Міністерство освіти і науки України

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра інформатики

Кваліфікаційна робота

освітній ступінь – бакалавр

на тему «**РЕАЛІЗАЦІЯ ІНТЕРПРЕТАТОРА БАЗОВОЇ ЧАСТИНИ АДРЕСНОЇ МОВИ ДЛЯ ОБРОБКИ ДЕРЕВОПОДІБНИХ ФОРМАТІВ**»

Виконав: студент 4-го року навчання,

Спеціальності

121 Інженерія програмного забезпечення

Санченко Георгій Олександрович

Керівник доцент, к.ф.-м.н. Ющенко Ю. О.

Київ – 2024

Тема: “Реалізація інтерпретатора базової частини Адресної мови для обробки деревоподібних форматів”

Календарний план виконання роботи:

№ п/п	Назва етапу роботи	Термін виконання	Примітка
1	Отримання завдання на курсову роботу	26.10.2023	
2	Огляд технічної літератури за темою роботи	29.01.2024	
3	Реалізація практичної частини	20.03.2024	
4	Написання текстової частини	02.05.2024	
5	Створення слайдів для доповіді	17.05.2024	

Студент _____

Керівник _____

“_____”_____

ЗМІСТ

ВСТУП

У сучасному інформаційному світі мови програмування відіграють ключову роль у вирішенні різноманітних завдань. Однак, попри багатство сучасних мов програмування, варто звернутися до історичних коренів для виявлення цінних ідей та концепцій.

У 1950-60-х роках відбувалося народження ряду інновацій у світі програмування, серед яких значне місце займала Адресна мова програмування, запропонована Катериною Ющенко, та яку вона створила за участю Володимира Королюка. Ця мова вперше ввела поняття опосередкованої адресації вищих рангів та потужних операцій з адресами, зокрема багатократного застосування “штрих-операції”. Слід зауважити, що адресація 2-ого рангу згодом, у 1964р. Гарольдом Лоусоном була названа терміном Pointers, який з того часу дотепер використовується світовою спільнотою програмістів. Наразі багатократне застосування “штрих-операції” називають Multiple indirection of Pointers.

Вказівники, адресна арифметика, розіменування та багатократне розіменування вказівників відіграли важливу роль у зародженні та розвитку мов програмування високого рівня. Використання зазначених засобів сприяло суттєвому розширенню сфер застосування комп’ютерів з розрахункових задач до розв’язку логічних, інформаційних задач та задач штучного інтелекту. Розширення сфер застосування досягнуто за рахунок можливості за допомогою адресної арифметики та розіменування вказівників потрібним для розв’язку задач чином групувати дані та довільним чином визначати зв’язки між ними.

Метою дослідження є створення специфікації Адресної мови програмування на основі попередніх досліджень, реалізація інтерпретатора за даною специфікацією за допомогою мови програмування Haskell та розробка на Адресній мові прикладів програм, що створюють та опрацьовують спискові

ланцюжки та деревоподібні формати, які являють собою складні ієрархічні структури.

Основним джерелом дослідження є роботи Катерини Ющенко, зокрема монографія «Адресное программирование» [3], де детально описані синтаксис та принципи роботи Адресної мови програмування, а також наведено численні приклади програм цією мовою, включаючи обробку спискових ланцюжків та складних ієрархічних структур (деревоподібних форматів). Значною проблемою знайомства сучасних програмістів з Адресною мовою програмування, є те, що вона введена в 50-х роках.

Важливими для розуміння концепції Адресного програмування є статті [6, 7, 8], у яких надаються роз'яснення Адресного програмування з використанням загальноприйнятої термінології програмування, пояснено поняття області доступності адреси (вказівника) та розглянуто принципи роботи в Адресній мові зі списковими ланцюжками та деревоподібними форматами. Зокрема в статтях розглядаються обробка деревоподібних форматів, які відповідають сучасному розумінню поняття бінарних дерев. Також важливим джерелом є дослідження Каріни Чередник, яка запропонувала перекодування особливих та унікальних символів та позначень Адресної мови засобами стандартних текстових редакторів та реалізувала інтерпретатор Адресної мови на мові C++ [9, 10].

Існує лише декілька закордонних вчених, які визнають за Україною першість винайдення адресації 2-ого рангу (Pointers). При цьому за кордоном фактично нічого не знають про Адресну мову програмування та наявність в ній Pointers (адресації 2-ого рангу).

Актуальність даної роботи полягає у тому, що дуже важливо для України відновити історичну справедливість. Реалізація інтерпретатора основної частини Адресної мови програмування надасть можливість програмістам власноруч скласти і запустити Адресні програми з використанням адресації вищих рангів.

Приклади програм обробки спискових ланцюжків та інших складних ієрархічних структур являють собою наочне, додаткове та вагоме підтвердження наявності Pointers в Адресній мові програмування (1955р.) яка була запропонована на декілька років раніше закордонних мов програмування: FORTRAN (1958), LISP (1958), COBOL (1959) та ALGOL-60 (1960).

Основною метою цієї роботи є розробка інтерпретатора основної частини Адресної мови, який надає програмістам можливість ознайомитись з потужними засобами адресації Адресної мови програмування, включаючи опосередковану адресацію вищих рангів та використання двох відношень слідування для представлення складних ієрархічних структур - деревоподібних форматів. Відтепер програмісти мають можливість скласти, налагодити програми на Адресній мові з використанням спискових ланцюжків та деревоподібних форматів.

Реалізація інтерпретатора основної частини Адресної мови відкриває нові перспективи у сфері обробки деревоподібних форматів і забезпечить підґрунтя для подальших досліджень у цьому напрямку.

РОЗДІЛ 1. ОГЛЯД АДРЕСНОЇ МОВИ

1.1. Огляд основних засобів Адресної мови

Згідно з описом Катерини Ющенко, запис алгоритмів Адресною мовою заснований на традиційних математичних позначеннях. Проте Адресна мова вводить додаткові поняття, символи та позначення, необхідні для опису алгоритмів [2, стор.10].

Адресна програма складається з рядків. Кожен рядок може містити одну або декілька формул. Кожна формула являє собою опис дій та дуже нагадує сучасний термін “оператор”. Для забезпечення можливості посилання на рядок з інших частин Адресної програми він може позначатись мітками.

Окрім звичайних математичних виразів в Адресній мові вводиться вираз із використанням ***штрих-операції***, який записується як $'a$. Штрих-операція – це операція отримання значення в пам’яті за його адресою. Наприклад, у виразі:

$$'a = b$$

a позначає деяку умовну адресу, а b – значення, яке знаходиться у пам’яті за цією адресою. Подвійне застосування штрих-операції являє собою операцію розіменування вказівника.

Також для позначення повторного застосування штрих-операції вводиться позначення ^{n}a, де n – кількість повторних застосувань операції. Наприклад, вираз ^{2}a є еквівалентним виразу $''a$. Таке позначення називається багаторазовим розіменуванням вказівника (Multiple Indirection of Pointers), або багаторазовою штрих-операцією.

Крім того, в Адресній мові введено ***мінус штрих операцію*** та, відповідне поняття багаторазової ***мінус штрих операції***. Мінус штрих операції , протилежна до штрих-операції, яка дозволяє отримати множину адрес, за якими в пам’яті знаходиться певне значення. Багаторазова штрих-операція записується

таким чином: ^{-n}a або $_{n}a$. Якщо ***n*** має значення **1**, то результатом виразу буде множина адрес, які посилаються на адресу ***a***. Якщо ж ***n*** більше **1**, то операція повторюється **n** разів відносно до проміжних результатів.

1.2. Підпрограми та методи у якості складових частин деревоподібних форматів

Для демонстрації використання підпрограм у якості складових елементів деревоподібних форматів наведемо приклад реалізації Адресною мовою функцій *map* та *fmap* [11, з 32:39].

Функція *map* використовується для застосування певної трансформації до кожного елементу списку. В Адресній мові їй відповідає наступний запис:

Ц {a, CØ ⇨ π }
F('(π ⊕1))

Функція *fmap*, згідно з поясненням в [11], приймає на вхід список кортежів «дані + функція трансформації». В Адресній мові її можна записати таким чином:

Ц {a, CØ ⇨ π }
'(π ⊕1)('(π ⊕2))

Пояснення позначень у реалізації цих функцій адресною мовою наведено у таблиці 1.1.

Таблиця 1.1. Пояснення позначень

№	Позначення	Сенс	Сучасне трактування
1	***Ц***	позначка формули циклування	ключове слово заголовку циклу
2	***a***	ведуча адреса	адреса голови списку, список або його ім’я
3	***Ø***	закінчення списку	Null-Pointer

4	$\boldsymbol{\pi}$	змінна циклу	змінна циклу типу Pointer, яка “пробігає” всі вузли списку: від голови до його кінця, тобто: поки $\pi \neq Ø$
5	$\oplus$	арифметична дія з адресою: додавання цілого числа до адреси	операція адресної арифметики: додавання цілого числа до адреси; “плаваюча адресація” (“floating addressing”)
6	$'(\boldsymbol{\pi} \oplus 1)$	для map: значення даних у вузлі $\boldsymbol{\pi}$ списку; для fmap функція: адреса функції, яка міститься в якості першого елемента у вузлі $\boldsymbol{\pi}$ списку	для map: поточний елемент списку: [..., ***e***, …]; для fmap: функція (перший елемент поточного вузла списку) ***e1***: a = [..., [***e1***,_] ,…]
7	$'(\boldsymbol{\pi} \oplus 2)$	дані: значення даних у другому елементі поточного вузла списку	другий елемент поточного вузла списку ***e2***: a = [..., [_, ***e2***] ,…]
8	$'(\pi \oplus 1)('(\pi \oplus 2))$	виклик функції $'(\boldsymbol{\pi} \oplus 1)$ з параметром $'(\boldsymbol{\pi} \oplus 2)$	Застосування/Виклик функції ***e1*** до значення ***e2***: ***e1(e2)***

Задля спрощення у цих прикладах не надаються деталі повернення результатів застосування функцій. Зокрема, функція *F* може провертати значення за адресою аргументу ($\pi \oplus 1$), тобто: поелементно перетворювати значення елементів початкового спискового ланцюжка. Для *fmap* можна створювати відповідний новий списковий ланцюжок.

1.3. Відношення слідування адрес

В Адресному програмуванні, на відміну від концепції Абстрактних типів даних для групування даних та підпрограм використовується два відношення слідування на множині адрес комп'ютера: природнє слідування та слідування, яке визначається штрих-операцією (розіменуванням вказівника).

Розглянемо ці два відношення слідування на множині адрес.

1.3.1. Відношення природнього слідування адрес

Адреси комірок в оперативній пам'яті комп'ютерів мають цілочисельні значення.

Адреси сусідніх комірок пам'яті відрізняються одна від іншої на 1. Таким чином всі комірки оперативної пам'яті комп'ютера впорядковані відношенням сдідування, яке в Адресному програмуванні отримало назву природнє слідування адрес.

Природньому слідуванню адрес відповідає так звана «плаваюча адресація» («floating addressing»), яка використовувалась в програмах у машинних кодах в перших комп'ютерах фон-нейманівської архітектури для доступу до елементів масиву. Зауважимо, що «плаваюча адресація» являє собою опосередковану (непряму) адресацію 1-ого рангу.

Розглянемо детально використання плаваючої адресацію для визначення адреси розташування елементу масиву.

Нехай у нас є масив **A** з **n** елементів. Нехай **B** - база, початкова адреса, яка є адресою першого елемента масиву, який у програмування прийнято визначати номером "**0**".

O - це зсув або зміщення, яке вказує на різницю значення адреси певного, **i**-ого елементу масиву зі значенням адреси першого елементу масиву, тобто базою.

Значення зсуву визначається за формулою:

$$\mathbf{O = i * q,}$$

де **q** - кількість комірок пам’яті, яка потрібна для збереження одного елементу масиву. Всі елементи масиву мають один скалярний тип та займають однакову кількість комірок в оперативній пам’яті.

В Адресному програмування використовуються відповідні терміни: “фіксатор адреси”, “адреса зсуву” та “лічильник” (або “індекс”).

Аналогічним чином можна ввести операції слідування над елементами багатовимірних масивів. В [2, стор. 12, 13] наведено приклад визначення операцій слідування по рядкам та по стовбцям для квадратної матриці (масиву).

Адреса певного, **i**-ого (**i є [0, n-1]**) елементу масиву визначається за формулою:

$$\mathbf{A_i = B + O = B + i * q}$$

Для визначення адреси елементу лінійних структур мови програмування COBOL (1959р.) також використовується плаваюча адресація. Адреса довільного, певного елементу лінійної структури визначається за формулою:

$$\mathbf{A_i = B + Q,}$$

де **Q** - кількість комірок пам’яті які займають всі елементи структури, які передують певному.

Термін “плаваюча адресація” (“floating addressing”) було введено для визначення адрес елементів масиву при програмуванні у машинних кодах на комп’ютері WhirlWind I (1951) [1]. Оскільки киянам не були доступні закордонні джерела, то вони ввели та застосовувати термін “природнє” слідування адрес, а для арифметичних операцій над аресами використовували плюс, мінус, добуток та ділення в кружечках, наприклад: $\oplus$ та $\ominus$, $\otimes$.

1.3.2. Адресне відношення слідування

У даному підрозділі представлено опис Адресного відношення слідування, яке визначається штрих-операцією.

Як зазначено на стор. 28 в [3] “Операція слідування для елементів початкової інформації може встановлюватися за допомогою штрих-операції згідно операції слідування на множині адрес”.

1.4. Слідування на множинах значень даних

Окрім визначених вище операцій або відношень слідування над множинами адрес в Адресному програмуванні використовується поняття слідування (відношення слідування) над множинами даних. Прикладом операцію слідування на множині натуральних числах можна визначити як операцію додавання 1 [3, стор. 12].

РОЗДІЛ 2. СПЕЦИФІКАЦІЯ СИНТАКСИСУ

Оскільки Адресна мова програмування була розроблена покладаючись на математичні нотації з використанням не існуючих на клавіатурі позначок, то важливо адаптувати синтаксис цієї мови до сучасних стандартних текстових редакторів. У цьому розділі наведено модифікацію синтаксису Адресної мови програмування для забезпечення можливості запису Адресних програм сучасними текстовими редакторами.

Видозміну синтаксису проведено з урахуванням загальноприйнятої системи кодування символів ASCII: цифр, літер та спеціальних символів.

Запропонована адаптація синтаксису Адресної мови програмування до ASCII названо «ADPL» (від англ. «**AD**dress **P**rogramming **L**anguage»).

2.1. Загальні принципи синтаксису Адресної мови

В ADPL загальний принцип запису програми такий самий, як і в оригінальному описі Адресної мови. Програма записується як послідовність рядків, розділених символом переносу \n. Кожний рядок може мати одну або декілька міток. Проте ADPL вводить обмеження на припустимі назви міток: дозволяється використовувати лише символи латинського алфавіту, цифри та спеціальний символ _, при цьому назва має завжди починатися із літери.

Для оголошення міченого рядка в Адресній мові використовується такий запис: *мітка1 ... мітка2 ... <зміст рядка>*. Тобто після кожної назви мітки має бути доданий символ трикрапки. В ADPL вводиться додаткове правило: для оголошення мітки перед назвою необхідно додати символ @, наприклад: @label_1 … @label_2 … *<зміст рядка>*. Це правило необхідне, щоб краще відрізняти оголошення міток від посилання на них у програмі.

2.2. Припустимі вирази Адресної мови

Згідно із описом Катерини Ющенко, вирази, які припустимі в Адресній мові програмування, схожі до тих, що можна знайти у математичних нотаціях, зокрема числові константи, змінні, застосування унарних та бінарних операторів,

а також застосування математичних функцій. Крім того вона вводить такі власні операції, як штрих-операція, багаторазова штрих-операція та мінус-штрих операція.

Числові константи (зокрема цілі та десяткові числа) не потребують складних модифікацій синтаксису. Специфікація мови ADPL має сприймати числові константи у вигляді: *1*, *0.2*, *-5*. Тобто для десяткових чисел кома замінюється крапкою, як це прийнято записувати у сучасних мовах програмування.

Змінні в ADPL можна називати за тими ж правилами, як і мітки: перший символ має бути латинською літерою, решта символів – цифри, латинські літери або символ _.

Запис застосування бінарних операцій в ADPL також є подібним до сучасних мов програмування. Таким чином, символи +, −, ×, : заміняються на +, -, *, /. Крім того, вводяться додаткові бінарні оператори для роботи з вказівниками: ⊕, ⊖, ⊗, та “:” які в ADPL представляються як <+>, <->, <*>, <:> відповідно. Такий синтаксис мотивований подібністю запису до кружечка навколо оригінального оператора. Також цей запис подібний до мови програмування Haskell, де прийнято визначати оператори які оточені символами «<» та «>».

Для позначення логічних операцій вводяться ключові слова *and*, *or*, *not*, подібно до мови програмування Python.

Запис застосування математичних функцій в ADPL також заснований на сучасних мовах програмування. Тобто наприклад функція √ буде записана в ADPL як *sqrt*. Передача аргументів в ADPL записується подібно до синтаксису мови Haskell: *sqrt 1*, *min 1 2*.

В Адресній мові програмування вводиться штрих-операція, яка записується як *‘a*. В ADPL для цього використовується символ *'*, наприклад *'a*. Також Адресна мова вводить операцію багаторазового розіменування, яка

записується у вигляді ^{n}a. В ADPL для запису подібних виразів вводиться синтаксис: *`n`a*. Такий запис мотивований тим, що символи `` навколо n створюють візуальну асоціацію того, що *n* знаходиться вище, ніж *a*.

В Адресній мові також вводиться мінус-штрих операція, що позначається як $_{n}a$. В ADPL для запису подібних виразів вводиться синтаксис: *m`n`a*, де символ *m* додається на позначення слова «мінус».

Також в Адресній мові присутнє використання у якості виразу символа $\varnothing$. Цей символ використовується як «placeholder» для певного значення. В ADPL для запису такого виразу вводиться ключове слово *Nil*.

Також в ADPL додаються додаткові допоміжні вирази, які не згадуються в оригінальному описі мови. Наприклад, у статті про «Деревоподібні формати» згадується можливість передачі функції (підпрограми) за її адресою. Проте підпрограми ідентифікуються мітками, а мітки мають набір символів подібний до назв змінних. Тому, щоб не допустити конфліктів між змінними та мітками, для отримання адреси (відступу) певної мітки вводиться запис *&a*, де *a* – назва мітки. Це схоже на операцію отримання адреси змінної у C/C++.

Крім того, оскільки Адресна мова пропонує потужні можливості для роботи зі списками, в ADPL додається синтаксичний цукор для позначення списків: *[1, 2, 3]*. Такий запис є досить розповсюдженим у сучасних мовах програмування. Незважаючи на те, що таке позначення не згадується в описі оригінальної версії Адресної мови, цей синтаксис є більш лаконічним і вводиться заради зручності для розробника.

2.3. Припустимі формули Адресної мови

В Адресній мові наявні такі твердження, як формули присвоєння, засилання, обміну, переходу до мітки, зупину. А також складніші формули: предикатна формула, формула циклу, формула входження (або виклику) підпрограми, формула заміни.

В Адресній мові для позначення формули засилання використовується символ правої подвійної стрілки: $a \Rightarrow b$. В ADPL цей символ імітується за допомогою символів «дорівнює» та «більше»: *a => b*. Подібним чином спеціальний символ у формулі обміну $a \Leftrightarrow b$ замінюється на «менше-дорівнює-більше»: *a <=> b*.

Формула безумовного переходу не потребує додаткової видозміни в ADPL, оскільки її запис складається лише із назви мітки, до якої необхідно перейти. Інші формули переходу (обчислюваний перехід та відносний перехід) вирішено не включати в реалізацію ADPL.

Для запису формул абсолютного та відносного зупину в Адресній мові використовуються символи *!* та *Я* відповідно. Позначення абсолютного зупину не потребує додаткової модифікації. Проте для позначення відносного зупину в ADPL вводиться ключове слово *Ret*. Це позначення (від англійського «**Ret**urn» - «повернутися») вибрано через те, що операція відносного зупину використовується для виходу (повернення) із виклику підпрограми.

Для запису предикатної формули в ADPL вводиться позначення:

P { <condition> } <thenStatements> | <elseStatements>.

Тобто використання літери *P* (від англ. «**P**redicate») залишається таким самим, як і в початковому описі Адресної мови.

Для запису формули циклу в оригінальній версії Адресної мови використовується літера *Ц* (від слова «**Ц**икл»). Оскільки в ADPL припустимі лише латинські літери, літера *Ц* замінюється на *L* (від англ. «**L**oop» – «цикл»). Тобто загальна форма запису виглядає таким чином:

L { <start> (<step>) <end> => <counter> } <scopeLabel>, <nextLabel>.

В Адресній мові для запису формули входження підпрограми використовується літера *П* (від слова «**П**ідпрограма»). В ADPL літера *П*

замінюється на *Pg* (від англ. «Sub**P**ro**g**ram» – «підпрограма»). Загальний вигляд формули буде таким:

Pg <label>{ <arguments> }.

Також у разі передачі адреси мітки підпрограми через вираз припускається позначення *P [<expr>] { <arguments> }*, наприклад:

Pg [&label]{ a, b, c }.

Такий запис є розширенням над оригінальною версією Адресної мови, і є необхідним перш за все для спрощення передачі посилань на підпрограми через змінні.

В Адресній мові формула заміни – це інструмент для певного різновиду метапрограмування, який має схожість до макросів у C/C++. Ця формула записується у вигляді: $3\ \{ + \rightarrow -,\ a_1 \rightarrow c_1 \}\ a, b$. Тобто кожний крок заміни записується через символ стрілки →, а між собою кроки розділені комою. Літера *3* використовується як скорочення від слова «**З**аміна».

В ADPL літера *3* замінюється на латинську літеру *R* (від англ. «**R**eplacement» – «заміна»). Символ стрілки позначається як «->». Окрім того, припускається заміна не лише виразів та бінарних операцій, але і заміна цілих тверджень (формул). Оскільки формули можуть містити в собі символ коми, то для розділення кроків заміни в ADPL використовується крапка з комою «;». Наприклад:

R { + -> -; a1 -> c1; v => nv -> Pg foo { v, nv } } a, b.

Таким чином, формула заміни дозволяє замінити формулу засилання *v => nv* на формулу входження підпрограми *Pg foo { v, nv }*, що може бути корисним у розробці узагальнених алгоритмів.

РОЗДІЛ 3. СПЕЦИФІКАЦІЯ ІНТЕРПРЕТАЦІЇ

Окрім синтаксису важливою частиною специфікації є опис того, як ті чи інші позначення мають виконуватися інтерпретатором. Зокрема це стосується типів даних та роботи із пам’яттю.

3.1. Типи даних

Адресна мова є динамічно типізованою, і загалом працює із числовими даними. Проте для специфікації ADPL необхідно визначити більш конкретні правила роботи із типами даних.

Найпростішими типами даних є цілі та десяткові числа. В ADPL вони вводяться як різні типи для більш зручної роботи із числовими даними. Також тип ціле число використовується для представлення булевих даних.

Окрім скалярних даних у статті «Деревоподібні формати» згадується окремий тип вказівника. Цей тип вводиться для більш строгого розділу між даними які несуть якесь значення та даними, які служать адресами в програмі.

Також у статті згадується поняття «області доступності» вказівника або адреси. Це множина адрес, та значення, які в них містяться, та які доступні або «покриває» даний вказівник. Тобто тип даних вказівник має містити інформацію не тільки про саму адресу, на яку він посилається, але й всі адреси та значення, які в них містяться, до яких можна дістатись шляхом застосування допустимих операцій природнього та адресного слідувань адрес. Областю доступності адреси довільного елементу масиву є адреси та значення всіх елементи масиву. Областю доступності елементу спискового ланцюжка є об’єднання множин доступності всіх слідуючих за цим елементом елементів цього ланцюжкового списку та областей доступності всіх адрес, які входять у якості елементів у цей списковий ланцюжок.

Також для реалізації масивів необхідно зберігати інформацію про кількість елементів у масиві. Тобто вказівник має містити ще одне поле – кількість

елементів, які лежать у пам’яті починаючи з певної адреси. Це дає змогу чітко визначити операції над адресами <+>, які дозволяють ітеруватися над масивами різних типів (див. статтю про «Деревоподібні формати»).

3.2. Змінні та пам’ять

Згідно із оригінальним описом Адресної мови, усі змінні в програмі є константами, значення яких є адресами, визначеними окремо від тексту програми. Сама програма не модифікує значення змінних, а лише працює зі значеннями, на які ті вказують.

В ADPL вирішено зробити компроміс між оригінальним підходом та підходом, звичним для сучасних мов програмування. Тобто всі змінні в програмі автоматично ініціалізуються унікальним значенням певної вільної адреси. Це відповідає оригінальному опису Адресної мови. Водночас дозволяється пряме присвоєння значення змінній. Це зроблено для більшої зручності розробника, а також щоб не алокувати додаткову пам’ять.

3.3. Вбудовані функції

Адресна мова припускає визначення певних вбудованих функцій та процедур. Наприклад, в описі Адресної мови зустрічається процедура виводу значення («друк»).

Специфікація ADPL визначає додаткові функції для роботи із пам’яттю. Функція **alloc n** приймає параметром n-ну кількість клітинок пам’яті, які необхідно алокувати, і повертає адресу, за якою алоковано відповідну кількість клітинок. При цьому розмір створеного вказівника буде дорівнювати **n**, а кількість елементів буде рівна 1.

Для створення масивів визначена інша вбудована функція **mulalloc n k**. Вона приймає аргументами розмір та кількість елементів масиву, і повертає вказівник на пам’ять, де алоковано **n*k** клітинок пам’яті. При цьому розмір та кількість елементів вказівника будуть рівними **n** та **k** відповідно.

3.4. Область видимості міток

В оригінальному описі Адресної мови немає чіткого визначення області видимості міток. Інакше кажучи, усі мітки в програмі є глобальними. Проте це може бути проблематичним у випадку зростання розміру програми та кількості функцій (підпрограм) у ній, та може призвести до проблем із занадто довгими назвами міток.

Тому в ADPL запропоновано ввести поняття області видимості міток у межах підпрограм. Таким чином мітки, визначені в межах підпрограми, мають бути унікальними тільки в межах цієї підпрограми, а не всієї програми.

Це дуже спрощує задачу створення назв для міток, а також допоможе запобігти переходу із глобального контексту до мітки, визначеної всередині підпрограми.

РОЗДІЛ 4. РЕАЛІЗАЦІЯ ІНТЕРПРЕТАТОРА

4.1. Вибір архітектури інтерпретатора

Перш за все для реалізації інтерпретатора необхідно обрати архітектуру. Орієнтуючись на підручник «Crafting Interpreters» [5], є декілька опцій дизайну інтерпретатора. Перший варіант – це виконання коду шляхом обходу синтаксичного дерева. Другий – реалізація віртуальної машини із проміжним представленням (байткод). У цьому випадку необхідно реалізувати компіляцію із вихідного коду в набір інструкцій віртуальної машини, а також реалізувати виконання цих інструкцій самою машиною. Також необхідно обрати архітектуру самої ВМ – ВМ на основі стеку операндів або ВМ на основі регістрів.

Попередня реалізація Адресної мови К. Чередник використовує віртуальну машину на основі стеку операндів. У реалізації ADPL також вирішено використовувати подібну архітектуру. З одного боку, Haskell є добре пристосованим для реалізації інтерпретатора з прямим обходом синтаксичного дерева. Проте це має більше сенсу у випадку реалізації мов із функціональною парадигмою. Оскільки Адресна мова має багато елементів імперативного програмування (зокрема стрибки подібні до goto), віртуальна машина із байткодом краще підходить для цієї задачі.

Також існує варіант реалізації віртуальної машини на основі регістрів. Такі ВМ найчастіше більш ефективні ніж ВМ на основі стеку операндів. Проте вони також є складнішими у реалізації. Оскільки досягнення максимальної ефективності не входить в межі цього дослідження, то прийнято рішення продовжити роботу К. Чередник та зосередитись на розвитку ідеї ВМ зі стеком.

Також варто помітити, що в дослідженні К. Чередник реалізовано компілятор байткоду «в один обхід». Тобто програма в рамках одного циклу зчитує набір токенів і одразу трансформує його в байткод. Це має перевагу меншого використання пам’яті, оскільки в C++ представлення синтаксичного

дерева було б додатковою алокацією пам’яті, за якою треба слідкувати. Проте в Haskell представлення та робота із синтаксичними деревами є набагато більш природною, тому в реалізації ADPL використаний підхід компіляції в декілька обходів: набір токенів трансформується в синтаксичне дерево, а вже потім це дерево компілюється в байткод.

4.2. Опис прийому реалізації адресації вищих рангів

У цьому підрозділі розглянемо метод моделювання стану пам’яті для реалізації адресації вищих рангів. В Адресній мові стан пам’яті складається із двох основних компонент: адреси та їх вміст (значення). Це відповідає поняттю адресного відображення. Проте крім цього кожній адресі може відповідати певна назва змінної. Таким чином, стан пам’яті програми на Адресній мові можна представити у вигляді таблиці з 3-х стовпчиків (таблиця 4.1).

Таблиця 4.1. Моделювання стану пам’яті.

Назва змінної	Адреса	Значення
A	100	2.5
H	900	1000
	1000	1100
	1001	1
	1100	1200
	1101	2
	1200	0
	1201	3

Розглянемо приклад інтерпретації виразу з використанням штрих-операції *'("H + 1)* на основі наведеної таблиці. Спочатку необхідно взяти значення, яке у таблиці відповідає змінній *H* - це число 1000. Далі потрібно до цього числа двічі застосувати штрих-операцію. Одноразове застосування штрих-операції до

певної адреси *A* означає знаходження рядка, де число у стовпчику “Адреса” дорівнює *A* і взяття із цього рядка числа у стовпчику “Значення”. Отже для даного виразу це буде послідовність 1000 -> 1100 -> 1200. Далі до отриманого результату додається 1 (1200 + 1 = 1201) і знову застосовується штрих-операція. У таблиці за адресою 1201 знаходиться значення 3, що і буде кінцевим результатом.

4.3. Реалізація лексичного аналізатора

Для реалізації лексичного аналізатора використано бібліотеку alex. Це бібліотека для Haskell, заснована на lex для C/C++. Вона пропонує власний синтаксис для визначення правил лексичного аналізу, а також дозволяє трансформувати токени в їх відповідне представлення у Haskell.

Для початку визначаємо набір токенів як структуру даних у Haskell.

```
data TokenType =
    TNewLine | TAnd | TBar | TSemi | TComma
    | TPlus | TCirclePlus | TMinus | TTimes | TDiv | TMod
    | TLPar | TRPar | TLCurly | TRCurly | TLBrack | TRBrack
    | TStop | TAssign | TEq | TNeq | TGt | TGe | TLt | TLe
    | TSQuote | TBackTick | TMBackTick | TSArrow | TDArrow | TExchange
    | TLabel String | TIdentifier String | TKeyword String
    | TProc String | TFn String | TInt Int | TFloat Double
    | TError
  deriving (Show, Eq)

data Token  = Token {
  tokenPos :: AlexPosn,
  tokenStr :: String,
  tokenType :: TokenType
} deriving (Eq, Show)
```

Лістинг 4.1. Визначення типу для представлення токенів

Далі необхідно визначити правила для зчитування кожного типу токена. Спочатку визначаються патерни на основі регулярних виразів, зокрема для літер, цифр, пробілів, символу нового рядка, а також для ідентифікаторів та ключових слів.

```
$alpha    = [a-zA-Z]
$digit    = 0-9
$white_no_nl = [\ \t]
$lf = \n
$cr = \r
@eol_pattern = $lf | $cr $lf | $cr $lf
@identifier = $alpha ($alpha | $digit | _)*
@decimal = $digit+
@kw = (P|L|R|Pg|Nil|Ret|Cj|not|and|or)
@builtinProc = (print|printList|printRefs)
@builtinFn = (alloc|ptr|id|mulalloc)
```

Лістинг 4.2. Визначення патернів для розпізнавання послідовностей символів

Наступним кроком визначаються правила, які ігнорують вхідний текст, зокрема це пробіли та коментарі. Далі визначено правила для зчитування знаків пунктуації. При цьому символ переносу вважається окремим токеном, оскільки він визначає розподіл на рядки, які є важливим елементом синтаксису Адресної мови. Наступним кроком є визначення правил для розпізнавання бінарних та унарних операторів, а також спеціальних позначень різних формул Адресної мови.

```
tokens :-
  $white_no_nl+ ;
  "//".*        ;
  @eol_pattern      { tok' TNewLine }
  "("               { tok' TLPar }
  ")"               { tok' TRPar }
  "{"               { tok' TLCurly }
  "}"               { tok' TRCurly }
  "["               { tok' TLBrack }
  "]"               { tok' TRBrack }
  "|"               { tok' TBar }
  ";"               { tok' TSemi }
  ","               { tok' TComma }
  "+"               { tok' TPlus }
  "<+>"             { tok' TCirclePlus }
  "-"               { tok' TMinus }
  "*"               { tok' TTimes }
  "/"               { tok' TDiv }
  "%"               { tok' TMod }
  "=="              { tok' TEq }
  "/="              { tok' TNeq }
  ">"               { tok' TGt }
  ">="              { tok' TGe }
  "<"               { tok' TLt }
  "<="              { tok' TLe }
  "'"               { tok' TSQuote }
  "`"               { tok' TBackTick }
  "m`"              { tok' TMBackTick }
  "&"               { tok' TAnd }
  "->"              { tok' TSArrow }
  "=>"              { tok' TDArrow }
  "<=>"             { tok' TExchange }
  "!"               { tok' TStop }
  "="               { tok' TAssign }
```

Лістинг 4.3. Визначення правил для розпізнавання спецальних позначень

Далі визначено правила для зчитування міток рядків, назв змінних, а також цілочисельних та десяткових констант. При цьому оголошення мітки сприймається як один цілісний токен, що спрощує синтаксичний аналіз на наступному етапі.

```
"@" @identifier $white_no_nl* "..." { tok (\s -> TLabel (getLabelName s)) }
@kw                   { tok (\s -> TKeyword s) }
@builtinProc          { tok (\s -> TProc s) }
@builtinFn            { tok (\s -> TFn s) }
@identifier                 { tok (\s -> TIdentifier s) }
@decimal              { tok (\s -> TInt (read s)) }
@decimal \. @decimal  { tok (\s -> TFloat (read s)) }
_                     { tok' TError }
```

Лістинг 4.4. Визначення правил для розпізнавання ключових слів, ідентифікаторів тощо

Нарешті, оголошується функція *scanTokens*, яка приймає параметром вхідний код та трансформує його у список токенів. Для визначення цієї функції використовується функція із бібліотеки alex «*alexScanTokens*».

```
scanTokens :: String -> [Token]
scanTokens = alexScanTokens
```

Лістинг 4.5. Визначення функції scanTokens

На цьому реалізація лексичного аналізу Адресної мови завершена, і лексичний аналізатор готовий для подальшого використання в синтаксичному аналізі.

4.3. Реалізація синтаксичного аналізатора

Для синтаксичного аналізу вирішено використати бібліотеку happy. Ця бібліотека часто використовується разом із alex і заснована на бібліотеці yacc для C/C++. Вона дозволяє декларативно описати граматику мови разом із визначенням структури синтаксичного дерева на Haskell.

Отже перш за все необхідно визначити типи даних, які будуть представляти відповідні вузли синтаксичного дерева. Зокрема визначено типи

даних *UnOp* та *BinOp* для представлення доступних бінарних операцій. Також визначено тип *Expr* для представлення усіх припустимих виразів Адресної мови.

```
You, 18 hours ago | 1 author (You)
data BinOp =
  Add | PtrAdd | Sub | Mul | Div | Mod
  | Greater | Less | Equal
  | NotEqual | GreaterEqual | LessEqual
  | And | Or | MulDeref | MinDeref
  deriving (Eq, Show)

data UnOp = Negate | Deref | Not deriving (Eq, Show)

You, 19 hours ago | 1 author (You)
data Expr
  = Lit Value
  | Nil
  | Var String
  | UnOpApp UnOp Expr
  | BinOpApp BinOp Expr Expr
  | BuiltinFn String [Expr]
  | LabelRef String Bool
  deriving (Eq, Show)
```

Лістинг 4.6. Визначення типів для представлення виразів

Далі визначається тип даних Statement для представлення формул Адресної мови. Зокрема це формули присвоєння, засилання та обміну, предикатна формула та формула циклу, формула заміни та входження підпрограми, формула виклику вбудованої команди (наприклад «друк»), а також формули переходу та зупину. Для представлення формул циклу та заміни визначено допоміжні типи LoopEnd та Replacement.

```
data Statement
  = Assign Expr Expr
  | Send Expr Expr
  | Exchange Expr Expr
  | Predicate Expr [Statement] [Statement]
  | Loop Expr LoopStep LoopEnd Expr (Maybe String) (Maybe String)
  | Replace [Replacement] String String
  | SubprogramCall Expr [Expr] (Maybe String)
  | BuiltinProc String [Expr]
  | Jump String
  | Ret
  | Stop
  deriving (Eq, Show)
```

Лістинг 4.7. Визначення типу для представлення тверджень

Нарешті, на основі типів, описаних раніше, визначається тип для представлення рядків програми (з інформацією про мітки), а також тип для представлення всієї програми.

```
data ProgLine = ProgLine
  { labels :: [String],
    stmts :: [Statement],
    lineNum :: Int
  }
  deriving (Eq, Show)

newtype Program = Program {pLines :: [ProgLine]}
  deriving (Eq, Show)
```

Лістинг 4.8. Визначення типів для представлення рядків та програми

Далі необхідно визначити умовні позначення для токенів, згенерованих лексичним аналізатором. Це можна зробити за допомогою директиви %token, яка дозволяє визначити відповідність «умовне позначення терміналу – значення токена». Такий підхід дозволяє зробити опис елементів граматики близьким до позначень у вихідному коді.

```
%token
    intConst      { Token _ _ (TInt $$) }
    floatConst    { Token _ _ (TFloat $$) }
    identifier    { Token _ _ (TIdentifier $$) }
    labelDecl     { Token _ _ (TLabel $$) }
    builtinProc   { Token _ _ (TProc $$) }
    builtinFn     { Token _ _ (TFn $$) }
    eol           { Token _ _ TNewLine }
    "Ret"         { Token _ _ (TKeyword "Ret") }
    "P"           { Token _ _ (TKeyword "P") }
    "L"           { Token _ _ (TKeyword "L") }
    "R"           { Token _ _ (TKeyword "R") }
    "Pg"          { Token _ _ (TKeyword "Pg") }
    "not"         { Token _ _ (TKeyword "not") }
    "and"         { Token _ _ (TKeyword "and") }
    "or"          { Token _ _ (TKeyword "or") }
    "Nil"         { Token _ _ (TKeyword "Nil") }
    "&"           { Token _ _ TAnd }
```

Лістинг 4.9. Визначення відповідності терміналів до токенів

Наступним кроком є опис правил граматики Адресної мови. Для початку за допомогою вбудованих можливостей бібліотеки happy визначаються правила пріоритетності наявних операцій.

```
%left "or"
%left "and"
%nonassoc "==" "/="
%nonassoc ">" "<" "<=" ">="
%left "+" "-" "<+>"
%left "*" "/" "%"
%left NEG
%left DEREF
```

Лістинг 4.10. Визначення відповідності терміналів до токенів

Далі визначаються правила для зчитування припустимих виразів Адресної мови. При цьому правило зчитування виразу є рекурсивним, оскільки вираз може бути записаний як інший вираз в дужках '(' Exp ')'.

```
-- Expressions
Exp :  Exp "or" Exp                    { BinOpApp Or $1 $3 }
    | Exp "and" Exp                    { BinOpApp And $1 $3 }
    | Exp "==" Exp                     { BinOpApp Equal $1 $3 }
    | Exp "/=" Exp                     { BinOpApp NotEqual $1 $3 }
    | Exp "<=" Exp                     { BinOpApp LessEqual $1 $3 }
    | Exp ">=" Exp                     { BinOpApp GreaterEqual $1 $3 }
    | Exp "<" Exp                      { BinOpApp Less $1 $3 }
    | Exp ">" Exp                      { BinOpApp Greater $1 $3 }
    | Exp "+" Exp                      { BinOpApp Add $1 $3 }
    | Exp "-" Exp                      { BinOpApp Sub $1 $3 }
    | Exp "*" Exp                      { BinOpApp Mul $1 $3 }
    | Exp "/" Exp                      { BinOpApp Div $1 $3 }
    | Exp "%" Exp                      { BinOpApp Mod $1 $3 }
    | Exp "<+>" Exp                    { BinOpApp PtrAdd $1 $3 }
    | "-" Exp %prec NEG                { UnOpApp Negate $2 }
    | "'" Exp %prec DEREF              { UnOpApp Deref $2 }
    | "`" Exp "`" Exp %prec DEREF      { BinOpApp MulDeref $2 $4 }
    | "m`" Exp "`" Exp %prec DEREF     { BinOpApp MinDeref $2 $4 }
    |  "&" identifier                  { LabelRef $2 True }
    | intConst                         { Lit (IntVal $1) }
    | floatConst                       { Lit (FloatVal $1) }
    | identifier                       { Var $1 }
    | "Nil"                            { Nil }
    | builtinFnExp                     { $1 }
    | listExp                          { $1 }
    | "(" Exp ")"                      { $2 }
```

Лістинг 4.11. Визначення правил граматики

Далі аналогічним чином визначається решта правил граматики.

4.4. Основна концепція компіляції у байткод

У цьому підрозділі розглянуто загальний підхід до генерації байткоду. Для реалізації цього процесу потрібно визначити типи даних, які представлятимуть проміжні результати на кожному етапі. Після цього необхідно визначити основні та допоміжні функції, які реалізують алгоритм перетворення синтаксичного дерева на список інструкцій байткоду.

4.4.1. Визначення необхідних типів даних

Для реалізації процесу генерації байткоду перш за все необхідно визначити множину доступних інструкцій, а також додаткові типи даних для представлення результатів.

Тип даних *OpCode* визначає множину інструкцій байткоду як набір різних конструкторів. Тут представлені інструкції для маніпуляції стеку операндів, арифметичних операцій, операцій з пам’яттю, а також інструкції для більш складних випадків застосування. Також використовується автоматична реалізація класу типів *Enum*, щоб можна було перетворити даний конструктор на числове представлення.

```
data OpCode
  = OP_RETURN | OP_CONSTANT
  | OP_NOT | OP_AND | OP_OR | OP_NEGATE
  | OP_ADD | OP_PTR_ADD | OP_SUB | OP_MUL | OP_DIV | OP_MOD
  | OP_EQUAL | OP_GREATER | OP_LESS
  | OP_POP | OP_JUMP | OP_JUMP_IF_FALSE
  | OP_SEND | OP_EXCHANGE | OP_DEREF | OP_MUL_DEREF | OP_MIN_DEREF
  | OP_DEFINE_VAR | OP_SET_VAR | OP_GET_VAR | OP_ALLOC
  | OP_CALL | OP_CALL_PROC | OP_CALL_FN
  deriving (Eq, Show, Enum)
```

Лістинг 4.12. Визначення типу для представлення інструкцій байткоду

Далі необхідно визначити тип даних для представлення різних типів значень, які доступні в Адресній мові. Конструктори *IntVal* та *FloatVal* призначені для представлення цілих та десяткових чисел як скалярних даних. Конструктор *PointerVal* потрібен для представлення значення вказівника разом

із його розміром. Конструктор *StringVal* призначений для внутрішнього представлення назв змінних. Конструктор *NilVal* представляє пусте значення у пам’яті.

```
data Value
  = IntVal !Int
  | PointerVal !Int !Int !Int
  | FloatVal !Double
  | StringVal !String
  | NilVal
```

Лістинг 4.13. Визначення типу для представлення значень у програмі

Наступним кроком є визначення типу даних для представлення результату компіляції в байткод *Chunk*. Поле *code* містить список цілих чисел, які відповідають кодам інструкцій *OpCode*. Поле *constants* містить список значень типу *Value* – це реалізація «constant pool» – набору констант, які використовуються в коді. Нарешті, поле *codeLines* – це список цілих чисел довжини, рівної довжині списку *code*. Тут містяться номери рядків вихідного коду, які відповідають інструкціям у *code*. Це необхідно, щоб забезпечити можливість налагодження програм.

```
data Chunk = Chunk
  { code :: [Int],
    codeLines :: [Int],
    constants :: [Value]
  }
  deriving (Eq, Show)
```

Лістинг 4.14. Визначення типу для представлення списку інструкцій

Нарешті, можна визначити ряд допоміжних функцій для роботи із *Chunk*. Функція *writeChunk* додає в кінець списку інструкцій нову інструкцію. Функція *addConstant* додає константу до набору констант.

```haskell
writeChunk :: Int -> Int -> Chunk -> Chunk
writeChunk byte codeLine ch@(Chunk {code, codeLines}) =
  ch
    { code = code ++ [byte],
      codeLines = codeLines ++ [codeLine]
    }

addConstant :: Value -> Chunk -> (Chunk, Int)
addConstant val ch@(Chunk {constants}) =
  let newChunk = ch {constants = constants ++ [val]}
  in (newChunk, length constants)
```

Лістинг 4.15. Визначення допоміжних функцій для роботи із Chunk

Тепер можна перейти до реалізації процесу компіляції. Для цього необхідно визначити тип даних *CompState* для зберігання стану компілятора, а також додаткові допоміжні типи. Тип *LabelOffsetMap* – це словник, де ключ – назва мітки, а значення – відступ від початку скомпільованого списку інструкцій, на який вказує ця мітка. Тип *FnVarMap* – це словник, де ключ – назва функції (підпрограми), а значення – список змінних, оголошених у цій підпрограмі (це використовується для реалізації області видимості локальних змінних).

```haskell
data CompState = CompState
  { curLine :: IORef Int,
    curChunk :: IORef Chunk,
    labelOffsetMap :: IORef LabelOffsetMap,
    jumpPatches :: IORef [(Int, String)],
    loopPatches :: IORef [LoopPatch],
    labelRefPatches :: IORef [(Int, String)],
    csFnVars :: FnVarMap,
    csFnMap :: LineFnMap,
    csProgLines :: [ProgLine],
    csReps :: IORef [Int]
  }
```

Лістинг 4.16. Визначення типу для представлення стану компілятора байткоду

Головними полями у стані компілятора *curChunk* та *curLine*. Ці поля зберігають поточний список скомпільованих інструкцій та номер поточного рядка відповідно. Для деяких полів використовується тип *IORef* для того, щоб зробити їх мутабельними. *IORef* дозволяє безпечно працювати із мутабельним станом в монаді *IO*.

4.4.2. Визначення операцій над типом Value

Для більш зручної роботи із типом *Value* можна використати можливості мови Haskell, яка дозволяє визначати власну реалізацію для вбудованих класів типів.

Спочатку можна визначити реалізацію класу типів *Show*, який відповідає за форматування при перетворенні типу даних на рядок. Зокрема, для відображення вказівників вводиться позначення у вигляді *(Ptr v)*, або *(Ptr[s*c] v)* у випадку, якщо розмір вказівника більший за 1.

```
instance Show Value where
  show :: Value -> String
  show (IntVal v) = show v
  show (FloatVal v) = show v
  show (PointerVal v 1 1) = "(Ptr " ++ show v ++ ")"
  show (PointerVal v s c) =
    "(Ptr[" ++ show s ++ "*" ++ show c ++ "] " ++ show v ++ ")"
  show (StringVal s) = s
  show NilVal = "N"
```

Лістинг 4.17. Реалізація класу типів Show для типу Value

Оскільки Адресна мова є динамічно типізованою, то необхідно правильно описати роботу порівняльних та арифметичних операцій для комбінацій різних типів. Для цього можна використати класи типів *Eq*, *Ord*, *Num* та *Fractional*.

Зокрема, під час порівняння цілочисельних та десяткових значень цілочисельне значення перетворюється у тип *Double* за допомогою функції *fromIntegral*. При порівнянні вказівників використовується лише адреса, на яку вони вказують, а розмір вказівника не враховується.

```haskell
instance Eq Value where
  -- a + a
  (==) :: Value -> Value -> Bool
  (==) (IntVal a) (IntVal b) = a == b
  (==) (FloatVal a) (FloatVal b) = a == b
  (==) (PointerVal a _ _) (PointerVal b _ _) = a == b
  (==) (StringVal a) (StringVal b) = a == b
  (==) NilVal NilVal = True
  -- Int + Double
  (==) (IntVal a) (FloatVal b) = fromIntegral a == b
  (==) (FloatVal a) (IntVal b) = a == fromIntegral b
  -- Int + Pointer
  (==) (IntVal a) (PointerVal b _ _) = a == b
  (==) (PointerVal a _ _) (IntVal b) = a == b
  -- Pointer + Double
  (==) (PointerVal a _ _) (FloatVal b) = fromIntegral a == b
  (==) (FloatVal a) (PointerVal b _ _) = a == fromIntegral b
  (==) _ _ = False
```

Лістинг 4.18. Реалізація класу типів Eq для типу Value

Для реалізації арифметичних операцій над *Value* реалізовано функції із класу типів *Num* (а також *Fractional* для операції ділення). Також визначено допоміжні функції *addV*, *mulV* та *divV*. Вони описують логіку перетворення типів при застосуванні арифметичних операцій до комбінацій значень різних типів.

```haskell
instance Num Value where
  (+) :: Value -> Value -> Value
  (+) = addV
  (*) :: Value -> Value -> Value
  (*) = mulV
```

Лістинг 4.19. Фрагмент реалізації класу типів Num для типу Value

4.4.3. Реалізація алгоритму генерації байткоду

У цьому підрозділі розглянемо алгоритм перетворення вихідного коду у список інструкцій байткоду. Для цього визначається функція *compileSrc*, яка виконує лексичний та синтаксичний аналіз вихідного коду (*(parseProg . scanTokens) src*) і передає результат у функцію *compileProg*. Також у функції *compileSrc* описана логіка обробки та форматування помилок, що можуть виникати під час процесу генерації байткоду.

```
compileSrc :: String -> IO (Either String Chunk)
compileSrc src = (Right <$> compileProg ((parseProg . scanTokens) src)) `catch` handler
  where
    handler (ErrorCallWithLocation msg _) = return $ Left $ "Compilation error: \n" ++ msg
```

Лістинг 4.20. Реалізація функції compileSrc

У свою чергу функція *compileProg* отримує на вхід синтаксичне дерево програми і використовує його для ініціалізації стану компілятора. Основна частина програми компілюється у виразі *compileLines pLines cs*. Далі виконуються додаткові модифікації байткоду, після чого повертається набір інструкцій байткоду *Chunk*.

```
compileProg :: Program -> IO Chunk
compileProg (Program {pLines = pLines1}) = do
  let pLines = numerateLines pLines1
  cs <- initCs pLines (collectProgVars pLines) (collectProgFns pLines)
  compileGlobalVars cs
  compileLines pLines cs
  patchJumps cs
  patchLabelRefs cs
  compileEof cs
  getCurChunk cs
```

Лістинг 4.21. Реалізація функції compileProg

4.5. Основна концепція віртуальної машини

Реалізацію віртуальної машини для інтерпретації байткоду необхідно почати із визначення типів даних для представлення стану машини.

Тип *VmIp* представляє мутабельне посилання на instruction pointer – вказівник на наступну інструкцію, яку необхідно виконати. Тип *VmStack* представляє стек операндів. Тип *VmMemory* – це мутабельний масив *IOArray* зі значеннями типу *Value*. Він використовується для симуляції пам’яті в програмі.

```
data VM = VM
  { chunk :: !VmChunk,
    ip :: !VmIp,
    stack :: !VmStack,
    memory :: !VmMemory,
    varsMap :: !VmVarsMap,
    vmCalls :: !VmCalls
  }
```

Лістинг 4.22. Визначення типу для представлення стану віртуальної машини

Далі необхідно визначити допоміжні функції для роботи зі станом машини. Функції *push* та *pop* дозволяють модифікувати стек операндів, а функція *peek* надає доступ до певного елемента стеку за відступом від вершини. Для реалізації цих функцій використано допоміжний модуль *Stack*, який реалізовано на основі мутабельного масиву. Також визначено функцію *popN*, яка дістає значення із вершини стеку *n* разів і повертає результат у вигляді списку.

```
push :: VM -> Value -> IO ()
push vm val = Stack.push val (stack vm)

pop :: VM -> IO Value
pop vm = Stack.pop (stack vm)

popN :: Int -> VM -> IO [Value]
popN 0 _ = return []
popN n vm = do
  val <- pop vm
  restValues <- popN (n - 1) vm
  return $ val : restValues

peek :: Int -> VM -> IO Value
peek offset vm = Stack.peek' offset (stack vm)
```

Лістинг 4.23. Визначення функцій для роботи зі станом віртуальної машини

Нарешті, можна описати основний цикл виконання програми. Функція *runVm* повторно зчитує наступну інструкцію та виконує її, допоки результат виконання типу *Maybe InterpretResult* не буде наявним (*isJust*).

```
runVm :: VM -> IO InterpretResult
runVm vm = do
  (Just intRes) <- untilM isJust runStep Nothing
  return intRes
  where
    runStep _ = do
      instr <- toEnum <$> readByte vm
      execInstruction instr vm `catch` handler
      where
        handler (ErrorCallWithLocation msg _) = do
          lineNum <- getCurrentLine vm
          putStrLn $ "Runtime error at line " ++ show lineNum
          putStrLn msg
          return $ Just RUNTIME_ERR
```

Лістинг 4.24. Визначення основного циклу роботи віртуальної машини

Функція *execInstruction* використовується для опису кроків виконання кожної із доступних інструкцій байткоду. Ця функція визначена через pattern-matching, що дозволяє окремо визначати алгоритм виконання для кожної інструкції.

4.6. Реалізація операцій Адресної мови

4.6.1. Утиліти для роботи з пам’яттю

Пам’ять у віртуальній машині представляється як мутабельний масив значень типу *Value*. Для роботи з пам’яттю необхідно визначити ряд допоміжних функцій.

Функції *memRead* та *memWrite* є такими собі «обгортками» над функціями роботи з масивом, які визначені у модулі Data.Array.IO. Вони забезпечують, що користувач не зможе записати або прочитати значення з нульової адреси.

```
memRead :: Int -> VM -> IO Value
memRead addr vm
  | addr > 0 = IA.readArray (memory vm) addr
  | otherwise = error $ "Cannot access memory at " ++ show addr

memWrite :: Int -> Value -> VM -> IO ()
memWrite addr val vm
  | addr > 0 = IA.writeArray (memory vm) addr val
  | otherwise = error $ "Cannot write to memory at address " ++ show addr
```

Лістинг 4.25. Визначення утиліт для роботи з пам’яттю

4.6.2. Штрих-операція

У байткоді для представлення штрих-операції вводиться спеціальна інструкція *OP_DEREF*. Виконання цієї інструкції доволі простим. Спочатку зі стеку операндів дістається останнє значення, яке представляє певну адресу. Далі виконується читання пам’яті за цією адресою (використовується утиліта *memRead*), після чого результат додається назад до стеку операндів.

```
execInstruction OP_DEREF vm = do
  addr <- asInt <$> pop vm
  val <- memRead addr vm
  push vm val
  return'
```

Лістинг 4.26. Реалізація штрих-операції

4.6.3. Багаторазова штрих-операція

Для представлення багаторазової штрих-операції вводиться окрема іструкція *OP_MUL_DEREF*. Виконання цієї інструкції вимагає реалізації нової допоміжної функції *mulDeref*. Вона приймає на вхід кількість та адресу для застосування багаторазової штрих-операції. У випадку, якщо кількість дорівнює 0, повертається те саме значення, яке було передане в якості адреси. Це відповідає поняттю штрих-операції нульового рангу. В іншому випадку зчитується значення в пам’яті за адресою *addrVal* для отримання наступної адреси *nextAddr*. Далі операція повторюється за рахунок рекурсивного застосування функції *mulDeref*.

```
mulDeref :: Int -> Value -> VM -> IO Value
mulDeref count addr vm
  | count < 0 = mulDerefError count
  | count == 0 = return addr
  | otherwise = do
      nextAddr <- memRead (asInt addr) vm
      mulDeref (count - 1) nextAddr vm
```

Лістинг 4.27. Реалізація допоміжної функції mulDeref

Тепер за допомогою утиліти *mulDeref* можна визначити виконання інструкції *OP_MUL_DEREF*. Значення для кількості та адреси дістаються зі стеку операндів, після чого результат роботи функції *mulDeref* додається до стеку.

```
execInstruction OP_MUL_DEREF vm = do
  addrVal <- pop vm
  count <- asInt <$> pop vm
  val <- mulDeref count addrVal vm
  push vm val
  return'
```

Лістинг 4.28. Реалізація багаторазової штрих-операції

4.6.4. Мінус штрих-операція

Розглянемо реалізацію мінус штрих-операції. Спочатку необхідно визначити допоміжну функцію *minDeref*. Вона приймає на вхід параметри *count* та *startAddr*, які відповідають кількості застосування операції та початковій адресі. Якщо адреса є валідною (тобто більшою за нуль), то вона додається до початкового списку поточних адрес. Далі виконується рекурсивний пошук адрес, значення за якими посилаються на одну зі списку поточних адрес (сам пошук здійснюється за допомогою функції *getRefsToAddr*). Цей процес повторюється стільки разів, скільки вказано параметром *count*. Нарешті, з отриманого результату *refs* за допомогою функції *constructList* у пам'яті програми створюється ланцюжковий (зв'язний) список. Вказівник на початок цього списку повертається як результат функції.

```
minDeref :: Int -> Int -> VM -> IO Value
minDeref count startAddr vm
  | count <= 0 = minDerefError count
  | otherwise = do
      refs <- minDeref' count [startAddr]
      constructList (map IntVal refs) vm
  where
    minDeref' :: Int -> [Int] -> IO [Int]
    minDeref' cnt addrs
      | cnt <= 0 = return addrs
      | otherwise = do
          refs <- concat <$> mapM (`getRefsToAddr` vm) addrs
          minDeref' (cnt - 1) refs
```

Лістинг 4.29. Реалізація допоміжної функції minDeref

Тепер можна описати роботу операції *OP_MIN_DEREF*. Зі стеку операндів отримуються параметри *addr* та *count*, які передаються у функцію *minDeref*, після чого вказівник на список результатів додається на стек операндів.

```
execInstruction OP_MIN_DEREF vm = do
  addr <- asInt <$> pop vm
  count <- asInt <$> pop vm
  listHead <- minDeref count addr vm
  push vm listHead
  return'
```

Лістинг 4.30. Реалізація мінус штрих-операції

4.7. Реалізація формул Адресної мови

4.7.1. Формула засилання

Компіляція формули засилання використовує два вирази, які є операндами засилання, а також нову інструкцію *OP_SEND*.

```
compileStmt (Send valEx addrEx) cs = do
  compileExpr valEx cs
  compileExpr addrEx cs
  emitOpCode OP_SEND cs
```

Лістинг 4.31. Компіляція формули засилання

Далі можна реалізувати виконання інструкції. Першим кроком зі стека дістається значення та адреса, за якою необхідно його заслати. Далі виконується перевірка на валідність адреси за допомогою функції *checkAddrForSend*. Нарешті, у пам’ять за адресою записується передане значення. Оскільки формула засилання не вважається виразом, то додавати значення на стек не потрібно.

```
execInstruction OP_SEND vm = do
  addr <- asInt <$> pop vm
  val <- pop vm
  memWrite (checkAddrForSend addr) val vm
  return'
```

Лістинг 4.32. Реалізація виконання формули засилання

4.7.2. Формула обміну

Розглянемо реалізацію формули обміну. Ця формула дозволяє поміняти місцями значення за двома даними адресами.

Для представлення формули обміну у байткоді використовується операція *OP_EXCHANGE*. Процес її виконання можна описати таким чином: спочатку зі стеку операндів дістаються дві адреси, які були передані у формулу. Після цього отримуються значення, які знаходяться в пам’яті за даними адресами (при цьому використовується функція *memRead*). Нарешті, за допомогою функції *memWrite*

за тими самими адресами записуються отримані значення, але у зворотному порядку.

```
execInstruction OP_EXCHANGE vm = do
  addrB <- asInt <$> pop vm
  addrA <- asInt <$> pop vm
  valA <- memRead addrA vm
  valB <- memRead addrB vm
  memWrite addrA valB vm
  memWrite addrB valA vm
  return'
```

Лістинг 4.33. Реалізація виконання формули обміну

4.7.3. Формула безумовного переходу

У цьому підрозділі розглянемо компіляцію та виконання формули безумовного переходу.

Спочатку до назви мітки *label* застосовується функція *toScopedLabel*, яка допомагає правильно відформатувати назву для реалізації обмеженої області видимості міток у межах підпрограм. Наступним кроком до списку інструкцій додається інструкція *OP_JUMP* із порожнім аргументом 0. Цей аргумент пізніше буде замінено на правильний відступ для здійснення переходу завдяки виклику утиліти *addJumpPatch*, яка додає до стану компілятора інформацію про необхідність виправити перехід до мітки. Такий підхід обраний через те, що мітка може зустрітись у коді пізніше, ніж перехід до неї.

```
compileStmt (Jump label) cs = do
  scopedLabel <- toScopedLabel label cs
  chunkCount <- curChunkCount cs
  addJumpPatch chunkCount scopedLabel cs
  emitOpCode OP_JUMP cs
  emitByte 0 cs
```

Лістинг 4.34. Компіляція формули безумовного переходу

Далі розглянемо алгоритм виправлення переходів до міток, який реалізує функція *patchJumps*. Ця функція викликається вже після того, як усі рядки програми були опрацьовані та перетворені в байткод. Спочатку зі стану компілятора дістається інформація про всі мітки програми, а також збережені

посилання на формули безумовного переходу в програмі. Після цього для кожної формули переходу обчислюється крок, який має зробити інтерпретатор, щоб перейти до наступної інструкції. Далі початкове пусте значення після кожної інструкції *OP_JUMP* замінюється на розрахований крок.

```
patchJumps :: CompState -> IO ()
patchJumps cs = do
  labelMap <- getLabelOffsetMap cs
  jumps <- getJumpPatches cs
  forM_ jumps $ \(curOffset, label) -> do
    let jumpToInstr = labelMap `readMap` label
    let jumpOffset = jumpToInstr - curOffset - 2
    jumpOffset `seq`
      patchChunkCode (curOffset + 1) jumpOffset cs
```

Лістинг 4.35. Реалізація алгоритму виправлення формул безумовного переходу

Оскільки основна логіка формули переходу до мітки реалізована на етапі компіляції, під час виконання цієї формули достатньо просто використати розрахований раніше крок. Отже, інструкція *OP_JUMP* читає із байткоду аргумент *jumpOffset* і додає його до вказівника на поточну інструкцію.

```
execInstruction OP_JUMP vm = do
  jumpOffset <- readByte vm
  addIp jumpOffset vm
  return'
```

Лістинг 4.36. Реалізація виконання формули безумовного переходу

4.7.4. Предикатна формула

У цьому підрозділі розглянемо реалізацію предикатної формули, яка використовується для контролю над ходом виконання програми залежно від певної умови.

У синтаксичному дереві предикатна формула представляється за допомогою трьох складових: вираз, який описує певну умову (*ifExp*), список формул для виконання, коли умова правдива (*thenStmts*), а також список формул для виконання в іншому випадку (*elseStmts*).

Спочатку компілюється умовний вираз, після чого до списку інструкцій додається інструкція *OP_JUMP_IF_FALSE*, яка виконує перехід до іншої інструкції, якщо значення на вершині стеку є неправдивим. Власне інструкція, до якої треба перейти, вказується пізніше у виразі *patchJump toElseJump cs*. Після компіляції формул *thenStmts* додається інший перехід *OP_JUMP*, який допомагає перестрибнути через інструкції, що відповідають списку формул *elseStmts*. Також в обидвох гілках виконання формули використовується інструкція *OP_POP* для того, щоб прибрати значення умовного виразу зі стека операндів.

```
compileStmt (Predicate ifExp thenStmts elseStmts) cs = do
  -- condition
  compileExpr ifExp cs
  toElseJump <- emitJump OP_JUMP_IF_FALSE cs
  -- then clause
  emitOpCode OP_POP cs
  compileStmts thenStmts cs
  toEndJump <- emitJump OP_JUMP cs
  -- else clause
  patchJump toElseJump cs
  emitOpCode OP_POP cs
  compileStmts elseStmts cs
  -- end
  patchJump toEndJump cs
```

Лістинг 4.37. Компіляція предикатної формули

4.7.5. Формула циклування

Розглянемо реалізацію формули циклування. Спочатку за допомогою функції *getLoopRange* вхідні дані, якими задається цикл, приводяться до єдиного формату. Це необхідно через те, що формула циклування має декілька варіацій синтаксису, які забезпечують спрощений запис. Далі компілюється твердження для ініціалізації циклу *initStmt*, яке складається із формули засилання початкового значення за адресою фіксатора (рахівника) циклу.

```
compileStmt st@(Loop _ _ _ _ scope next) cs = do
  let (initStmt, stepStmt, endCondition) = getLoopRange st
  -- initialize
  compileStmt initStmt cs
  -- condition
  loopStart <- curChunkCount cs
  compileExpr endCondition cs
  exitJump <- emitJump OP_JUMP_IF_FALSE cs
  -- body
  emitOpCode OP_POP cs
  bodyJump <- emitJump OP_JUMP cs
  -- step
  stepStart <- curChunkCount cs
  compileStmt stepStmt cs
  emitLoop loopStart cs
  --
  patchJump bodyJump cs
  addLoopPatch (LoopPatch scope next stepStart exitJump) cs
```

Лістинг 4.38. Компіляція формули циклування

Наступним кроком компілюється вираз умови виходу із циклу *endCondition*. При цьому індекс першої інструкції виразу зберігається у змінній *loopStart*, оскільки саме на цю інструкцію необхідно буде здійснювати перехід на кожній ітерації циклу. Для реалізації процесу виходу із циклу використано інструкцію *OP_JUMP_IF_FALSE*.

Далі компілюється безумовний перехід на першу інструкцію тіла циклу. Цей перехід виправляється пізніше у виразі *patchJump bodyJump cs*. Далі компілюється твердження, яке представляє крок циклу *stepStmt*. На цьому етапі до рахівника циклу застосовується певна функція модифікації (наприклад, функція слідування). Після цього здійснюється безумовний перехід на початок циклу *loopStart*.

Після компіляції рядків, які представляють тіло циклу, застосовується функція *patchLoop*, яка завершує процес компіляції формули циклування. Одразу після інструкцій тіла циклу додається інструкція безумовного переходу на першу інструкцію кроку циклу *stepStart*, а після неї здійснюється виправлення інструкції умовного переходу *OP_JUMP_IF_FALSE*, яка була скомпільована раніше для виходу із циклу. Решта рядків у функції *patchLoop* реалізують

компіляцію опціональної мітки наступного рядка, на який необхідно перейти після завершення роботи циклу.

```
patchLoop :: LoopPatch -> CompState -> IO ()
patchLoop (LoopPatch {stepStart, exitJump, nextLabel}) cs = do
  emitLoop stepStart cs
  patchJump exitJump cs
  emitOpCode OP_POP cs
  case nextLabel of
    Just next -> compileStmt (Jump next) cs
    Nothing -> return ()
```

Лістинг 4.39. Реалізація алгоритму виправлення інструкцій формули циклування

4.7.6. Реалізація підпрограм

У цьому підрозділі розглянемо реалізацію оголошення та виклику підпрограм.

В Адресній мові оголошення підпрограми складається із рядка, позначеного міткою, який містить декілька формул засилання вигляду “Nil => <variable name>“. Для того, щоб розпізнати такий формат, використовується допоміжна функція *isSubprogramHead*. Далі виконується компіляція оголошень локальних змінних за допомогою функції *compileVars*. Після цього формули засилання значень у параметри підпрограми *args* компілюються у зворотному порядку, щоб забезпечити правильний процес отримання переданих значень зі стеку операндів.

```
compileLine :: ProgLine -> CompState -> IO ()
compileLine pl@(ProgLine labs _ lineNum) cs = do
  setCurLine lineNum cs
  compileLineLabels labs cs
  if isSubprogramHead pl
    then compileSubprogramHead pl cs
    else do
      lps <- getLoopPatches cs
      patchLoops lps pl cs
      compileStmts (stmts pl) cs

compileSubprogramHead :: ProgLine -> CompState -> IO ()
compileSubprogramHead (ProgLine labs args _) cs = do
  let fnName = head labs
  compileVars (csFnVars cs `readMap` fnName) cs
  compileStmts (reverse args) cs
  emitOpCode OP_POP cs
```

Лістинг 4.40. Компіляція програмних рядків

При компіляції формули входження (виклику) підпрограми спочатку обробляється вираз, який відповідає посиланню на мітку підпрограми *callValue*. Далі компілюються передані параметри *args*, після чого додається інструкція *OP_CALL*, а також кількість переданих параметрів *argCount*.

```
compileStmt (SubprogramCall callValue args _) cs = do
  let argCount = length args
  compileExpr callValue cs
  compileExprs args cs
  emitOpCode OP_CALL cs
  emitByte argCount cs
```

Лістинг 4.41. Компіляція формули виклику підпрограми

При виконанні інструкції *OP_CALL* першим кроком зчитується кількість фактично переданих параметрів *argCount*. Це значення використовується як відступ, щоб отримати посилання на підпрограму, яка викликається (*fnOffset*). До стеку викликів додається посилання на підпрограму разом із адресою наступної інструкції, яку необхідно виконати. Це необхідно для реалізації повернення із виклику підпрограми. Нарешті, значення вказівника на поточну інструкцію змінюється на адресу першої інструкції підпрограми за допомогою функції *setIp*.

```
execInstruction OP_CALL vm = do
  argCount <- readByte vm
  fnOffset <- asInt <$> peek argCount vm
  curIp <- readIp vm
  pushCall (fnOffset, curIp) vm
  setIp fnOffset vm
  return'
```

Лістинг 4.42. Реалізація виконання формули виклику підпрограми

Далі розглянемо реалізацію формули відносного зупину, яка відповідає поняттю повернення із виклику підпрограми. Спочатку зі стану віртуальної машини дістається стек викликів *curVmCalls*. Якщо він пустий, то виконується повна зупинка виконання програми. В іншому випадку виконується очищення локальних змінних за допомогою функції *freeVars*, після чого зі стеку викликів видаляється поточний виклик. Нарешті, значення вказівника на наступну

інструкцію змінюється на адресу інструкції, яка була збережена перед викликом підпрограми.

```
execInstruction OP_RETURN vm = do
  curVmCalls <- readCalls vm
  execReturn curVmCalls vm

execReturn :: [VmCallFrame] -> VM -> IO (Maybe InterpretResult)
execReturn [] _ = returnOk
execReturn ((_, returnAddr) : _) vm = do
  freeVars vm
  popCall vm
  setIp returnAddr vm
  return'
```

Лістинг 4.43. Реалізація виконання повернення із виклику підпрограми

4.7.7. Формула заміни

У даному підрозділі розглянемо реалізацію формули заміни. Оскільки ця формула має призначення синтаксичного цукру, то вона не вимагає реалізації нових інструкцій байткоду.

Основна ідея реалізації формули заміни заснована на роботі макросів у C/C++: формула вказує певний діапазон рядків (це відповідає тілу макроса) і список правил заміни (у C/C++ це аргументи, які передаються в макрос). Компілятор напряму замінює кожне застосування формули на видозмінені рядки, вказані мітками.

Спочатку виконується пошук оригінальних рядків, які записані в програмі між мітками *start* та *end*. Далі виконується заміна операцій, виразів та формул у цих рядках за допомогою допоміжної функції *lineReplacements*. Ця функція приймає на вхід список правил заміни *reps*, послідовно застосовує їх до синтаксичного дерева рядків та повертає оновлену версію рядків *newLines*. Після цього оновлені рядки компілюються за допомогою функції *compileLines*. При цьому функції *pushReplacement* та *popReplacement* модифікують внутрішній стан компілятора для забезпечення правильної області видимості міток всередині замінених рядків.

```
compileStmt (Replace reps start end) cs = do
  srcLines <- findReplaceRange start end cs
  let newLines = map (`lineReplacements` reps) srcLines
  curLn <- getCurLine cs
  pushReplacement curLn cs
  compileLines newLines cs
  popReplacement cs
```

Лістинг 4.44. Компіляція формули заміни

РОЗДІЛ 5. РЕАЛІЗАЦІЯ ПРОГРАМ АДРЕСНОЮ МОВОЮ

5.1. Реалізація стеку

Однією із найпопулярніших структур даних є стек. Це структура даних типу FIFO, яка підтримує додавання елемента на вершину стеку через операцію push, а також вилучення верхнього елемента через операцію pop.

Для реалізації стеку Адресною мовою можна використати підхід зв'язного списку. Для створення нового стеку визначається функція *stack_new*, яка засилає значення 0 до наданої адреси. Таким чином формується представлення порожнього списку[1], згідно з яким порожній список представляється вказівником на нульову адресу.

```
@stack_new ... Nil => res_addr
    0 => 'res_addr
    Ret
```

Лістинг 5.1. Реалізація функції stack_new.

Наступним кроком можна визначити функцію *stack_is_empty*, яка дозволяє перевірити стек на порожність. Вона реалізується покладаючись на той самий принцип, що порожній стек представляється вказівником на нульову адресу.

```
@stack_is_empty ... Nil => stack_addr, Nil => res_addr
    head = 'stack_addr
    'head == 0 => 'res_addr
    Ret
```

Лістинг 5.2. Реалізація функції stack_is_empty.

Далі необхідно визначити функцію додавання елемента *stack_push*. На початку вона зберігає стару адресу *old_addr*, на яку раніше посилався вказівник *head*. Тобто *old_addr* – це адреса вузла списку, який раніше був вершиною стеку. Далі ця функція алокує в пам'яті новий кортеж розміру 2 (наступна адреса +

[1] В Адресному програмуванню порожньому списку відповідає адреса зі значенням спеціального символа "Ø" або зі значенням числа "0".

значення) для нової вершини стеку та записує нову адресу в змінну *new_addr* . Тепер залишається лише оновити адресне відображення та заслати значення у відповідні адреси. Значення *new_addr* засилається за адресою *head* як вказівник на нову вершину стеку. При цьому значення *old_addr* засилається за адресою *new_addr* як вказівник на другий від вершини елемент стеку. Нарешті, нове значення *'val* засилається на другу клітинку нового вузла, тобто *new_addr + 1*.

```
@stack_push ... Nil => val, Nil => stack_addr
    head = 'stack_addr
    old_addr = 'head

    new_addr = alloc 2

    new_addr => head
    old_addr => new_addr
    'val => new_addr + 1
    Ret
```

Лістинг 5.3. Реалізація функції stack_push.

Операцію *stack_push* можна проілюструвати діаграмою, представленою на рис. 1.

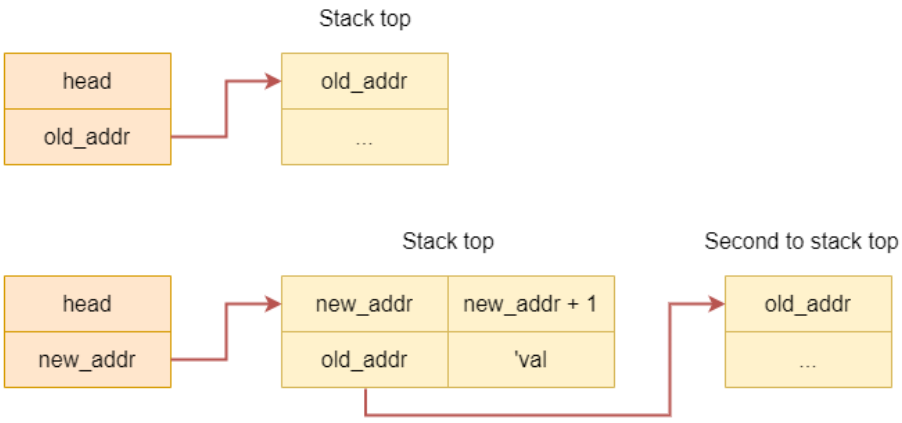

Рисунок 5.1. Візуалізація операції stack_push

Наступною можна визначити функцію *stack_pop*. Вона приймає на вхід адресу стеку та адресу для повернення результату. Спочатку вона зберігає поточну адресу вершини стеку в змінну *old_addr* і перевіряє, що стек не порожній. У разі, якщо стек порожній, виконується ранній вихід із функції.

Наступним кроком зчитується значення, яке було на вершині стеку, та засилається в адресу для повернення результату. Решта інструкцій оновлюють адресне відображення. Адреса другого від вершини стеку вузла *next_addr* засилається за адресою *head*, а вказівник old_addr, який раніше вказував на *next_addr*, затирається значенням 0.

```
@stack_pop ... Nil => stack_addr, Nil => res_addr
    head = 'stack_addr
    old_addr = 'head
    P { old_addr == 0 } Ret |

    old_val = '(old_addr + 1)
    old_val => 'res_addr

    next_addr = 'old_addr
    next_addr => head
    0 => old_addr
    Ret
```

Лістинг 5.4. Реалізація функції stack_pop.

Операцію *stack_pop* можна проілюструвати діаграмою, поданою на рис. 2.

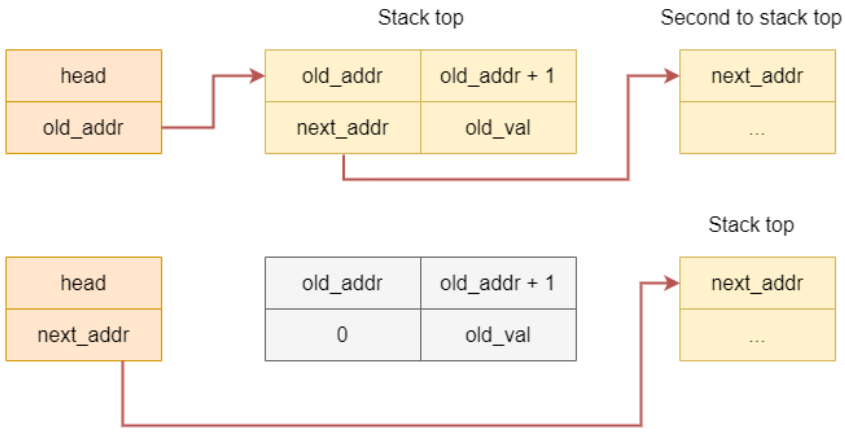

Рисунок 5.2. Візуалізація операції stack_pop

Тепер можна продемонструвати роботу визначених функцій за допомогою тестової програми. Вона створює порожній стек, в циклі додає до нього зростаючі числа за допомогою функції *stack_push*, а також виводить результат на екран використовуючи вбудовану інструкцію *printList*. Далі за допомогою функцій *stack_is_empty* та *stack_pop* реалізований цикл, який на кожній ітерації

перевіряє стек на порожність та в разі наявності значення на вершині стеку виводить його на екран разом із оновленим стеком.

```
Pg stack_new { s }
printList s

L { 1(1)5 => i } l1
    Pg stack_push { 'i, s }
@l1 ...
printList s

Pg stack_is_empty { s, s_em }
L { 0(1) P { 's_em /= 1 } => pi } l2
    Pg stack_pop { s, top_val }
    print 'top_val
    printList s
    Pg stack_is_empty { s, s_em }
@l2 ...
```

Лістинг 5.5. Реалізація тестової програми для стеку.

У результаті виконання тестової програми на екран виведено дані, зображені на рис. 3. Отже, можна зробити висновок про те, що реалізація основних операцій зі стеком пройшла успішно.

```
PS D:\DiplomaFiles\addr-lang1> stack --silent run stack
[]
[5,4,3,2,1]
5
[4,3,2,1]
4
[3,2,1]
3
[2,1]
2
[1]
1
[]
PS D:\DiplomaFiles\addr-lang1>
```

Рисунок 5.3. Вивід на екран результатів роботи зі стеком

5.2. Реалізація функцій для роботи зі списками

Списки є однією із ключових структур даних у функціональних мовах програмування, зокрема у Haskell. Теоретична основа роботи зі списками в Адресній мові розглянута детально у розділі 1.2.

5.2. Функція map

Haskell має багато вбудованих функцій для роботи зі списками. Функція *map :: (a -> b) -> [a] -> [b]* дозволяє застосувати певну трансформаційну функцію до елементів даного списку та отримати на виході новий список. В Адресній мові для роботи зі списками також можна визначити функцію *map*. На вхід вона отримує три аргументи: посилання на функцію трансформації *f*, адресу списку *list_addr*, а також адресу для повернення результату *r*.

Спочатку виконується створення нового списку за адресою *'r*, при цьому використовується допоміжна функція *list_empty*. Далі у випадку, якщо вхідний список порожній, виконується ранній вихід із функції. Наступний крок описує цикл, який ітерується по ланцюжку вказівників, із яких складається список, до тих пір, поки не зустріне посилання на нульову адресу.

У тілі циклу розраховується значення поточного елементу списку *val*. Далі за допомогою спеціального синтаксису викликається функція, на яку посилається *'f*. Параметрами у цю функцію передається поточне значення *val*, а також адреса для нового значення *new_val*. Нарешті, нове значення *'new_val* додається до списку, створеного за адресою *'r*. Таким чином, після закінчення виконання циклу за цією адресою буде створено список, який відповідає поелементному застосуванні даної функції до вхідного списку.

```
@map ... Nil => f, Nil => list_addr, Nil => r
    head = 'list_addr
    Pg list_empty { 'r }

    P { 'head == 0 } Ret |

    L { 'head, 'Nil, P { 'pi /= 0 } => pi } l1
        val = '('pi + 1)
        Pg ['f] { val, new_val }
        Pg list_add { 'new_val, 'r }
    @l1 ...
    Ret
```

Лістинг 5.6. Реалізація функції map.

Для демонстрації роботи функції *map* реалізовано тестову програму, яка створює список цілих чисел від 1 до 5 і застосовує до нього функцію *map* разом із функцією множення на 2.

```
list = [1,2,3,4,5]
printList list

Pg map { &double, list, new_list }

printList new_list

!
@double ... Nil => val, Nil => res_addr
    'val * 2 => 'res_addr
    Ret
```

Лістинг 5.7. Реалізація тестової програми для функції map.

Результати роботи тестової програми зображено на рис. 4. На екран виведено список цілих чисел від 1 до 5, а потім список чисел від 2 до 10 з кроком 2. Отже, функція *map* працює правильно.

```
PS D:\DiplomaFiles\addr-lang1> stack --silent run list_map
[1,2,3,4,5]
[2,4,6,8,10]
PS D:\DiplomaFiles\addr-lang1>
```

Рисунок 5.4. Вивід на екран результатів роботи функції map

5.3. Реалізація бінарного дерева з використанням мінус-штрих операції

Бінарні дерева є однією із найбільш розповсюджених структур даних та використовуються для реалізації ефективного пошуку, сортування, представлення множин тощо.

В Адресній мові бінарне дерево можна реалізувати подібно до зв’язних списків. Наприклад, кожний вузол може бути представлений як кортеж із трьох значень: значення поточного вузла, адреса лівого дочірнього вузла, адреса правого дочірнього вузла. При цьому відсутність відповідного дочірнього вузла можна представляти нульовим вказівником.

Проте Адресна мова надає інструменти і для альтернативного підходу до представлення дерев. Це підхід, який використовується в реляційних базах даних. Суть цього підходу полягає в тому, що посилання направлені у зворотньому напрямку: кожний вузол є кортежем із двох значень, де перше – адреса батьківського вузла, а друге – власне значення даного вузла. При цьому корінь дерева має замість посилання на батька нульовий вказівник.

Для роботи із такою структурою даних є необхідною мінус-штрих операція. Вона дозволяє для певної адреси отримати адреси інших комірок пам’яті, які посилаються на дану. Отже, маючи адресу кореня дерева, можна отримати адреси його дочірніх вузлів. Аналогічним чином, рекурсивно застосовуючи мінус-штрих операцію до дочірніх вузлів, можна обійти все дерево.

5.3.1. Функції для створення та пошуку вузлів

Для створення вузлів бінарного дерева можна визначити допоміжні функції *bst_create_root* та *bst_create_child*. Вони мають подібну реалізацію: кожна алокує в пам’яті кортеж довжини 2 та засилає у першу клітинку адресу батька, а в другу – значення вузла. При цьому у випадку *bst_create_root* адреса батька відсутня, тому у першу клітинку засилається значення 0.

```
@bst_create_root ... Nil => val, Nil => res_addr
    n = (alloc 2) + 0
    ptr(0) => n + 0
    'val => n + 1
    n => 'res_addr
    Ret

@bst_create_child ... Nil => val, Nil => parent_addr, Nil => res_addr
    n = (alloc 2) + 0
    ptr('parent_addr) => n + 0
    'val => n + 1
    n => 'res_addr
    Ret
```

Лістинг 5.8. Реалізація функцій створення вузлів дерева

Також необхідно визначити допоміжну функцію для пошуку дочірніх вузлів *bst_find_children*. Вона приймає на вхід адресу певного вузла, а також

адреси для повернення лівого та правого дочірнього вузла. Значення поточного вузла записується у змінну *node_val*. Далі до адреси поточного вузла застосовується мінус-штрих операція, щоб отримати список адрес дочірніх вузлів *children*. Далі за допомогою циклу значення кожного дочірнього вузла порівнюється зі значенням поточного вузла. Це потрібно, щоб забезпечити правильну відповідність між лівим та правим дочірнім вузлом, адже мінус-штрих операція може повернути адреси у «перемішаному» порядку. Після цього отримані значення *left* та *right* повертаються через адреси, надані раніше для передачі результатів.

```
@bst_find_children ... Nil => node, Nil => left_addr, Nil => right_addr
    node_addr = 'node
    node_val = '(node_addr + 1)

    children = m`1`(node_addr)
    left = 0; right = 0

    L { 'children, 'Nil, P { 'i /= 0 } => i } b
        child_addr = '('i + 1)
        child_val = '(child_addr + 1)
        P { child_val < node_val } left = child_addr | right = child_addr
    @b ...

    left => 'left_addr
    right => 'right_addr
    Ret
```

Лістинг 5.9. Реалізація функції знаходження дочірніх вузлів

5.3.2. Функція додавання значення у дерево

Наступним кроком є визначення функції додавання значення до дерева *bst_insert*. Вона отримує на вхід нове значення, а також адресу вузла. Спочатку нове значення порівнюється зі значенням поточного вузла, якщо вони рівні, виконується ранній вихід з функції, бо дане значення вже присутнє в дереві.

Далі для знаходження дочірніх вузлів застосовується раніше визначена допоміжна функція *bst_find_children*. Знову виконується порівняння нового значення для вставки із поточним значенням, цього разу для того, аби визначити, в який із двох дочірніх вузлів необхідно вставити значення.

Нарешті, здійснюється перевірка на існування даного дочірнього вузла. Якщо його не існує, створюється новий вузол за допомогою раніше визначеної функції *bst_create_child*. Якщо ж даний вузол існує, операція вставки продовжується шляхом рекурсивного виклику функції *bst_insert*.

```
@bst_insert ... Nil => val, Nil => node
    node_addr = 'node
    node_val = '(node_addr + 1)

    P { 'val == node_val } Ret |

    Pg bst_find_children { node_addr, left, right }

    is_left = 'val < node_val
    P { is_left } cur_child = 'left | cur_child = 'right

    P { cur_child == 0 } case_create | case_insert
    @case_create ...
        Pg bst_create_child { 'val, node_addr, cur_child_new }
        end
    @case_insert ...
        Pg bst_insert { 'val, cur_child }
        end

    @end ...
    Ret
```

Лістинг 5.10. Реалізація функції додавання значення до дерева

5.3.3. Функція видалення значення із дерева

Далі можна визначити функцію видалення значення із дерева *bst_delete*. Оскільки видалення з бінарного дерева є більш складною операцією, ніж вставка, то для реалізації необхідно створити декілька допоміжних функцій.

Спочатку виконується перевірка на порожність. Далі значення поточного вузла записується у змінну *node_val*, а також виконується пошук дочірніх вузлів за допомогою *bst_find_children*. Якщо значення, яке необхідно видалити, відрізняється від значення поточного, то виконується рекурсивний виклик функції *bst_delete*. Якщо ж необхідно видалити поточне значення, виконується алгоритм видалення та перебудови дерева залежно від кількості дочірніх вузлів.

```
@bst_delete ... Nil => val, Nil => node
    node_addr = 'node
    P { node_addr == 0 } Ret |
    node_val = '(node_addr + 1)

    Pg bst_find_children { node_addr, left, right }

    P { 'val == node_val } case_cur | case_child
    @case_child ...
        P { 'val < node_val } child = left | child = right
        Pg bst_delete { 'val, 'child }
        end
    @case_cur ...
        parent_addr = 'node_addr
        @case_leaf ...
            P { 'left == 0 and 'right == 0 } | case_no_left
            0 => node_addr
            end
        @case_no_left ...
            P { 'left == 0 } | case_no_right
            Pg bst_shift_child { node_addr, 'right }
            end
        @case_no_right ...
            P { 'right == 0 } | case_both
            Pg bst_shift_child { node_addr, 'left }
            end
        @case_both ...
            Pg bst_min_value { 'right, min_val }
            Pg bst_set_value { node_addr, 'min_val }
            Pg bst_delete { 'min_val, 'right }
    @end ...
    Ret
```

Лістинг 5.11. Реалізація функції видалення значення із дерева

5.3.4. Демонстрація роботи бінарного дерева

Для демонстрації роботи бінарного дерева створено тестову програму. Вона створює бінарне дерево на основі списку, а потім поступово видаляє із нього елементи. При цьому на кожному етапі на екран виводиться представлення дерева у вигляді списку, а також розмір дерева.

```
list1 = [10, 5, 15, 2, 8, 12, 20]
printList list1

Pg bst_from_list { list1, bst }
Pg bst_to_list { 'bst, tree_list }
printList tree_list
Pg bst_size { 'bst, sz }
print 'sz

list_to_delete = [10, 12, 15, 5]

L { 'list_to_delete, 'Nil, P { 'i /= 0 } => i } l1
    val = '('i + 1)
    Pg bst_delete { val, 'bst }
    Pg bst_to_list { 'bst, tree_list }
    printList tree_list
    Pg bst_size { 'bst, sz }
    print 'sz
@l1 ...
```

Лістинг 5.12. Реалізація програми для тестування роботи бінарного дерева

Результати роботи програми зображено на рис. 5. На кожному етапі список, який представляє бінарне дерево, залишається відсортованим. Отже, можна зробити висновок про успішну реалізацію.

```
PS D:\DiplomaFiles\addr-lang1> stack --silent run "test/data/bin_tree_short.adpl"
[10,5,15,2,8,12,20]
[2,5,8,10,12,15,20]
7
[2,5,8,12,15,20]
6
[2,5,8,15,20]
5
[2,5,8,20]
4
[2,8,20]
3
```

Рисунок 5.5. Вивід на екран результатів роботи з бінарним деревом

ВИСНОВКИ

Дана робота присвячена дослідженню концепції Адресного програмування, реалізації інтерпретатора її головної частини мовою Haskell та демонстрації на прикладах алгоритмів обробки Адресною мовою спискових ланцюжків та деревоподібних форматів, зокрема таких, які відповідають бінарним деревам у сучасному розумінні поняття Абстрактних типів даних.

У даній роботі наведено дослідження можливостей Адресної мови програмування. Створено специфікацію мови ADPL, засновану на попередніх дослідженнях Адресної мови, описано видозміну синтаксису. За допомогою мови Haskell реалізовано інтерпретатор Адресної мови на основі проміжного представлення (байткоду) та віртуальної машини. Наведено приклади програм Адресною мовою для роботи з різними структурами даних, зокрема списками та бінарними деревами.

Основними результатами цієї роботи є:

- розробка видозміни синтаксису Адресної мови - специфікації ADPL, яка призначення для забезпечення можливості складати програми Адресною мовою з використанням сучасних стандартних текстових редакторів;
- розробка методів та прийомів реалізації інтерпретатора мови програмування, яка містить засоби вказівників (опосередковану адресацію вищих рангів);
- розроблено прийом моделювання адрес, імен змінних та значень, які відповідають основній концепції Адресного програмування для реалізації опосередкованої адресації вищих рангів та реалізації багатократного розіменування Pointers;
- реалізація мовою функційного програмування Haskell інтерпретатора ADPL - основної частини Адресної мови програмування.
- в інтерпретаторі реалізовано програми обробки спискових ланцюжків та деревоподібних форматів, зокрема реалізовано програми, які обробляють

деревоподібні формати, які ідентичні сучасному розумінню бінарних дерев;

- реалізовано операції обробки бінарних дерев, які представлено специфічним методом Адресного програмування з використанням мінус-штрих операції
- надано доступ до вільного користування основною частиною Адресної мови програмування (ADPL).

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. COMPREHENSIVE SYSTEM MANUAL. A System of Automatic Coding for the Whirlwind Computer. By: C. Adams, D. Arden, S. Best and other. Digital Computer Laboratory Massachusetts Institute of Technology Cambridge 39, Massachusetts. 1953. – [Електронний ресурс] – Режим доступу до ресурсу: URL: http://www.bitsavers.org/pdf/mit/whirlwind/M-series/M-2539-2_Comprehensive_System_Manual_Dec55.pdf (дата звернення: 15.05.2024).

2. Вычислительная машина “Киев”: Математическое описание / В.М.Глушков, Е.Л.Ющенко // К.: Государственное издательство технической литературы, 1962. − 183 с. [Електронний ресурс] – Режим доступу до ресурсу: URL: https://files.infoua.net/yushchenko/Vychislitelnaya-mashyna-Kiev_VHlushkov_EYushchenko_1962.pdf (дата звернення: 10.05.2024).

3. Ющенко Е.Л., Адресное программирование // К. : – Гос. издательство технической литературы, 1963. – 287 с., [Електронний ресурс] Режим доступу до ресурсу: URL: https://files.infoua.net/yushchenko/Adresnoeprogrammirovanie_EYushchenko_1963.pdf (дата звернення: 10.05.2024).

4. Некоторые вопросы теории алгоритмических языков и автоматизации программирования / Рукопись доклада по совокупности выполненных и опубликованных работ: Диссертация доктора физ.-мат. наук / Ющенко Екатерина Логвиновна – Киев, 1964. – 51 с. [Електронний ресурс] – Режим доступу до ресурсу: URL: https://files.infoua.net/yushchenko/Nekotorye-voprosy-teorii-algoritmicheskih-yazykov_etc_KYushchenko_Doklad_1964.pdf (дата звернення: 10.05.2024).

5. Nystrom R. Crafting Interpreters/ Robert Nystrom. – 2015. – [Електронний ресурс] Режим доступу до ресурсу: URL: https://craftinginterpreters.com/ (дата звернення: 10.05.2024).

6. Ющенко Ю. О. Окремі аспекти декларативності «мінус штрих-операції» / Ю. О. Ющенко // Наукові записки НаУКМА. Комп’ютерні науки. – 2020. – Т. 3. – С. 19–26. [Електронний ресурс] – Режим доступу до ресурсу: URL: https://doi.org/10.18523/2617-3808.2020.3.17-26 (дата звернення: 10.05.2024).
7. Ющенко Ю.О. Деревоподібні формати Адресного програмування / Ющенко Ю. О. // Наукові записки НаУКМА. Комп'ютерні науки. - 2021. - Т. 4. - С. 78-87. [Електронний ресурс] – Режим доступу до ресурсу: URL: https://doi.org/10.18523/2617-3808.2021.4.78-87 (дата звернення: 10.05.2024).
8. Ющенко Ю. О. Розробка архітектури комп'ютера «Київ» за концепцією адресного методу програмування / Ю. О. Ющенко // Проблеми програмування. — 2021. — № 4. — С. 103—118. — Бібліогр.: 46 назв. — укр. [Електронний ресурс] – Режим доступу до ресурсу: URL: http://dspace.nbuv.gov.ua/bitstream/handle/123456789/183499/09-Yuschenko.pdf?sequence=1 https://doi.org/10.15407/pp2021.04.103 (дата звернення: 10.05.2024).
9. Чередник К. Реалізація компілятора базової частини адресної мови. Текстова частина до курсової роботи. / К. Чередник. – 2022. – [Електронний ресурс] – Режим доступу до ресурсу: URL: https://ekmair.ukma.edu.ua/handle/123456789/28337 (дата звернення: 10.05.2024).
10. Чередник К. Репозиторій компілятора базової частини адресної мови: Address Programming Language interpreter, – 2022. – [Електронний ресурс] – Режим доступу до ресурсу: URL: https://github.com/karina-cherednyk/AddressProgrammingLanguage (дата звернення: 10.05.2024).
11. Ющенко Ю. О. Як в Україні винайшли Pointers. Екзотична потужність Адресної мови (1955р.). / Ю. О. Ющенко - 2024. - [Електронний ресурс] –

Режим доступу до ресурсу: URL: https://youtu.be/P1UnThQvkkI, Youtube Channel: https://www.youtube.com/c/ЮрийЮщенкоСенсей (дата звернення: 10.05.2024).

ДОДАТОК А. ПОСИЛАННЯ НА ВІЛЬНИЙ ДОСТУП ДО ІНТЕРПРЕТАТОРА

A Haskell implementation of an interpreter for the main part of Address Programming Language. . [Електронний ресурс] – Режим доступу до ресурсу: URL: GitHub - Jorge3129/address_lang_1 (дата звернення: 13.05.2024).