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

РЕАЛІЗАЦІЯ МОВИ SCHEME НА HASKELL

**Текстова частина до курсової роботи
за спеціальністю „Інженерія програмного забезпечення” 121**

Керівник курсової роботи
к.ф-м.н. Проценко В. С.

_____
(підпис)
“____” _____ 2020 р.

Виконала студент
Магур К. В.

“____” _____ 2020 р.

Київ 2020

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

ЗАТВЕРДЖУЮ
Зав. кафедри інформатики,
доц., к.ф.-м.н.
_____С. С. Гороховський
(підпис)
„_____”_____2019 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на курсову роботу

студенту факультету інформатики 4 курсу Магур Ксенії Володимирівні
ТЕМА : “Реалізація мови Scheme на Haskell”
Вихідні дані:

- інтерпретатор для мови Scheme
- приклад коду на Scheme

Зміст ТЧ до курсової роботи:

Анотація
Вступ
1. Scheme як діалект Lisp
2. Реалізація Scheme на Haskell
3. Застосування проекту та тестування
Висновки
Список літератури використаноъ

Дата видачі „___” _____ 2019 р. Керівник _____
(підпис)
Завдання отримав _____
(підпис)

Тема: “Реалізація мови Scheme на Haskell”

Календарний план виконання роботи:

№ п/п	Назва етапу дипломного проекту (роботи)	Термін виконання етапу	Примітка
1.	Отримання завдання на курсову роботу.	30.10.2019	
2.	Огляд технічної літератури за темою роботи.	10.01.2020	
3.	Реалізація практичної частини.	25.02.2020	
3.	Написання текстової частини за результатами роботи.	10.04.2020	
4.	Створення слайдів для доповіді.	19.04.2020	
5.	Оцінка курсової роботи викладачем.	24.04.2020	

Студент _____

Керівник _____

“_____” _____

ЗМІСТ

АНОТАЦІЯ

Метою цієї курсової роботи є реалізація на основі Haskell інтерпретатора для мови Scheme – одного з діалектів Lisp. За основу взято роботу Джонатана Танга "Write Yourself A Scheme in 48 Hours", яку проаналізовано, переписано на сучасний Haskell та доповнено виправленням помилок і розширенням реалізованої частини Scheme. Реалізація Scheme, наведена в рамках цієї курсової роботи, втілює ключові функції, описані в стандарті Scheme R5RS. Також реалізовано бібліотеку стандартних функцій на Scheme та додано модульні тести для окремих частин програми.

Ключові слова: Scheme, Lisp, R5RS, Haskell, функціональна мова, s-вираз, гомоїконічність, монада.

ВСТУП

Мови програмування з’являються і зникають. З появою нової мови завжди виникає питання, що стоятиме в основі її реалізації – компілятор, інтерпретатор чи поєднання обох. В межах цієї роботи зосереджено увагу на роботі інтерпретатора, а саме інтерпретатора для мови Scheme і його реалізації на Haskell.

Чому саме Scheme? Scheme як діалект Lisp є однією з перших інтерпретованих мов високого рівня. Ще однією перевагою Scheme є простота синтаксису : як і всі мови сімейства Lisp, вона використовує одні й ті ж синтаксичні конструкції для представлення коду і даних – особливий запис списків, названий s-виразами (про які буде йти мова в розділі 1.1). Також порівняно з іншими діалектами Lisp Scheme вирізняється найбільшою мінімалістичністю.

Чому саме Haskell? Haskell – класичний приклад чисто функціональної мови програмування з широким набором можливостей. Попри те, що Scheme притаманні деякі з можливостей процедурних мов, основна частина Scheme реалізує функціональну парадигму.

На реалізацію цієї роботи багато в чому вплинули роботи на цю тему Адама Веспісера “Write You A Scheme” та особливо Джонатана Танга – “Write Yourself a Scheme in 48 Hours”.

Робота складається з 3 основних розділів.

Перший розділ містить загальний опис Scheme : її властивостей як мови програмування, особливостей синтаксису, – а також визначення функціоналу Scheme, реалізованого в рамках цієї роботи.

У другому розділі описано реалізацію окремих частин програми, що відповідають за синтаксичний аналіз, обчислення, обробку помилок, роботу із середовищем, ввід/вивід.

Третій розділ присвячено використанню реалізованого інтерпретатора Scheme : наведено реалізацію стандартної бібліотеки функцій, яка може бути оброблена даним інтерпретатором. Також цей розділ звертає увагу на імплементацію автоматизованого тестування для ключових функцій програми.

Створено програмний продукт, який являє собою інтерпретатор мови Scheme, що забезпечує ключовий набір функцій, описаних в стандарті R5RS. Для підтримки продукту реалізовано автоматизоване модульне тестування. З використанням синтаксису Scheme, який підтримується наведеним в рамках роботи інтерпретатором, написано бібліотеку стандартних функцій Scheme.

1 SCHEME ЯК ДІАЛЕКТ LISP

1.1 Загальна характеристика

Scheme – один з основних діалектів Lisp, разом з такими мовами, як Common Lisp і Clojure.

Відмінна ознака усіх мов, які можна віднести до сімейства Lisp – це те, що усі дані і код в програмі представлені у вигляді так званих s-виразів (англ. s-expression). S-вираз – це спосіб запису вкладених списків даних. Такий список оточений дужками “()” і якщо він містить ідентифікатор функції або оператора з параметрами, то на першому місці у списку стоятиме ідентифікатор цієї функції/оператора, а за ним слідуватимуть його аргументи. Наприклад, додавання двох чисел, записане у вигляді s-виразу виглядатиме наступним чином : *(+ 1 2).*

Завдяки тому, що все в мові Lisp є s-виразами, їй притаманна така властивість, як гомоїконічність (англ. homoiconicity). Мова є гомоїконічною, якщо вона дозволяє маніпулювати кодом, написаним на ній так само, як даними цієї ж мови (“code as data”). Наприклад, в Lisp все є списками, тому програма, написана на Lisp, може використовувати інший код на Lisp як дані для параметрів своїх функцій або навіть створювати нові функції і змінювати їх без застосування якихось низькорівневих маніпуляцій. Наслідком такої специфіки є надзвичайна гнучкість Lisp : це дозволяє створювати програми, які маніпулюють іншими програмами, що часто називають метапрограмуванням (англ. metaprogramming).

Ще одна ознака Lisp, яка притаманна також і Scheme - строга динамічна типізація, тобто типи значень, використовуваних програмою,

визначаються на етапі виконання, та існують певні чіткі обмеження приведення одних типів до інших.

Scheme – це насамперед функціональна мова, однак вона підтримує також імперативну парадигму програмування.

Стандарти Scheme прописані в офіційному стандарті IEEE та де факто у стандарті RnRS (“Revised n Report on the Algorithmic Language Scheme”). Найбільш широко відомим є стандарт R5RS 1998 року. Саме на нього буде орієнтуватися імплементація, представлена в рамках цієї роботи.

Основна відмінність Scheme від інших діалектів Lisp – це її мінімалістичність. Scheme містить мінімум примітивних конструкцій і дозволяє виразити майже все, що завгодно, шляхом надбудови над ними.

Scheme був першим діалектом Lisp, який використовував виключно статичну область видимості змінних (англ. static scoping або lexical scoping), тобто для кожної змінної, визначеної в коді програми, завжди можна однозначно визначити її область видимості, прочитавши текст програми і не враховуючи стан стеку часу виконання, в контексті якого цей код буде викликано.

Також розробники Scheme в першому описі мови вперше ввели поняття стилю продовження-проходження (англ. continuation-passing style). Цей стиль передбачає, що функція приймає додатковий параметр – функцію від одного аргумента, як “явне продовження”(explicit continuation). Коли така функція завершує обчислення значення, вона повертає його шляхом виклику функції-продовження зі значенням обчисленого результату. Цей підхід дозволив елегантно ввести деякі можливості, притаманні імперативним і декларативним мовам, таким як ALGOL і Fortran, а також мовам Lisp з динамічною областю видимості

змінних, завдяки використанню лямбда-виразів не лише для звичайного визначення функцій, а і як “керуючі структури і модифікатори середовища” [7, 1]. Продовження (англ. continuations) – це по суті спосіб представлення стану програми в певний момент, яке можна зберегти і використати для переходу в цей стан. Продовження містять всю необхідну інформацію, щоб продовжити виконання програми з певної точки.

Продовження в Scheme є об’єктами першого класу (англ. first-class objects). Тобто вони можуть:

– бути параметрами функцій;
– бути повернуті як результат виконання функції;
– бути збережені в змінних;
– перевірятися на рівність.

Також продовження можуть бути використані для імітації поведінки виразу return з імперативних мов програмування. Стандарт Scheme визначає функцію *call-with-current-continuation*, яка приймає на вхід один аргумент-функцію. По суті ця функція робить так званий “знімок” (snapshot) у вигляді об’єкту поточного стану контексту управління і застосовує до нього функцію-параметр.

Ще однією особливістю Scheme є покращена оптимізація хвостової рекурсії (англ. tail recursion). Реалізації Scheme, що відповідають стандарту, гарантують підтримку необмеженої кількості активних рекурсивних викликів, що дозволяє безпечно писати ітеративні алгоритми, використовуючи рекурсивні структури. Scheme має ітеративну конструкцію *do*, однак використання рекурсії для вираження ітерації вважать більш ідіоматичних підходом.

У Scheme усі дані і процедури належать до спільного простору імен (англ. namespace). Тобто одночасно в програмі не може існувати функція і змінна з однаковими іменами. На противагу цьому, наприклад, Common Lisp визначає різні простори імен для змінних і функцій, що дозволяє змінній і функції мати однакові імена, однак вимагає використання спеціальної нотації для звернення до значення функції.

Scheme визначає досить повний набір числових типів даних, відомий як “числова вежа” (англ. numerical tower), що означає те, що математично числові типи Scheme можна скласти у так звану вежу підтипів, в якій кожен наступний рівень є підмножиною рівня, що знаходиться над ним:

- number;
  - complex;
  - real;
  - rational;
  - integer.

Більшість діалектів Lisp визначають порядок обчислення аргументів функцій. Однак стандарт Scheme це не визначає: вирази-аргументи функцій можуть бути обчислені в довільному порядку, залежно від реалізації Scheme, основне, щоб такий порядок був консистентним з певним лінійним порядком виконання.

У більшості діалектів Lisp значення *NIL* у булевих виразах обчислюється як false. У Scheme, починаючи із стандарту IEEE 1991 року, всі значення, крім *#f* (false), включаючи *'()* – еквівалент *NIL* в Scheme, – обчислюються як true. Слід зазначити, що в більшості мов Lisp true позначається як *T*, однак в Scheme це *#t*.

У Scheme примітивні типи не пересікаються, тобто для даного об’єкту тільки один з предикатів визначення типу, таких як *boolean?, pair?, symbol?, number?, char?, string?, vector?, port?, procedure?*, може повернути true. Що ж до числових типів даних, то тут є можливим перетин. Наприклад, ціле число (*integer*) задовільняє усі наступні предикати : *integer?, rational?, real?, complex?* та *number?* водночас.

Scheme визначає 3 різні види еквівалентності між довільними об'єктами:

- *eq?*. Повертає *#t*, якщо його параметри представляють один і той самий об'єкт у пам'яті;
- *eqv?*. Повертає *#t*, якщо *eq?* для даних параметрів повертає *#t*. Однак також оцінює примітивні типи за значенням і якщо вони співпадають, повертає *#t;*
- *equal?*. Повертає *#t*, якщо для даних параметрів *eq?* і *eqv?* повертає #t. Однак також порівнює такі типи, як списки, вектори і стрічки, за структурою і повертає *#t*, якщо для кожних відповідних елементів, що входять до їх складу, виконується *eqv?*.

Ввід та вивід в мові Scheme базується на використанні типу даних порт. Стандарт R5RS визначає два порти за замовчуванням, доступні за допомогою функцій *current-input-port* та *current-output-port*, які відповідають поняттям Unix стандартного вводу та виводу. Більшість імплементацій Scheme визначають також *current-error-port*. Також стандарт Scheme визначає функції перенаправлення вводу/виводу та читання/запис у файл.

У Scheme функції прив’язані до змінних. Стандарт R5RS дозволяє користувацькому коду перевизначати вбудовані функції Scheme, змінюючи їх прив’язку (англ. binding) до відповідних змінних.

За стандартами Scheme, функції, що конвертують значення з одного типу в інший містять у своєму імені "->". Предикати закінчуються на символ "?". Функції, що змінюють значення вже розміщених в пам’яті даних, в кінці імені містять символ "!". Цих конвенцій часто дотримуються і при написанні клієнтського коду.

1.2.Синтаксис Scheme

Як уже було зазначено, весь код в Scheme являє собою s-вирази. Вони записуються у вигляді списків, оточених дужками, елементи яких відокремлені пробілами. Наприклад, список цілих чисел від 1 до 3 виглядатиме наступним чином:

(1 2 3)

У первісному варіанті Lisp було два основних типи даних – атоми та списки. Атоми могли бути числами або символами. Основною ознакою атомів, на відміну від списків, була унікальність та незмінність. Однак зі збільшенням кількості типів даних в мовах Lisp (в тому числі і Scheme) цей термін втратив своє первісне значення і почав використовуватись для позначення усіх типів, що не є списками.

Списки – впорядковані скінченні послідовності, елементами яких можуть бути атоми або інші списки.

Списки в Lisp – це однозв’язні списки, кожна клітинка яких називається *cons* (в Scheme вона має назву pair). *Cons* містить вказівник на поточний елемент (*car*) і на решту списку (*cdr*).

Один і той самий список можна записати кількома способами за допомогою позначення “точкова пара” (англ. dotted-pair notation). Вона виглядає наступним чином:

(c1 . c2)

де *c1* – значення поля *car*, *c2* – значення поля *cdr*.

Наприклад:

(a b c d e) = (a . (b . (c . (d . (e . ()))))

Як було зазначено раніше, вирази записуються у вигляді списків з використанням префіксної нотації. Наприклад, виклик функції виглядатиме наступним чином:

(some-fun arg1 arg2)

де *some-fun* – назва функції; *arg1, arg2* – аргументи.

Аналогічно виглядатиме і застосування операторів:

(+ 1 2 3 4)

Для присвоєння значення змінній використовується функція *set!*. Наприклад :

(set! x 4)

Слід зазначити, що перед присвоєнням значення змінній, відбувається обчислення виразу. Тобто наступне присвоєння значення змінній *x* буде еквівалентним до попереднього:

(set! x (+ 2 2))

Якщо змінна не була визначена раніше, *set!* кидатиме помилку.

Подібну роль виконує конструкція *define*. Наприклад :

(define x 4)

Однак між виразами *set!* та *define* існує відмінність. Конструкція *define* додає визначення нової змінної в середовище. Якщо змінна була оголошена раніше, повторний *define* затіняє попереднє визначення змінної, роблячи її недоступною, що робить таку операцію подібною, однак не еквівалентною присвоєнню.

Прикладом умовних конструкцій в Scheme може бути оператор *if*. Загальний синтаксис виглядає так:

(if <test> <consequent> <alternate>)

If приймає три аргументи : якщо перший аргумент (<*test*>) обчислюється як *#f*, тоді обчислюється третій (<*alternate*>) і

повертається результат цього обчислення, інакше – другий (<*consequent*>). Наприклад:

```
(if (number? 2)
    "number"
    "not a number")
```

Результатом, що поверне така конструкція буде *"number"*.

Спеціальний оператор *lambda* використовується для створення нової анонімної функції. Значення такої функції можна використати для присвоєння змінній за допомогою *set!* або оголошення нової змінної через конструкцію *define*. Аргументами для *lambda* є список аргументів та вираз/вирази, які необхідно обчислити. Відповідно синтаксис виглядає наступним чином:

(lambda <formals> <body>),

де <*formals*> – список аргументів;

<*body*> – послідовність виразів.

Наприклад:

```
(lambda (arg) (+ arg 1))
```

Такий запис обчислюється як функція, яка у випадку її застосування, приймає один аргумент і, додавши до нього 1, повертає результат.

Визначення нових функцій можна створювати за допомогою оператора *define*. Синтаксис використання виглядає наступним чином:

(define <variable> <expression>)

або

(define (<variable> <formals>) <body>)

або

(define <variable>

(lambda (<formals>) <body>)),

де <*variable*> – назва функції;

<*formals*> – список аргументів;

<*expression*> – вираз;

<*body*> – список виразів.

Наприклад функцію додавання двох чисел можна визначити кількома способами, використовуючи наведені вище конструкції:

(define add +)

або

(define (add x y) (+ x y))

або

(define add (lambda (x y) (+ x y)))

Тоді додавання чисел 2 і 3 за допомогою функції *add* виглядатиме так:

(add 2 3)

Виклик функції поверне значення 5.

Також можливим є використання такого синтаксису:

(define (<variable> . <formal>) <body>)

або

(define <variable>

(lambda <formal> <body>))

Слід зауважити, що *<formal>* у такій конструкції – це одна змінна. Наприклад:

(define (add1 . x) (+ x x))

або

(define add2 (lambda x (+ x x)))

Спеціальний оператор *quote* (') використовується для створення списків, при цьому вказавши Scheme, що елементи цього списку не треба обчислювати. Окремі елементи такого списку можна зробити “unquoted”, поставивши перед ними кому (,). “Unquoted” елементи Scheme обчислить як під час створення звичайного списку. Наприклад:

(1 2 (+ 2 3) (+ 3 4)) => (1 2 5 7)

'(1 2 (+ 2 3) (+ 3 4)) => (1 2 (+ 2 3) (+ 3 4))

'(1 2 (+ 2 3) ,(+ 3 4)) => (1 2 (+ 2 3) 7)

Також *quote* може бути застосований до інших типів даних. В таком випадку він лише повертає значення цих об’єктів без їх обчислення. Тому *quote* може бути використаний для створення літералів. Наприклад:

'a => a

'() => ()

''a => (quote a)

Можливе також таке використання оператора *quote:*

(quote a) => a

1.3.Постановка задачі: яку частину Scheme буде реалізовано

Реалізація Scheme, наведена в рамках цієї роботи, базується на стандарті R5RS та роботі “Write You a Scheme in 48 Hours” Джонатана Танга. Незважаючи на простоту і лаконічність Scheme, порівняно з іншими мовами Lisp, стандарт R5RS містить досить багато визначень основних конструкцій Scheme, тому дана робота реалізує тільки основні моменти, визначені стандартом, при цьому дещо доповнюючи і розширюючи реалізацію, наведену Тангом.

Нижче наведено повний список реалізованих можливостей Scheme:

- основні типи.
    a) Number – для спрощення не реалізовано повну “числову вежу”, лише цілочисельні значення;
    b) Boolean;
    c) Character (наприклад, #\a, #\space);
    d) String;

e) List – включно з нотацією “точкова пара” або dotted-pair notation

f) Port – для вводу/виводу;

g) Function – включно з функціями вводу/вивиду;

- оператор *quote*.

  Зупиняє Scheme від обчислення елементів списку. Для спрощення не реалізовано знак коми (,) , який робить окремі елементи “unquoted”, про що було згадано в розділі 1.2;

- числові оператори.

  Стандартні операції над цілими числами: додавання (+), віднімання (-), ділення (/), ділення по модулю (*mod*), частка від ділення (*quotient*), остача від ділення (*remainder*);

- булеві оператори.

  Стандарті булеві операції “і” (&&) та “або” (||);

- оператори порівняння.

  Операції порівняння чисел : “дорівнює” (=), “більше” (>), “більше-рівне” (>=), “менше” (<), “менше-рівне” (<=), “не дорівнює” (/=).

  Операції порівняння стрічок: “дорівнює” (*string=?*), “більше” (*string>?*), “більше-рівне” (*string>=?*), “менше” (*string<?*), “менше-рівне” (*string<=?*);

  Рівність об’єктів: оператори *eq?*, *eqv?*, *equal?*, про які детально йдеться в розділі 1.1.

- оператори визначення типів.

  a) *string?* – повертає *#t*, якщо об’єкт є стрічкою (*String*), інакше повертає *#f*.

b) *char?* – повертає *#t*, якщо об’єкт є символом (*Character*), інакше *#f*.

c) number? – повертає *#t*, якщо об’єкт є числом, інакше *#f*.

d) *boolean?* – повертає *#t*, якщо об’єкт є *#t* або *#f* , повертає *#f* в іншому випадку;

- функції роботи зі списками.

  a) *car* – повертає значення поля *car* пари, по суті перший елемент списку;

  b) *cdr* – повертає значення поля *cdr* пари, по суті решту елементів або, іншими словами, – хвіст списку;

  c) *cons* – повертає пару, в якій *car* дорівнює першому аргументу функції, а *cdr* – другому;

- функції роботи зі стрічками.

  Окрім наведених вище функцій порівняння стрічок, реалізовано також:

  a) *string-length* – повертає довжину стрічки;

  b) *string-append* – повертає конкатенацію стрічок, переданих як параметри функції;

  c) *string* – об'єднує в стрічку символи, передані як параметри функції;

- функція *apply*.

  Застосовує вхідну функцію-параметр до списку аргументів.

- *if*

  Про умовний оператор *if* детально йдеться в розділі 1.2.

- *cond*.

  Загальний синтаксис виглядає так:

(cond <clause1> <clause2> ...),

де <*clause*> може набувати вигляду:

(<test> <expression1> ...)
(<test> => <expression>)
(else <expression1> <expression2> ...)

Послідовно оцінюються значення <*test*> кожного з <*clause*>, поки один з них не поверне значення true. Тоді послідовно обчислюється список виразів <*expression*> цього <*clause*> і результат обчислення останнього повертається як результат *cond.* Альтернативна форма з використанням => передбачає, що <*expression*> приймає один аргумент, а в якості параметра виступає значення, отримане внаслідок обчислення виразу <*test*> даного <*clause*>. Для спрощення, підтримка цього особливого випадку не реалізована в рамках цієї роботи. Якщо жоден <*test*> не повернув значення true, тоді виконується блок *else* і результат обчислення останнього виразу цього блоку повертається як результат *cond.*

- *case.*

Синтаксис виглядає наступним чином:

(case <key> <clause1> <clause2> ...),

де <*clause*> може набувати вигляду:

((<datum1> ...) <expression1> <expression2> ...)

(else <expression1> <expression2> ...)

Обчислюється значення виразу <*key*> і порівнюється з кожним елементом <*datum*> за допомогою оператора *eqv?*. Далі послідовно обчислюються вирази <*expression*> відповідного <*clause*> і результат обчислення останнього повертається як результат *case*. Якщо жодного співпадіння <*key*> і <*datum*> не знайдено, виконується блок *else* і результат обчислення останнього виразу повертається як результат *case*.

- *lambda*.

  Про *lambda* детально йдеться в розділі 1.2.

- Прив'язка змінних.

  Для прив'язки значень до змінних реалізовано обидва оператори *set!* і *define*, описані в розділі 1.2.

- визначення функцій.

  Реалізована підтримка усіх конструкцій синтаксису *define* для визначення функцій, описаних в розділі 1.2

- читання та запис у файл.

  a) *open-input-file* – приймає на вхід шлях до файлу і повертає порт, з якого можна здійснювати його зчитування;

  b) *open-output-file* – створює новий файл і повертає порт, з якого можна здійснювати запис у цей файл;

  c) close-input-port – закриває порт, з якого здійснювалось читання з файлу

  d) *close-output-port* – закриває порт, з якого здійснювався запис у файл;

e) *read* – приймає на вхід об’єкт і порт для читання, зчитує об’єкт з файлу, повертає наступний зчитаний об’єкт;

f) *write* – приймає на вхід об’єкт і порт для запису, записує об’єкт у файл;

g) *read-contents* – приймає на вхід шляд до файлу і зчитує його як в одну стрічку;

h) *read-all* – приймає на вхід шляд до файлу, зчитує з нього об’єкти і повертає їх як список;

- функція *load*.

  Приймає на вхід шлях до файлу, зчитує з нього вирази та визначення і обчислює їх.

- REPL.

  Також реалізовано цикл “читання-обчислення-виводу” (англ. read-eval-print loop), який дозволяє вводити команди Scheme з консолі, відразу отримуючи результат їх обчислення. Наприклад:

```
Lisp >>> (+ 1 2 3)
6
Lisp >>>
```

Рисунок 1.1 – Цикл “читання-обчислення-виводу”

2 РЕАЛІЗАЦІЯ SCHEME НА HASKELL

2.1 Загальний огляд та основні кроки реалізації

Щоб реалізувати власну мову програмування (у нашому випадку це Scheme), необхідно перш за все зробити лексичний аналіз тексту

програми, введеного користувачем. Тобто перетворити послідовність символів тексту на послідовність токенів певних типів. Далі – синтаксичний аналіз, який побудує дерево розбору. Наступний крок – певним чином його проаналізувати, і повернути користувачу результат. В цілому завдання виглядає досить складним, однак завдяки властивості гомоїконічності, яка притаманна, як Scheme, так і решті діалектів Lisp, задача значно спрощується. Нагадаємо, що гомоїконічність по суті означає те, що немає синтаксичної різниці між кодом і даними програми. У нашому випадку це дозволяє використовувати одні й ті самі структури для оперування кодом і даними, про що детальніше буде йти мова у наступних розділах.

Світ не ідеальний: завжди виникатимуть ситуації, обробка яких не передбачена можливостями програми. Наприклад, користувач може зробити синтаксичну помилку в коді або використовувати некоректні дані для виклику функцій. Тому також необхідно забезпечити механізм обробки помилок на всіх етапах проограми.

Haskell – чисто функціональна мова програмування. У Haskell функція не може змінити певний стан, такий як значення змінної. Так, наприклад, коли додаємо елемент до списку, по суті отримуємо новий список із доданим елементом. Оскільки ввід/вивід у Haskell пов'язаний із “виходом у зовнішній світ”, тобто зміною стану пристроїв вводу/виводу, він потребує окремої обробки, про що йтиме мова в наступних розділах.

Підсумовуючи усе сказане, наведемо діаграму, зображену на рисунку 2.1.

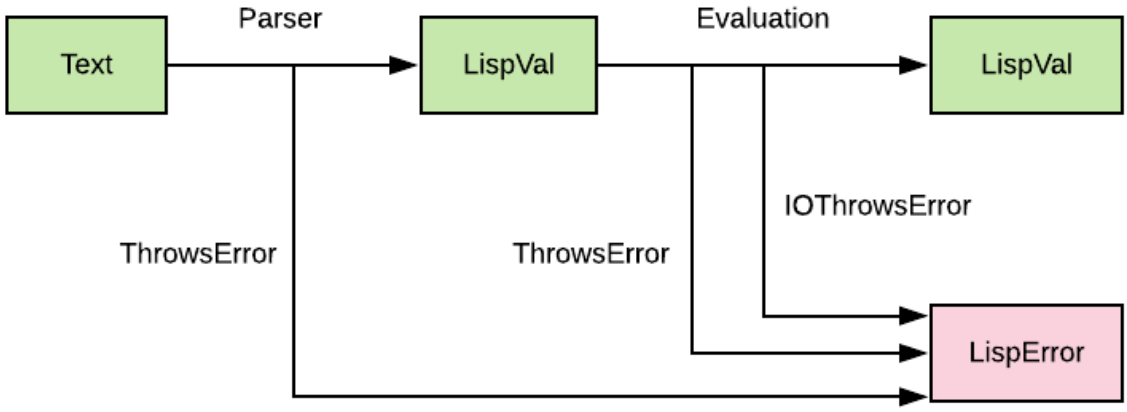

Рисунок 2.1 – Робота інтерпретатора Scheme

На рисунку 2.1 Text – текст коду на Scheme, *LispVal* – тип, який використано для репрезентації усіх даних і коду, *LispError* – тип для обробки помилок, Parser – частина програми, яка відповідає за лексичний і синтаксичний аналіз, Evaluation – обчислення вхідних даних і повернення результату, *ThrowsError* – тип для перехоплення помилок, не пов'язаних з вводом/виводом , *IOThrowsError* – тип для перехоплення помилок, що стосуються вводу/виводу.

2.2 Парсер

Для парсера використано бібліотеку Parsec.

Як уже було сказано, завдання парсера – перетворити текст в *LispVal.* Тому почнемо з визначення типу *LispVal* як зображено на лістингу 2.1.

Тобто *LispVal* може бути:

- *Atom* (по суті виконує роль ідентифікатора), який зберігає стрічку з назвою атома;

- *List*, який містить в собі список інших *LispVal;*
- *DottedList*, який відображає позначення “точкова пара” (dotted-pair). Містить список *LispVal* і останній елемент в окремій змінній;
- *Number*, який зберігає значення типу *Integer* – типу Haskell для цілого числа;
- *String*, який зберігає значення типу *String* – типу Haskell для стрічок;
- *Character*, який зберігає значення Haskell типу *Char;*
- *Bool*, який зберігає значення Haskell типу *Bool*.

```
data LispVal = Atom String
             | List [LispVal]
             | DottedList [LispVal] LispVal
             | Number Integer
             | String String
             | Character Char
             | Bool Bool
```

Лістинг 2.1 – Визначення типу *LispVal*

У наведеному *LispVal* немає конструкторів для функцій і виразів. Для парсера це не потрібно, оскільки весь код на Scheme являє собою s-вирази (розділ 1.1), які по суті є те ж саме, що й списки в Scheme.

Основною функцією парсера є *parseExpr*, зображена на лістингу 2.2.

```
parseExpr :: Parser LispVal
parseExpr = parseString
        <|> try parseBool
        <|> try parseChar
        <|> try parseNumber
        <|> try parseAtom
        <|> parseQuoted
        <|> do char '('
               x <- (Text.ParserCombinators.Parsec.try parseList) <|> parseDottedList
               char ')'
               return x
```

Лістинг 2.2 – Визначення функції *parseExpr*

З лістингу 2.2 зрозуміло, що *parseExpr* виконує розбір однієї з форм *LispVal*: *String*, *Bool*, *Char*, *Number*, *Atom*, *List*, *DotterdList*. Комбінатор <|> означає альтернативний вибір, тобто спробу застосувати по черзі одну з функцій (*parseString*, *parseBool*, і т.д.), поки одна з них не поверне бажаний результат. Звернімо увагу на ще одну функцію з бібліотеки Parsec – *try*. Вона застосовує функцію-парсер і у випадку помилки поводить себе так, ніби функцію не було застосовано. По суті вона є реалізацією механізму “підглядання наперед” (англ. look-ahead) синтаксичного аналізу. В даному випадку її застосування необхідне, оскільки деякі конструкції *LispVal* можуть починатися з однакових символів.

Звернімо увагу на *parseQuoted*, яка виконує розбір конструкції *quote* (детальніше в розділі 1.2). Для *quote* не передбачено окремого конструктора, оскільки ця форма лише являє собою *LispVal*, якому передує символ “ ' ”. Відповідно для неї використано конструктор *List*, першим елементом якого є *Atom “ quote ”* (див. лістинг 2.3).

```
parseQuoted :: Parser LispVal
parseQuoted = do
    char '\''
    x <- parseExpr
    return $ List [Atom "quote" , x]
```

Лістинг 2.3 – Визначення функції *parseQuoted*

Наведемо ще один приклад розбору *LispVal*, використаний у функції *parseExpr*. Наприклад, *parseAtom* (див. лістинг 2.4).

```
symbol :: Parser Char
symbol = oneOf "!$%&|*+-/:<=?>@^_~"

parseAtom :: Parser LispVal
parseAtom = do first <- letter <|> symbol
               rest <- many (letter <|> digit <|> symbol)
               let atom = [first] ++ rest
               return $ Atom atom
```

Лістинг 2.4 – Визначення функції *parseAtom*

На лістингу 2.4 використано такі функції бібліотеки Parsec: *letter*, що розпізнає одну будь-яку літеру латинського алфавіту, *digit* – одну будь-яку цифру, *many* – 0 або більше виразів, які визначаються функцією-параметром. З реалізації функції *parseAtom* зрозуміло, що атом – це літера або один з дозволених символів (що визначаються допоміжною функцією *symbol*), за яким слідує будь-яка кількість літер, цифр та символів.

Майже всі функції бібліотеки Parsec повертають значення типу *ParsecT s u m a*, який по суті є монадичним трансформером, або просто кажучи, “монадою, що інкапсулює іншу монаду”. На лістингах 2.1-2.4

наведено функції, що оперують типом *Parser LispVal,* який насправді є зручним синонімом *ParsecT* з певним набором параметрів:

type Parser = Parsec String ()
type Parsec s u = ParsecT s u Identity
data ParsecT s u m a

Насправді, щоб розібрати вхідний текст на *LispVal*, функції *readExpr* недостатньо : її необхідно використати як параметр до Parsec функції *parse,* яка повертає значення типу *Either ParseError a*, де *ParseError* – стандартний тип помилки бібліотки Parsec, *a* – тип результату (*LispVal*). Про застосування функції *parse* і обробку помилок парсера буде йти мова в розділі 2.3.

2.3 Обробка помилок

Для обробки помилок, визначено окремий тип – *LispError* (див. лістинг 2.5).

```
data LispError = NumArgs Integer [LispVal]
               | TypeMismatch String LispVal
               | Parser ParseError
               | BadSpecialForm String LispVal
               | NotFunction String String
               | UnboundVar String String
               | IOError SomeException
               | Default String
```

Лістинг 2.5 – Визначення типу *LispError*

LispError визначає декілька конструкторів для різних типів помилок:

- *NumArgs* – неправильна кількість аргументів;
- *TypeMismatch* – неправильний тип параметрів;
- *Parser* – помилки парсера;
- *BadSpecialForm* – неправильне використання визначених реалізацією спеціальних форм. Наприклад, відсутність в виразі *case* блоку, умова якого повертає *#t*;
- *NotFunction* – спроба використати в якості функції те, що не належить до типу функції;
- *UnboundVar* – відсутність у середовищі змінної або функції із зазначеною назвою;
- *IOError* – помилки вводу/виводу;
- *Default* – усі помилки, для яких не підходить визначення попередніх конструкторів.

Окрім типу помилок, необхідно також визначити механізм їх виникнення і перехоплення.

Один з варіантів забезпечення цього механізму в Haskell є клас монад *MonadError*:

class Monad m => MonadError e m | m -> e where ,

де e – тип помилки,

m – конструктор типу монади.

Мінімальним визначенням цього класу є дві функції *throwError*, яка використовується в рамках монадичних обчислень, щоб почати обробку

помилок, та *catchError*, яка виконує обробку помилок та повернення до нормального виконання, приймаючи в якості параметра функцію-обробник помилок типу *(e -> m a):*

throwError :: e -> m a
catchError :: m a -> (e -> m a) -> m a

Монада *Either* є екземпляром класу *MonadError*. Монада *Either*, параметризована типом *LispError*, використана для перехоплення звичайних помилок, які не пов'язані з вводом/виводом:

type ThrowsError = Either LispError

Як приклад такої обробки помилок наведемо обробку помилок парсера (див. лістинг 2.6).

```
readExpr = readOrThrow (do x <- parseExpr
                           eof <|> spacesEof
                           return x)
readExprList = readOrThrow (endBy parseExpr spaces)

readOrThrow :: Parser a -> String -> ThrowsError a
readOrThrow parser input = case parse parser "lisp" input of
    Left err -> throwError $ Parser err
    Right val -> return val
```

Лістинг 2.6 – Обробка помилок парсера

Як було зазначено в розділі 2.2 , функція *parse* повертає результат типу *Either ParseError LispVal*, тому допоміжна функція *readOrThrow* виконує обробку результату виклику функції *parse* таким чином, що у

випадку виникнення помилки *ParseError*, повертається *LispError*, побудований за допомогою передачі в конструктор *Parser* помилки, що виникла, і “запакований” в *ThrowsError*.

Перехоплення помилок вводу/виводу пов’язане з деякими труднощами, оскільки всі обчислення відбуваються в межах монади *IO*. До того ж, ввід/вивід в Haskell пов’язаний з іншою схемою обробки помилок, основою якої є перехоплення помилок типу *SomeException*. По суті усі помилки, які можуть виникати у функціях вводу/виводу, належать до цього типу.

Для обробки *IO* помилок використано монадичний трансформер *ExceptT*:

*newtype ExceptT e (m :: * -> *) a*,

де e – тип помилки,

m – внутрішня монада.

Конструктор *ExceptT* виглядає так:

ExceptT (m (Either e a))

Можна сказати, що єдине, що робить *ExceptT* – це додає обробку помилок в монаду, яка не є екземпляром класу *MonadError*.

ExceptT, як і *Either*, є екземпляром *MonadError*, тому також визначає функції *throwError* та *catchError*.

Для обробки помилок вводу/виводу визначено тип *IOThrowsError*, який є синонімом для *ExceptT*, параметризованим типом помилки *LispError* та монадою *IO*.

type IOThrowsError = ExceptT LispError IO

Як було зазначено вище, необхідно забезпечити спосіб перехоплення *SomeException.* Для цього визначено функцію *catchIOExceptionOrThrow* (див. лістинг 2.7).

```
catchIOExceptionOrThrow :: (a -> IO b) -> a -> IOThrowsError b
catchIOExceptionOrThrow action input = ExceptT (do result <- Control.Exception.try (action input)
                                                   case result of
                                                     Left err -> return $ Left (IOError err)
                                                     Right res -> return $ Right res)
```

Лістинг 2.7 – перехоплення *SomeException*

Як показано на лістингу 2.7, *catchIOExceptionOrThrow* використовує функцію *try* з модуля *Control.Exception*. Ця функція у випадку виникнення помилки всередині монади *IO,* повертає *IO (Left ex)*, де *ex* – помилка типу *SomeException,* яка виникла. Якщо жодної помилки не було перехоплено, повертає результат у вигляді *IO (Right a).* Функція *catchIOExceptionOrThrow* перехоплює *SomeException* і “запаковує” його в *LispError* і *IOThrowsError* або у випадку відсутності помилки аналогічно “запаковує” результат в *IOThrowsError,* повертаючи це як результат виконання.

Забігаючи наперед, наведемо приклад використання *catchIOExceptionOrThrow* у функції *load*, яка приймає на вхід шлях до файлу, зчитує і виконує розбір його вмісту на список *LispVal* за допомогою функції *readExprList* (див. лістинг 2.8).

```
load :: String -> IOThrowsError [LispVal]
load filename = (catchIOExceptionOrThrow readFile filename)
    >>= liftEither . readExprList
```

Лістинг 2.8 – приклад використання *catchIOExceptionOrThrow*

До цього моменту було визначено різні способи перехоплення і прокидування на наступний рівень помилок у форматі *LispError*. Однак необхідно також визначити місце їх реальної обробки. З цією метою визначено функцію *runIOThrows* (див. лістинг 2.9).

```
extractValue :: ThrowsError a -> a
extractValue (Right val) = val

trapError action = catchError action (return . show)

runIOThrows :: IOThrowsError String -> IO String
runIOThrows action = runExceptT (trapError action) >>= return . extractValue
```

Лістинг 2.9 – визначення функції *runIOThrows*

Функція *runIOThrows* приймає на вхід *IOThrowsError*, параметризований типом *String* і повертає *String* всередині монади *IO*, позбуваючись *ExceptT* за допомогою функції *runExceptT*:

runExceptT :: ExceptT e m a -> m (Either e a)

Обробка помилки відбувається за допомогою допоміжної функції *trapError*, яка використовує згадану вище функцію *catchError*. В якості функції-обробника помилок виступає композиція функцій *return* та *show*, що по суті означає, що у випадку помилки *trapError* поверне стрічкове представлення цієї помилки всередині *IOThrowsError*.

Зазираючи наперед, наведемо приклад використання *runIOThrows* у функції *evalString* (див. лістинг 2.10).

```
evalString :: Env -> String -> IO String
evalString env expr = runIOThrows $ liftM show $ (liftEither $ readExpr expr)
    >>= eval env
```

Лістинг 2.10 – приклад використання *runIOThrows*

Як показано на лістингу 2.10, *evalString* перетворює стрічку на *LispVal* за допомогою *readExpr*, обчислює одержаний *LispVal*, використовуючи *eval* (розділ 2.4), перетворює результат у стрічкове представлення і перехоплює всі помилки за допомогою *runIOThrows*.

2.4 Обчислення

Ключовим місцем обробки всіх *LispVal* є функція *eval* (див. лістинг 2.11).

```
eval :: Env -> LispVal -> IOThrowsError LispVal
eval env val@ (String _) = return val
eval env val@ (Character _) = return val
eval env val@ (Number _) = return val
eval env val@ (Bool _) = return val
eval env (List [Atom "quote" , val]) = return val
```

Лістинг 2.11 – Початок реалізації функції *eval*

Як показано на лістингу 2.11, цей розділ лише вводить поняття функції *eval* для обробки примітивних значень, повертаючи те саме

значення. В наступних розділах (2.5 і 2.6) відбувається доповнення *eval* для підтримки змінних та різних типів функцій.

2.5 Змінні та присвоювання

У Scheme можна зберігати результат обчислення виразу у змінних та доступатись до цього значення пізніше під час виконання програми. Також існує можливість присвоювати вже існуючим змінним нові значення.

Така особливість ускладнює реалізацію на Haskell, оскільки в Haskell модель виконання побудована на функціях, які лише повертають значення, але ніколи не змінюють його.

Однак існує кілька способів моделювання стану в Haskell, усі вони пов’язані з використанням монад. Мабуть, найпростіший – застосування монади *State*, яка дозволяє “приховати” довільний стан всередині монади і передавати його “непомітно” між функціями. Однак в даному випадку використання монади *State* не є доцільним, оскільки типи даних, які необхідно зберігати, є досить складними. Це пов’язано з реалізацією викликів функцій, де виникає необхідність мати справу зі стеком вкладених середовищ та змінними, які існують в середовищі незалежно від функцій, в яких вони використовуються.

Більш оптимальним рішенням, ніж монада *State,* є використання потоків стану (англ. state threads). Вони дозволяють працювати зі змінними, що можуть змінювати своє значення, як і в інших мовах програмування, використовуючи функції для отримання (get) та встановлення їх значення (set).

Одним з варіантів використання потоків стану є модуль *Data.IORef*. Він дозволяє використовувати змінні, значення яких може змінюватись, всередині монади *IO*. В рамках цієї роботи використано саме його, оскільки велика частина реалізації пов'язана з операціями всередині цієї монади.

Для середовища, що зберігає змінні, визначено тип *Env,* який являє собою *IORef,* що містить в собі список пар *String-IORef LispVal,* де перший елемент (*String*) – це назва змінної, а другий – її значення, яке може змінюватись:

type Env = IORef [(String, IORef LispVal)]

Також наведемо функцію *nullEnv* для створення порожнього середовища, яка використовує функцію *newIORef* з модуля *Data.IORef* , яка створює новий *IORef* всередині монади *IO* (див. лістинг 2.12)

```
nullEnv :: IO Env
nullEnv = newIORef []
```

Лістинг 2.12 – Створення порожнього середовища

Варто звернути увагу також на функцію перевірки, чи існує визначення даної змінної в середовищі, яка необхідна для коректної підтримки конструкції *define* (див. лістинг 2.13).

```
isBound :: Env -> String -> IO Bool
isBound envRef var = readIORef envRef >>=
    return . maybe False (const True) . lookup var
```

Лістинг 2.13 – перевірка існування змінної в середовищі

На лістингу 2.13 *isBound* зчитує фактичне значення середовище за допомогою *readIORef*, а потім здійснює пошук в ньому змінної із вказаною назвою, використовуючи функцію *lookup*.

Також визначено функції оголошення змінних, отримання їх значення та присвоєння їм нового значення.

На лістингу 2.14 функція *getVar* отримує значення змінної шляхом зчитування поточного значення середовища і пошуку в ньому змінної з вказаною назвою. Якщо змінну не знайдено, *getVar* кидає помилку *UnboundVar*.

```
getVar :: Env -> String -> IOThrowsError LispVal
getVar envRef var = do env <- liftIO $ readIORef envRef
                       maybe (throwError $ UnboundVar "Getting an unbound variable" var)
                           (liftIO . readIORef) (lookup var env)
```

Лістинг 2.14 – отримання значення змінної

На лістингу 2.15 функція *setVar* зчитує значення середовища, здійснює в ньому пошук змінної і за допомогою функції *writeIORef* записує в *IORef,* пов’язаним з нею, нове значення. У випадку якщо змінну не знайдено, *getVar* кидає помилку *UnboundVar*.

```
setVar :: Env -> String -> LispVal -> IOThrowsError LispVal
setVar envRef var value = do env <- liftIO $ readIORef envRef
                             maybe (throwError $ UnboundVar "Setting an unbound variable" var)
                                (liftIO . (flip writeIORef value)) (lookup var env)
                             return value
```

Лістинг 2.15 – Присвоєння значення змінній

На лістингу 2.16 *defineVar* за допомогою визначеної раніше функції *isBound* перевіряє існування змінної в середовищі. Якщо така змінна існує, використовує *setVar* для присвоєння їй нового значення, інакше – за допомогою *newIORef* створює нову змінну з вказаним значенням і записує її в середовище, використовуючи writeIORef.

```
defineVar :: Env -> String -> LispVal -> IOThrowsError LispVal
defineVar envRef var value = do
        alreadyDefined <- liftIO $ isBound envRef var
        if alreadyDefined
            then setVar envRef var value >> return value
            else liftIO $ do
                valueRef <- newIORef value
                env <- readIORef envRef
                writeIORef envRef ((var , valueRef) : env)
                return value
```

Лістинг 2.16 – Оголошення змінної

Під час виконання функцій виникає необхідність зберігати в середовищі кілька змінних одночасно, тому з цією метою визначено ще одну функцію для роботи з середовищем – *bindVars* (див. лістинг 2.17). Всередині *bindVars* оголошені допоміжні функції:

- *addBinding*, яка приймає на вхід пару із назви змінної та її значення, створює новий *IORef*, який утримує це значення, і повертає пару із назви змінної та створеного *IORef*;

- *extendEnv*, яка застосовує *addBinding* до кожного члена списку *bindings* за допомогою *mapM* та додає новостворений список змінних на початок поточного середовища *(++ env)*.

Сама ж функція *bindVars* зчитує середовище *envRef* , витягуючи його з *IORef*, додає в нього *bindings* за допомогою *extendEnv* і повертає нове середовище з доданими значеннями, використовуючи *newIORef*.

```
bindVars :: Env -> [(String, LispVal)] -> IO Env
bindVars envRef bindings = readIORef envRef >>=
    extendEnv bindings >>= newIORef
    where extendEnv bindings env = liftM (++ env) (mapM addBinding bindings)
          addBinding (var , value) = do ref <- newIORef value
                                        return (var , ref)
```

Лістинг 2.17 – Визначення функції *bindVars*

З додаванням функцій роботи з середовищем і його змінними, розширено можливості функції *eval* – додано підримку конструкцій *set!* та *define* для змінних (див. лістинг 2.18).

```
eval env (List [Atom "set!" , Atom var, form]) =
    eval env form >>= setVar env var
eval env (List [Atom "define" , Atom var , form]) =
    eval env form >>= defineVar env var
```

Лістинг 2.18 – Підтримка *set!* та *define*

2.6 Функції

Для підтримки функцій визначено кілька додаткових конструкторів для *LispVal* (див. лістинг 2.19):

- *PrimitiveFunc* – для зберігання примітивних функцій, визначених стандартом мови (+, *eqv?* і т. д.). Містить у собі функцію, яка приймає список *LispVal* як аргумент і повертає результат у вигляді *ThrowsError LispVal*;
- *IOFunc* – для зберігання примітивних функцій вводу/виводу, визначних стандартом мови (*open-input-file, close-output-port*). Інкапсулює функцію, яка приймає на вхід список *LispVal*, як і *PrimitiveFunc*, однак повертає результат у вигляді *IOThrowsError LispVal*, який місить всередині монаду *IO*;
- *Func* – для функцій, визначених користувачем. Зберігає всередині себе таку інформацію:
    a) params – назви параметрів функції;
    b) varargs – чи має функція змінну кількість аргументів, і якщо так, то містить назву змінної, до якої вони прив’язані;
    c) body – тіло функції, список виразів;
    d) closure – середовище, в якому створено функцію.

```
| PrimitiveFunc ([LispVal] -> ThrowsError LispVal)
| IOFunc ([LispVal] -> IOThrowsError LispVal)
| Func {params :: [String] , vararg :: (Maybe String),
        body :: [LispVal] , closure :: Env}
```

Лістинг 2.19 – Визначення конструкторів функцій

Виклик наведених вище функцій з реальними значеннями параметрів здійснює функція *apply* (див. лістинг 2.20).

Для конструкторів примітивних функцій *PrimitiveFunc* і *IOFunc* вона просто викликає функцію, яку вони інкапсулюють, з переданим набором параметрів.

Для *Func* функція *apply* перш за все перевіряє, чи співпадає кількість очікуваних аргументів функції та кількість реальних параметрів за допомогою локальної функції *num*. Якщо ні, кидає помилку *NumArgs*. Далі за допомогою *zip* об'єднує у список пар списки назв аргументів і значень реальних параметрів та, використовуючи його і середовище функції *closure*, створює нове середовище для обчислення результату за допомогою *bindVars*. За наявності varargs необхідно їх також прив'язати до середовища функції. Для цього визначено локальну функцію *bindVarArgs*. Якщо обчислювана функція не має жодних *varArgs* (*Nothing*), то *bindVarArgs* просто повертає те саме середовище. Інакше створює список з єдиним значенням – парою, ключем якої є назва змінної для *varArgs*, а значенням – список реальних параметрів, що залишились. Далі за допомогою *bindVars* створює нове середовище із доданим в нього *varArgs*. Параметри, що залишились, отримуються шляхом відкидання від вхідних параметрів перших n значень, де n – кількість звичайних аргументів функції, за допомогою функції *drop*. Останній крок в *apply* – це обчислення тіла функції у новому створеному середовищі. Для цього існує локальна функція *evalBody*. Вона застосовує функцію *eval* до кожного виразу в тілі функції і повертає результат обчислення останнього.

```
apply :: LispVal -> [LispVal] -> IOThrowsError LispVal
apply (PrimitiveFunc func) args = liftEither $ func args
apply (IOFunc func) args = func args
apply (Func params varargs body closure) args =
    if num params /= num args && varargs == Nothing
       then throwError $ NumArgs (num params) args
       else (liftIO $ bindVars closure $ zip params args) >>=
         bindVarArgs varargs >>= evalBody
       where remainingArgs = drop (length params) args
             num = toInteger . length
             evalBody env = liftM last $ mapM (eval env) body
             bindVarArgs arg env = case arg of
                Just argName -> liftIO $ bindVars env [(argName, List $ remainingArgs)]
                Nothing -> return env
apply someVal _ = throwError $ BadSpecialForm "Unrecognized special form" someVal
```

Лістинг 2.20 – Визначення функції *apply*

За допомогою функції *apply* та функцій роботи із середовищем (розділ 2.5), в *eval* додано підтримку конструкцій *lambda* (див. лістинг 2.21), *define* для оголошення функцій (див. лістинг 2.21) та виклику функцій з вказаними параметрами (див. лістинг 2.23).

```
eval env (List (Atom "define" : List (Atom var : params) : body)) =
    makeNormalFunc env params body >>= defineVar env var
eval env (List (Atom "define" : DottedList (Atom var : params) varargs : body)) =
    makeVarargs varargs env params body >>= defineVar env var
eval env (List (Atom "lambda" : List params : body)) =
    makeNormalFunc env params body
eval env (List (Atom "lambda" : DottedList params varargs : body)) =
    makeVarargs varargs env params body
eval env (List (Atom "lambda" : varargs@ (Atom _) : body)) =
    makeVarargs varargs env [] body
```

Лістинг 2.21 – Підтримка *lambda* та *define* для оголошення функцій

У рівняннях для підтримки *define* функція *eval* будує об’єкт *LispVal* функції з вхідних параметрів за допомогою однієї з допоміжних функцій

makeNormalFunc або *makeVarargs* та додає цей об’єкт в середовище як змінну з назвою нової функції за допомогою *defineVar*.

Як показано на лістингу 2.22 *makeNormalFunc* та *makeVarargs* лише будують відповідно звичайну функцію та функцію зі змінною кількістю аргументів, використовуючи допоміжну функцію *makFunc*. Вона будує об’єкт *LispVal* на основі вхідних параметрів за допомогою конструктора *Func*.

```
makeFunc varargs env params body = return $ Func (map show params) varargs body env
makeNormalFunc = makeFunc Nothing
makeVarargs = makeFunc . Just . show
```

Лістинг 2.22 – Допоміжні функції *makeNormalFunc* та *makeVarargs*

Виклик функції з вказаними параметрами показано на лістингу 2.23. Спершу обчислюється функція та її вхідні параметри за допомогою *eval*, після чого здійснюється виклик функції *apply*, яка застосовує функцію до вказаних аргументів.

```
eval env (List (function@ (Atom _) : args)) = do
    func <- eval env function
    argVals <- mapM (eval env) args
    apply func argVals
```

Лістинг 2.23 – Підтримка викликів функцій

2.7 Визначення основних примітивів Scheme

Для зберігання примітивних функцій визначено функцію *primitives*, яка повертає список пар, де ключем кожної пари є назва примітивної

функції, а значенням – Haskell функція типу *[LispVal] -> ThrowsError LispVal* (див. лістинг 2.24).

```haskell
primitives :: [(String , [LispVal] -> ThrowsError LispVal)]
primitives = [("+" , numericBinop (+)),
              ("-" , numericBinop (-)),
              ("*" , numericBinop (*)),
              ("/" , numericBinop div),
              ("mod" , numericBinop mod),
              ("quotient" , numericBinop quot),
              ("remainder" , numericBinop rem),
              ("=" , numBoolBinop (==)),
              ("<" , numBoolBinop (<)),
              (">" , numBoolBinop (>)),
              ("/=" , numBoolBinop (/=)),
              (">=" , numBoolBinop (>=)),
              ("<=" , numBoolBinop (<=)),
              ("&&" , boolBoolBinop (&&)),
              ("||" , boolBoolBinop (||)) ,
              ("string=?" , strBoolBinop (==)),
              ("string>?" , strBoolBinop (>)),
              ("string<?" , strBoolBinop (<)),
              ("string<=?" , strBoolBinop (<=)),
              ("string>=?" , strBoolBinop (>=)),
              ("string-length", strLength),
              ("string-append",strAppend),
              ("string", charsToString),
              ("car" , car),
              ("cdr" , cdr),
              ("cons" , cons),
              ("eq?" , eqv),
              ("eqv?" , eqv),
              ("equal?" , equal),
              ("string?", isString),
              ("char?", isChar),
              ("number?", isNumber),
              ("boolean?", isBool)]
```

Лістинг 2.24 – Примітивні функції

На лістингу 2.24 з метою уникнення дублікатів коду для реалізації подібних операторів для різних типів даних, використано функції-обгортки *numericBinop, numBoolBinop, boolBoolBinop, strBoolBino*p, які

приймають на вхід відповідний оператор/функцію Haskell та список аргументів *LispVal*, до яких цей оператор/функцію буде застосовано.

Реалізації функцій *numBoolBinop, boolBoolBinop, strBoolBinop* показано на лістингу 2.25. Вони необхідні для обгортки булевих операторів Haskell. Усі вони використовують допоміжну функцію *boolBinop*, яка приймає додатковий параметр *unpacker* – функцію, яка витягує значення з-під *LispVal*. Застосувавши *unpacker* до кожного з аргументів, *boolBinop* викликає відповідний булевий оператор Haskell і повертає це як результат *boolBinop*.

```
boolBinop :: (LispVal -> ThrowsError a) -> (a -> a -> Bool) -> [LispVal] -> ThrowsError LispVal
boolBinop unpacker op args = if length args /= 2
                             then throwError $ NumArgs 2 args
                             else do left <- unpacker $ args !! 0
                                     right <- unpacker $ args !! 1
                                     return $ Bool $ left `op` right

numBoolBinop = boolBinop unpackNum
strBoolBinop = boolBinop unpackStr
boolBoolBinop = boolBinop unpackBool
```

Лістинг 2.25 – Визначення функцій *numBoolBinop, boolBoolBinop, strBoolBinop*

Наведемо приклад функції *unpacker*. На лістингу 2.26 показано *unpacker* для символів (*Character*).

```
unpackChar:: LispVal -> ThrowsError Char
unpackChar (Character ch) = return ch
unpackChar notChar = throwError $ TypeMismatch "character" notChar
```

Лістинг 2.26 – Визначення функції unpackChar

Оскільки примітивні функції та примітивні функції вводу/виводу повинні зберігатися в середовищі під час запуску програми, для досягнення цієї мети визначено функцію *primitiveBindings* (див. лістинг 2.27). Вона визначає локальну функцію *makeFunc*, яка приймає на вхід конструктор *LispVal* та пару з назви змінної та функції, застосовує до функції відповідний конструктор, будуючи об'єкт *LispVal,* та повертає пару з назви змінної та побудованого *LispVal*. Сама ж функція *primitiveBindings* застосовує *makeFunc* до кожної пари, які повертають функції *primitives* (див. лістинг 2.24) та *ioPrimitives* (див. лістинг 2.28), об'єднує результуючі списки пар, та створює на основі цього результату та порожнього середовища нове середовище за допомогою *bindVars* (див. лістинг 2.17).

```
primitiveBindings :: IO Env
primitiveBindings = nullEnv >>= (flip bindVars $ map (makeFunc IOFunc) ioPrimitives
    ++ map (makeFunc PrimitiveFunc) primitives)
        where makeFunc constructor (var, func) = (var, constructor func)
```

Лістинг 2.27 – Визначення функції *primitiveBindings*

2.8 Визначення основних примітивів вводу/виводу

Для зберігання примітивів вводу/виводу визначено функцію *ioPrimitives*, яка повертає список пар, де ключем кожної пари є назва функції вводу/виводу, а значенням – Haskell функція типу *[LispVal] -> IOThrowsError LispVal* (див. лістинг 2.28).

```
ioPrimitives :: [( String, [LispVal] -> IOThrowsError LispVal)]
ioPrimitives = [("apply" , applyProc),
                ("open-input-file" , makePort ReadMode),
                ("open-output-file" , makePort WriteMode),
                ("close-input-port" , closePort),
                ("close-output-port" , closePort),
                ("read", readProc),
                ("write", writeProc),
                ("read-contents" , readContents),
                ("read-all" , readAll)]
```

Лістинг 2.28 – Примітивні функції вводу/виводу

Як було зазначено в розділі 2.7, *ioPrimitives* разом з *primitives* (див. лістинг 2.24) використовується у функції *primitiveBindings* для створення середовища, яке містить визначення усіх примітивних функцій (див. лістинг 2.27).

Наведемо один з прикладів примітивної функції вводу/виводу.

На лістингу 2.29 показано реалізацію функції *readContents*, яка приймає на вхід шлях до файлу, зчитує його і повертає *LispVal*, який містить стрічку зі зчитаним вмістом файлу. Функція *catchIOExceptionOrThrow* використовується для перехоплення помилок типу *SomeException*, які можуть виникати під час виконання операцій, пов’язаних з вводом/виводом (розділ 2.3).

```
readContents :: [LispVal] -> IOThrowsError LispVal
readContents [String filename] = fmap String (catchIOExceptionOrThrow readFile filename)
readContents [notString] = throwError (TypeMismatch "string" notString)
```

Лістинг 2.29 – Визначення функції

2.9 Цикл “Читання-обчислення-виводу” (REPL)

Цикл “Читання-обчислення-виводу” (англ. read-eval-print loop) передбачає реалізацію можливості для користувача введення виразів з консолі, результат обчислення яких відображається відразу після введення кожного окремого виразу. Для забезпечення такого функціоналу визначено функцію *runRepl* (див. лістинг 2.30).

```
runRepl :: IO ()
runRepl = primitiveBindings
    >>= until_ (== "quit") (readPrompt "Lisp >>> " ) . evalAndPrint
```

Лістинг 2.30 – Визначення функції *runRepl*

Функція *runRepl* використовує допоміжну функції *until_* для побудови циклу, який переривається, коли користувач вводить слово “*quit*”. У цьому циклі на кожній ітерації виводиться повідомлення “Lisp>>>” і зчитується з консолі значення за допомогою *readPrompt*, після чого виконується допоміжна функція *evalAndPrint*, яка обчислює введений вираз і виводить результат в консоль. Для отримання середовища, в якому відбуваються обчислення виразів використовується функція *primitiveBindings*, яка повертає середовище з усіма визначеними примітивними функціями.

На лістингу 2.31 показано реалізацію функції *until_*. Вона приймає на вхід предикат-умову закінчення циклу *pred*, дію, яка зчитує значення, *prompt*, та функцію, яку необхідно застосувати до зчитаного значення – *action*. Спершу зчитується значення за допомогою *prompt*, далі якщо умова предикату *pred* справджується, повертається порожня монада, що

припиняє виконання циклу. Інакше виконується *action* та рекурсивно викликається функція *until_*.

```
until_ :: Monad m => (a -> Bool) -> m a -> (a -> m ()) -> m ()
until_ pred prompt action = do
    result <- prompt
    if pred result
    then return ()
    else action result >> until_ pred prompt action
```

Лістинг 2.31 – Визначення функції *until_*

На лістингу 2.32 зображено реалізацію функції *evalAndPrint.* Для обчислення виразу *evalAndPrint* використовує допоміжну функцію *evalString*, яка перетворює вхідну стрічку на *LispVal* за допомогою *readExpr* та обчислює його значення через функцію *eval*.

```
evalAndPrint :: Env -> String -> IO ()
evalAndPrint env expr = evalString env expr >>= putStrLn

evalString :: Env -> String -> IO String
evalString env expr = runIOThrows $ liftM show $ (liftEither $ readExpr expr)
    >>= eval env
```

Лістинг 2.32 – Визначення функції *evalAndPrint*

На лістингу 2.33 показано реалізацію функції *readPrompt.* Вона виводить повідомлення за допомогою допоміжної функції *flushStr* і зчитує значення з консолі через стандартну функцію *getLine*. Функція *flushStr* після виведення повідомлення в консоль, відразу очищує потік за допомогою *hFlush*, примусово виводячи дані, які могли залишитись в буфері для виводу. Це необхідно для виключення ситуацій, коли

деякі повідомлення, залишившись в буфері, ніколи не були виведені в консоль.

```
flushStr :: String -> IO ()
flushStr str = putStr str >> hFlush stdout

readPrompt :: String -> IO String
readPrompt prompt = flushStr prompt >> getLine
```

Лістинг 2.33 – Визначення функції *readPrompt*

Не завжди існує необхідність у виконанні циклу “Читання-обчислення-виводу”. Повинна бути можливість також виконати код, прочитавши його з файлу. З цією метою визначено функцію *runOne* (див. лістинг 2.34).

Вона приймає на вхід список аргументів, додає усі аргументи, крім першого, як змінну з назвою “*args*” в середовище , що повертає функція *primitiveBindings*. Далі використовує одне з рівнянь функції *eval*, щоб зчитати і виконати код з файлу, шлях до якого – це перший аргумент списку, який не було додано в середовище.

```
runOne :: [String] -> IO ()
runOne args = do
    env <- primitiveBindings >>= flip bindVars [("args" , List $ map String $ drop 1 args)]
    (runIOThrows $ liftM show $ eval env (List [Atom "load" , String (args !! 0)]))
        >>= hPutStrLn stderr
```

Лістинг 2.34 – Визначення функції *runOne*

Рівняння *eval* для зчитування коду з файлу і його виконання наведено на лістингу 2.35. У ньому відбувається зчитування вмісту файлу і його перетворення в список *LispVal* за допомогою допоміжної

функції *load*. Далі до кожного елементу з отриманого списку застосовується функція *eval* і як результат виконання повертається результат обчислення останнього виразу зі списку *LispVal*.

```
eval env (List [Atom "load" , String filename]) =
    load filename >>= liftM last . mapM (eval env)
  . . .
load :: String -> IOThrowsError [LispVal]
load filename = (catchIOExceptionOrThrow readFile filename) >>= liftEither . readExprList
```

Лістинг 2.35 – Підтримка зчитування і виконання коду з файлу

Як результат, основна функція застосунку залежно від наявності вхідних аргументів або запускає цикл “Читання-обчислення-виводу” через функцію *runRepl*, або викликає *runOne*, передаючи в неї зчитані аргументи (див. лістинг 2.36).

```
main :: IO()
main = do args <- getArgs
          if null args then runRepl else runOne $ args
```

Лістинг 2.36 – Основна функція застосунку

2.10 Stack і поділ на модулі

Stack - це кросплатформний інструмент для розробки проектів на Haskell та управління їх залежностями. Stack має ряд переваг. Stack встановлює усі залежності проекту в ізольованому середовищі, що не дозволяє їм конфліктувати з іншими бібліотеками, встановленими на обладнанні користувача. Також він значно спрощує побудову проекту

та його тестування. Проект, реалізований в рамках цієї роботи, не має багато залежностей – це лише бібліотека *parsec*. Однак, з метою його доповнення реалізацією тестування використано stack.

Структура модулів проекту зображена на рисунку 2.2. Стрілками показано включення модулів одне в одного.

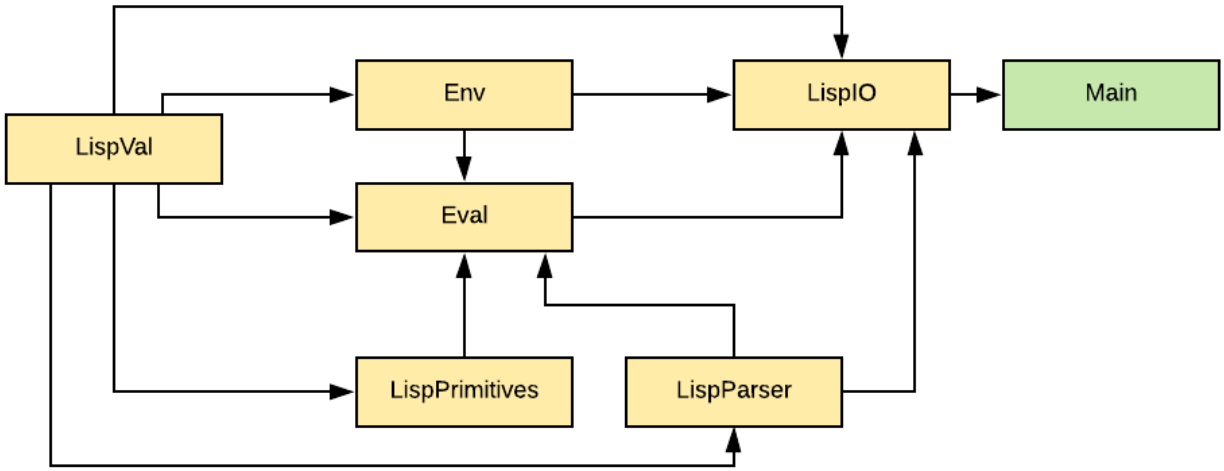

Рисунок 2.2 – Структура проекту

Проект поділено на такі модулі:

- *LispVal* – визначення типів *LispVal* та *LispError*, функції обробки помилок;
- *LispParser* – парсер, що перетворює вхідний текст на *LispVal;*
- *LispPrimitives* – визначення примітивних функцій;
- *Env* – визначення функцій роботи із середовищем та змінними;
- *Eval* – обчислення *LispVal,* примітивні функції вводу/виводу;
- *LispIO* – REPL, функція зчитування коду з файлу і його виконання;
- *Main* – модуль, з якого починається виконання програми.

3 ЗАСТОСУВАННЯ ПРОЕКТУ ТА ТЕСТУВАННЯ

3.1 Бібліотека на Scheme

Для зручності використання наведеної реалізації Scheme також імплементовано бібліотеку стандартних функцій, написану на ній.

Ключовими функціями цієї бібліотеки є рекурсивні функції *foldl, foldr* та *unfold*.

Функція *foldl* зображена на лістингу 3.1. Приймає на вхід список *lst*, початкове значення-накопичувач результату *accum* та функцію *func*, яку буде застосовано до *accum* та поточного елемента списку. Умовою зупинки рекурсії є перевірка чи список порожній за допомогою оператора *null?* (див. лістинг 3.2). *foldl* рекурсивно викликає саму себе, застосовуючи *func* до *accum* та поточного елементу списку, починаючи з першого, та передаючи отриманий результат як параметр *accum* в наступний рекурсивний виклик *foldl.*

```
(define (foldl func accum lst)
  (if (null? lst)
      accum
      (foldl func (func accum (car lst)) (cdr lst))))
```

Лістинг 3.1 – Визначення функції *foldl*

```
(define (null? obj)
    (if (eqv? obj '())
        #t
        #f))
```

Лістинг 3.2 – Визначення оператора *null?*

Функція *foldr* приймає такий самий список аргументів, що й *foldl* (див. лістинг 3.3). Так само, як і *foldl* , застосовує *func* до поточного елемента списку та *accum*, накопичуючи результат в *accum*, однак починає прохід по списку, починаючи з його кінця.

```
(define (foldr func accum lst)
  (if (null? lst)
      accum
      (func (car lst) (foldr func accum (cdr lst)))))
```

Лістинг 3.3 – Визначення функції *foldr*

Функція *unfold* зображена на лістингу 3.4. Вона приймає на вхід функцію від одного аргумента *func*, початкове значення *init* та унарний предикат *pred*.

Поки умова предиката *pred* не справдиться, вона будує список, починаючи від значення *init*, а для побудови кожного наступного елемента застосовує *func* до значення *init*, передаючи одержаний результат як параметр *init* в наступний рекурсивний виклик функції *unfold*.

```
(define (unfold func init pred)
  (if (pred init)
      (cons init '())
      (cons init (unfold func (func init) pred))))
```

Лістинг 3.4 – Визначення функції *unfold*

Наведені вище функції дозволяють спрощувати написання рекурсивних та ітеративних алгоритмів. Багато функцій з цієї ж

стандартної бібліотеки використовують функції *foldl* та *foldr* для реалізації операцій над списками.

Наприклад, функція знаходження довжини списку *length* використовує *foldl*, передаючи в якості аргумента *accum* число 0, а в ролі *func* – анонімну функцію від двох аргументів, що інкрементує перший аргумент на число 1 і повертає як результат. Таким чином, проходячи по списку, *foldl* на кожному наступному кроці додає до *accum* число 1, що в результаті дозволяє отримати кількість елементів цього списку (див. лістинг 3.5).

```
(define (length lst)
  (foldl (lambda (x y) (+ x 1)) 0 lst))
```

Лістинг 3.5 – Визначення функції *length*

Функція *map*, яка застосовує функцію-аргумент *func* до кожного елемента зі списку *lst*, використовує *foldr* (див. лістинг 3.6). В якості *accum* використовується порожній список. В якості параметра *func* до *foldr* потрапляє лямбда-функція, яка приймає на вхід два аргументи та будує список, де головою списку є результат застосування вхідної функції *map* до першого аргументу, а хвостом – другий аргумент. Таким чином *foldr* проходячи по списку і застосовуючи до кожного наступного аргументу вхідну функцію, додає його до списку, який накопичується в *accum*.

```
(define (map func lst)
  (foldr (lambda (x y) (cons (func x) y)) '() lst))
```

Лістинг 3.6 – Визначення функції *map*

3.2 Тестування

Для написання автоматичних тестів було використано фреймворк Hspec. Використання Stack, про який йшлося в розділі 2.10, значно спрощує інтеграцію тестування в Haskell проект.

Було обрано саме Hspec, оскільки він простий у використанні для типових випадків, однак містить широкий набір функцій для більш неочевидних ситуацій, може бути інтегрований з іншими фреймворками тестування для Haskell.

Ключовими частинами програми є модулі *Parser* (синтаксичний аналіз) та *Eval* (обчислення) та відповідно їх функції – *readExpr* і *eval.* Тому саме ці функції є об'єктом тестування за допомогою Hspec.

Для описання тестів в Hspec використовуються ключові слова *it* (описує кожен тест зокрема) та *describe* (групує кілька тестів). Наприклад, на лістингу 3.7 показано використання Hspec для тестування функції *readExpr*.

```
main :: IO ()
main = do
    hspec $ describe "Test suit for LipsParser" $ do
-- Test parser
      it "Test parse Atom" $
        readExpr "x&_" `shouldBe` (Right $ Atom "x&_")

      it "Test parse positive Number" $
        readExpr "3" `shouldBe` (Right $ Number 3)
```

Лістинг 3.7 – Тестування *readExpr* за допомогою Hspec

Для порівняння результату функції з очікуваним результатом використовується функція *shouldBe*, яка перевіряє на рівність ліву та

праву частини. Тому ліва та права частини повинні повертати дані, що належать до типу, який є екземпляром класу *Eq*.

В усіх реалізованих тестах відбувається порівняння з *LispVal* та *LispError*, тому відповідно вони є екземплярами класу *Eq*.

Тестування *eval* є дещо складнішим випадком, оскільки вона вимагає наявність середовища як вхідного параметра та повертає значення, що знаходиться всередині *IOThrowsError*. Тому для спрощення процесу написання тестів, визначено допоміжну функцію *runTestEval* (див. лістинг 3.8).

```
--extract value from lispVal evaluation
runTestEval::Maybe FilePath ->LispVal -> IO (ThrowsError LispVal)
runTestEval libFile lispExpr = runExceptT (testEval libFile lispExpr)

-- evaluate LispVal
testEval::Maybe FilePath -> LispVal -> IOThrowsError LispVal
testEval Nothing lispExpr = do env <- liftIO primitiveBindings
                               res <- eval env lispExpr
                               return res
testEval (Just libFile) lispExpr = do env <- liftIO primitiveBindings
                                      eval env (List [Atom "load" , String libFile])
                                      res <- eval env lispExpr
                                      return res
```

Лістинг 3.8 – Визначення функції *runTestEval*

Функція *runTestEval* приймає на вхід параметр *libFile*, який містить шлях до файлу бібліотеки всередині конструктора *Just*, інакше *Nothing*, та параметр *lispExpr – LispVal,* який необхідно обчислити. Використовує допоміжну функцію *testEval* для обчислення виразу *lispExpr*. *testEval* бере за основу середовище, що повертає функція *primitiveBindings*, за наявності файлу бібліотеки зчитує і обчислює його вирази, обчислює *lispExpr* та повертає результат обчислення. Функція *testEval* повертає

значення всередині *IOThrowsError (*що еквівалентно *ExceptT LispError IO)*. Для того, щоб позбутися обгортки з *ExceptT* , *runTestEval* використовує *runExceptT*.

Функція *runTestEval* повертає значення всередині монади *IO*. Для порівняння значення з монади *IO* з очікуваним результатом Hspec надає спеціальну функцію – *shouldReturn*. Саме її використано для усіх тестів, в яких використовується *runTestEval* (див. лістинг 3.9).

```
-- Test evaluation
    describe "Test suit for Eval" $ do
      it "Test eval addition" $
        runTestEval Nothing
          (List [Atom "+", Number 2, String "3"])
           `shouldReturn` (Right $ Number 5)

      it "Test eval string>?" $
        runTestEval Nothing
          (List [Atom "string>?", String "abc", String "aba"])
          `shouldReturn` (Right $ Bool True)
```

Лістинг 3.9 – Тестування *eval* за допомогою Hspec

Запуск тестів відбувається за допомогою таких команд Stack:

- *stack test* (звичайний запуск тестів);
- *stack build --test* (запуск тестів під час побудови проекту).

Приклад результату виконання тестів показано на рисунку 3.1.

```
Test suit for LipsParser
  Test parse Atom
  Test parse positive Number
  Test parse postive Number with sign
  Test parse negative Number with sign
  Test parse String
  Test parse Bool True
  Test parse Bool False
  Test parse List: simple
```

Рисунок 3.1 – Виконання тестів

3.3 Запуск та робота з проектом

Проект можна запустити за допомогою команди Stack:

stack exec haskell-scheme-exe

При першому запуску також необхідно попередньо виконати команди:

stack setup
stack build

Перша команда за необхідності завантажує компілятор в ізольованому середовищі, друга – будує проект. На рисунку 3.2 показано приклад використання деяких стандартних функцій у циклі REPL.

```
Lisp >>> (+ 2 3)
5
Lisp >>> (mod 12 7)
5
Lisp >>> (|| #f #t)
#t
Lisp >>> (&& #f #t)
#f
Lisp >>> (boolean? #f)
#t
Lisp >>> (char? #f)
#f
Lisp >>> (char? #\space)
#t
Lisp >>> (string-append "Hel" "lo," "Haskell")
"Hello,Haskell"
Lisp >>> (string>? "abc" "abcd")
#f
Lisp >>> (string<? "abc" "abcd")
#t
Lisp >>> (car (1 2 3 4))
1
Lisp >>> (cdr (1 2 3 4))
(2 3 4)
```

Рисунок 3.2 – Використання стандартних функцій

На рисунку 3.3 показано використання функцій з реалізованої бібліотеки на Scheme у циклі REPL.

Спершу за допомогою функції load завантажуються конструкції, визначені в файлі бібліотеки – *"./test/scheme/lib/basic_library.scm"*. Далі показано виклик деяких з цих функцій з конкретними параметрами.

```
Lisp >>> (load "./test/scheme/lib/basic_library.scm")
(lambda ("pred" . lst) ...)
Lisp >>> (filter even? '(1 2 3 4))
(2 4)
Lisp >>> (map (curry + 2) '(1 2 3 4))
(3 4 5 6)
Lisp >>> (sum 1 2 3 4 5)
15
Lisp >>> (product 1 2 3 4 5)
120
Lisp >>> (min 10 30 20)
10
Lisp >>> (max 10 30 20)
30
Lisp >>> (every? odd? 1 2 3 4)
#f
Lisp >>> (any? odd? 1 2 3 4)
#t
Lisp >>> (length '(1 2 3 4 4 4))
6
Lisp >>> (or #f #f #t)
#t
Lisp >>> (and #f #f #t)
#f
Lisp >>>
```

Рисунок 3.3 – Приклад роботи з бібліотекою на Scheme

Наведемо також приклад з використанням функції вводу/виводу. У файлі "*./test/scheme/lisp_example.scm*" визначено функцію *readDay*, яка приймає на вхід шлях до файлу, зчитує вміст файлу за допомогою *read-contents*, і повертає результат залежно від того, який день тижня було зчитано. Якщо у файлі записано назву робочого дня тижня, функція поверне результат "weekday", якщо назву вихідного – "weekend", інакше – "not a day" (див. лістинг 3.10).

```
(define (readDay filename)
    (case (read-contents filename)
      (("Monday" "Tuesday" "Wednesday" "Thursday" "Friday") "weekday")
      (("Saturday" "Sunday") "weekend")
      (else "not a day")))
```

Лістинг 3.10 – Визначення функції *readDay*

Для тестування роботи *readDay* використаємо 3 різні файли:

- "*./test/scheme/test_weekday.scm*" - містить стрічку "Tuesday"
- "*./test/scheme/test_weekend.scm*" - містить стрічку "Saturday"
- "*./test/scheme/test_notday.scm*" - містить стрічку "sdpfjelsfnsnf".

Як показано на рисунку 3.4, спочатку за допомогою функції load завантажується файл, що містить визначення функції *readDay*, далі здійснюється її виклик з використанням файлів, наведених вище, в якості параметрів.

```
Lisp >>> (load "./test/scheme/lisp_example.scm")
(lambda ("filename") ...)
Lisp >>> (readDay "./test/scheme/test_weekday.txt")
"weekday"
Lisp >>> (readDay "./test/scheme/test_weekend.txt")
"weekend"
Lisp >>> (readDay "./test/scheme/test_notday.txt")
"not a day"
```

Рисунок 3.4 – Виклик функції *readDay*

Для ілюстрації обробки некоректного вводу користувача, на рисунку 3.5 наведено кілька прикладів помилкового вводу та їх обробку застосунком.

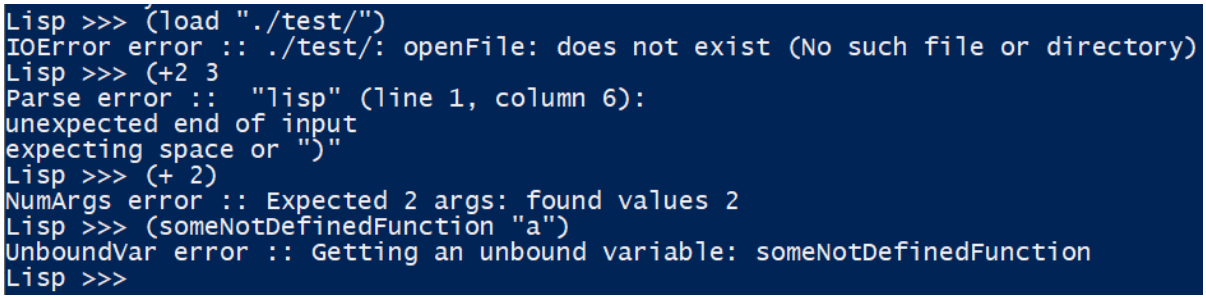

Рисунок 3.5 – Обробка некоректного вводу користувача

ВИСНОВКИ

В межах цієї курсової роботи було проаналізовано принцип роботи інтерпретатора для Scheme на основі робіт Джонатана Танга “Write Yourself a Scheme in 48 Hours”, Адама Веспісера “Write You A Scheme” та стандарту Scheme R5RS.

На основі Haskell було реалізовано ключову частину функцій зі стандарту R5RS мови Scheme, включаючи підтримку основних типів, визначень функцій, лямбда-виразів, оголошення та присвоєння змінних, область видимості змінних, операції на списками та примітивними типами, завантаження коду з файлу та цикл “Читання-обчислення-виводу” (REPL). Також написано на Scheme бібліотеку стандартних функцій, що сприймається даним інтерпретатором. Реалізовано автоматизоване модульне тестування.

В якості перспективи для покращення цієї роботи може бути покриття більшої частини функцій стандарту R5RS, можливо, також підтримка функціоналу, описаного в пізніших стандартах, таких як R6RS та R7RS. Наприклад, реалізація так званих “*let-bindings*” та роботи з продовженнями (англ. *continuations*) і функціями управління потоком виконання (наприклад, *call-with-current-continuation,* про яку було згадано в розділі 1.1). Також можливе розширення роботи в іншому напрямку – реалізації бібліотек функцій на Scheme, використовуючи можливості наведеного інтерпретатора.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Jonathan Tang. Write Yourself a Scheme in 48 Hours: An Introduction to Haskell through Example [Електронний ресурс]. – 2007. – 130 с. – Режим доступу: https://upload.wikimedia.org/wikipedia/commons/a/aa/Write_Yourself_a_Scheme_in_48_Hours.pdf.
2. Adam Wespiser. Write You A Scheme, Version 2 [Електронний ресурс]. – 2016.– Режим доступу: https://wespiser.com/writings/wyas/home.html.
3. Will Kurt. Get Programming with Haskell.– 2018. – 599 с.
4. Lisp (programming language). *Wikipedia, the free encyclopedia* [Електронний ресурс]. – 2020. – Режим доступу: https://en.wikipedia.org/wiki/Lisp_(programming_language).
5. John McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I [Електронний ресурс]. – Massachusetts Institute of Technology, Cambridge, Mass, 1960. – 34 с. – Режим доступу: https://web.archive.org/web/20131004232653/http:/www-formal.stanford.edu/jmc/recursive.pdf.
6. Revised 5 Report on the Algorithmic Language Scheme [Електронний ресурс] / [H. Abelson, N. I. Adams IV, D. H. Bartley and others]. – 1998. – Режим доступу: https://schemers.org/Documents/Standards/R5RS/
7. Guy Lewis Steele, Jr. Lambda: The Ultimate Declarative [Електронний ресурс]. – 1976. – 45 с. – Режим доступу: https://web.archive.org/web/20160315162926/http:/repository.readscheme.org/ftp/papers/ai-lab-pubs/AIM-379.pdf

8. Scheme (programming language). *Wikipedia, the free encyclopedia* [Електронний ресурс]. – 2020. – Режим доступу: https://en.wikipedia.org/wiki/Scheme_(programming_language).
9. call-with-current-continuation. *Wikipedia, the free encyclopedia* [Електронний ресурс]. – 2020. – Режим доступу: https://en.wikipedia.org/wiki/Call-with-current-continuation.
10. Continuation. *Wikipedia, the free encyclopedia* [Електронний ресурс]. – 2020. – Режим доступу: https://en.wikipedia.org/wiki/Continuation.
11. Static and Dynamic Scoping. *GeekForGeeks : A computer science portal for geeks* [Електронний ресурс]. – Режим доступу: https://www.geeksforgeeks.org/static-and-dynamic-scoping/.
12. Control.Monad.Except. *Hackage: The Haskell Package Repository* [Електронний ресурс]. – Режим доступу: http://hackage.haskell.org/package/mtl-2.2.2/docs/Control-Monad-Except.html.
13. Control.Exception. *Hackage: The Haskell Package Repository* [Електронний ресурс]. – Режим доступу: http://hackage.haskell.org/package/base-4.11.0.0/docs/Control-Exception.html.
14. Data.IORef. *Hackage: The Haskell Package Repository* [Електронний ресурс]. – Режим доступу: https://hackage.haskell.org/package/base-4.12.0.0/docs/Data-IORef.html.
15. User guide. *The Haskell Tool Stack* [Електронний ресурс]. – Режим доступу: https://docs.haskellstack.org/en/stable/GUIDE/.
16. User's Manual. *Hspec : A Testing Framework for Haskell* [Електронний ресурс]. – Режим доступу: http://hspec.github.io/.

17. Prelude. *Zvon* [Електронний ресурс]. – Режим доступу: http://zvon.org/other/haskell/Outputprelude/.