні синтаксичні конструкції: Тестування вкладених абстракцій (\x y. x), складних застосувань (f g h), виразів з дужками ((f x) y та f (g y)), а також коректної обробки пробілів та різних символів лямбда (\ та λ).
   - Некоректний синтаксис: Перевірка того, що парсер належним чином повідомляє про помилки синтаксису (наприклад, незакриті дужки, відсутні крапки в абстракціях).
2. Тести для підстановки:
   - Мета: Підтвердити, що функція subst (підстановка) працює коректно, а головне – без захоплення змінних.
   - Випадки:
   - Проста підстановка: subst "x" (Var "a") (Var "x") повинно дати Var "a".
   - Підстановка в абстракцію: subst "x" (Var "a") (Lam "y" (App (Var "x") (Var "y"))) повинно дати Lam "y" (App (Var "a") (Var "y")).
   - Тестування α-перетворення: Критично важливі випадки, що демонструють запобігання захопленню змінних:
   - subst "x" (Var "y") (Lam "y" (Var "x")) повинно дати Lam "y1" (Var "y"), а не Lam "y" (Var "y").
   - Перевірка генерації свіжих імен freshVar у контексті складних виразів.
3. Тести для оцінки (редукції):
   - Мета: Переконатися, що механізм обчислення (функції step та evalSteps) правильно виконує β-редукцію та приводить вирази до їхньої нормальної форми.
   - Випадки:
   - Базова β-редукція: (\x. x) y редукується до y.
   - Комбінатори: Тестування відомих комбінаторів, таких як I (ідентичність), K (константа), S (композиція).
   - Числа Черча: Перевірка, що succ zero редукується до one, add one one редукується до two, mult two two до four тощо.
   - Булеві вирази: and true false редукується до false, not true до false, if true A B до A.

- Нескінченні редукції: Тестування виразів, що не мають нормальної форми (наприклад, комбінатор $\Omega=(\lambda x.xx)(\lambda x.xx)$), щоб перевірити, як спрацьовує обмеження кроків (evalStepsLimited).

Окрім модульних тестів, перевірено інтеграційні тестові випадки, які перевіряють взаємодію між різними компонентами та загальну функціональність інтерпретатора: Сценарії, де користувач визначає власні функції через присвоєння та потім використовує їх. Сценарії завантаження та збереження визначень з/до SQLite, перевірки коректності збереження та відновлення даних.

6. Порівняння з аналогічними роботами

6.1. Інші лямбда-інтерпретатори на Haskell

Існує безліч реалізацій лямбда-інтерпретаторів, особливо в академічному середовищі та як навчальні проєкти, і Haskell є дуже популярним вибором для їх написання завдяки своїй функціональній парадигмі. Багато з цих інтерпретаторів зосереджуються на основних аспектах: парсингу, підстановці та редукції. Проте, вони часто відрізняються у своїх пріоритетах та додаткових функціях:

Більшість простих інтерпретаторів працюють у консольному режимі, приймаючи вираз як вхідні дані та повертаючи його нормальну форму. Вони підходять для демонстрації базових концепцій, але обмежують інтерактивність.

Окремі проєкти можуть включати розширення лямбда-числення, такі як рекурсивні визначення (через Y-комбінатор або фіксовані точки), арифметику з "великими" числами, або інтеграцію з монадними ефектами для IO.

Даний інтерпретатор, у порівнянні, вирізняється графічним інтерфейсом користувача (GUI) на базі GTK+ та надійним зберіганням визначень у базі даних SQLite. Це зміщує фокус з чистої демонстрації редукції до створення більш інтерактивного та практичного інструменту, що дозволяє користувачеві не тільки обчислювати вирази, але й ефективно керувати бібліотекою власних лямбда-функцій.

6.2. Відмінності з типізованим лямбда-численням та інтерпретаторами

Важливо розрізняти нетипізоване лямбда-числення (яке реалізовано в цьому проєкті) та типізоване лямбда-числення.

Нетипізоване лямбда-числення – фундаментальна модель обчислень, де немає поняття "типу" для виразів. Будь-який вираз може бути застосований до будь-якого іншого виразу, і єдиним механізмом є β-редукція. Простота нетипізованого лямбда-числення робить його ідеальним для вивчення базових принципів обчислень та рекурсії. Однак, воно дозволяє писати вирази, які не мають "сенсу" або призводять до нескінченної редукції (наприклад, (λx.xx)(λx.xx)).

Типізоване лямбда-числення: Вводить систему типів, яка накладає обмеження на те, які вирази можуть бути застосовані один до одного. Наприклад, функція, яка приймає ціле число, не може бути застосована до булевого значення. Системи типів забезпечують типову безпеку, гарантуючи, що "добре типізовані" програми не застрягнуть і не призведуть до невизначеної поведінки.

Реалізація інтерпретатора типізованого лямбда-числення значно складніша, оскільки вимагає визначення розширеного AST для типів, реалізації алгоритму перевірки або виведення типів та обробки потенційних типових помилок.

Даний інтерпретатор фокусується на нетипізованому лямбда-численні для демонстрації основних механізмів обчислення у їхній найчистішій формі. Це дозволило зосередитися на коректній реалізації парсингу, підстановки, α-перетворень та β-редукції, не ускладнюючи початкову реалізацію системою типів.

6.3. Компроміси в розробці реалізації

Під час розробки даного інтерпретатора було зроблено кілька ключових компромісів, що вплинули на його функціональність та архітектуру:

Використання "повного" нормального порядку обчислення, хоча й є теоретично чистим, може бути менш ефективним для деяких виразів, ніж слабка нормальна форма. Операції підстановки можуть створювати багато проміжних структур даних, що має бути оптимізовано в більш масштабних проєктах.

Фокус на інтерактивному GUI та персистентності за допомогою SQLite означав менше уваги до реалізації зокрема розширених типів даних або вбудованих I/O. Цей компроміс зробив інтерпретатор більш доступним та зручним для кінцевого користувача.

Вибір бібліотек:

- Parsec: Був обраний через його елегантність та простоту для написання парсерів без необхідності використовувати зовнішні генератори. Це прискорило розробку та зробило граматику ближчою до коду.
- Haskell GTK: Надає потужний, але низькорівневий контроль над GUI. Вибір цієї бібліотеки був зроблений для створення повноцінного десктопного застосунку на Haskell, хоча існують і вищі рівні абстракції для GUI для веб-браузерів.
- SQLite.Simple: Ця бібліотека забезпечує зручний та типобезпечний інтерфейс для роботи з SQLite, що було достатньо для потреб зберігання визначень.

Висновки

У даній роботі розроблено та реалізовано інтерактивний інтерпретатор нетипізованого лямбда-числення за допомогою мови програмування Haskell. Проєкт має на меті створити не просто функціональний обчислювач, а освітній інструмент, що дозволяє користувачам зрозуміти фундаментальні концепції обчислень та функціонального програмування.

Вибір Haskell як мови реалізації виявився оптимальним, оскільки його чиста функціональна парадигма, потужна система типів та вбудована підтримка алгебраїчних типів даних і зіставлення зі зразком ідеально відображають математичну природу лямбда-числення. Це дозволило створити елегантну та типобезпечну модель абстрактного синтаксичного дерева (AST), що є основою для всіх подальших операцій.

Ключові компоненти інтерпретатора включають:

- Надійний парсер, розроблений за допомогою бібліотеки Parsec, який ефективно перетворює текстові лямбда-вирази в AST, коректно обробляючи змінні, абстракції, застосування та дужки.
- Механізм обчислення, що реалізує повний нормальний порядок редукції. Це забезпечує приведення виразів до їхньої найпростішої форми, включно з редукцією всередині тіл функцій. Основою цього механізму є коректна функція підстановки (subst), яка завдяки α-редукції та механізму генерації свіжих змінних (freshVar) повністю запобігає проблемі захоплення змінних.
- Підтримку зовнішніх середовищ для зберігання та завантаження користувацьких визначень з бази даних SQLite. Це дозволило створювати та повторно використовувати іменовані лямбда-вирази для розширення функціональності.
- Інтеграцію з графічним інтерфейсом (GUI), розробленим за допомогою Haskell GTK. GUI забезпечив зручне введення виразів, візуалізацію їхніх результатів та головне – покрокове трасування обчислення.

Новизна даної роботи полягає в комплексному підході до реалізації інтерпретатора лямбда-числення, який перетворює його з суто обчислювального інструменту на інтерактивну та візуалізовану платформу. Основними елементами тут є:

Створення повноцінного та інтерактивного GUI для нетипізованого лямбда-інтерпретатора: На відміну від більшості існуючих консольних реалізацій, даний інтерпретатор надає зручний графічний інтерфейс, що значно знижує поріг входження для користувачів та дозволяє їм легше взаємодіяти з теорією.

Детальна покрокова візуалізація β-редукції: Відображення кожного етапу обчислення, включно з механізмами підстановки та α-перетворення, дозволяє користувачам наочно розуміти динаміку лямбда-числення, що є критично важливим для розуміння даної абстрактної концепції.

Система управління користувацькими визначеннями: Інтеграція з базою даних дозволяє користувачам зберігати власні лямбда-функції та створювати персоналізовані бібліотеки. Це підвищує практичну цінність інтерпретатора як інструменту для навчання та дослідження складніших виразів з часом.

Список використаної літератури

1) Глибовець М. М., Кирієнко О. В., Проценко В. С. : Моделі обчислень у програмній інженерії : Навч. посіб. 54-те вид. Київ : Вид. дім «Києво-Могилянська академія», 2019. 212 с.
2) Pierce, B. C. : Types and Programming Languages. Ukraine: MIT Press, 2002, 655 p.
3) Harrison J. : Introduction to Functional Programming : Лекції. Cambridge : Computer Laboratory, University of Cambridge, 1997. 159 c. URL: https://www.cl.cam.ac.uk/teaching/Lectures/funprog-jrh-1996/all.pdf
4) Thompson S. : Haskell the craft of functional programming : [3rd ed.]. Harlow : Pearson Education Limited, 2011. 598 c.
5) O'Sullivan B., Goerzen J., Stewart D. B. : Real World Haskell : [1st ed.]. United States: O'Reilly Media, 2008. 714 c.

Додатки

Вихідний код програми:

```
{-# LANGUAGE OverloadedStrings #-}

module Main where

import Graphics.UI.Gtk
import Graphics.UI.Gtk.General.Enums (Modifier(..))
import qualified Graphics.UI.Gtk.Gdk.Events as Gdk
import Database.SQLite.Simple hiding ((:=), Statement)
import Database.SQLite.Simple.FromRow
import Database.SQLite.Simple (execute_)
import Data.Text (Text)
import Data.Char (isAlphaNum)
import Data.Map (Map)
import Data.IORef
import qualified Data.Text as T
import qualified Data.Text as T
import qualified Data.Set as S
import qualified Data.Map as Map
import Text.Read (readMaybe)
import Text.Parsec
import Text.Parsec.String (Parser)
import Control.Applicative (many)
import Control.Monad.IO.Class (liftIO)
import Control.Monad (when)
import qualified Control.Exception as E


-- Define a record for the table row
data Record = Record Int Int Int String String String deriving Show

-- Define the database schema
instance FromRow Record where
  fromRow = Record <$> field <*> field <*> field <*> field <*> field <*> field

-- Define the row conversion for inserting into the database
instance ToRow Record where
  toRow (Record idModel id num name txBody txComm) =
    toRow (idModel, id, num, name, txBody, txComm)

-- Lambda calculus expression AST
data Expr
  = Var String        -- variable, e.g., "x"
  | Lam String Expr   -- lambda abstraction, e.g., \x. expr
  | App Expr Expr     -- application, e.g., e1 e2
  deriving (Eq, Show)

-- Statement for assignment or expression evaluation
data Statement
  = Assign String Expr
  | ExprOnly Expr
  deriving (Show)
```

```
type Env = Map.Map String Expr

-- ----------------------

-- Parsing Statements
statement :: Parser Statement
statement = try assignment <|> exprStmt

-- Parse an assignment statement
assignment :: Parser Statement
assignment = do
  name <- many1 letter
  spaces
  char '='
  spaces
  e <- expr
  return $ Assign name e

-- Parse an expression statement
exprStmt :: Parser Statement
exprStmt = ExprOnly <$> expr

-- Parse an expression from a string, returning an Expr or a ParseError
parseUnsafe :: String -> Expr
parseUnsafe str =
  case parseExpr str of
    Right e -> e
    Left err -> error ("Failed to parse built-in definition: " ++ show err)

-- ----------------------

-- Printing Expr with parentheses
showExpr :: Expr -> String
showExpr (Var x) = x
showExpr (Lam x e) = "λ" ++ x ++ "." ++ showExpr e
showExpr (App e1 e2) = "(" ++ showExpr e1 ++ " " ++ showExpr e2 ++ ")"

-- Load the database and print records
loadDatabase :: FilePath -> IO [Record]
loadDatabase dbPath = do
  conn <- open dbPath
  results <- query_ conn "SELECT idModel, id, num, name, txBody, txComm FROM eLambda" :: IO [Record]
  close conn
  return results

saveDatabase :: FilePath -> [Record] -> IO ()
saveDatabase dbPath records = do
  conn <- open dbPath
  execute_ conn "CREATE TABLE IF NOT EXISTS eLambda (idModel INTEGER, id INTEGER, num INTEGER, name TEXT,
txBody TEXT, txComm TEXT)"
  execute_ conn "DELETE FROM eLambda"
  mapM_ (execute conn "INSERT INTO eLambda (idModel, id, num, name, txBody, txComm) VALUES (?, ?, ?, ?, ?,
?)") records
  close conn
```

```
-- ----------------------
-- Parsing
-- ----------------------
lambdaExpr :: Parser Expr
lambdaExpr = do
  char '\\' <|> char 'λ'
  spaces
  vars <- many1 (many1 letter <* spaces)  -- parse one or more variables
  char '.'
  spaces
  body <- expr
  return $ foldr Lam body vars          -- nest them: \x y . E  => Lam x (Lam y E)

varExpr :: Parser Expr
varExpr = Var <$> many1 letter

parens :: Parser Expr
parens = between (char '(' >> spaces) (spaces >> char ')') expr

appExpr :: Parser Expr
appExpr = do
  es <- many1 (spaces *> (parens <|> varExpr <|> lambdaExpr) <* spaces)
  return $ foldl1 App es

expr :: Parser Expr
expr = appExpr <|> varExpr <|> lambdaExpr <|> parens

parseExpr :: String -> Either ParseError Expr
parseExpr = parse (spaces *> expr <* spaces <* eof) ""

programParser :: Parser [Statement]
programParser = sepEndBy statement (many1 (oneOf "\n;")) <* eof

parseProgram :: String -> Either ParseError [Statement]
parseProgram = parse (spaces *> programParser <* spaces) ""

-- ---------------------------
-- Environment handling
-- ----------------------------

-- Resolve named variables
resolveEnv :: Map String Expr -> Expr -> Expr
resolveEnv env (Var x) = Map.findWithDefault (Var x) x env
resolveEnv env (App e1 e2) = App (resolveEnv env e1) (resolveEnv env e2)
resolveEnv env (Lam x e) = Lam x (resolveEnv env e)

-- Extract name-body map from DB records
buildEnvFromRecords :: [Record] -> Map String Expr
buildEnvFromRecords records = Map.fromList [ (name, parseUnsafe body)
                                   | Record _ _ _ name body _ <- records ]

-- ---------------------------
-- Evaluation
-- ---------------------------
```

```
-- Free variables in an expression
freeVars :: Expr -> S.Set String
freeVars (Var x)     = S.singleton x
freeVars (App e1 e2) = freeVars e1 `S.union` freeVars e2
freeVars (Lam x e)   = S.delete x (freeVars e)

-- Generate a fresh variable name not in the set of used variables
freshVar :: S.Set String -> String -> String
freshVar used base = head $ filter (`S.notMember` used) candidates
  where candidates = [base ++ show i | i <- [1..]]

-- Substitute a variable in an expression
subst :: String -> Expr -> Expr -> Expr
subst var val expr@(Var x)
  | x == var  = val
  | otherwise = expr
subst var val (App e1 e2) =
  App (subst var val e1) (subst var val e2)
subst var val lam@(Lam x e)
  | x == var = lam  -- variable is bound, skip
  | x `S.member` freeVars val =
      let used = freeVars val `S.union` freeVars e
          x' = freshVar used x
          e' = subst x (Var x') e
      in Lam x' (subst var val e')
  | otherwise = Lam x (subst var val e)

-- One step of normal-order reduction
step :: Expr -> Maybe Expr
step (App (Lam x e1) e2) = Just (subst x e2 e1)
step (App e1 e2) =
  case step e1 of
    Just e1' -> Just (App e1' e2)
    Nothing ->
      case step e2 of
        Just e2' -> Just (App e1 e2')
        Nothing -> Nothing
step (Lam x e) = fmap (Lam x) (step e)
step _ = Nothing

-- Evaluate expression with a maximum step limit (returns only computed steps)
evalStepsLimited :: Int -> Expr -> [Expr]
evalStepsLimited maxSteps = go 0 []
  where
    go n acc e
      | n >= maxSteps = acc ++ [e]  -- return partial result up to maxSteps
      | otherwise = case step e of
          Just e' -> go (n + 1) (acc ++ [e]) e'
          Nothing -> acc ++ [e]     -- normal form reached

-- Evaluate fully using normal-order
eval :: Expr -> Expr
eval e = case step e of
  Just e' -> eval e'
```

```
  Nothing -> e

-- Evaluate a program with an environment
evalProgram :: Env -> [Statement] -> Either String ([Expr], Expr)
evalProgram env stmts =
  case splitAtAssignments stmts of
    (defs, ExprOnly e) ->
      let newEnv = foldl insertDef env defs
          resolved = resolveEnv newEnv e
          steps = evalStepsLimited 1000 resolved
      in Right (steps, last steps)
    _ -> Left "No expression to evaluate."
  where
    insertDef acc (Assign name expr) = Map.insert name expr acc
    insertDef acc _ = acc

    splitAtAssignments xs = (init xs, last xs)

-- ----------------------------

-- Show error dialog
errorDialog :: Window -> String -> String -> IO ()
errorDialog parent title message = do
  dialog <- messageDialogNew
            (Just parent)
            [DialogDestroyWithParent]
            MessageError
            ButtonsClose
            message
  set dialog [ windowTitle := title ]
  _ <- dialogRun dialog
  widgetDestroy dialog

-- Set input error highlight
setInputErrorHighlight :: TextView -> Bool -> IO ()
setInputErrorHighlight tv isError = do
  let color = if isError then Color 0xffff 0xcccc 0xcccc else Color 0xffff 0xffff 0xffff
  widgetModifyBase tv StateNormal color

-- Handle evaluation button click
onEval :: TextView -> TextView -> IORef [Record] -> IO ()
onEval lambdaInputView recordPreviewView recordRef = do
  buf <- textViewGetBuffer lambdaInputView
  (startIter, endIter) <- textBufferGetBounds buf
  input <- textBufferGetText buf startIter endIter True
  bufOut <- textViewGetBuffer recordPreviewView
  let trimmedInput = T.unpack $ T.strip $ T.pack input
  if null trimmedInput
    then do
      setInputErrorHighlight lambdaInputView True
      textBufferSetText bufOut ("Input is empty. Please enter a lambda expression." :: String)
    else
      case parseProgram trimmedInput of
        Left err -> do
          setInputErrorHighlight lambdaInputView True
```

```
      textBufferSetText bufOut ("Parse error:\n" ++ show err)
     Right stmts ->
      if null stmts
        then do
          setInputErrorHighlight lambdaInputView True
          textBufferSetText bufOut ("No expression to evaluate." :: String)
        else do
          setInputErrorHighlight lambdaInputView False
          records <- readIORef recordRef
          let dbEnv = buildEnvFromRecords records
          case evalProgram dbEnv stmts of
            Left msg -> textBufferSetText bufOut msg
            Right (steps, result) -> do
              let numberedSteps = zipWith (\i e -> show i ++ ". " ++ showExpr e) [0..] steps
                  output = unlines
                    [ "Evaluation Steps:"
                    , unlines numberedSteps
                    , ""
                    , "Final Result:"
                    , showExpr result
                    ]
              textBufferSetText bufOut output


-- ====================================================================== --

main :: IO ()
main = do
  _ <- initGUI
  window <- windowNew
  set window [windowTitle := ("Lambda GUI" :: String), windowDefaultWidth := 600, windowDefaultHeight := 600]

  mainVBox <- vBoxNew False 10
  set mainVBox [containerBorderWidth := 10]
  containerAdd window mainVBox

  ----------------------
  -- Section 1: Expression Evaluation
  ----------------------

  evalFrame <- frameNew
  frameSetLabel evalFrame ("Evaluate Lambda Expression" :: String)
  boxPackStart mainVBox evalFrame PackNatural 0

  evalVBox <- vBoxNew False 5
  containerAdd evalFrame evalVBox

  lambdaInputView <- textViewNew
  lambdaScrollWindow <- scrolledWindowNew Nothing Nothing
  scrolledWindowSetPolicy lambdaScrollWindow PolicyAutomatic PolicyAutomatic
  containerAdd lambdaScrollWindow lambdaInputView
  widgetSetSizeRequest lambdaInputView 400 100
  boxPackStart evalVBox lambdaScrollWindow PackGrow 0
  textViewSetWrapMode lambdaInputView WrapWordChar
```

```
-- Evaluate button and output label
evalButton <- buttonNewWithLabel ("Evaluate" :: String)
widgetSetTooltipText evalButton (Just ("Ctrl+Enter" :: String))
boxPackStart evalVBox evalButton PackNatural 0

outputLabel <- labelNew (Just ("Output will appear here" :: String))
boxPackStart evalVBox outputLabel PackNatural 0

-- Top-level frame for displaying records
outputScroll <- scrolledWindowNew Nothing Nothing
scrolledWindowSetPolicy outputScroll PolicyAutomatic PolicyAutomatic
boxPackStart mainVBox outputScroll PackGrow 5

recordPreviewView <- textViewNew
textViewSetEditable recordPreviewView False
containerAdd outputScroll recordPreviewView

----------------------
-- Section 2: Record Entry
----------------------

recordRef <- newIORef []

-- Toggle button to show/hide the entry form
toggleEntryButton <- buttonNewWithLabel ("Show Entry Form" :: String)
boxPackStart mainVBox toggleEntryButton PackNatural 0

-- Container for the entry form
entryContainer <- vBoxNew False 5
boxPackStart mainVBox entryContainer PackNatural 0

-- Frame for the entry form
entryFrame <- frameNew
frameSetLabel entryFrame ("New Record" :: String)
boxPackStart entryContainer entryFrame PackNatural 0

entryGrid <- tableNew 5 2 False
containerAdd entryFrame entryGrid

-- Create entries for each field
idModelEntry  <- entryNew
numEntry      <- entryNew
nameEntry     <- entryNew
txBodyEntry   <- entryNew
txCommEntry   <- entryNew

let addRow row labelText entry = do
    label <- labelNew (Just labelText)
    miscSetAlignment label 0 0.5
    tableAttachDefaults entryGrid label 0 1 row (row+1)
    tableAttachDefaults entryGrid entry 1 2 row (row+1)

addRow 0 ("Model ID" :: String) idModelEntry
addRow 1 ("Num / ID" :: String) numEntry
addRow 2 ("Name" :: String) nameEntry
```

```
addRow 3 ("Lambda Expression" :: String) txBodyEntry
addRow 4 ("Comment" :: String) txCommEntry

-- Save button for the entry form
saveDbButton <- buttonNewWithLabel ("Save to .db File" :: String)
--boxPackStart entryContainer saveDbButton PackNatural 0

-- Clear All Fields button
clearFieldsButton <- buttonNewWithLabel ("Clear All Fields" :: String)
--boxPackStart entryContainer clearFieldsButton PackNatural 0

entryButtonBox <- hBoxNew False 10
boxPackStart entryContainer entryButtonBox PackNatural 0

boxPackStart entryButtonBox saveDbButton PackGrow 0
boxPackStart entryButtonBox clearFieldsButton PackGrow 0

-- Connect the button to the clearing action
on clearFieldsButton buttonActivated $ do
  entrySetText idModelEntry ("" :: String)
  entrySetText numEntry ("" :: String)
  entrySetText nameEntry ("" :: String)
  entrySetText txBodyEntry ("" :: String)
  entrySetText txCommEntry ( "" :: String )

-- Toggle logic
visibleRef <- newIORef False
on toggleEntryButton buttonActivated $ do
  isVisible <- readIORef visibleRef
  if isVisible
    then do
      widgetHide entryContainer
      set toggleEntryButton [buttonLabel := ("Show Entry Form" :: String)]
      writeIORef visibleRef False
    else do
      widgetShowAll entryContainer
      set toggleEntryButton [buttonLabel := ("Hide Entry Form" :: String)]
      writeIORef visibleRef True

-- ----------------------
-- Section 3: Record Display
-- ----------------------

-- Toggle button to show/hide the DB records
toggleDbButton <- buttonNewWithLabel ("Show Database Records" :: String)
boxPackStart mainVBox toggleDbButton PackNatural 0

-- Container for the database section
dbContainer <- vBoxNew False 5
boxPackStart mainVBox dbContainer PackGrow 0

-- Frame to hold the treeView
displayFrame <- frameNew
frameSetLabel displayFrame ("Database Records" :: String)
boxPackStart dbContainer displayFrame PackGrow 10
```

```
-- Scrolled window for the treeView
recordScrollWindow <- scrolledWindowNew Nothing Nothing
containerAdd displayFrame recordScrollWindow

-- TreeView and ListStore setup
listStore <- listStoreNew ([] :: [Record])
treeView <- treeViewNewWithModel listStore
containerAdd recordScrollWindow treeView

-- Helper function to define and add columns
let addColumn title accessor = do
      col <- treeViewColumnNew
      renderer <- cellRendererTextNew
      treeViewColumnSetTitle col title
      cellLayoutPackStart col renderer True
      cellLayoutSetAttributes col renderer listStore $ \record ->
        [ cellText := accessor record ]
      treeViewAppendColumn treeView col

-- Add all columns to the TreeView
addColumn ("idModel" :: String) (T.pack . show . \(Record idModel _ _ _ _ _) -> idModel)
addColumn ("id" :: String)      (T.pack . show . \(Record _ id _ _ _ _) -> id)
addColumn ("num" :: String)     (T.pack . show . \(Record _ _ num _ _ _) -> num)
addColumn ("name" :: String)    (T.pack . \(Record _ _ _ name _ _) -> name)
addColumn ("txBody" :: String)  (T.pack . \(Record _ _ _ _ txBody _) -> txBody)
addColumn ("txComm" :: String)  (T.pack . \(Record _ _ _ _ _ txComm) -> txComm)

-- Button to load the database file
loadDbButton <- buttonNewWithLabel ("Load .db File" :: String)
boxPackStart dbContainer loadDbButton PackNatural 0

-- Toggle logic
dbVisibleRef <- newIORef False
on toggleDbButton buttonActivated $ do
  isVisible <- readIORef dbVisibleRef
  if isVisible
    then do
      widgetHide dbContainer
      set toggleDbButton [buttonLabel := ("Show Database Records" :: String)]
      writeIORef dbVisibleRef False
    else do
      widgetShowAll dbContainer
      set toggleDbButton [buttonLabel := ("Hide Database Records" :: String)]
      writeIORef dbVisibleRef True

----------------------
-- Section 4: Handlers
----------------------

-- Load button functionality
_ <- on loadDbButton buttonActivated $ do
  dialog <- fileChooserDialogNew
          (Just ("Open File" :: String))
          (Just window)
```

```
      FileChooserActionOpen
      [ ("Cancel", ResponseCancel), ("Open", ResponseAccept) ]
 widgetShow dialog
 response <- dialogRun dialog
 case response of
  ResponseAccept -> do
   mpath <- fileChooserGetFilename dialog
   case mpath of
    Just path -> do
     records <- loadDatabase path
     listStoreClear listStore
     mapM_ (listStoreAppend listStore) records
     writeIORef recordRef records
    Nothing -> return ()
  _ -> return ()
 widgetDestroy dialog

-- Save button functionality
_ <- on saveDbButton buttonActivated $ do
 -- Collect data from entries
 idModelStr <- entryGetText idModelEntry
 numStr    <- entryGetText numEntry
 name      <- entryGetText nameEntry
 txBody    <- entryGetText txBodyEntry
 txComm    <- entryGetText txCommEntry

 let parseInt str = maybe 0 id (readMaybe (T.unpack str) :: Maybe Int)
 let idModel = parseInt idModelStr
   num     = parseInt numStr
   id      = num
   record  = Record idModel id num name txBody txComm

 -- Open file chooser dialog to save
 dialog <- fileChooserDialogNew
       (Just ("Save to .db File" :: String))
       (Just window)
       FileChooserActionSave
       [ ("Cancel", ResponseCancel), ("Save", ResponseAccept) ]
 fileChooserSetDoOverwriteConfirmation dialog True
 widgetShow dialog
 response <- dialogRun dialog
 case response of
  ResponseAccept -> do
   mpath <- fileChooserGetFilename dialog
   case mpath of
    Just dbPath -> do
     result <- E.try (E.bracket (open dbPath) close $ \conn -> do
      execute_ conn $
       "CREATE TABLE IF NOT EXISTS eLambda (\
       \ idModel INTEGER,\
       \ id INTEGER,\
       \ num INTEGER,\
       \ name TEXT,\
       \ txBody TEXT,\
       \ txComm TEXT,\
```

```
      \ PRIMARY KEY (idModel, id))"
     execute conn
      "INSERT INTO eLambda (idModel, id, num, name, txBody, txComm) VALUES (?, ?, ?, ?, ?, ?)"
      record
     ) :: IO (Either E.SomeException ())

    case result of
     Left e  -> errorDialog window "Database Error" (show e)
     Right _ -> return ()
   Nothing -> return ()
  _ -> return ()
 widgetDestroy dialog

_ <- on evalButton buttonActivated $ onEval lambdaInputView recordPreviewView recordRef

-- Handle Ctrl+Enter in recordPreviewView to trigger the Evaluate button
_ <- on lambdaInputView keyPressEvent $ do
 key <- eventKeyName
 modifiers <- eventModifier
 liftIO $ when (key == "Return" && Control `elem` modifiers) $
  buttonClicked evalButton
 return False

_ <- on window objectDestroy mainQuit
widgetShowAll window
widgetHide entryContainer
widgetHide dbContainer
mainGUI
```