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

РЕАЛІЗАЦІЯ ІМПЕРАТИВНОЮ МОВОЮ ПРОГРАМУВАННЯ ОСНОВНОЇ ЧАСТИНИ АДРЕСНОЇ МОВИ ДЛЯ ОБРОБКИ СКЛАДНИХ ІЄРАРХІЧНИХ СТРУКТУР

Текстова частина

магістерської роботи

за спеціальністю «Інженерія Програмного Забезпечення» 121

Керівник магістерської роботи

Доцент Ющенко Ю. О.

_____

(підпис)

“____” _____ 2024 р.

Виконав студент

Чорнокозинський К. С.

“____” _____ 2024 р.

КИЇВ 2024

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

ЗАТВЕРДЖУЮ
Зав. кафедри інформатики
к.ф-м.н., доц. Гороховський С. С.
_____
(підпис)
“_____” _____2023 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на магістерську роботу

студенту 2 р.н. магістерської програми Інженерія Програмного Забезпечення Чорнокозинському Кирилу Сергійовичу
Розробити Компілятор для основної частини Адресної мови для обробки складних ієрархічних структур

Зміст текстової частини до магістерської роботи:

Зміст
Анотація
Вступ
1 Дослідження предметної області
2 Видозміна специфікації
3 Огляд використаних технологій
4 Проєктування та розробка
5 Обробка ієрархічних структур
Висновки
Список використаних джерел

Дата видачі “_____” _____ 2024 р.

Керівник
Ю. О. Ющенко, доцент _____
(підпис)

Завдання отримав
К. С. Чорнокозинський _____
(підпис)

Тема: Реалізація імперативною мовою програмування основної частини адресної мови для обробки складних ієрархічних структур

Календарний план виконання роботи:

№ п/п	Назва етапу дипломного проекту (роботи)	Термін виконання етапу	Примітка
1.	Отримання завдання на дипломну роботу	02.09.2023	
2.	Огляд технічної літератури за темою роботи	25.11.2023	
3.	Робота над видозміною специфікації	30.11.2023	
4.	Вибір елементів для реалізації системи	10.12.2023	
5.	Побудова лексичного аналізатору	20.12.2023	
6.	Побудова синтаксичного аналізатору	15.01.2024	
7.	Побудова демонстраційного інтерпретатору	31.1.2024	
8.	Побудова віртуальної машини та генератору байт-коду	16.05.2024	
9.	Попередній захист магістерської роботи	17.05.2024	
10.	Коригування роботи за результатами попереднього захисту	27.05.2024	
11.	Остаточне оформлення пояснювальної записки та створення слайдів	02.06.2024	
12.	Захист магістерської роботи	10.06.2024	

Студент Чорнокозинський К. С.

Керівник Ющенко Ю. О. “____”_____ 2024 р.

ЗМІСТ

АНОТАЦІЯ

У даній роботі розроблені інтерпретатор та компілятор Адресної мови програмування. Ці розробки призначено для реставрації ІТ-історії та ознайомлення програмістів з мовою, яка стала основою для багатьох сучасних технологій програмування. Вона дозволяє ефективно працювати зі складними ієрархічними структурами, включаючи інноваційні для свого часу спискові ланцюжки, та потужні засоби програмування: «штрих-операцію» (розіменування вказівників) та «мінус штрих-операцію», яка є оберненою до розіменування вказівників. Реалізація Адресної мови допомагає розробникам програмного забезпечення зрозуміти витоки сучасних технологій і методології програмування, а також зберегти знання про важливі технологічні досягнення минулого.

Знайомство з концепцією адресації вищих рангів значно поширює світогляд програмістів на вказівники та їх можливості, бо надає суттєво поглиблене розуміння їх сутності. Знайомству з цією концепцією перешкоджає розбіжність термінології, запровадженої в Україні, з загально прийнятою термінологією сучасного програмування. Перепоною такому знайомству є відсутність реалізацій Адресної мови, що унеможливлює програмістам писати та налагоджувати програми для практичного використання концепції адресації вищих рангів.

Окрім зазначеного, актуальність та доцільність цієї роботи обумовлюється необхідністю встановлення історичної справедливості, адже в Адресній мові, вперше в світі, ще у 1955 році було винайдено інноваційний засіб в програмуванні – адресація 2-ого (англ. Pointers) та вищих рангів, але ця інформація не є всесвітньо відомою, оскільки досі помилково приписується винайдення вказівників у Швеції американському вченому Гарольду Лоусону у 1964 р. у мові програмування PL/1.

Ключові слова: Адресна мова, Address Language, compiler, interpreter, lexical analysis, syntax analysis, Rust, Cargo, bytecode, генерація коду, CLI, інтерфейс командного рядка, lexer, parser, vm, віртуальна машина, ast, інтерпретатор, abstract syntax tree, компілятор, структура, типи, лексичний аналіз, лексичний аналізатор, сканер, опосередкована адресація, адресація 2-ого рангу, адресація вищих рангів, штрих операція, розіменування вказівників, Pointers, списки, дерева, деревоподібні формати, абстрактні типи даних.

СПИСОК УМОВНИХ СКОРОЧЕНЬ

AST (Abstract Syntax Tree) – абстрактне синтаксичне дерево.

CLI (Command Line Interface) – інтерфейс командного рядка.

VM (Virtual Machine) – віртуальна машина.

ADL (Address Language) – найменування реалізованого діалекту Адресної мови програмування.

DRY (Don’t repeat yourself) – принцип розробки програмного забезпечення.

ВСТУП

Для нащадків надзвичайно важливо зберегти історію зародження інформаційних технологій, зокрема причино-наслідкові зв’язки появи найпотужніших засобів сучасних технологій програмування. Адресна мова програмування має значну історичну та технічну цінність. В цій дипломній роботі сучасною імперативною мовою програмування реалізовано інтерпретатор та компілятор основної частини маловідомої людству Адресної мови, яка була першою в світі мовою програмування високого рівня зі значним практичним значенням. В цій роботі реалізовано програми обробки складних ієрархічних структур – деревоподібних форматів. Концепція деревоподібних форматів Адресної мови програмування розбігається з класичною концепцією абстрактних типів даних імперативних засобів програмування та, як не дивно, у певному сенсі споріднена з декларативними засобами програмування та підтримує можливість використовувати парадигму об’єктно-орієнтованого програмування, яка з’явилась у програмуванні на два десятки років пізніше Адресної мови. В роботі наведено приклади об’єктно-орієнтованих програм на Адресній мові програмування.

Особливо слід зазначити про налагодження прикладів програм з представленням дерев декларативними методами, які не притаманні імперативним засобам програмування, ба більше: імперативні засоби не допускають таке представлення дерев без застосування вишуканих методів модулювання декларативності.

Адресна мова програмування, розроблена в середині 20-го століття, стала важливим кроком у розвитку комп’ютерних наук. Вона дозволила ефективно працювати зі складними ієрархічними структурами – дерево-подібними форматами, та заклала фундамент для сучасних мов програмування. Однією з найбільших інновацій Адресної мови було впровадження концепції спискових ланцюжків, яким подібні однозв’язні та двозв’язні списки. Це була перша мова програмування з динамічними структурами даних і дала потужний

інструмент для розробки вишуканих алгоритмів розв'язку логічних задач та задач штучного інтелекту. Саме на Адресній мові програмуванні вперше в світі проведено аналіз речень природньою мовою, розпізнавання простих геометричних фігур, друкованих та рукописних літер та цифр [13, 14] з використанням методів машинного навчання. Саме наявність можливості необхідним чином групувати дані у складні ієрархічні структури та можливість визначати зв'язки між даними за допомогою «штрих-операції» (розіменування Pointers) дозволила українцям першим в світі розпочати на комп'ютері «Київ» розв'язувати перелічені вище задачі. Новаторські підходи українців тих часів значно розширили сферу застосування комп'ютерів, як в самій Україні, так і за її межами.

Популяризація інформації про Адресну мову та її історію є важливим завданням, оскільки це дозволяє сучасним розробникам і дослідникам краще зрозуміти витоки сучасних технологій та їх методології. Це також сприяє реставрації ІТ-історії, допомагаючи зберегти знання про важливі технологічні досягнення минулого. Вивчення та відновлення історичних мов програмування разом з наданням цікавих уроків про зародження та еволюцію програмування дозволяє глибше усвідомити концепцію вказівників та абстрактних типів даних та ширше зрозуміти можливості та призначення вказівників.

Інноваційність Адресної мови програмування в її часовому контексті полягала в новаторських підходах до роботи з даними та ефективному використанні обчислювальних ресурсів. Завдяки Адресній мові стало можливим розширити коло задач, які здатні розв'язувати комп'ютери за рахунок використання складних ієрархічних структур.

Адресна мова програмування значно розширило можливості програмування тих часів. Про це мають дізнатись всі програмісти. Людство має зберегти правдиву історію зародження інформаційних технологій для нащадків.

Таким чином, ця робота розв'язує важливу задачу для всього людства, оскільки, окрім технічної реалізації Адресної мови сприяє популяризації знань

про історію появи найпотужніших засобів інформаційних технологій та збереженню цієї історії.

1. ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ

Адресна мова доволі наближена до мови математичних формул. У ній прийнято всі символи, якими в математиці позначають величини, вектори, функції, множини, тобто дужки, цифри, букви з індексами і без них, а також усі знаки математичних операцій. Однак в Адресну мову вводяться спеціальні поняття і відповідні символи, завдяки яким вона стає більш пристосованою для опису алгоритмів.

Перейдемо до визначення основних понять і опису засобів Адресної мови.

1.1. Рядки

Адресний запис алгоритму складається з рядків.

У кожному з них записується одна або кілька алгоритмічних дій. Запис кожної дії називається формулою. Якщо в рядку є кілька формул, то між ними ставиться крапка з комою або кома. Під час лінійного запису рядків між ними ставиться крапка.

Зазвичай дії адресного алгоритму виконуються одна за одною за іншою в порядку запису рядків, але цей порядок може бути змінений за допомогою деяких із допустимих у мові формул.

1.2. Мітки та мічені рядки

Рядки адресного алгоритму для вказання порядку їх слідування можуть бути відзначені мітками. Такі рядки називаються міченими. Міткою може бути цифра, літера, число, ідентифікатор з цифр і літер, наприклад: *214, A31, K1, Соколов*. Мітки можуть вибиратися майже довільно. Мітка з наступним трьома крапками ставиться ліворуч від відзначеного рядка. Три крапки в запису алгоритму завжди означають, що ліворуч від них стоїть мітка, що відзначає даний рядок. Рядки можуть бути відзначені кількома мітками.

Мітки безумовного переходу – це окремий рядок може складати мітка без наступних трьох крапок. У цьому випадку вона називається міткою безумовного переходу. При реалізації алгоритму виконання такого рядка полягає в переході до рядка, позначеного тією ж міткою (або має цю мітку як одну з своїх). Таким чином, мітка, що складає рядок, має сенс тільки тоді, коли в алгоритмі є інший рядок, позначений тією ж міткою.

Окремий рядок алгоритму (без наступних трьох крапок) може складати адреса будь-якого рангу. У цьому випадку вміст за даним рангом є міткою. Виконання такого рядка полягає у витягуванні вмісту за даним рангом і в переході до рядка алгоритму, позначеного міткою, визначеною цим вмістом.

1.3. «Штрих-операція», «мінус штрих-операція» та адресне відображення

Одним з основних понять Адресної мови є «штрих-операція» (операції розіменування вказівника), яка визначає відображення множини адрес (А) на множину вмісту цих адрес (В). «Штрих-операція» записується як $'a = b$, де a – аргумент, а b – результат операції. Таким чином «штрих-операція» визначає функцію одного аргументу, яка називається «штрих-функцією». Аргумент «штрих-функції» називається адресою, а її значення – вмістом адреси.

«Штрих-функція» є однозначною, тобто одному адресу відповідає тільки один вміст. «Штрих-операція» алгоритмічно виконується, тобто за адресою завжди можна дізнатися її вміст.

Адреси можуть мати різні ранги. Повторне застосування «штрих-операції» приводить до поняття адреси другого рангу. Наприклад, якщо $'a_1 = a_2$ *і* $'a_2 = b$, то [2]

$a_1 = '('a_1) = ''a_1 = 'a_2 = b$, де a_1 є адресою другого рангу для b.

Зворотне відображення може бути неоднозначним, однаковий вміст може відповідати кільком різним адресам. «Мінус штрих-операція» (або "–′А") є операцією, що була введена як обернена до «штрих-операції». У той час, як

«штрих-операція» дозволяє отримати значення за адресою, «мінус штрих-операція» дозволяє отримати адреси, які посилаються на певну адресу.

Мінус штрих-операція визначається як операція, яка повертає сукупність всіх адрес, значення яких дорівнюють даній адресі. Формально це можна записати як:

$$-'A = \{ B \mid 'B=A \}$$

Тут *A* це певна адреса, а результатом $-'A$ є всі адреса *B*, для значення за адресою *B* дорівнює *A*.

1.4. Формула зупину

В Адресній мові програмування використовуються два спеціальні символи, які називаються формулами відносного і безумовного зупину. Використання символу $\exists$ пов'язане з формулами входження (будуть розглянуті далі); символ ! означає дію – кінець алгоритму. Символ $\exists$ також означає кінець алгоритму, якщо йому не передувала відповідна формула входження. Формули зупину можуть складати окремі рядки алгоритму або входити як складові частини до інших формул Адресної мови.

1.5. Формули засилання та обміну

Вирази виду $f_1 => f_2$, де f_1 і f_2 – адресні функції, можуть складати окремі рядки алгоритму і називаються формулами засилання. Вони символізують наступну алгоритмічну операцію: значення адресної функції f_1 засилається за адресою, рівною значенню функції f_2. Виконання формули засилання змінює адресне відображення і тим самим впливає на значення ряду адресних функцій. Наприклад, якщо $'a = b$, $'c = d$, то вираз $c => a$ є формулою засилання, після виконання якої $''a = d$, так як $''a = 'c = d$. Це означає, що виконання цієї

формули змінює значення всіх адресних функцій, які включають адресу *a* з будь-яким рангом.

$$a \Rightarrow b; \quad a \Rightarrow 'c; \quad a \Rightarrow d + 2$$

Рисунок 1.5.1. Приклад формул засилання

Вирази виду $f_1 <=> f_2$, де f_1 і f_2 – адресні функції, можуть складати окремі рядки алгоритму і називаються формулами обміну. Вони є записом наступної алгоритмічної операції: при заданому адресному відображенні обчислюються значення адресних функцій f1 і f2, які сприймаються як адреси, і здійснюється обмін між вмістом цих адрес. Вміст інших адрес залишається незмінним. Наприклад, якщо ′a = b, ′c = d, то вираз c <=> a є формулою обміну, після виконання якого ′a = d, так як ′c = b.

1.6. Предикатна формула

Предикатними формулами називаються вирази виду *P { L } α ↓ β*, де *P* – символ предикатної формули; *L* – деякий булевий вираз; ↓ – розділовий знак; *α* і *β* – верхнє і нижнє значення предикатної формули, кожне з них може бути одним з описаних раніше видів рядків.

$$\boldsymbol{P}\{'a = 'c\}\ 'f - 'g \Rightarrow d \downarrow 'h + 1 \Rightarrow b;\ f$$

Рисунок 1.6.1. Приклад предикатної формули

Предикатні формули складають окремі рядки алгоритму. Відповідна предикатній формулі алгоритмічна дія полягає у виконанні рядка, що представляє верхнє значення предикатної формули, якщо булевий вираз є істинним, а інакше – виконується нижнє значення предикатної формули.

1.7. Формула циклювання

Предикатними формулами називаються вирази виду вказаному на рисунку 1.7.1.

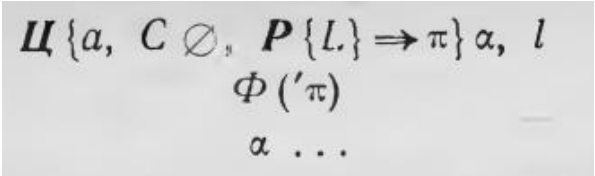


Рисунок 1.7.1. Приклад формули циклювання

Де *Ц* – символ формули циклювання; *α* – мітка рядка, до якої усі рядки після заголовку формули циклювання називаються областю дії формули циклування; *l* – мітка рядка, до якої необхідно перейти після кінця роботи формули циклування; С ∅ – операція слідування (в даному випадку операція додавання одиниці над вмістом адресу π); а – початкове значення вмісту адресу π.

1.8. Формула входження і підпрограми

Формули входження застосовуються для переходу до виконання перетворення, що знаходиться в іншому місці запису програми. Саме перетворення називається підпрограмою. В описі підпрограми повинні бути вказані:

1. Мітка, присвоєна даній підпрограмі;

2. Формула відносного зупину;

3. Впорядкований список вхідних та вихідних параметрів.

Підпрограми в Адресній мові мають наступний вигляд:

name ... Ø => a1 , Ø => a1, Ø => a2, ... , Ø => an
'a1 => an
Ƨ

Під час виконання алгоритму параметри (*a1, ..., an*) ініціалізуються списком адресів, що входять в формулу входження.

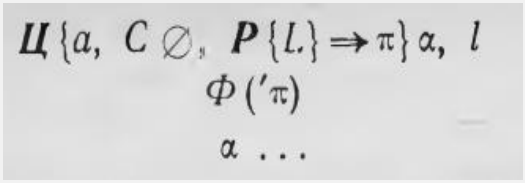
Формулами входження називаються вирази виду *П α { a1, ..., an } β, де П* – формула входження; – мітка початку підпрограми; *a1, ..., an* – список параметрів; *β* – мітка рядка, до якої необхідно перейти після кінця роботи формули входження.

У цьому розділі проведено дослідження основних понять Адресної мови програмування, яка за своєю структурою наближена до мови математичних формул і використовує широкий набір символів та операцій для опису алгоритмів. Було розглянуто поняття рядків, міток, штрих-операцій, мінус штрих-операцій, формул зупину, формул засилання та обміну, предикатних формул, формул циклювання та формул входження. Ці елементи дозволяють ефективно маніпулювати адресами та їх вмістом, визначати точки завершення алгоритму, передавати значення між адресами, описувати умови та цикли, а також викликати підпрограми.

2. ВИДОЗМІНА СПЕЦИФІКАЦІЇ АДРЕСНОЇ МОВИ

Синтаксис Адресної мови програмування, хоча і є потужним інструментом для маніпуляції складними ієрархічними структурами даних, виявився не зовсім зручним для написання програм на сучасних комп’ютерах. Оригінальний синтаксис Адресної мови використовує багато символів, які не завжди доступні на стандартній клавіатурі, що значно ускладнює і уповільнює процес написання коду. Для спрощення та пришвидшення написання програм на Адресній мові було вирішено провести певні заміни в позначеннях.

У цьому розділі буде розглянуто зміни, внесені в синтаксис Адресної мови, щоб зробити її більш зручною для використання. Ці зміни покликані полегшити написання та читання програм, зберігаючи при цьому всю потужність і гнучкість, яку надає Адресна мова.

Назва елементу	**Оригінальний синтаксис**	**Оновлений синтаксис**
Формула відносного зупину	*Ꞵ*	return
Формула відносного переходу	↓n	|n
Формула входження	*П* α *{ a1, ..., an }* β	*SP a { a1, ..., an } b*
Предикатна формула	*P { L }* α ↓ β	P { L } α | β
Формула циклювання		*L{a, step, condition => pi }* *b l f('pi)* *b...*
Пуста множина	∅	null
Багатократна штрих-операція	^{k}n	D { n, k }
Мітки безумовного переходу	label	@label
Формула заміни	З{ …}	R{ …}

У цьому розділі були розглянуті зміни, внесені в синтаксис Адресної мови, щоб зробити її більш зручною для використання. Новий синтаксис зберігає всю потужність і гнучкість, яку надає Адресна мова, але робить її більш доступною та легкою для написання і читання програм.

3. ОГЛЯД ВИКОРИСТАНИХ ТЕХНОЛОГІЙ

Вибір технологій для розробки проєкту є важливим підготовчим етапом, тому що вдало вибрані технології та інструменти можуть значно полегшити розробку проєкту або ж збільшити ефективність його роботи.

3.1. Стислий опис мови Rust

Rust є сучасною мовою програмування, розробленою для забезпечення безпеки пам’яті та високої продуктивності. У неї є багато особливостей, які роблять її привабливою для написання компіляторів:

- Безпека пам’яті.

  Rust має систему запозичень (англ. borrowing) і перевірки часу життя (англ. lifetime), щоб переконатися, що показники ніколи не показують недійсну пам’ять. Це зменшує ймовірність виникнення помилок, таких як сегментаційні помилки (англ. segmentation faults) і витоки пам’яті, які є поширеними проблемами в C і C++.

- Високий рівень продуктивності.

  Rust компілюється у виконуваний код, що дає швидкодію, подібну до C і C++. Компіляторам, які повинні ефективно обробляти великі кількості даних і виконувати складні алгоритми, це дуже важливо.

- Прості та зрозумілі макроси.

  Розробка складних програм, таких як компілятори, стає простішою завдяки макросам Rust, які дозволяють створювати повторно використовувані фрагменти коду. Вони уникають надмірної деталізації коду та дозволяють мові розширюватися.

- Екосистема та ресурси.

  Rust має розвинену екосистему з багатьма бібліотеками та інструментами, включаючи Cargo (систему управління пакетами та збірками). Це полегшує процеси створення, тестування та розгортання програм.

3.2. Cargo

Cargo – це офіційний менеджер пакетів та інструмент збірки для Rust, спеціально розроблений, щоб допомогти розробникам ефективно керувати своїми Rust-проєктами. Cargo спрощує ряд завдань, таких як контроль залежностей (які в Rust називаються «crates»), компіляція та тестування коду, а також розповсюдження пакетів через crates.io, офіційного реєстру пакетів Rust.

3.3. LALRPOP

LALRPOP – це фреймворк генератора синтаксичних аналізаторів Rust, основною метою якого є те, щоб його було легко використовувати. Користувач має можливість реалізовувати компактні, DRY, читабельні граматики. LALRPOP пропонує низку переваг:

- Інформативні та однозначні повідомлення про помилки в разі збою конструктора синтаксичного аналізатора.
- Макроси, що дозволяють витягувати загальні частини вашої граматики. Це означає, що ви можете відмовитися від простого повторення типу Id* і використовувати такі речі, як Comma<Id> для списку ідентифікаторів, розділених комами.
- Крім того, макроси можуть створювати підмножини, тому можна легко використовувати такі функції, як *Expr<"all">*, щоб показати повний діапазон виразів, або *Expr<"if">*, щоб показати підмножину виразів, які можуть з’явитися у виразі *if*.
- Підтримується такі оператори, як * і ?.
- Виведення типів, що дозволяє часто опускати нетермінальні типи.
- LALRPOP автоматично використовує LR(1).

3.4. Clap

Clap (Command Line Argument Parser) – це бібліотека для обробки командного рядка в мові Rust. Вона забезпечує простий і зручний спосіб

створення командних інтерфейсів користувача (CLI), дозволяючи розробникам легко додавати та обробляти аргументи командного рядка у своїх програмах. Бібліотека *Clap* пропонує наступні переваги:

- Простота використання: інтуїтивний інтерфейс для визначення та обробки аргументів командного рядка.
- Гнучкість: підтримка підкоманд, опціональних і позиційних аргументів.
- Автоматичне генерування документації: *Clap* автоматично створює довідку та повідомлення про помилки.
- Безпека та надійність: допомагає уникнути поширених помилок при обробці аргументів командного рядка.

Використання Clap у даному проєкті спрощує взаємодію з користувачем через командний рядок, забезпечуючи зручний і зрозумілий інтерфейс для виконання різних операцій.

Цей розділ присвячений вибору інструментів розробки. Кожна технологія, обрана для цієї роботи, коротко описана, а також перераховані її переваги порівняно з іншими аналогічними інструментами розробки.

4. ПРОЄКТУВАННЯ ТА РОЗРОБКА

Майбутній системі була обрана назва «ADL», що є скороченням від перекладу Адресної мови, тобто – Address Language.

Декомпозиція задачі на окремі етапи та визначення компонентів системи були першими кроками в розробці компілятора та інтерпретатора Адресної мови програмування. Перед усім необхідно було розробити граматику для мови, а після того вирішено поділити всю систему на окремі компоненти, кожен з яких виконує певну стадію компіляції або інтерпретації коду.

Отже, проєкт ADL було вирішено поділити на такі основні компоненти:

- Лексичний аналіз (назва модуля – *lexer*).
- Синтаксичний аналіз (назва модуля – *parser*).

- Генерація байт-коду (назва модуля – *codegen*).
- Віртуальна машина (назва модуля – *vm*).
- Інтерпретація (назва модуля – *interpreter*).
- Робота з примітивними типами даних (назва модуля – *value*).
- Утиліти та допоміжні функції (назва модуля – *common*).

Далі надано короткі коментарі та пояснення щодо кожного з компонентів.

lexer

Лексичний аналізатор ділить вихідний код на токени або лексеми. Ключові слова, ідентифікатори, літерали, оператори, розділові знаки та інші елементи мови програмування є лексеми. Лексичний аналізатор перетворює текст програми в послідовність токенів, які синтаксичний аналізатор може використовувати для побудови синтаксичного дерева.

parser

Синтаксичний аналізатор приймає на вхід послідовність токенів, створених лексичним аналізатором, і будує на їх основі абстрактне синтаксичне дерево.

codegen

Генератор байт-коду перетворює синтаксичне дерево у проміжний байт-код. Байт-код є низькорівневим представленням програми, яке може бути виконане віртуальною машиною.

vm

Віртуальна машина відповідає за виконання байт-коду. Віртуальна машина містить механізм управління адресами змінних і стек для зберігання проміжних результатів виконання. Для більш точного опису його функцій цей компонент можна назвати «виконавцем байт-коду».

interpreter

Інтерпретатор забезпечує виконання програми в режимі реального часу, транслюючи кожен вузол абстрактного синтаксичного дерева у відповідні дії.

Інтерпретатор включає в себе механізм управління адресами змінних для виконання операцій над даними та управління пам’яттю.

value

Важливим компонентом системи є модуль value, який відповідає за визначення та управління основними типами даних, які використовуються в Адресній мові програмування. Його основні обов’язки включають:

- Визначення типів даних: модуль підтримує різні типи даних, включаючи рядки, цілі числа, нульові значення та логічні значення. Зазначені типи даних використовуються під час виконання програм на віртуальній машині та представлення значень у байт-коді.
- Операції над значеннями: модуль підтримує арифметичні та логічні операції, а також операції порівняння для основних типів даних.

common

У модулі *common* доступні загальні утиліти та допоміжні функції, які підтримують інші компоненти системи. Його основні функції включають:

- Функції загального призначення: модуль надає можливості для роботи з структурою *Location* та інші утиліти, які використовуються в інших модулях системи.
- Читання та запис файлів: *common* включає функції для роботи з файлами, такі як читання вихідного коду з файлів та запис результатів роботи системи у файли.

Усі частини системи працюють разом, щоб утворити єдину цілісну систему. Лексичний аналізатор розбиває вихідний код на токени, які передаються синтаксичному аналізатору для побудови синтаксичного дерева. Генератор байт-коду перетворює синтаксичне дерево у байт-код, який виконується віртуальною машиною, або ж інтерпретатор перетворює абстрактне синтаксичне дерево у відповідні йому дії.

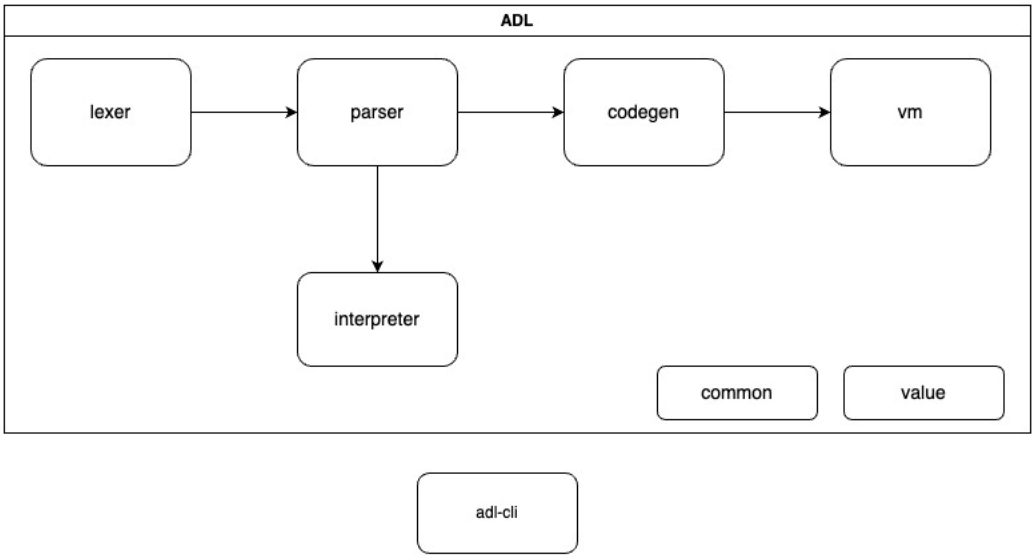


Рисунок 4.1. Архітектура системи ADL

У подальших підрозділах 4.1–4.6 детально описано розробку основних компонентів.

4.1. Граматика

4.1.1. Поняття граматики, LR(1)

Основою для побудови, аналізу та трансляції мов програмування є граматика, одна з основних теорій формальних мов і автоматів. Граматика дозволяє формалізувати синтаксис мови, розробляючи правила, що визначають правильні послідовності символів у мові програмування.

Для Адресної мови програмування було обрано LR(1) граматику. Вона використовується для створення синтаксичних аналізаторів з однією одиницею передбачення, розпізнаючи граматику зліва направо. Лівий розбір (L), праве найглибше введення (R) і один символ передбачення (1) використовуються аналізаторами LR(1) для розбору вхідного потоку символів.

Переваги використання LR(1) граматики включають:

- Ефективність: синтаксичні аналізатори, побудовані на основі LR(1) граматик, мають високу продуктивність і можуть ефективно обробляти великі обсяги даних.

- Визначеність: LR(1) аналізатори запобігають неоднозначності, оскільки кожне рішення приймається на основі одного символу передбачення та поточного стану.

Після вибору типу граматики, безпосередньо переходимо до розробки її правил для Адресної мови програмування.

4.1.2. Граматика Адресної мови програмування

Граматика для Адресної мови програмування визначається наступними правилами:

1. Algorithm, FileLine, LabelsDeclaration, Labels

```
<Algorithm> ::= <FileLine> { <FileLine> }

<FileLine> ::= <LabelsDeclaration> <Statements> ("END_OF_FILE" | "NEW_LINE")

<LabelsDeclaration> ::= <Labels> "..."

<Labels> ::= { <Identifier> "," } <Identifier>?
```

Лістинг 4.1.2.1. Визначення нетерміналів Algorithm, FileLine, LabelsDeclaration, Labels

Граматика починається з основного блоку <*Algorithm*>, який визначає програму як послідовність рядків файлу (<*FileLine*>). Рядок файлу складається з декларації міток (<*LabelsDeclaration*>) та тверджень (<*Statements*>), завершених кінцем файлу або новим рядком. Декларація міток містить список міток, розділених комами, з обов’язковим термінальним символом «...». Мітки (<*Labels*>) є множиною ідентифікаторів, розділених комами, також може бути пустою множиною.

Statements

```
<Statements> ::= { <SimpleStatement> ";" }
                 | <OneLineStatement>

<SimpleStatement> ::= <Expression> "=" <Expression>
                    | <Expression> "<=>" <Expression>
                    | <Expression> "=>" <Expression>
                    | <Expression>

<OneLineStatement> ::= <CallSubProgram>
                     | <UnconditionalJump>
                     | <Loop>
                     | <Predicate>
                     | "!"
                     | "RETURN"
```

Лістинг 4.1.2.2 Визначення нетерміналів Statements, SimpleStatement, OneLineStatement

Цей фрагмент граматики описує твердження. Твердження включають деяку множину простих тверджень (<*SimpleStatement*>), розділених крапкою з комою, або одне однорядкове (складне) твердження (<*OneLineStatement*>). Просте твердження включає присвоєння, обмін або обчислення виразу <Expression>, а однорядкове твердження може бути викликом підпрограми (<*CallSubProgram*>), безумовним переходом (<*UnconditionalJump*>), формулою циклювання (<*Loop*>), предикатною формулою (<*Predicate*>), формулою безумовного зупину ("!") або формулою відносного зупину ("RETURN").

```
<CallSubProgram> ::= "SUB_PROGRAM" <Identifier> "{" <Parameters> "}"
<Identifier>?


<Predicate> ::= "PREDICATE" "{" <ExpressionPrecedence8> "}"
<Statements> "|" <Statements>


<Loop> ::= "LOOP" "{" <Expression> "," <Expression> "," <Expression>
"=>" <Expression> "}" <Identifier> <Identifier>?


<UnconditionalJump> ::= "@" <Identifier>
```

Лістинг 4.1.2.3. Визначення нетерміналів CallSubProgram, SimpleStatement, OneLineStatement

Виклик підпрограми містить ідентифікатор підпрограми, параметри, укладені в фігурні дужки і ідентифікатор мітки, для переходу після її виконання. Предикатна формула визначає умовний оператор з виразом, який перевіряється, та двома наборами тверджень для кожного з випадків істинності. Формула циклювання складається з початкового значення, кроку, фінального значення або умови, та два ідентифікатори для маркування кінця циклу та мітку, на яку перейти після його роботи, а безумовний перехід – з ідентифікатору для переходу до вказаної мітки.

Expressions

```
<Expression> ::= <ExpressionPrecedence8>

<ExpressionPrecedence8> ::= <ExpressionPrecedence8> "OR" <ExpressionPrecedence7>
                          | <ExpressionPrecedence7>

<ExpressionPrecedence7> ::= <ExpressionPrecedence7> "AND" <ExpressionPrecedence6>
                          | <ExpressionPrecedence6>

<ExpressionPrecedence6> ::= <ExpressionPrecedence6> "==" <ExpressionPrecedence5>
                          | <ExpressionPrecedence6> "!=" <ExpressionPrecedence5>
                          | <ExpressionPrecedence6> "<" <ExpressionPrecedence5>
                          | <ExpressionPrecedence5>

<ExpressionPrecedence5> ::= <ExpressionPrecedence5> "+" <ExpressionPrecedence4>
                          | <ExpressionPrecedence5> "-" <ExpressionPrecedence4>
                          | <ExpressionPrecedence4>

<ExpressionPrecedence4> ::= <ExpressionPrecedence4> "*" <ExpressionPrecedence3>
                          | <ExpressionPrecedence4> "/" <ExpressionPrecedence3>
                          | <ExpressionPrecedence3>

<ExpressionPrecedence3> ::= <ExpressionPrecedence2>

<ExpressionPrecedence2> ::= "NOT" <ExpressionPrecedence2>
                          | "'" <ExpressionPrecedence2>
                          | <ExpressionPrecedence1>

<ExpressionPrecedence1> ::= <LiteralExpression>
                          | <List>
                          | <FunctionCall>
                          | <MultipleDereference>
                          | <Variable>
                          | "(" <Expression> ")"

<LiteralExpression> ::= <BoolLiteral>
                      | <IntLiteral>
                      | <StringLiteral>
                      | <FloatLiteral>
                      | <NullLiteral>
```

Лістинг 4.1.2.4. Визначення нетерміналів Expression різного пріоритету

Цей фрагмент граматики керує різними рівням пріоритету для визначення правил синтаксичного аналізу виразів у Адресній мові програмування. Вони пронумеровані від 0 до 8, і чим менший номер, тим більша пріоритетність виразу. Наприклад, оператор «OR» має найнижчий пріоритет (8), тому вони мають здатність об’єднувати вирази з вищим рівнем пріоритету логічних операцій типу «AND», які мають вищий пріоритет, ніж операції типу «OR», але нижчий, ніж оператори порівняння.

"==", "!=", "<" мають пріоритет 6 . Вони об’єднують вирази з рівнем пріоритету арифметичних операцій додавання та віднімання. Оператори додавання та віднімання мають вищий пріоритет, ніж оператори порівняння, але нижчий, ніж оператори множення та ділення. Хоча оператори множення та ділення мають більший пріоритет, ніж оператори додавання та віднімання, оператори множення та ділення мають нижчий пріоритет, ніж унарні оператори.

Унарні оператори, такі як розіменування та заперечення, мають найвищий пріоритет. Відповідно інші вирази в залежності від їх номеру мають певну пріоритетність.

Цей порядок виконання операцій дозволяє граматиці коректно розставляти дужки та визначати пріоритети виразів, що є важливим для точного синтаксичного аналізу та виконання програм.

Literals, Function, List …

```
<FunctionCall> ::= "IDENTIFIER" "{" <Parameters> "}"

<MultipleDereference> ::= "DEREF" "{" <Expression> "," <Expression> "}"

<Parameters> ::= { <Expression> "," }

<Variable> ::= <Identifier>

<List> ::= "[" <Parameters> "]"

<NullLiteral> ::=  ...

<IntLiteral> ::= ...

<FloatLiteral> ::= ...

<BoolLiteral> ::= ...

<StringLiteral> ::= ...

<Identifier> ::= ...
```

Лістинг 4.1.2.5. Визначення базових нетермналів

В цьому фрагменті описані правила, що визначають базові елементи мови програмування, такі як виклики: функцій, розіменування, параметри, змінні, списки та літерали.

4.2. Лексичний аналіз

Лексичний аналіз є важливою складовою процесу проєктування компілятора: відіграє важливу роль у інтерфейсі компілятора, де відповідає за розбиття вихідного коду на менші, більш керовані одиниці, які називаються лексемами. Вони слугують будівельними блоками для наступного етапу проєктування компілятора, який називається синтаксичним аналізом.

Рисунок 4.2.1. Роль лексичного аналізу в роботі компілятора

Основною метою лексичного аналізу є виконання попереднього аналізу вихідного коду, перевірка на наявність будь-яких лексичних помилок або проблем, перш ніж вони можуть спричинити більш серйозні проблеми на більш пізніх етапах проектування компілятора.

Фаза лексичного аналізу передбачає написання лексичного аналізатора (англ. Lexer), також відомого як сканер (англ. Scanner), який сканує вхідний текст і розпізнає лексеми.

Тому, оскільки граматика була вже розроблена, то наступним кроком було створення компоненту, який відповідає за лексичний аналіз. Він включає в себе такі основні елементи:

- Модуль *token (визначення токенів)*.
- Модуль *lexer (реалізація лексичного аналізатору)*.
- Модуль *error (обробка помилок)*.

Тому відповідно до послідовності елементів списку, зазначеного вище, першою задачею було розробити модуль token.

4.2.1. Визначення токенів

Модуль token включає в себе лише тип даних, що відповідає за представлення абстракції лексеми, та допоміжних функцій для форматування.

Перелічуваний тип даних *TokenKind* визначає всі можливі лексеми, які лексичний аналізатор може ідентифікувати у вхідному тексті. Кожен варіант TokenKind представляє певний тип токенів, що інкапсулює різні синтаксичні

елементи Адресної мови. На лістингу 4.2.1.1 можна побачити у який спосіб це було реалізовано.

```
#[derive(Clone, Debug, PartialEq)]
pub enum TokenKind {
    // Literals
    Identifier(String), IntegerLiteral(i64), FloatLiteral(f64), StringLiteral(String),
    True, False, Null,

    // Structural Tokens
    NewLine, EndOfFile, LeftParenthesis, RightParenthesis, LeftSquareBracket,
    RightSquareBracket, Colon, Comma, Semicolon,
    Slash, VerticalBar, LeftCurlyBrace,
    RightCurlyBrace, Ellipsis,

    // Operators
    Multiply, Plus, Minus, Ampersand,
    LessThan, GreaterThan, Equal, Percent,
    EqualEqual, NotEqual, LessThanEqual, GreaterThanEqual,
    Send, Apostrophe,  Exchange,
    At, Bang,

    // Keywords:
    Return, And, Del, Not,
    Or,Loop, SubProgram, Predicate,
    Replace, Deref,
}
```

Лістинг 4.2.1.1. Імплементація переліку TokenKind

Кожна лексема представлена варіантом перелічуваного типу даних *TokenKind* та може містити додаткову інформацію, якщо це необхідно. Наприклад, варіанти *Identifier*, *IntegerLiteral*, *FloatLiteral* і *StringLiteral* містять відповідні значення, що дозволяє сканеру зберігати фактичний текст або числове значення з вхідного коду.

Таке повне визначення лексем гарантує, що лексичний аналізатор може точно і ефективно токенізувати весь синтаксис Адресної мови, уможливлюючи подальший синтаксичний і семантичний аналіз на наступних етапах процесу компіляції.

4.2.2. Реалізація модуля lexer

Модуль *lexer* відповідає за перетворення вхідного тексту на послідовність токенів, які використовуються для подальшого синтаксичного та семантичного аналізу.

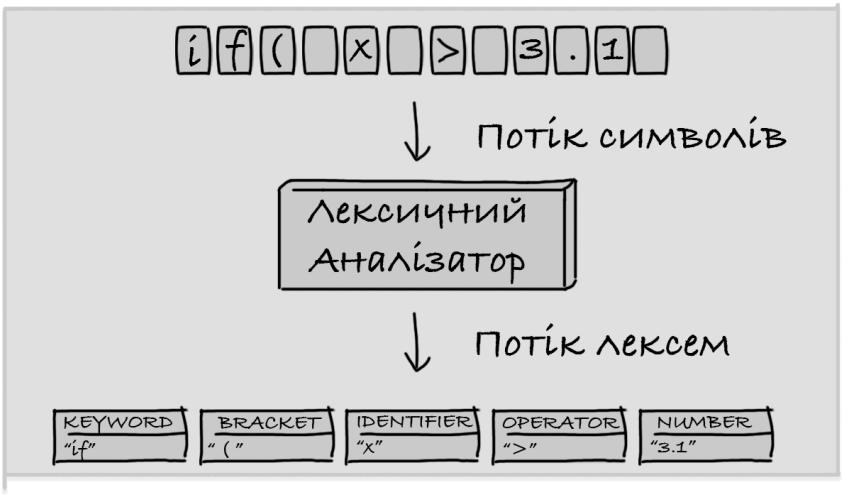


Рисунок 4.2.2.1. Приклад роботи лексичного аналізатору

Тому першим кроком було визначення структури *Lexer*.

```
pub struct Lexer<'a> {
    input: &'a str,
    char_indices: Peekable<CharIndices<'a>>,
    current_index: usize,
    current_char: Option<(usize, char)>,
    location: Location,
    is_eof: bool,
    skipped_chars: Queue<Option<(usize, char)>>,
}
```

Лістинг 4.2.2.1. Визначення структури Lexer

Структура *Lexer* містить наступні поля:

- *input*: вхідний текст, який потрібно токенізувати.
- *char_indices*: ітератор, що дозволяє переглядати множину з кортежів з індексу та символу у вхідному рядку.
- *current_index*: поточний індекс у вхідному рядку.
- *current_char*: поточний символ, що обробляється.
- *location*: поточна позиція у вхідному рядку для звітування про помилки.

- *is_eof*: індикатор, що вказує, чи досягнуто кінця файлу.

Структура *Lexer* реалізує трейт (англ. Trait) Iterator, що дозволяє її використання у циклах як ітератор, повертаючий токени або помилки. Це забезпечує зручний спосіб обробки вхідного тексту по одному токену за раз, дозволяючи сканеру створювати відповідні лексеми для кожного розпізнаного елемента.

Насправді, створюється не токен, а трохи складніший тип під назвою *Span*.

```
pub type Span = (Location, TokenKind, Location);
```

Лістинг 4.2.2.2. Визначення типу Span

Тип *Span* є кортежем, який складається з трьох елементів: початкової позиції (типу *Location*), виду токена (типу *TokenKind*) і кінцевої позиції (типу *Location*).

Span використовується для зберігання інформації про лексеми, включаючи їх тип і розташування у вхідному тексті. Це дозволяє точно відслідковувати, де кожен токен починається і закінчується, що є критичним для коректного синтаксичного аналізу та генерації коректних повідомлень про помилки.

Після того як були визначені основні типи і структури, було розроблено такий алгоритм:

Ініціалізація

Спочатку створюється екземпляр сканеру, який зберігає оригінальний вхідний текст. Також він зберігає поточну позицію, символ, що обробляється, та іншу інформацію, необхідну для моніторингу процесу токенізації.

Пропуск пробілів та коментарів

Наступним кроком є видалення всіх коментарів і пробілів у вхідному тексті. Це можна досягти за допомогою методу *skip_whitespace_and_comments*, який пропускає всі пробіли та коментарі, доки не зустрінеться символ, який може бути частиною токена.

Читання поточного символу

Метод *peek_char* використовується для отримання поточного символу без зміни позиції. Це дозволяє визначити або припустити тип токена, до якого належить поточний символ, без переходу до наступного символу.

Обробка різних типів токенів

Залежно від поточного символу, лексичний аналізатор визначає, який тип токена потрібно згенерувати. Це може бути ключове слово, ідентифікатор, числовий літерал, рядковий літерал або символ.

- Ключові слова та ідентифікатори: метод *next_keyword_or_identifier_literal* зчитує послідовність буквено-цифрових символів або підкреслень та визначає, чи є ця послідовність ключовим словом чи ідентифікатором.
- Числові літерали: метод *determine_number* обробляє послідовність цифр та визначає, чи є це цілочисельним чи дійсним (з плаваючою комою) літералом. Якщо це дійсне число, викликається метод *next_float* для завершення обробки.
- Рядкові літерали: метод *next_quoted_str_literal* обробляє рядкові літерали, зчитуючи символи до зустрічі закриваючої лапки.
- Символи: методи *next_symbol_token*, *next_double_symbol_token* та *next_double_symbol_token* обробляють окремі символи, пари символів та трійки символів, які можуть представляти оператори або інші спеціальні символи.

Створення токенів

Після визначення типу токена створюється та повертається відповідний кортеж типу *Span*, що був згаданий раніше, з інформацією про тип лексеми та розташування у вхідному тексті.

Обробка кінця файлу

Оскільки структура Lexer наслідує інтерфейс Iterator, то при кожному виклику методу *next*, описані вище кроки будуть повторюватися поки не зустрінеться останній символ вхідного тексту, і тоді лексичний аналізатор повертає спеціальний токен *EndOfFile*, який сигналізує про завершення токенізації.

Описаний алгоритм забезпечує ефективну та коректну токенізацію оригінального коду Адресної мови програмування, готуючи його для подальшого синтаксичного та семантичного аналізу.

4.2.3. Обробка помилок

Обробка помилок є важливою складовою роботи синтаксичного аналізатору, оскільки вона дозволяє виявляти та повідомляти про лексичні помилки у вхідному коді. У даному сканері визначено кілька типів помилок, які можуть виникати під час токенізації. Для представлення помилок використовується перелік *LexError*.

```
pub enum LexError {
    Unexpected(Location, char),
    UnterminatedStringLiteral(Location),
    FloatFormatError(Location, String),
    IntegerFormatError(Location, String),
}
```

Лістинг 4.2.3.1. Визначення переліку LexError

Нижче наведено пояснення типів помилок, які можуть виникнути під час токенізації:

- *Unexpected(Location, char)*: ця помилка виникає, коли лексичний аналізатор зустрічає несподіваний символ, який не може бути частиною жодного токена. Наприклад, якщо у вхідному коді з'являється незнайомий символ, сканер генерує цю помилку, вказуючи місце, де сталася помилка, та сам символ.
- *UnterminatedStringLiteral(Location)*: ця помилка виникає, коли рядковий літерал не завершено. Якщо сканер зустрічає початок рядкового літералу, але не знаходить закриваючої лапки, він генерує цю помилку, вказуючи місце, де починається незавершений рядковий літерал.
- *FloatFormatError(Location, String)*: ця помилка виникає, коли лексичний аналізатор не може коректно інтерпретувати число з плаваючою точкою. Наприклад, якщо у вхідному коді число з плаваючою точкою має неправильний формат, сканер генерує цю помилку, вказуючи місце помилки та саме некоректне значення.
- *IntegerFormatError(Location, String)*: ця помилка виникає, коли сканер не може коректно інтерпретувати ціле число. Наприклад, якщо у вхідному коді

ціле число має неправильний формат, сканер генерує цю помилку, вказуючи місце помилки та саме некоректне значення.

Завдяки цим механізмам обробки помилок модуль *lexer* забезпечує надійну та ефективну токенізацію вхідного коду.

4.3. Синтаксичний аналіз

Якщо лексичний аналіз розбиває вхідні дані на лексеми, то метою синтаксичного аналізу (також відомого як синтаксичний розбір) є рекомбінація цих лексем в деяку сутність, що відображає структуру тексту. Цією сутністю зазвичай є структура даних, яка називається абстрактним синтаксичним деревом тексту (англ. Abstract syntax tree). Як видно з назви, це деревоподібна структура.

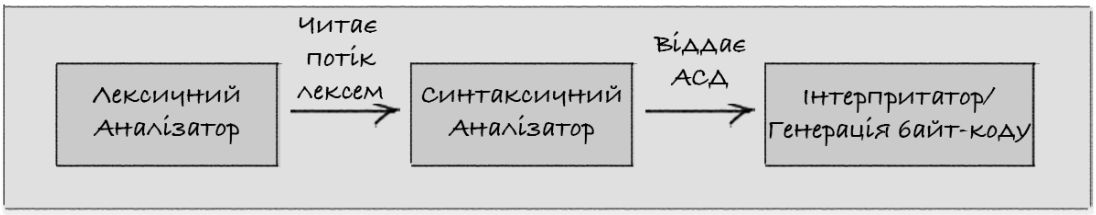


Рисунок 4.3.1. Роль синтаксичного аналізу у роботі компілятора

В даній роботі реалізація синтаксичного аналізу передбачає написання кількох компонентів, які разом формують процес синтаксичного розбору і генерації абстрактного синтаксичного дерева. Ось короткий опис кожного з цих компонентів:

- Модуль *ast*: визначає структури для представлення абстрактного синтаксичного дерева.
  - Модуль серіалізації та десеріалізації: забезпечує серіалізацію і десеріалізацію абстрактного синтаксичного дерева у текстовий формат і назад.
  - Модуль visitor: реалізує шаблон проєктування «Відвідувач» (англ. Visitor) для операцій над різними елементами АСД.
- Модуль *parser*: визначає граматику мови програмування та правила для кожного конструкта мови.

- Модуль *lib* : інтегрує всі інші модулі і забезпечує основний інтерфейс для синтаксичного аналізу.

4.3.1. Розробка абстрактного синтаксичного дерева

Абстрактне синтаксичне дерево (АСД) – це ієрархічна структура, яка точно відображає розташування елементів у вихідному коді, з урахуванням синтаксису Адресної мови. Воно вважається абстрактним, оскільки не зберігає в своїй структурі всі специфічні елементи синтаксису вихідного коду, такі як дужки або коми. Дерево надає пріоритет представленню логічної структури коду, що має вирішальне значення для подальшої обробки.

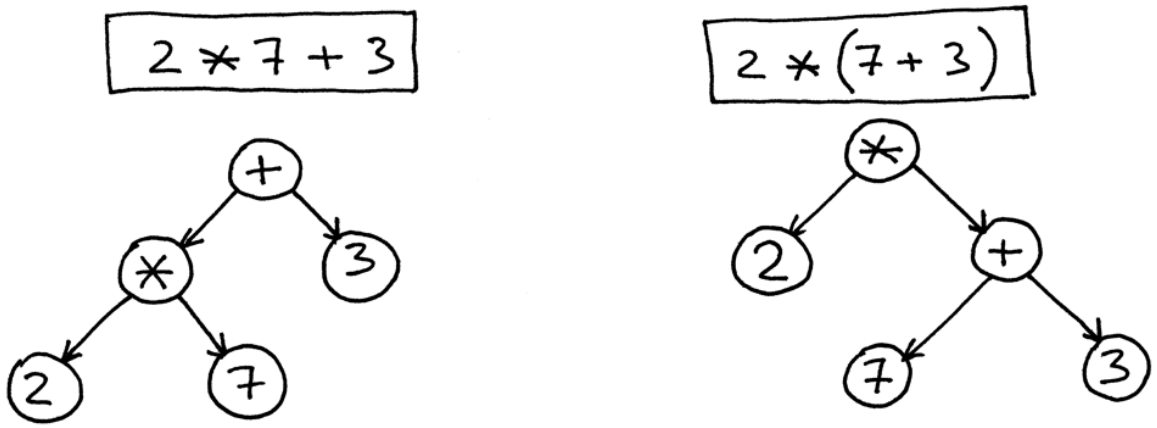


Рисунок 4.3.1.1 Приклад абстрактного синтаксичного дерева

Модуль *ast* визначає структури для представлення абстрактного синтаксичного дерева відповідно до вищевказаної граматики Адресної мови. Система типів в Rust з їх перелічуваним типом даних дуже зручна для визначення подібних структур, що дозволяє легко представити різні синтаксичні елементи мови програмування.

Спочатку в цілях створення точної діагностики помилок було визначено тип-обгортку *Located*.

```
pub struct Located<T = ()> {
pub l_location: Location,
pub r_location: Location,
pub node: T,
}
```

Лістинг 4.3.1.1. Визначення типу Located

Головне призначення структури *Located* полягає в тому, щоб зберігати інформацію про місцезнаходження кожної одиниці граматики у оригінальному коді відповідно до вузла АСД.

Далі було реалізовано структури і перелічувані типи даних, що визначають абстрактне синтаксичне дерево. Ось основні з них:

- *Algorithm*: представляє весь алгоритм, який складається зі списку типу *FileLine*.
- *FileLine*: представляє окремий рядок файлу, який може містити мітки та твердження.
- *Statements*: визначає типи тверджень, які можуть бути в одному рядку файлу. Може містити або однорядковий оператор (*OneLineStatement*), або вектор простих операторів (*SimpleStatements*).
- *OneLineStatement*: тип, що представляє однорядкове твердження.
- *OneLineStatementKind*: визначає різні види однорядкових операторів, такі як підпрограма (*SubProgram*), цикл (*Loop*), умовний оператор (*Predicate*), вихід (*Exit*), повернення (*Return*) і безумовний перехід (*UnconditionalJump*).
- *SimpleStatementKind*: визначає типи простих операторів, таких як імпорт (*Import*), звільнення комірки пам'яті (*Del*), присвоєння (*Assign*), засилання (*Send*), обмін (*Exchange*) і вираз (*Expression*).
- *Path*: представляє шлях до імпортованого модуля.
- *Expression*: тип, що представляє вираз.
- *ExpressionKind*: визначає різні типи виразів, такі як пуста множина (*Null*), числові (*Float, Int*), булеві (*Bool*), строкові (*String*), змінні (*Var*), списки (*List*), виклики функцій (*Call*), унарні оператори (*UnaryOp*) та бінарні оператори (*BinaryOp*).

Ці структури забезпечують гнучкий та зручний спосіб представлення різних синтаксичних елементів мови програмування в деревоподібній формі, що полегшує подальшу обробку та аналіз коду. На лістингу 4.3.1.2 продемонстровано реалізацію частини з переліками, що були пояснені раніше.

```
pub enum FileLine {
    Line {
        labels: Vec<String>,
        statements: Statements,
    },
}

pub enum Statements {
    OneLineStatement(OneLineStatement),
    SimpleStatements(Vec<SimpleStatement>),
}
```

Лістинг 4.3.1.2. Визначення перерахувань FileLine, Statements

Для більш простого обходу АСД було вирішено використати шаблон проєктування «Відвідувач». Він є одним з поведінкових шаблонів проєктування та використовується, коли потрібно виконати операцію над групою однотипних об’єктів. За допомогою цього шаблону вдалося перенести логіку роботи з структури *AST* на поведінку іншої структури. Даний шаблон складається з двох частин:

- метод *visit*, який реалізується структурою «Відвідувачем» і викликається для кожного елементу в основній структурі даних;
- відвідувані класи надають методи *accept*, які приймають «Відвідувача» у вигляді аргументу.

Таким чином було реалізовано інтерфейс (в контексті мови Rust аналогом є Trait) *Visitor*. Для кожного типу вузла в інтерфейсі були створені відповідні методи. Аналогічно для кожної структури, що представляє деякий вузол, було додано по методу *accept* (див. *Лістинг 4.3.1.3*.).

```
impl SimpleStatement {
    pub fn accept<V: Visitor>(&self, visitor: &mut V) {
        visitor.visit_simple_statement(self);
    }
}
```

Лістинг 4.3.1.3. Визначення методу accept

Додатково модуль *ast* також включає важливий підмодуль серіалізації та десеріалізації *serializer*.

Модуль серіалізації та десеріалізації забезпечує можливість перетворення АСД у текстовий JSON формат і в зворотному напрямку.

4.3.2. Розробка синтаксичного аналізатору

Функція аналізатора синтаксису полягає в тому, щоб перетворити послідовність токенів, створених сканером, у абстрактне синтаксичне дерево, яка відображає синтаксис вихідного коду. У цьому проєкті був використаний фреймворк LALRPOP для генерації синтаксичного аналізатору.

LALRPOP – це генератор синтаксичних аналізаторів для Rust, який дозволяє описувати граматики та генерувати синтаксичні аналізатори на їх основі. LALRPOP підтримує контекстно-вільні граматики та базується на алгоритмі LALR(1), що дозволяє йому ефективно обробляти синтаксичні конструкції різної складності.

LALRPOP пропонує опис синтаксису граматик, який дуже схожий на класичну форму Бекуса-Наура (BNF), що значно спрощує процес розробки.

На основі описаної граматики LALRPOP генерує синтаксичний аналізатор, який здатний перетворювати послідовність токенів у абстрактне синтаксичне дерево з клієнтського коду .

Одним із плюсів використання LALRPOP є те, що він автоматично створює повідомлення про помилки. Оскільки розробнику не потрібно самостійно визначати та обробляти синтаксичні помилки, це значно полегшує процес налагодження.

Нижче наведено приклад граматики Адресної мови, описаної за допомогою синтаксису LALRPOP:

```
MultipleDereference: Expression = {
        <l_location:@L>  "Deref" "{" <expression:Expression>  "," <r: Expression> "}"
<r_location:@L>  => Expression
        {
        l_location,
        r_location,
        node: ExpressionKind::UnaryOp{op: UnaryOp::MultipleDereference(Box::new(r)),
expr: Box::new(expression)},
        },
}
```

Лістинг 4.3.2.1. Приклад синтаксису LALRPOP

4.4. Обчислення АСД

Розробка компоненту для обчислення абстрактного синтаксичного дерева була здійснена з метою надання міжнародної премії Tony Sale Award демонстрації можливостей Адресної мови. Таким чином, процес виконання цього компонента був достатньо простим, щоб забезпечити швидку та ефективну демонстрацію основних функціональних можливостей.

Компонент, що відповідає за обчислення АСД призначений для інтерпретації та виконання програм, представлених у вигляді абстрактного синтаксичного дерева. Він виконує обчислення виразів, виконання циклів і умовних операторів, а також виклики підпрограм.

4.4.1. Визначення примітивних типів даних

Перед безпосередньою інтерпретацією АСД необхідно визначити примітивні типи даних, якими можна маніпулювати під час роботи програми. У Адресній мові програмування було визначено кілька примітивних типів даних, щоб забезпечити базову функціональність. Визначені примітивні типи включають:

Null: представляє відсутність значення.

Float: для чисел з плаваючою комою.

String: для текстових рядків.

Bool: для булевих значень (істина або хиба).

Int: для цілих чисел.

Function: для збереження вбудованих функцій.

Для цих типів були описані основні операції, такі як додавання, множення, порівняння та інші. Це дозволяє виконувати стандартні математичні та логічні операції між значеннями.

Для зручності програмування були реалізовані методи для конвертації та перевірки типів значень, такі як *type_of*, що дозволяє визначити тип значення в час виконання програми.

Таким чином, визначення примітивних типів даних і реалізація базових операцій забезпечують міцну основу для роботи з даними в Адресній мові програмування, що полегшує написання та відлагодження коду.

4.4.2. Управління пам’яттю та контекстом виконання

Для виконання абстрактного синтаксичного дерева Адресної мови було вирішено моделювати стан пам’яті множиною трійок, який включає в себе значення таких типів: ім’я змінної, адреса (номер комірки), та значення.

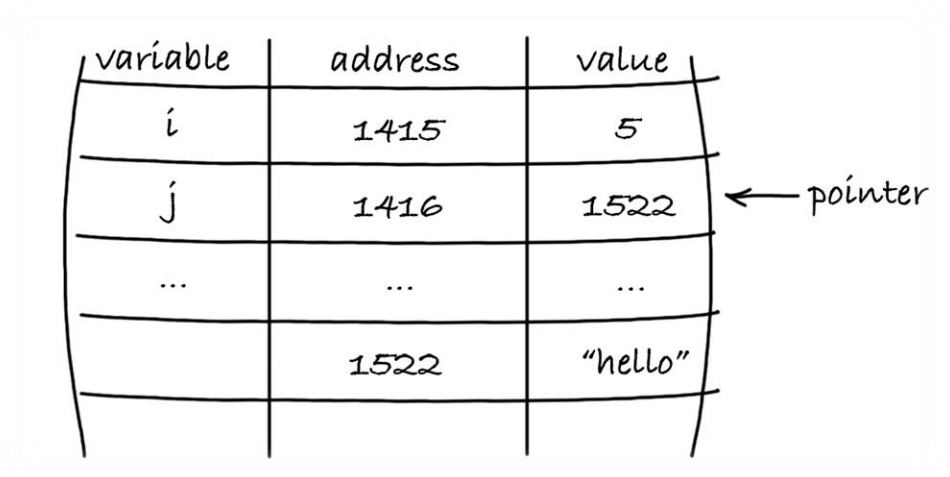


Рисунок 4.4.2.1 Ілюстрація емульованої пам’яті

Оскільки, очевидний підхід, в якому є необхідність ітерації по списку з кортежів з трьох елементів є досить складною операцією, то була обрана інша версія реалізації цієї абстракції.

Остаточна реалізація має таку конструкцію: повинно існувати дві геш-таблиці (англ. Hashmap). Перша зберігає відповідність імені змінної до адреси, на яку вона дивиться, а друга – відповідність адреси до безпосереднього значення.

Також до контексту виконання програми було додано зберігання вбудованих функцій і міток. Це було реалізовано за рахунок двох додаткових геш-таблиць, перша встановлює відповідність між назвою функції та її способом виклику і друга – між назвою мітки та номером стрічки коду, до якої треба перейти.

```
pub struct RuntimeContext {
    functions: HashMap<String, Value>,
    variable_addresses: HashMap<String, i64>,
    values_by_address: HashMap<i64, Value>,
    labels: HashMap<String, usize>,
}
```

Лістинг 4.4.2.1. Визначення структури RuntimeContext

Наступним кроком було реалізовано функції для керування станом виконання програми. Вони поділені на такі групи за областю їх застосування:

- Управління пам’яттю.
- Управління змінними.
- Управління функціями.
- Управління мітками.
- Управління пам’яттю

Обраний алгоритм виділення пам’яті базується на простому підході – пошуку першої вільної адреси в пам’яті. Цей підхід відомий як пошук першого

вільного блоку (англ. First-Fit Allocation). Він задовольняє умови для даної частини проєкту, оскільки має наступні переваги:

- Простота реалізації.
- Висока швидкість пошуку першої вільної адреси для невеликої кількості змінних.

Очевидно, що даний алгоритм має відчутні недоліки, а саме:

- З часом можуть утворюватися «дірки» в пам’яті, що призводить до неефективного використання пам’яті.
- Пошук вільної адреси може стати повільним для великої кількості змінних, оскільки алгоритм перебирає всі зайняті адреси.

Однак для цієї демонстраційної роботи ці недоліки можна не брати до уваги.

Наступним кроком було визначення функцій для звільнення певної адреси, читання даних відповідно до адреси, виділення адреси, виділення множини адресів для запису списку, запису даних за адресою.

Управління змінними

Інтерфейс для управління змінними включає до себе такий функціонал:

- Асоціація змінної з певною адресою.
- Отримання адреси, на яку посилається певна змінна.
- Видалення змінної.

Управління функціями

Для роботи з функціями визначені лише два методи: *get_function* і *add_function,* додавання вбудованої функції асоційованої з деяким ім’ям та отримання функції за її ім’ям відповідно.

Управління мітками

Робота з мітками відбувається аналогічно до управління вбудованими функціями, теж є два методи, що відповідають за процес додавання мітки

асоційованої з номером стрічки коду та отримання номеру стрічки коду за ім’ям мітки, що на неї вказує.

4.4.3. Інтерпретація

У реалізації інтерпретатора для Адресної мови програмування головну роль відіграє обхід абстрактного синтаксичного дерева (АСД) та інтерпретація його вузлів. Основна мета обходу АСД – виконання обчислень, що відповідають структурі та змісту програмного коду. У цьому розділі ми розглянемо, як відбувається обхід дерева в даному інтерпретаторі, та детально розглянемо обробку підпрограм і циклів.

Обхід АСД здійснюється за допомогою структури *Evaluator*, яка відповідає за інтерпретацію вузлів дерева. *Evaluator* містить АСД, поточний контекст виконання та індекс поточної стрічки коду.

```
pub struct Evaluator {
    lines: Vec<FileLine>,
    context: RuntimeContext,
    current_line: usize,
}
```

Лістинг 4.4.3.1. Визначення структури Evaluator

Основний метод, який здійснює обхід дерева, – це *eval*. Він реалізує основний цикл, у якому інтерпретуються вузли дерева по черзі. Цей метод також включає механізми для обробки міток та переходів між ними.

Основний цикл інтерпретації в методі *eval* виглядає наступним чином: спочатку витягуються та реєструються всі мітки, потім інтерпретуються вузли до моменту досягнення кінця списку стрічок коду (представлені в АСТ окремим вузлом) або до зустрічі інструкції завершення. Вузли *FileLine* та *Statements* обробляються методами *eval_file_line* та *eval_statements* відповідно, які в свою чергу викликають методи для обробки конкретних типів інструкцій.

Результати виконання інструкцій представлені в переліку *StatementResult*. Ця структура використовується для управління потоком виконання програми та включає кілька варіантів результату:

- *Continue*: продовжує виконання наступної інструкції.
- *FullStop*: завершує виконання всієї програми.
- *LocalStop*: завершує виконання поточної підпрограми або блоку.
- *JumpTo(usize)*: переходить до інструкції за заданим індексом.

Ця гнучка структура дозволяє легко реалізувати різні сценарії управління потоком виконання, зокрема цикли, умовні переходи та виклики підпрограм.

Однією з важливих частин інтерпретації є обробка циклів. У даній реалізації це здійснюється за допомогою методу *eval_loop*. Цей метод оцінює початкове значення, крок, умову завершення та ітератор циклу. Після ініціалізації відбувається виконання його тіла в циклі, який продовжується до досягнення умови завершення. Кожна ітерація циклу оновлює значення ітератора, після чого перевіряється умова завершення. У випадку істинності умови *Evaluator* переходить до рядка зазначеного в декларації формули циклювання міткою, а інакше – повертається до стрічки, що є початком тіла циклу.

```
    pub fn eval_loop(&mut self, statement: OneLineStatement) ->
Result<StatementResult, EvaluationError> {
        // Ініціалізація початкових значень та оцінка умов
        // Виконання тіла циклу
        // Оновлення ітератора та перевірка умови завершення
        // якщо умова true: перехід до початку тіла циклу
        // якшо умова false: перехід до стрічки, яка вказана в декларації циклу
    }
```

Лістинг 4.4.3.2. Демонстрація алгоритму роботи eval_loop

Ще однією важливою частиною є обробка підпрограм, яка здійснюється методом *eval_subprogram_call*. Цей метод здійснює пошук рядка, де починається підпрограма, за допомогою мітки. Потім відбувається оцінка виразів, що передаються як аргументи підпрограми. Після цього інтерпретатор виконує тіло підпрограми, а після завершення звільняються всі змінні, створені для підпрограми. Це забезпечує коректне управління пам’яттю та виконання підпрограм.

```
    pub fn eval_subprogram_call(&mut self, statement: OneLineStatement) ->
Result<StatementResult, EvaluationError> {
        // Пошук рядка підпрограми за міткою
        // Оцінка аргументів підпрограми
        // Виконання тіла підпрограми
        // Звільнення змінних після завершення підпрограми
    }
```

Лістинг 4.4.3.3. Демонстрація алгоритму роботи eval_subprogram_call

Серед складних інструкцій умовні вирази є останніми, вони позначені як вузли *Predicate* в АСД, обробляються в методі e*val_one_line_statement*. Спочатку оцінюється вираз умови. Якщо він повертає значення *true*, виконується один блок інструкцій, якщо *false* – інший. Це дозволяє реалізувати умовні переходи та набори інструкцій, що виконуються за певних умов.

```
OneLineStatementKind::Predicate {
                condition,
                if_true,
                if_false,
            } => {
                let cond = match self.eval_expression(condition.clone()) {
                    Ok(value) => match value {
                        Value::Bool(value) => value,
                        v => {
                            return Err(EvaluationError::RuntimeError(
                                condition.l_location,
                                condition.r_location,

RuntimeError::TypeError(Value::_raise_unexpected_type_error(
                                    vec![Type::Int, Type::Bool],
                                    &v,
                                )),
                            ))
                        }
                    },
                    Err(e) => return Err(e),
                };
                if cond {
                    self.eval_statements(*if_true)
                } else {
                    self.eval_statements(*if_false)
                }
            }
```

Лістинг 4.4.3.4. Інтерпретація умовних виразів

Таким чином, обхід абстрактного синтаксичного дерева та інтерпретація коду в даному інтерпретаторі реалізовано через рекурсивну обробку виразів та

інструкцій, що забезпечує гнучкість та простоту в підтримці та розширенні. Такий підхід дозволяє ефективно реалізовувати обробку циклів, підпрограм та інших конструкцій мови програмування, що демонструється в прикладах реалізації інтерпретатора.

4.4.4. Обробка помилок

Помилки в інтерпретаторі необхідні для виявлення та діагностики проблем у програмному коді під час його виконання. Вони дозволяють розробникам і користувачам інтерпретатора зрозуміти, що пішло не так, і швидко виправити помилки, які можуть бути синтаксичними, семантичними або виникати під час виконання. Ефективна обробка помилок також сприяє поліпшенню якості та надійності програмного забезпечення.

Перелік *RuntimeError* представляє помилки, які виникають під час виконання програми. Це можуть бути помилки доступу до пам’яті, арифметичні помилки, помилки типів та інші.

```
pub enum RuntimeError {
    NullReference,
    DivisionByZero,
    TypeError(ValueError),
    IndexOutOfBounds(usize, usize),
    VariableNotFound(String),
    LabelNotFound(String),
    LabelAlreadyRegistered(String, usize, usize),
    FunctionNotFound(String),
    InvalidArgument(String),
    FunctionCallError(String, String),
    InvalidArgumentsNumber(String, usize, usize),
}
```

Лістинг 4.4.4.1. Визначення RuntimeError

Кожен варіант *RuntimeError* містить опис специфічної проблеми:

- *NullReference*: ця помилка виникає, коли програма намагається звернутися до null-значення.
- *DivisionByZero*: арифметична помилка, яка виникає при діленні на нуль.

- *TypeError(ValueError)*: помилка типу, яка виникає, коли тип значення не відповідає очікуваному. ValueError містить додаткову інформацію про очікуваний та фактичний типи даних.
- *IndexOutOfBounds(usize, usize)*: помилка, яка виникає, коли індекс виходить за межі допустимого діапазону. Наприклад, спроба доступу до елемента масиву з індексом, який перевищує його розмір.
- *VariableNotFound(String)*: помилка, яка виникає, коли змінна з заданим ім’ям не знайдена у поточному контексті.
- *LabelNotFound(String)*: помилка, яка виникає, коли мітка з заданим ім’ям не знайдена у програмі.
- *FunctionNotFound(String)*: помилка, яка виникає, коли функція з заданим ім’ям не знайдена.
- *InvalidArgument(String)*: помилка, яка виникає, коли переданий аргумент є некоректним.
- *FunctionCallError(String, String)*: помилка, яка виникає під час виклику функції, коли виникає інша помилка (наприклад, внутрішня помилка функції).
- *InvalidArgumentsNumber(String, usize, usize)*: помилка, яка виникає, коли кількість переданих аргументів не відповідає очікуваній кількості.
- *LabelAlreadyRegistered(String, usize, usize)*: помилка, яка виникає, коли мітка вже зареєстрована на іншій стрічці коду.

Ці помилки не тільки допомагають визначити тип проблеми, але й надають корисну інформацію для користувача, щоб зрозуміти, що саме і де сталося.

Однією з ключових особливостей *RuntimeError* є його здатність форматувати помилки спеціально для користувача, надаючи чіткі та зрозумілі повідомлення (див. *Рисунок 4.4.4.1*.). Це досягається за допомогою реалізації трейтів *Display* і *Debug* для *RuntimeError*.

```
runtime error: Type error: Incompatible types for '+': (string: 4) and (int: 3)
  --> 3:1 .. 3:8
   |
3  | "4" + 3
   | ^^^^^^^
```

Лістинг 4.4.4.2. Приклад виводу помилки для користувача

Завдяки цим механізмам обробки помилок модуль *interpretator* забезпечує надійну інтерпретацію абстрактного синтаксичного дерева.

4.5. Байт-код генерація

Генерація байт-коду – це важливий етап у процесі компіляції програм, на якому вхідний код перетворюється в проміжний формат, придатний для виконання на віртуальній машині. Цей проміжний формат називається байт-кодом. Він являє собою набір інструкцій, які віртуальна машина може інтерпретувати і виконувати.

Байт-код – це проміжний машинно-незалежний код, який створюється компілятором з вихідного коду програми. Він складається з послідовності інструкцій, кожна з яких являє собою просту дію, яку віртуальна машина може виконати. Ці інструкції включають операції для виконання арифметичних обчислень, управління пам’яттю, роботи зі змінними, управління потоком виконання програми та інші.

Тому для розробки компоненту, що займається генерацією байт-коду першим кроком було визначення байт-код операцій.

4.5.1. Визначення байт-код операцій

Визначення байт-код операцій є ключовим етапом, оскільки ці операції визначають, які дії можуть бути виконані віртуальною машиною. У даній системі байт-код операції реалізовані у вигляді переліку *Bytecode*, де кожна операція представляє певну дію. Нижче наведено операції байт-коду, які були введені:

```
pub enum Bytecode {
    Halt, Return, Constant(Value), Not, And, Or,
    Negate, Add, Sub, Mul, Div, Mod, Equal, NotEqual,
    Greater, Less, Pop, Label(String), Jump(usize),
    JumpIfFalse(usize), Deref, MulDeref, StoreVar(String),
    LoadVar(String), Store, Alloc, AllocMany(usize),
    Dup, StoreAddr, BindAddr(String), FreeAddr,
    CallBuiltin(String, usize), CallSubProgram(usize, usize),
    PushScope, PopScope, Swap,
}
```

Лістинг 4.5.1.1. Визначення байт-код операцій

Для зберігання байт-коду було розроблено модуль серіалізації. Модуль серіалізації дозволяє записувати байт-код у файл у зрозумілому для людини форматі, а також зчитувати байт-код з файлу. Це спрощує процес відладки і тестування, а також дозволяє зберігати проміжні результати компіляції (див. *Лістинг 4.5.1.2.*).

```
0     LOAD_CONST              Null
1     STORE_ADDR
2     LOAD_CONST              1
3     ALLOC
4     STORE
5     BIND_ADDR               l
6     LOAD_NAME               l
```

Лістинг 4.5.1.2. Байт-код записаний у файл

4.5.2. Трансляція АСД в байт-код

Трансляція абстрактного синтаксичного дерева в байт-код є наступним у процесі компіляції програм. Цей процес включає в себе перетворення вузлів АСД у відповідні інструкції байт-коду, які можуть бути виконані віртуальною машиною.

У процесі трансляції використовується шаблон проєктування «Відвідувач», що був описаний раніше, який дозволяє відокремити алгоритм обходу АСД від структури його вузлів. Це забезпечує зручний спосіб обробки різних типів вузлів, генеруючи відповідні інструкції байт-коду для кожного з них.

Для коректної трансляції АСД в байт-код було визначено структуру *BytecodeGenerator* (див. *Лістинг 4.5.2.1*.), що містить усі необхідні компоненти.

```
pub struct BytecodeGenerator<'a> {
    bytecode: Vec<Bytecode>,
    labels: HashMap<String, usize>,
    jumps: Vec<(usize, String)>,
    loop_context: Vec<LoopContext>,
    ast: &'a Algorithm,
    current_position: usize,
}
```

Лістинг 4.5.2.1. Визначення структури BytecodeGenerator

BytecodeGenerator містить такі поля: список *bytecode*, що зберігає згенеровані інструкції байт-коду, геш-таблицю *labels* для зберігання міток і їх позицій у байт-коді, список *jumps* для зберігання переходів, які потрібно буде вирішити після завершення генерації, стек *loop_context* для відстеження контексту поточних циклів, посилання на АСД, яке потрібно транслювати, і поточну позицію (*current_position*) в АСД.

Трансляція АСД у байт-код включає кілька важливих етапів, що детально описані нижче.

1. Ініціалізація генератора.

Генератор байт-коду ініціалізується з початковими значеннями, включаючи вектор для збереження інструкцій байт-коду і структури для управління мітками та переходами.

2. Відвідування вузлів АСД.

Під час обходу АСД, кожен вузол викликає метод *accept*, передаючи йому екземпляр структури *BytecodeGenerator* як аргумент. Генератор реалізує методи для обробки кожного типу вузла, генеруючи відповідні інструкції байт-коду.

3. Генерація інструкцій для вузлів.

В залежності від типу вузла АСД, генеруються різні інструкції байт-коду. Наприклад, для арифметичних операцій генеруються інструкції завантаження операндів і виконання арифметичних обчислень (див. *Рисунок 4.5.2.1*.).

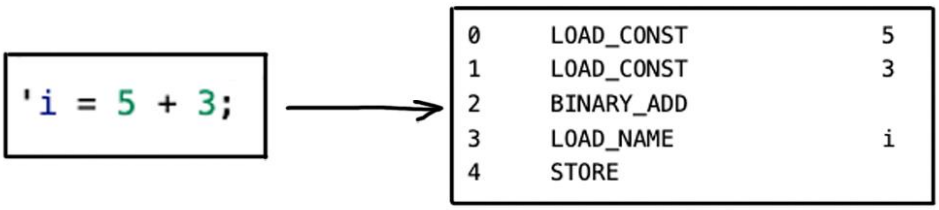


Рисунок 4.5.2.1. Приклад трансляції АСД у байт-код

4. Обчислення переходів.

Після завершення обходу АСД, генератор обчислює всі незавершені переходи, використовуючи збережені мітки для визначення кінцевих адрес переходів.

Трансляція АСД в байт-код є важливим етапом у процесі компіляції, який забезпечує перетворення вихідного коду у формат, придатний для виконання віртуальною машиною. Використання шаблону проєктування «Відвідувач» дозволяє зручно обробляти різні типи вузлів АСД, генеруючи відповідні інструкції байт-коду. Це забезпечує гнучкість і модульність системи, спрощуючи підтримку і розширення функціональності компілятора.

4.6. Віртуальна Машина

Віртуальна машина (ВМ) виконує проміжний байт-код, який створюють компілятори мов високого рівня. Це допомагає досягти високої продуктивності та безпеки у виконанні програм. Вона також забезпечує контроль за виконанням програм, управління пам’яттю та ресурсами для покращення надійності та безпеки середовища програмування.

У цьому розділі буде продемонстровано процес розробки віртуальної машини для Адресної мови.

Для обчислення байт-коду необхідно мати механізм збереження та обробки інструкцій, стеку значень, контекстів виконання та управління пам’яттю. Тому було створено структуру *VM* з полями, які представляють різні аспекти обчислювального процесу (див. *Лістинг 4.6.1.*).

```rust
pub struct VM {
    bytecode: Vec<Bytecode>,
    pc: usize,
    stack: Vec<Value>,
    scopes: Vec<Scope>,
    values_by_address: HashMap<i64, Value>,
    builtins: HashMap<String, BuiltinFunction>,
    next_address: i64,
    free_list: Vec<i64>,
    call_stack: Vec<usize>,
}
```

Лістинг 4.6.1. Визначення структури VM

Варто пояснити значення цих полів:

- *bytecode*: список байт-код інструкцій.
- *pc*: вказівник на поточну інструкцію.
- *stack*: стек використовується для зберігання проміжних значень під час виконання програм.
- *scopes*: стек, за допомогою якого відбуваються маніпуляції з областями видимості під час викликів підпрограм.
- *values_by_address*: геш-таблиця, що зберігає значення за відповідними адресами в пам’яті.
- *builtins*: геш-таблиця, що зберігає відповідність вбудованих функцій до їх імен.
- *next_address*: наступна вільна адреса.
- *free_list*: список звільнених адрес.
- *call_stack*: стек для зберігання адрес повернення.

4.6.1. Управління пам’яттю

Управління пам’яттю у віртуальній машині здійснюється за допомогою алгоритму наступної вільної адреси з використанням списку звільнених адрес.

```
    fn allocate_address(&mut self) -> i64 {
        if let Some(address) = self.free_list.pop() {
            address
        } else {
            let address = self.next_address;
            self.next_address += 1;
            address
        }
    }
```

Лістинг 4.6.1.1. Метод для виділення пам'яті

Цей алгоритм перевіряє наявність звільнених адрес у списку *free_list* і повертає одну з них, якщо вона є. Якщо звільнених адрес немає, генерується нова адреса, використовуючи лічильник *next_address*. Таким чином досягається ефективне виділення пам'яті задля запису даних за рахунок того, що список *free_list* забезпечує уникання «дірок» в пам'яті, а змінна *next_address* прискорює пошук наступної вільної комірки.

4.6.2. Область видимості

Області видимості реалізовані у вигляді стеку областей, де кожна область містить змінні та відповідні їм адреси. Це дозволяє відокремити глобальні змінні від інших змінних у різних функціональних блоках , а також гарантувати правильну роботу з локальними змінними.

```
pub struct Scope {
    variable_addresses: HashMap<String, i64>,
}

impl Scope {
    pub fn get_var(&self, name: &str) -> Option<i64> {
        self.variable_addresses.get(name).cloned()
    }

    pub fn set_var(&mut self, name: &str, address: i64) {
        self.variable_addresses.insert(name.to_string(), address);
    }
}
```

Лістинг 4.6.2.1. Визначення структури та поведінки Scope

Об’єкти *Scope* зберігають адреси змінних, які знаходяться у відповідній області видимості. Методи *get_var* і *set_var* можна використовувати для отримання та встановлення відповідності адрес до змінних (див. *Лістинг 4.6.2.1.*).

4.6.3. Інтерпретація байт-коду

У віртуальній машині інструкції виконуються в циклі в головному методі *run* (див. *Лістинг 4.6.3.1.*), де кожна операція байт-коду обробляється за її типом. Для обробки кожного типу інструкції окремо визначений метод для її обробки. Тому в залежності від типу інструкції викликається відповідний метод, що її обробляє.

```
pub fn run(&mut self) {
    while self.pc < self.bytecode.len() {
      let instruction = self.bytecode[self.pc].clone();
      self.pc += 1;
      match instruction {
                                  ...
        Bytecode::Constant(value) => self.stack.push(value),
        Bytecode::LoadVar(name) => self.get_var(&name),
        Bytecode::StoreVar(name) => self.set_var(&name),
        Bytecode::MulDeref => self.mul_deref(),
        Bytecode::Store => self.store(),
        Bytecode::Alloc => self.alloc(),
        Bytecode::AllocMany(count) => self.alloc_many(count),
        Bytecode::StoreAddr => self.store_addr(),
        Bytecode::BindAddr(name) => self.bind_addr(name),
        Bytecode::FreeAddr => self.free_addr(),
        Bytecode::Swap => self.swap(),
                                  ...
      }
    }
}
```

Лістинг 4.6.3.1. Реалізація методу run

Обробка підпрограм реалізується через збереження адреси повернення у *call_stack* та перехід до мітки, де підпрограма починається, що позначає початок підпрограми. Після завершення підпрограми виконується інструкція повернення, яка відновлює попередній контекст виконання (див. *Лістинг 4.6.3.2.*).

```
fn call_subprogram(&mut self, label: usize, argc: usize) {
    self.call_stack.push(self.pc + 1);
    self.pc = label;
}

fn handle_return(&mut self) {
    self.pop_scope();
    self.pc = self
        .call_stack
        .pop()
        .expect("Call stack underflow on return");
}
```

Лістинг 4.6.3.2. Реалізація виклику та виконання підпрограми

Оскільки цикли це «синтаксичний цукор» для набору умовних переходів, то й реалізуються вони через інструкції умовного переходу та безумовного стрибка. Умовні стрибки дозволяють контролювати виконання блоку інструкцій у циклі залежно від умови, що перевіряється (див. *Лістинг 4.6.3.3.*).

```
fn jump_if_false(&mut self, addr: usize) {
    let condition = self.stack.pop().unwrap();
    if !self.is_truthy(condition) {
        self.pc = addr;
    }
}
```

Лістинг 4.6.3.3. Реалізація умовного переходу

Умовні вирази у байт-коді представлені інструкціями умовного стрибка (див. *Лістинг 4.6.3.3.*). Це дозволяє контролювати потік виконання програми, приймаючи рішення на основі результату логічних операцій.

Таким чином, віртуальна машина забезпечує виконання байт-коду, управління пам’яттю, що дозволяє виконувати програми на Адресній мові та забезпечувати надійність і безпеку програмного середовища.

4.7.Розробка CLI

Компонент *adl-cli* дозволяє зручно використовувати систему ADL за допомогою інтерфейсу командного рядка. Було вирішено додати до інтерфейсу такий функціонал:

- Парсинг вхідного коду.
- Генерація байт-коду з вхідного коду.

- Виконання байт-коду віртуальною машиною.
- Лімітована інтерпретація абстрактного синтаксичного дерева.

Інтерфейс командного рядка зустрічає користувача ASCII-рисунком та поясненням існучих команд (див. *Рисунок 4.7.1.*).

Рисунок 4.7.1. Вікно привітання adl-cli

Інтерфейс було реалізовано за допомогою бібліотек *Clap* для обробки команд і аргументів командного рядка, а також *colored* для кольорового виведення тексту. Головна функція *main* містить цикл, що обробляє введення користувача, виконує відповідні команди та обробляє помилки. Цей цикл дозволяє інтерактивно вводити команди, які потім аналізуються і виконуються. Користувач може ввести одну з доступних команд, якщо введена команда некоректна або сталася помилка під час виконання, відображається відповідне повідомлення про помилку.

Кожна команда виконується своєю функцією, визначеною в компонентах розглянутих раніше. Наприклад, команда *parse* викликає функцію з модуля *parser*, яка проводить синтаксичний розбір вхідного тексту.

Інтерактивний підхід до обробки команд дозволяє користувачам легко виконувати необхідні операції та отримувати зворотній зв’язок про результати або помилки. Це робить *adl-cli* ефективним інструментом для розробників, які працюють з Адресною мовою, надаючи їм можливість швидко та зручно виконувати свої завдання.

В розділі був описаний процес створення системи ADL, її розбиття на компоненти та детальний опис їх розробки та залежності один від одного за допомогою схем.

5. ОБРОБКА ІЄРАРХІЧНИХ СТРУКТУР

В цьому розділі буде продемонстровано роботу зі списками та деревоподібними форматами за допомогою потужних засобів Адресної мови.

5.1. Списки

У багатьох мовах програмування списки є основними структурами даних, і Адресна мова не виняток. Вони дають змогу організувати дані в послідовність елементів, яка може бути ефективно оброблена . Адресна мова програмування використовує особливий підхід до роботи зі списками; зокрема, вона використовує «штрих-операції» та «мінус штрих-операції», що призводить до декларативного методу маніпулювання списками, який є простішим, порівнюючи з традиційними імперативними методами.

Списки в Адресній мові реалізовано як зв’язаний список, кожному елементу виділяється дві сусідніх комірки пам’яті. В першій зберігається адреса (вказівник) на наступний елемент, а в другій – значення елемента. Слід відмітити, що значення елементу може бути будь-якого типу.

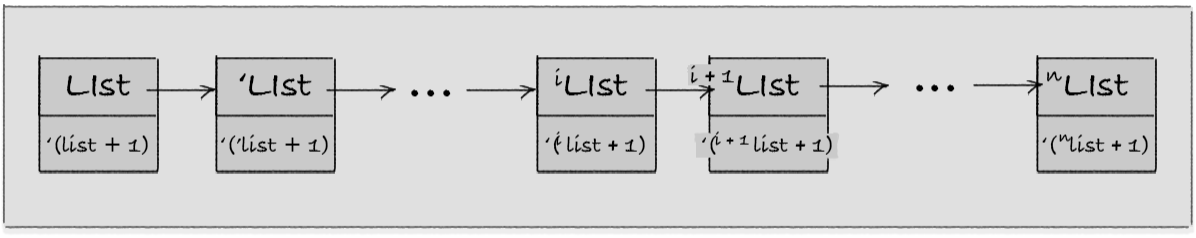


Рисунок 5.1.1. Однозв'язний список в Адресній мові

Далі наведено приклади підпрограм написані на Адресній мові, в кожному з прикладів надано варіанти з оригінальним та адаптованим синтаксисом для вирішення деякої проблеми.

На *Рисунку 5.1.2* наведено приклад ініціалізації списку *[c, d, e]*. В даному проєкті було реалізовано синтаксичний цукор для створення списків. Вираз виду *[c, d, e]* повертає вказівник на голову списку.

оригінальний синтаксис

```
1   110 => l; 18 =>'l
2
3   "c" => l + 1
4   "d" => 'l + 1
5   "e" => ''l + 1;
```

адаптований синтаксис

```
1   110 => l; 18 =>'l
2
3   "c" => l + 1
4   "d" => 'l + 1
5   "e" => ''l + 1;
```

синтаксичний цукор

```
1   l = [ "c", "d", "e" ]
```

Рисунок 5.1.2. Ініціалізація списку

Наступний приклад (див. *Рисунок 5.1.3*.) демонструє підпрограму *get*, яка приймає три параметри: *list, index, e*.

оригінальний синтаксис

```
1   get ... ø => list; ø => index; ø => e
2       '('indexlist + 1) => e
3   8
```

адаптований синтаксис

```
1   get ... null => list; null => index; null => e
2       'e = '(D {list, 'index} + 1)
3   return
```

Рисунок 5.1.3. Отримання елементу по індексу

Реалізація досить проста у порівняні з імперативними мовами програмування: використовується багатократна «штрих-операція», а після того отримане значення засилається за адресою останнього аргументу.

Отримання довжини списку було наступною задачою, для цього було реалізовано підпрограму *len* з двома параметрами: *list* і *len*.

оригінальний синтаксис

```
1   len ... ø => list; ø => len
2
3      Ц {0, 1, true => i } l1
4
5      P {'ⁱlist = null} | k
6              'i => len;
7              Ƨ
8          k...
9      l1 ...
10  Ƨ
```

адаптований синтаксис

```
1   len ... null => list; null => len
2
3      L {0, 1, true => i } l1
4
5      P {D {list, 'i} == null} | @k
6              'len = 'i;
7              return
8          k...
9      l1 ...
10  return
```

Рисунок 5.1.4. Отримання довжини списку

Для цього була використана формула циклювання, яка на кожній ітерації засилає індекс поточного елементу списку за адресою *len,* а у випадку, коли поточний елемент не має вказівника на наступний, підпрограма закінчує свою роботу.

Далі було написано підпрограму *concat* (див. *Рисунок 5.1.5*.), параметрами якої є два списки. Її задачею є об’єднати два списки в один. Першим кроком був виклик підпрограми *len*, яка була визначена раніше, для того щоб отримати довжину першого списку. Після того за допомогою багатократної «штрих-операції» було знайдений останній елемент, фінальної дією було засилання адреса голови другого списку у першу адресу (що вказує на наступний елемент) останнього елементу першого списку.

оригінальний синтаксис

```
1   concat ... ø => list1; ø => list2;
2
3       П len {list1, len1}
4       list2 => 'len1 - 1list1
5
6   Я
```

адаптований синтаксис

```
1   concat ... null => list1; null => list2;
2
3       SP len {list1, len1}
4       '(D {list1, 'len1 - 1}) = list2
5
6   return
```

Рисунок 5.1.5. Конкатенація списків

Параметрами підпрограми *concat* (див. *Рисунок 5.1.5.*) є два списки. Її ціллю є об’єднати два списки в один. Першим кроком був виклик підпрограми *len*, яка була визначена раніше, для того щоб отримати довжину першого списку. Після того за допомогою багатократної «штрих-операції» було знайдений останній елемент, фінальної дією було засилання адреси голови другого списку у першу адресу (що вказує на наступний елемент) останнього елементу першого списку.

Схожим чином було реалізовано підпрограму *append* (див. *Рисунок 5.1.6.*), яка додає в кінець списку елемент. Спочатку необхідно створити список з одного елементу (з того елементу, що треба додати до списку), а потім об’єднати їх в один.

оригінальний синтаксис

```
append ... ø => list; ø => el

    П len {list, len}
    ø => el - 1
    el - 1 => 'len-1list

Ǝ
```

адаптований синтаксис

```
append ... null =>list; null => el

    SP len {list, len}
    '(D {list, 'len - 1}) = [el]

return
```

Рисунок 5.1.6. Додавання елементу до кінця списку

Наступною задачею було видалення елемента зі списку за його індексом. Для цього була розроблена підпрограма *rem* (див. *Рисунок 5.1.7.*), параметрами якої є список (з якого треба вилучити елемент) та індекс елементу. Для цього використавши багатократну «штрих-операцію», в елемент, що знаходиться перед тим, що потрібно вилучити, засилається адреса на елемент, що знаходиться після нього.

оригінальний синтаксис

```
rem ... ø => list; ø => index;

    'index + 1list =>  'index-1list

Ǝ
```

адаптований синтаксис

```
rem ... null => list; null => index;

    D {list, 'index} = D {list, 'index + 1}

return
```

Рисунок 5.1.7. Видалення елементу списку по індексу

Також для демонстрації засобів Адресної мови програмування було написано підпрограму *bin_search* (див. *Рисунок 5.1.8.*) для двійкового пошуку (англ. Binary search) у впорядкованому списку. Вона була визначена у рекурсивний спосіб.

адаптований синтаксис

```
bin_search ... null => list; null => value; null => low; null => high; null => index
    P {high < low} | @not_found
        -1 => index
    return
    not_found...

    (low + high) / 2 => mid

    '(D{list, 'mid} + 1) => mid_val

    P {'mid_val > 'value} SP bin_search {list, value, low, 'mid - 1, index}
        |  P {'mid_val < 'value} SP bin_search {list, value, 'mid + 1, high, index}
            | 'mid => index
return
```

Рисунок 5.1.8. Реалізація двійкового пошуку (адаптований синтаксис)

Аналогічно, не дивлячись на деякі синтаксичні розбіжності, виглядає реалізація з використанням оригінального синтаксису (див. *Рисунок 5.1.9.*), що підтверджує слідування основним концепціям Адресної мови програмування під час розробки системи ADL.

оригінальний синтаксис

```
bin_search ... ø => list; ø => value; ø => low; ø => high; ø => index
    P {high < low} ↓ not_found
        -1 => index
    return
    not_found...

    (low + high) / 2 => mid

    '('midlist + 1) => mid_val

    P {'mid_val > 'value} П bin_search {list, value, low, 'mid - 1, index}
        ↓  P {'mid_val < 'value} П bin_search {list, value, 'mid + 1, high, index}
            ↓ 'mid => index
Я
```

Рисунок 5.1.9. Реалізація двійкового пошуку (оригінальний синтаксис)

Отже, засоби Адресної мови програмування дозволяють ефективно організовувати та обробляти елементи списків. А за рахунок використання «штрих-операцій» та «мінус штрих-операцій» робити маніпуляції зі списками більш декларативними та простими засобами у порівнянні з імперативними мовами програмування.

5.2. Структурні типи даних

Засобами Адресної мови програмування можливо визначати структурні типи даних та їх поведінку, які можна порівняти з класами або структурами в сучасних об’єктно-орієнтованих мовах, таких як Java та C++.

Доступ до складових (в сучасних мовах називають полями) таких типів даних здійснюється з використанням прямої, непрямої адресації вищих рангів та арифметичних операцій над адресами.

Для прикладу, було написано програму, в якій створюється структурний тип даних *Person* з такими полями: *name*, *age*, *married_on*. Також для демонстраційних цілей була визначена наступна поведінка: мати можливість вивести своє ім’я на екран, вивести свій вік, «одружитися» з іншим екземпляром цього типу та вивести на екран ім’я свого партнера, якщо такий є. Для кожної частини програми буде наводитися код написаний в адаптованому та оригінальному синтаксисах.

Спочатку було визначено конструктор (див. *Рисунок 5.2.1.*) та деструктор (див. *Рисунок 5.2.2.*) для типу даних *Person*.

оригінальний синтаксис

```
new ... ø => name; ø => age; ø => married_on; ø => person;
    'name => p
    'age => p + 1
    'married_on => p + 2;
    p => person
ß
```

адаптований синтаксис

```
new ... null => name; null => age; null => married_on; null => person;
    'name => p
    'age => p + 1
    'married_on => p + 2;
    p => person
return
```

Рисунок 5.2.1. Визначення конструктору для типу Person

Записано підпрограму *new* яка приймає чотири наступні параметри: *name*, *age*, *married_on* і *person*. Перші три аргументи – це складові даної структури, а останній – це адреса, відповідно до якої буде зберігатися вся інформація про новостворений екземпляр типу. Після запису значень аргументів до комірок пам'яті *p*, *(p + 1)*, *(p + 2)*, адрес першої комірки засилається до адреси *person*. Таким чином, після виконання програми буде існувати вказівник на створений екземпляр типу *Person*.

оригінальний синтаксис

```
rem ... ø => person;
    null => person
    null => person + 1
    null => person + 2;
ß
```

адаптований синтаксис

```
rem ... null => person;
    null => person
    null => person + 1
    null => person + 2;
return
```

Рисунок 5.2.2. Визначення деструктору для типу Person

Записано підпрограму *rem* яка приймає один параметр-вказівник *person*. Вона видаляє усі дані за адресами (асоційовані з полями екземпляру) *p*, *(p + 1)*, *(p + 2)*.

Отже, зараз були визначені засоби для створення екземпляру структурного типу даних та його видалення з пам'яті.

Наступним кроком буде визначення підпрограм відповідальних за реалізацію поведінки.

оригінальний синтаксис

```
say_hi ... ø => person;
    P {'person = null} Print {"Hello, I have no name :("} ↓ Print {"Hello, my name is ", 'person}
ß
```

адаптований синтаксис

```
say_hi ... null => person;
    P {'person == null} Print {"Hello, I have no name :("} | Print {"Hello, my name is ", 'person}
return
```

Рисунок 5.2.3. Визначення підпрограми say_hi

В записаній на *Рисунку 5.2.3.* підпрограмі *say_hi* у консоль виводиться стрічка зі значення поля *name*, що знаходиться за адресою *p*, якщо вона існує.

Аналогічно до підпрограми *say_hi* працює підпрограма *say_age* (див. *Рисунок 5.2.4.*), з єдиною відмінністю у тому, що значення віку знаходиться за адресою *(p + 1)*.

оригінальний синтаксис

```
say_age ... ø => person;
    P {'(person + 1) = null} Print {"I don`t how old I am :("} ↓ Print {"I`m ", '(person + 1), " years old"}
ꓭ
```

адаптований синтаксис

```
say_age ... null => person;
    P {'(person + 1) == null} Print {"I don`t how old I am :("} | Print {"I`m ", '(person + 1), " years old"}
return
```

Рисунок 5.2.4. Визначення підпрограми say_age

Поле *married_on* є вказівником на екземпляр типу *Person*, для того щоб продемонструвати зв'язок «один до одного». Для того щоб уможливити керування цим зв'язком було визначено підпрограму *marry* (див. *Рисунок 5.2.5.*). Вона дозволяє симетрично записати адреси один одного до зазначеного поля кожного з об'єктів.

оригінальний синтаксис

```
marry ... ø => person1; ø => person2
    'person2 => 'person1 + 2
    'person1 => 'person2 + 2
ꓭ
```

адаптований синтаксис

```
marry ... null => person1; null => person2;
    'person2 => 'person1 + 2
    'person1 => 'person2 + 2
return
```

Рисунок 5.2.5. Визначення підпрограми marry

Також, таким чином можна створити зв’язок «один до багатьох», в даній предметній області це б відображало багатоженство.

Для перевірки цієї складової структури *Person* була визначена підпрограма *say_who_you_married_on* (див. *Рисунок 5.2.6.*), яка виводить у консоль ім’я екземпляру, який знаходиться у зв’язку з поточним об’єктом.

оригінальний синтаксис

```
say_who_you_married_on ... ø => person;
    p = 'person + 2
    P { 'p = null} Print {"I`m not married yet"} ↓ Print {"I`m married on ", ''p}
ß
```

адаптований синтаксис

```
say_who_you_married_on ... null => person;
    p = 'person + 2
    P { 'p == null} Print {"I`m not married yet"} | Print {"I`m married on ", ''p}
return
```

Рисунок 5.2.6. Визначення підпрограми say_who_you_married_on

У результаті виконання повної програми був отриманий коректний результат (див. *Рисунок 5.2.7.*).

Послідовний виклик підпрограм

```
"Alice" => name1
23 => age1
null => married_on1

"Bob" => name2
25 => age2
null => married_on2

SP new {name1,age1, married_on1, alice}
SP new {name2,age2, married_on2, bob}

SP say_hi {'alice}
SP say_age {'alice}
SP marry {alice, bob}
SP say_who_you_married_on {alice}
!
```

Результат виконання

```
Code parsed successfully.
Bytecode generated successfully.
Hello, my name is Alice
I`m 23 years old
I`m married on Bob
Compilation result: ()
```

Рисунок 5.2.7. Результат виконання програми

5.3.Дерева

Дерева є важливим типом даних у багатьох мовах програмування, оскільки вони дозволяють ефективно організовувати ієрархічні структури. У цьому підрозділі розглянемо основні способи представлення дерев та їх реалізацію в Адресній мові програмування.

Першим варіантом представлення дерев (див. *Рисунок 5.3.1.*) є популярний підхід, в якому кожен вузол може зберігати дані та мати посилання на наступні вузли – дочірні вузли, для яких він є батьківським вузлом.

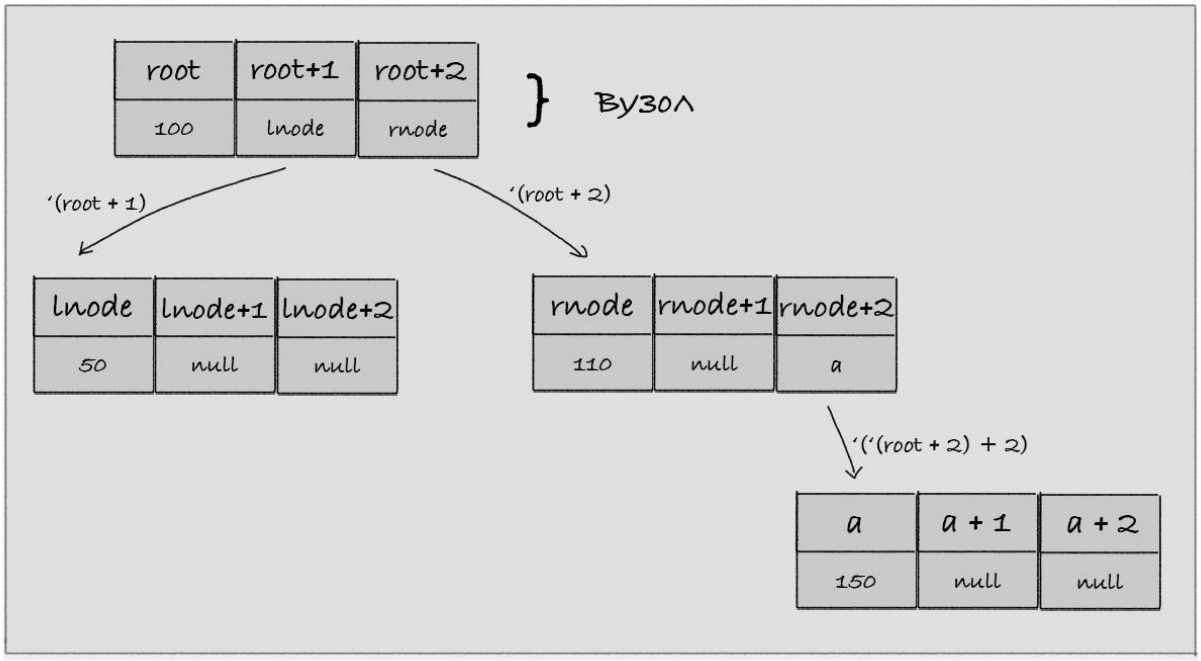


Рисунок 5.3.1. Представлення дерев в Адресній мові (1)

Для реалізації такого дерева в Адресній мові необхідно представити вузол як структурний тип даних (аналогічно до інших мов програмування). Наприклад, у випадку з бінарним (двійковим) деревом вузол повинен мати такі складові: значення вузла, вказівник на лівий елемент та на правий елемент. Після реалізації такої структури даних за допомогою «штрих-операції» можно створювати дерево та проводити його обхід. Наприклад (див. *Рисунок 5.3.1.*), щоб отримати значення вузла *a* необхідно написати наступний вираз:

"('(root +2)+2).

Засоби Адресної мови програмування дозволяють представити структуру дерева іншим шляхом. Кожен дочірній вузол може зберігати дані та мати посилання на попередній вузол – в даному випадку, на батьківський. Використовуючи «мінус штрих-операцію» на батьківському вузлі, можна отримати усі адреси, що на нього вказують, вони і будуть адресами дочірніх вузлів (див. *Рисунок 5.3.2.*).

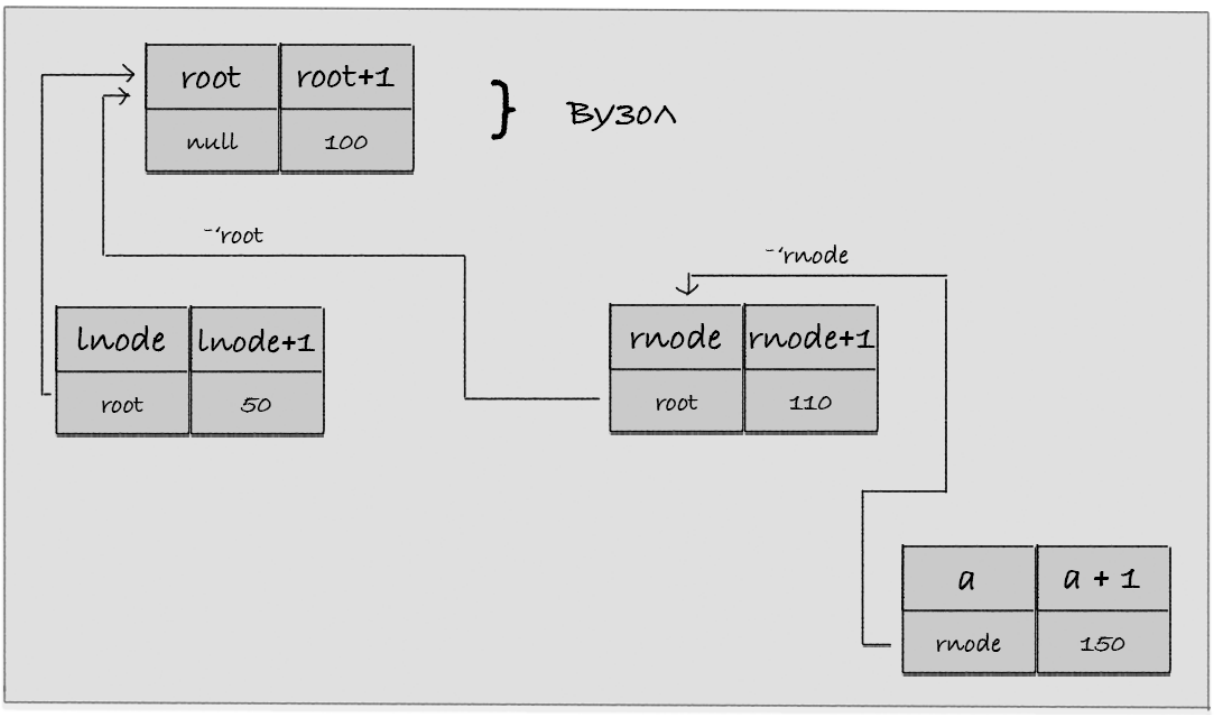


Рисунок 5.3.2. Представлення дерев в Адресній мові (2)

Для реалізації такого дерева в Адресній мові також необхідно представити вузол як структурний тип даних, наприклад, у випадку з бінарним (двійковим) деревом вузол повинен мати такі складові: значення вузла, вказівник на батьківський вузол. Реалізація такої структури даних дозволяє проводити обхід дерева за допомогою «мінус штрих-операції». Наприклад (див. *Рисунок 5.3.2.*), щоб отримати значення вузла *a* необхідно написати наступний вираз: *'(-''(root).*

Для демонстрації були написані програми, в яких створюється двійкові дерева згідно з кожним із підходів. Також для демонстраційних цілей був

визначений набір базових операцій для кожної реалізації дерева. Реалізації наведені в адаптованому та оригінальному синтаксисах.

Спочатку було визначено конструктор (див. *Рисунок 5.3.3.*) для типу даних *Node*.

оригінальний синтаксис

```
node... ø => val
        ø => val + 1
        ø => val + 2
Ǝ
```

адаптований синтаксис

```
node... null => val
        null => val + 1
        null => val + 2
return
```

Рисунок 5.3.3. Визначення пустого конструктору для типу Node

Записано підпрограму *node* яка приймає лише один параметр – адресу, відповідно до якої буде зберігатися вся інформація про новостворений вузол. Оскільки конструктор лише створює відокремлений вузол за адресами, в яких в подальшому буде записано адреси дочірніх вузлів, *(val + 1)*, *(val + 2)* засилаються нульові значення.

оригінальний синтаксис

```
insert ... ø => root;  ø => node
    P { 'root > 'node }  ↓ gte
        P { '(root + 1) = ø } node => root + 1 ↓ Π insert {'(root + 1), node}
            Ǝ
    gte ...
        P { '(root + 2) = ø } node => root + 2 ↓ Π insert {'(root + 2), node}
            Ǝ
Ǝ
```

адаптований синтаксис

```
insert ... null => root;  null => node
    P { 'root > 'node }  | @gte
        P { '(root + 1) == null } node => root + 1 | SP insert {'(root + 1), node}
            return
    gte ...
        P { '(root + 2) == null } node => root + 2 | SP insert {'(root + 2), node}
            return
return
```

Рисунок 5.3.4. Визначення операції вставки для двійкового дерева

Підпрограма *insert* приймає на вхід два параметри: *root* та *node*. Вона додає новий вузол до вже існуючого дерева, в залежності від того, значення нового вузла менше чи більше за значення поточного кореня піддерева, підпрограма рекурсивно викликає себе з деяким піддеревом та новим вузлом у якості аргументів. У випадку, коли буде знайдене відповідне вільне місце в дереві, новий вузол буде вставлений і підпрограма закінчить роботу.

Далі для демонстрації реалізації засобами Адресної мови програмування більш складного алгоритму було визначено операцію видалення (див. *Рисунок 5.3.5.*).

адаптований синтаксис

```
rem ... null => father; null => node; null => val;
    P { 'node == 'val } | @to_find
        SP is_leaf {node, is_leaf}
        P { 'is_leaf}  | @case2
        P { node == '(father + 1)} null => father + 1 | null => father + 2
        return
    case2 ...
        SP has_one_son {node, son}
        P { 'son == null  } @case3 |
        P { node == '(father + 1)} 'son => father + 1 | 'son => father + 2
        return
    case3 ...

    SP min {node, leaf}

    SP rem {father, node, 'leaf}

    P { node == '(father + 1)} 'leaf => father + 1 | 'leaf => father + 2
    '(node + 1) => 'leaf + 1; '(node + 2) => 'leaf + 2
    return

    to_find...
    P { 'node < 'val }  | @lte1
        P { '(node + 2) == null } Print{"elem not found"} | SP rem {node, '(node + 2), val}
        return
    lte1 ...
        P { '(node + 1) == null } Print{"elem not found"} | SP rem {node, '(node + 1), val}
        return
```

```

min ... null => node; null => leaf;
    SP is_leaf {node, is_leaf}
    P { 'is_leaf} | @not_leaf
        node => leaf
        return
    not_leaf...
    SP has_one_son {node, son}
    P { 'son == null } @two_sons|
        SP min {'son, leaf}
        return
    two_sons ...
    n1 = '(node + 1)
    n2 = '(node + 2)
    P { 'n1 < 'n2 } SP min {n1, leaf} | SP min {n2, leaf}
return
```

Рисунок 5.3.5. Визначення операції видалення для двійкового дерева (адаптований синтаксис)

Для реалізації операції видалення вузла за його значенням було визначено підпрограму *rem*. Вона охоплює всі 3 можливі випадки:

- Вузол без дітей (листовий вузол).
- Вузол з однією дитиною.
- Вузол з двома дітьми.

Для останнього випадку була визначена допоміжна підпрограма *min* (також рекурсивно), що знаходить вузол з найменшим значенням. Запис підпрограм оригінальним синтаксисом можна побачити на *Рисунку 5.3.6.*

оригінальний синтаксис

```
rem ... ø => father; ø => node; ø => val;
    P { 'node = 'val } ↓ to_find
        Π is_leaf {node, is_leaf}
        P { 'is_leaf}  ↓ case2
        P { node = '(father + 1)} ø => father + 1 ↓ ø => father + 2
        Θ
    case2 ...
        Π has_one_son {node, son}
        P { 'son = ø  } case3 ↓
        P { node = '(father + 1)} 'son => father + 1 ↓ 'son => father + 2
        Θ
    case3 ...

    Π min {node, leaf}

    Π rem {father, node, 'leaf}

    P { node = '(father + 1)} 'leaf => father + 1 ↓ 'leaf => father + 2
    '(node + 1) => 'leaf + 1; '(node + 2) => 'leaf + 2
    Θ

    to_find...
    P { 'node < 'val }  ↓ lte1
        P { '(node + 2) = ø } Print{"elem not found"} ↓ Π rem {node, '(node + 2), val}
        Θ
    lte1 ...
        P { '(node + 1) = ø } Print{"elem not found"} ↓ Π rem {node, '(node + 1), val}
        Θ
```

```
min ... ø => node; ø => leaf;
    Π is_leaf {node, is_leaf}
    P { 'is_leaf} ↓ not_leaf
        node => leaf
        Θ
    not_leaf...
    Π has_one_son {node, son}
    P { 'son = ø } two_sons↓
        Π min {'son, leaf}
        Θ
    two_sons ...
    n1 = '(node + 1)
    n2 = '(node + 2)
    P { 'n1 < 'n2 } Π min {n1, leaf} ↓ Π min {n2, leaf}
Θ
```

Рисунок 5.3.6. Визначення операції видалення для двійкового дерева (оригінальний синтаксис)

Для демонстрації роботи з деревами, де кожен дочірній вузол може зберігати дані та мати посилання на батьківський вузол було реалізовано алгоритм DFS (Depth-first search) (див. *Рисунок 5.3.7.*).

оригінальний синтаксис

```
dfs ... ø => root; ø => val; ø => found
    P { '(root + 1) = val } | not_equal
        true => found
        Θ
    not_equal ...
    Π sons {root, sons, len}
    P {len = 0} Θ |
    L {0, 1, 'i < 'len => i} l
        Π dfs {'ⁱ⁺¹sons, val, found}
        P { 'found } Θ |
    l ...

Θ

sons ... ø => n; ø => s; ø => len
    ⁻'n => s
    Π len_list {s, len}
Θ
```

адаптований синтаксис

```
dfs ... null => root; null => val; null => found
    P { '(root + 1) == val } | @not_equal
        true => found
        return
    not_equal ...
    SP sons {root, sons, len}
    P {len == 0} return |
    L {0, 1, 'i < 'len => i} l
        cur_node = D{'sons, 'i}
        SP dfs {cur_node, val, found}
        P { 'found } return |
    l ...

return

sons ... null => n; null => s; null => len
    D {n, -1} => s
    SP len_list {s, len}
return
```

Рисунок 5.3.7. Визначення алгоритму DFS

Для отримання дочірніх вузлів була окремо записана підпрограма *sons* з наступними параметрами: *n, s, len*, де *n* – це батьківський вузол, *s* – список дочірніх вузлів і *len* – кількість дочірніх вузлів. В тілі підпрограми обчислюється «мінус штрих-операція» на аргументом *n* і засилається голова списку з дочірніх вузлів за адресою *s*. Далі обчислюється довжина цього списку та засилається за адресою *len*. Після цього підпрограма *sons* закінчує свою роботу. Таким чином вдалося отримати усі дочірні вузли батьківського *n*.

Далі була визначена підпрограма *dfs*, яка шукає вузол дерева за його значенням і у випадку знаходження такого вузла, засилає значення істинності за адресою *found*. Першим кроком вона перевіряє чи є поточний вузол шуканим. Якщо це не так, отримує список дітей поточного вузла та, ітеруючись по кожному з них, рекурсивно викликає себе з поточним дочірнім вузлом у якості аргументу. Закінчує роботу у випадку знаходження шуканого вузла або коли всі вузли перебрані.

У цьому розділі було продемонстровано ефективність використання Адресної мови програмування для роботи зі списками та деревоподібними структурами даних. Робота показала, що Адресна мова може забезпечити простоту і декларативність у маніпуляціях зі списками через використання «штрих-операції» та «мінус штрих-операції».

Також було показано, як створювати та маніпулювати структурними типами даних, які можна порівняти з класами чи структурами в об’єктно-орієнтованих мовах програмування. Використання прямої та непрямої адресації дозволяє ефективно обробляти поля структурних типів даних та керувати зв’язками між об’єктами, демонструючи можливість створення комплексних типів даних.

Дослідження деревоподібних структур у Адресній мові програмування продемонструвало гнучкість та ефективність роботи з ієрархічними даними.

Було представлено два основні підходи до реалізації дерев: один, де кожен вузол має посилання на дочірні вузли, та інший, де дочірні вузли посилаються на свої батьківські. Обидва підходи були реалізовані з використанням конструкторів, операцій вставки та видалення, а також алгоритмів обходу дерев, таких як пошук у глибину (DFS).

ВИСНОВКИ

У ході виконання даної роботи було реалізовано інтерпретатор та компілятор для Адресної мови програмування, що дозволило детально вивчити особливості засобів розробки, які надає ця мова. Робота над проєктом включала кілька основних етапів, кожен з яких був спрямований на досягнення конкретних технічних і дослідницьких цілей.

Основні результати роботи можна узагальнити наступним чином: Адресна мова програмування була детально проаналізована з точки зору її структурних елементів, таких як рядки, мітки, «штрих-операція», «мінус штрих-операція», формула зупину, формула засилання та обміну, предикатні формули, формули циклювання та формули входження. Ці елементи дозволяють ефективно маніпулювати адресами та їх вмістом, визначати точки завершення алгоритму, передавати значення між адресами, описувати умови та цикли, а також викликати підпрограми.

Для спрощення написання програм на Адресній мові було внесено зміни в її синтаксис. Оригінальний синтаксис, хоча і є потужним інструментом для маніпуляції даними, виявився не зовсім зручним для написання програм в сучасних текстових редакторах. Новий синтаксис зберіг всі можливості і гнучкість, яку надає Адресна мова, але зробив її більш доступною та легкою для написання програм.

У роботі були використані сучасні технології, такі як Rust, Cargo, LALRPOP та Clap, які забезпечили безпеку пам’яті, простоту розробки складних програм та зручність обробки аргументів командного рядка. Використання цих технологій значно полегшило розробку проєкту.

Проєкт ADL був розбитий на окремі компоненти: лексичний аналізатор, синтаксичний аналізатор, генератор байт-коду, віртуальна машина, інтерпретатор та допоміжні модулі для роботи з примітивними типами даних і утилітами. Кожен з цих компонентів був реалізований для забезпечення коректної обробки та виконання програм на Адресній мові.

Було продемонстровано роботу зі списками, структурними та дерево-подібними структурами даних за допомогою засобів Адресної мови. Реалізація таких структур дозволила маніпулювати даними та вирішувати складні задачі, що підтвердило актуальність можливостей засобів Адресної мови у сучасних умовах.

Виконана робота продемонструвала високий потенціал Адресної мови програмування для розв’язання сучасних задач та підкреслила важливість їх вивчення та популяризації. Проект ADL наочно показав, що, використовуючи сучасні інструменти та технології, можна вдосконалити історичні досягнення в галузі програмування та зробити їх доступними для нових поколінь розробників. Ця робота внесла свій вклад у збереження історії комп’ютерних технологій.

Посилання на репозиторій реалізації

1. Головне посилання.
2. Плагін для Visual Studio Code.
3. Приклади роботи зі списками.
4. Приклади роботи з деревоподібними форматами.
5. Приклади об’єктно-орієнтованих програм на Адресній мові.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Ющенко Ю.О., Деревоподібні формати Адресного програмування// Наукові записки НаУКМА. – Т. 3 : Комп’ютерні науки. – К. : – 2021. – С. 79–87, – URL: http://nrpcomp.ukma.edu.ua/article/view/220657?fbclid=IwAR0N65fNMllivasK84zOOsEmzenblOFAv8IksOvQpzV6JSX54TJpBfOV7o (дата звернення: 03.10.2021).
2. Serokell. How to Implement LR1 Parser. URL: https://serokell.io/blog/how-to-implement-lr1-parser (дата звернення: 02.06.2024).
3. Nystrom, Robert. Crafting Interpreters. URL: https://craftinginterpreters.com/ (дата звернення: 02.06.2024).
4. Zheng, Wubing. Build Lua in Rust. URL: https://wubingzheng.github.io/build-lua-in-rust/en/PREFACE.html (дата звернення: 02.06.2024).
5. Rust-Hosted Languages. Interpreter and VM Implementation. URL: https://rust-hosted-langs.github.io/book/chapter-interp-vm-impl.html (дата звернення: 02.06.2024).
6. Ющенко Е.Л., Адресное программирование // К. : – Гос. издательство технической литературы, УРСР, 1963. – 287 с., – URL: https://files.infoua.net/vushchenko/AdresnoeprogrammirovanieEYushchenko1963.pdf (дата звернення: 03.10.2021).
7. Ющенко Ю.О., Окремі аспекти декларативності «мінус штрих-операції» // Наукові записки НаУКМА. – Т. 3 : Комп’ютерні науки. – К. : – 2020. – С. 19–26, – URL: http://nrpcomp.ukma.edu.ua/article/view/220657?fbclid=IwAR0N65fNMllivasK84zOOsEmzenblOFAv8IksOvQpzV6JSX54TJpBfOV7o (дата звернення: 03.10.2021).
8. LALRPOP. LALRPOP Documentation. URL: https://lalrpop.github.io/lalrpop/index.html (дата звернення: 02.06.2024).

9. Stan Development Team. BNF Grammars. URL: https://mc-stan.org/docs/2_22/reference-manual/bnf-grammars.html (дата звернення: 02.06.2024).
10. Peet, Simmy. Crafting a Compiler in Rust: Lexical Analysis. URL: https://dev.to/simmypeet/crafting-a-compiler-in-rust-lexical-analysis-1eb1 (дата звернення: 02.06.2024).
11. Better Programming. Building CLI Apps in Rust: What You Should Consider. URL: https://betterprogramming.pub/building-cli-apps-in-rust-what-you-should-consider-99cdcc67710c (дата звернення: 02.06.2024).
12. Visual Studio Code. Syntax Highlight Guide. URL: https://code.visualstudio.com/api/language-extensions/syntax-highlight-guide (дата звернення: 02.06.2024).
13. Glushkov V.M., & Yushchenko E.L., D 1966, The Kiev Computer; a Mathematical Description, USA, Ohio, Translation Division, Foreign Technology Div., Wright-Pattenon AFB, 234p., ASIN: B0007G3QGC. https://files.infoua.net/yushchenko/The-Kiev-Computer_a-Mathematical-Description_VHlushkov-EYushchenko_1966.pdf (дата звернення: 03.06.2024).
14. Глушков В.М., Вычислительная машина "Киев". Математическое описание / В.М. Глушков, Е.Л. Ющенко. // К. : – Гостехиздат УССР, 1962. – 183 с. : ил., – URL: files.infoua.net/yushchenko/Vychislitelnaya-mashyna-Kiev_VHlushkov_EYushchenko_1962.pdf (дата звернення: 03.06.2024).