Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук
(повна назва)
Кафедра Штучного інтелекту
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти перший (бакалаврський)

Мова програмування на базі акторної моделі мови Erlang
(тема)

Виконав:
здобувач четвертого року навчання,
групи ІТШІ-21-5
Євгеній Сехін
(власне ім'я, прізвище)
Спеціальність 122 Комп'ютерні науки
(код і повна назва спеціальності)
Тип програми освітньо-професійна
Освітня програма Штучний інтелект
(повна назва освітньої програми)
Керівник ст. викл. В'ячеслав Гребенюк
(посада, власне ім'я, прізвище)

Допускається до захисту

Завідувач кафедри ШІ _____ Олег ЗОЛОТУХІН
(підпис) (власне ім'я, прізвище)

2025 р.

Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук

Кафедра Штучного інтелекту

Рівень вищої освіти перший (бакалаврський)

Спеціальність 122 Комп'ютерні науки
(код і повна назва)

Тип програми освітньо-професійна

Освітня програма Штучний інтелект
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«_____» _____ 20 ___ р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Сехіну Євгенію Миколайовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Мова програмування на базі акторної моделі мови Erlang

затверджена наказом університету від 19 травня 20 25 р. № 378Ст

2. Термін подання студентом роботи до екзаменаційної комісії 18 червня 20 25 р.

3. Вихідні дані до роботи Дані Інтернет-джерел та документації щодо мов програмування та їх реалізації, книга «Програмування на С»

4. Перелік питань, що потрібно опрацювати в роботі

1) Аналіз предметної галузі

2) Постановка задачі

3) Програмна реалізація

4) Результати

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Строк / терміни виконання етапів роботи	Примітка
1	Отримання завдання на кваліфікаційну роботу	19.05.2025	виконано
2	Аналіз предметної галузі	20.05.2025	виконано
3	Дослідження існуючих рішень	21.05.2025	виконано
4	Постановка задачі	22.05.2025	виконано
5	Створення архітектури мови програмування	25.05.2025	виконано
6	Програмна реалізація	30.05.2025	виконано
7	Написання пояснювальної записки	01.06.2025	виконано
8	Перевірка на академічний плагіат	07.06.2025	виконано
9	Нормоконтроль	09.06.2025	виконано
10	Підготовка презентації та доповіді	11.06.2025	виконано
11	Попередній захист	13.06.2025	виконано
12	Рецензування	14.06.2025	виконано
13	Захист перед ЕК	17.06.2025	

Дата видачі завдання 19 травня 2025 р.

Здобувач _____
(підпис)

Керівник роботи _____ ст. викл. В’ячеслав Гребенюк
(підпис) (посада, власне ім’я, прізвище)

РЕФЕРАТ

Пояснювальна записка: 81 с., 5 рис., 1 дод., 16 джерел.

АКТОРИ, ВІРТУАЛЬНА МАШИНА, КОМПІЛЯТОР, МОВИ ПРОГРАМУВАННЯ, ПАРСЕР, СКАНЕР, ФУНКЦІЯ, ШТУЧНИЙ ІНТЕЛЕКТ.

Об’єкт дослідження – процес проектування мови програмування з акторною моделлю виконання.

Предмет дослідження – архітектура мов програмування з акторними моделями, можливі реалізації для віртуальної машини.

Мета роботи – розробка мови програмування на базі вірутальної машини з акторною моделлю виконання, яка забезпечує ізольований кожного актора та ефективну комунікацію між ними, можливістю виконання арифметичних операцій, їх виводу, використання потоку управління, нативних та звичайних функцій.

Методи дослідження – аналіз існуючих мов програмування з підтримко акторної моделі, аналіз літератури для створення мов програмувань.

У результаті роботи було зроблено аналіз, постановку задачі, проаналізовано проблеми та обмеження існуючих підходів, реалізовано мову програмування на базі віртуальної машини та акторної моделі.

ABSTRACT

Bachelor’s thesis contains: 81 pp., 5 fig., 1 ann., 16 references.

ACTORS, ARTIFICIAL INTELLIGENCE, COMPILER, FUNCTION, PARSER, PROGRAMMING LANGUAGES, SCANNER, VIRTUAL MACHINE

The object of research is the process of designing a programming language with an actor model of execution.

The subject of research is the architecture of programming languages with actor models, possible implementations for a virtual machine.

The purpose of the work is to develop a programming language based on a virtual machine with an actor model of execution, which provides isolation of each actor and effective communication between them, the ability to perform arithmetic operations, their output, use of the control flow, native and common functions.

Research methods are an analysis of existing programming languages with support for the actor model, analysis of literature for creating programming languages. As a result of the work, an analysis and problem statement were conducted, the issues and limitations of existing approaches were examined, and a programming language was implemented based on a virtual machine and the actor model.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ВМ – віртуальна машина;

ООП – об’єктно-орієнтоване програмування;

ШІ – штучний інтелект;

BEAM – Bogdan/Björn's Erlang Abstract Machine – абстрактна машина Bogdan/Björn's Erlang;

GC – Garbage Collection – збирання сміття;

REPL – Read-Eval-Print Loop – інтерактивне середовище цикл читання-обчислення-друк.

ВСТУП

Наразі штучний інтелект набирає все більше обертів у сучасному світі, через що гостро стоїть питання у напрямку вирішення проблем масштабованості обчислень, паралельне обчислення, ізоляцію цих процесів та ефективна взаємодія між ними.

Традиційні мови програмування, які використовують об’єктно-орієнтовану та імперативну парадигми мають обмеження при побудові таких систем. Під час розробки програм виникає купа проблем, де можна помилитися, як приклад змінивши глобальну змінну або використання спільних даних при паралельних обчисленнях. Це потребує контролю та відслідковування, що збільшує час на розробку та тестування програми.

Вирішенням цих проблем може бути акторна модель виконання, головною ідеєю якої є обчислення, якими займаються окремі актори, виконуючи їх незалежно одне від одного.

Багато сучасних мов програмування мають вбудовані бібліотеки, фреймворки, які реалізують дану модель, з них можна виокремити Erlang, Elixir, Dart, Akka (для Scala), і т.д. Але суттєва вада цих рішень у тому, що це надбудова над мовою програмування, а не реалізація всередині мови.

Тому метою роботи буде виступати розробка такої мови програмування з вбудованою підтримкою акторної моделі, де вона не буде виступати в якості допоміжної бібліотеки. Мова з такою моделлю буде краще відображати ідею ООП Алана Кея [1]. Вона повинна бути Тюрінг Повною, мати асинхронну систему обміну повідомленнями, компілятор та віртуальну машину.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Огляд мов програмування та фреймворків з акторними моделями

Акторна модель – це математична модель паралельних обчислень, яка розглядає актора як основний блок для цих обчислень. У відповідь на отримане повідомлення актор може: приймати локальні рішення, в межах своєї області видимості, створювати інших акторів, надсилати або не надсилати повідомлення і визначати як реагувати на наступне повідомлення. Актори можуть змінювати свій власний приватний стан, а на інших можуть впливати лише через повідомлення.

Спочатку розглянемо популярний фреймворк Akka, який розроблено для Java, Scala та .NET за допомогою мови програмування Scala. Він використовується для створення додатків із стійкістю до відмов у системах, які потребують розподіленого виконання або які можна розпаралелити. Akka реалізує легких ізольованих акторів, які отримують асинхронні повідомлення, вміють створювати нових акторів, надсилати повідомлення до інших акторів та змінювати внутрішню поведінку. З особливостей, Akka підтримує створення додатків з розподіленою системою виконання за допомогою спеціальних модулей. Це дозволяє розподілити робоче навантаження між декількома вузлами, щоб вільні не простоювали. Даний фреймворк також має стійкість до відмов, що дозволяє зберігати ланцюг повідомлень та відновлювати їх після збоїв. Окрім того, Akka містить потужну абстракцію, щоб створювати реактивні додатки та конвеєри обробки даних. Наостанок, цей фреймворк є масштабованим та типізованим, через що він може працювати з великою кількістю вузлів, та забезпечувати безпеку під час компіляції, для зменшення помилок під час виконання.

Erlang – це функціональна мова програмування, яка розроблювалася для створення розподілених, паралельних та стійких до відмови систем.

Вона була створена у 1986 році для того, щоб вирішити проблеми, пов’язані зі створенням масштабованих телекомунікаційних систем. Вони мали бути надійними, мати можливість оброблювати велику кількість різних з’єднання та оновлюватися без зупинки. Акторна модель у цій мові реалізовується за допомогою легковажних процесів, які виконується за допомогою Erlang Open Telecom Platform (BEAM або Erlang VM). Обмін повідомлень відбувається суто через асинхронну передачу повідомлень, за допомогою оператору ! та отримати повідомлення за допомогою receive. Кожен актор має власну скриньку повідомлень, тобто чергу повідомлень, в яку під час виконання програми потрапляють повідомлення і виконуються у порядку потрапляння. Щодо процесів, то Erlang не використовують спільну пам’ять і забезпечують ізоляцію акторів, за допомогою цього можна запобігти пошкодженню даних при одночасному доступі. Даними можна обмінюватися тільки через повідомлення, але перед цим треба їх скопіювати. Так само як і в Akka Erlang має всередині себе механізми відмовостійкості, що відстежують та перезапускають процеси, що вийшли з ладу. Ще одна особливість Erlang полягає у тому, що можна змінювати код у працюючій системі без його перезапуску, що є досить корисним для додатків високої доступності. Дана мова програмування також підтримує паралельні обчислення для взаємодії між різними вузлами у мережі. Також, варто згадати pattern matching, який дозволяє легше оброблювати різні типи повідомлень. Наостанок Erlang є функціональною мовою програмування, тому дані всередині функцій незмінні, самі по собі функції є first-class citizens, що якраз відповідає певним принципам акторної моделі.

Перейдемо до Elixir. Вона є динамічно типізованою, функціональною та розподіленою мовою програмування, яка також працює на основі Erlang VM. З’явися у 2011 році та створений Хосе Валімом. Сам по собі успадковує модель акторів, яка є у Erlang, але у противагу пропонує більш сучасний та зручний синтаксис, базуючись на іншій мові програмування Ruby. Надалі не будуть повторюватися особливості, які успадкувалися в Elixir від Erlang,

це паралелізм через акторів, стійкість до відмов, розподіленість. Щодо особливостей, можна виділити метапрограмування за допомогою системи макросів, яка дозволяє розробникам розширювати мову та створювати більш зручні та ефективні абстракції. Так як це більш нова мова програмування аніж Erlang, то вона постачається з додатковим інструментарієм який включає інструмент збірки Mix для управління проектами, а також управління залежностями проекту та тестування. Зазвичай Elixir використовується для веб-додатків, особливо активно, які працюють в реальному часі, розподілених систем, вбудованих систем з фреймворком Nerves. Щодо веб-додатків, то використовується фреймворк Phoenix, який є масштабованим, містить можливості роботи в реальному часі, побудований на основі акторної моделі та є досить швидким.

Далі йде Actix фреймворк для мови програмування Rust. Сама по собі мова зосереджена на продуктивності та безпечності, а також надає функціонал для роботи з низькорівневим паралельним програмуванням через потоки, канали, м’ютекси і не тільки. Але цього не завжди достатню, тому було створено Actix. Актори в цьому фреймворку є незалежними одиницями обчисленням, яка у свою чергу повністю інкапсулює стан і поведінку. Актори, так само як і в попередніх прикладах спілкуються між собою лише через повідомлення. Вони виступають в якості структур або переліку, тобто enum. Повідомлення також є асинхронними, тому не блокують загальний потік програми. Кожен актор має власну, унікальну адресу в системі Actix, а також поштову скриньку яка так само як і в Erlang та Elixir мовах по порядку у черзі оброблює повідомлення. До цього можна додати обробників, для відповідного повідомлення маємо відповідний обробник. Також контекст, який надає доступ до середовища виконання Actix, це дозволяє актору породжувати нових акторів, надсилати повідомлення іншим акторам, реєструвати таймери та періодичні завдання, доступ до власної адреси, зупинка себе або інших акторів. Цікавою особливістю для розгляду є система супервізорів. Вона дозволяє створювати

акторів-супервізорів, які в свою чергу можуть контролювати дочірніх акторів. В разі неполадки, він вирішує яким чином реагувати, перезапустити або зупинити актора.

1.2 Проблеми та обмеження існуючих підходів

Розглянуті вище мови програмування та фреймворки мають також різні недоліки, через які вони не завжди є найкращим рішенням у використанні для розробки додатків.

Почнемо з Erlang, ця мова має досить застарілий синтаксис, який пізніше покращили в Elixir. Для співставлення можна написати просту функцію з подвоєнням парних елементів у контейнері на мовах програмування Scala, Rust та Erlang.

У лістингу 1.1 можна побачити оголошення списку з шести елементів, після чого за допомогою зручної функції filter обираємо лише парні елементи, та наостанок за допомогою map подвоюємо ці елементи.

Лістинг 1.1 – Scala код для подвоєння парних елементів

```
val number = List(
  1, 2, 3, 4, 5, 6
)
val double_evens = numbers.filter(_ % 2 == 0).map(_ * 2)
println(s"Подвоєні парні: $double_evens")
```

Далі розглянемо Rust код у лістингу 1.2. Ідея аналогічна, тільки на рівні мови потрібно спочатку отримати ітератор на елементи контейнера, після чого всередині функції filter отримати значення посилання за допомогою оператора зірочки. В кінці зібрати усі елементи у контейнер.

Лістинг 1.2 – Rust код для подвоєння парних елементів

```
fn main() {
```

Продовження лістингу 1.2

```
  let numbers = vec![1, 2, 3, 4, 5, 6];
  let doubled_evens: Vec<i32> = numbers
    .iter()
    .filter(|x| *x % 2 == 0)
    .map(|x| x * 2)
    .collect();
  println!("Doubled evens: {:?}", double_evens);
}
```

Наостанок розглянемо Erlang, лістинг 1.3. Спочатку оголошуємо поточний модуль, після чого експортуємо функції doubled_even та main і вказуємо скільки аргументів вони приймають, у цьому випадку по одному. Основною складністю у читання коду, є функція doubled_even, у коді проходить ітерація по списку за допомогою синтаксису X <- List, далі виконується умова якщо елемент X є парним, тобто X rem 2 == 0, тоді виконується вираз 2*X і записується у список. Це є досить неочевидним і більш складним для читання ніж попередні два підходи.

Лістинг 1.3 – Erlang код для подвоєння парних елементів

```
-module(list_comp_example).
-export([doubled_even/1, main/1]).

doubled_even(List) ->
  [2*X || X <- List, X rem 2 == 0].

main(_) ->
  Numbers = [1, 2, 3, 4, 5, 6],
  Result = double_even(Numbers),
  io:format("Double evens: ~p~n", [Result]).
```

Також, через специфічний синтаксис та додатково функціональний підхід навчання може зайняти більше часу аніж на інших мовах

програмування. Щодо екосистеми, то Erlang обмежує вибір готових бібліотек, на противагу більш сучасному Elixir. Щодо продуктивності, то в Erlang кожен процес має свою власну копію пам’яті, це забезпечує ізольованість і зручність, якщо розглядати в контексті багатозадачності. Але при постійному копіюванні та серіалізації та десеріалізації повідомлень йде вище навантаження на процесор, через що знижується продуктивність. Окрім того, Erlang містить hot code swapping, через який BEAM погано оптимізує код, бо існують обмеження на зміни структури даних. До того ж пам’ять постійно збільшується, так як зберігається стара та нова версія коду. Варто зауважити, що Erlang, як і Elixir є динамічно типізованими мовами програмування, через що помилки з типами будуть виникати вже на етапі виконання коду, а не компіляції.

Тепер розглянемо Elixir. Хоч він і має більшу екосистему за Erlang, але вона все ще залишається досить малою відносно більш популярних мов, таких як Java, Scala та Rust. Існують складнощі з налагодженням розподілених систем. У звичайному монолітному застосунку, код виконується на одному пристрою, однопоточно чи багатопоточно. Але у розподіленій системі задача виконується на декількох пристроях одночасно. Тому відстеження послідовності подій та відладка є складнішими ніж у звичайних мовах програмування. Ще однією проблемою є залежність від Erlang VM, тому це може виступати як перевагою, так і недоліком. Окрім того є накладні витрати на абстрагування, наприклад GenSever behavior, в OTP. OTP – це набір бібліотек та шаблонів проектування для побудови масштабованих розподілених додатків на мові Erlang. GenServer у свою чергу надає стандартну структуру для створення акторів, які займаються обробкою повідомлень, підтримкою стану та керуванням своїм життєвим циклом.

Далі перейдемо до Scala фрейморку Akka. Він багатофункціональний та потужним, але достатньо складний через це, окрім того треба мати хоча б базові знання Scala, перед опануванням фреймворку, що збільшує криву

навчання. До проблем з продуктивністю можна віднести накладні витрати на JVM, це споживання пам’яті і швидкість виконання, так як Scala компілюється у байткод після чого виконується JVM. Варто зауважити що у Scala є Callback Hell проблема з ф’ючерсами. Основна проблема полягає в неправильному використання, після чого це призводить до вкладених зворотних викликів, через що код стає складно читати та підтримувати.

Наостанок розглянемо Actix, який також є фреймворком, але для мови програмування Rust. Основною проблемою є вивчення самої мови Rust, так як вона є достатньо складною для опануванням, через механізми запозичень та безпечним керувань пам’яті. Сам Rust надає підтримку асинхронного програмування за допомогою використання ключових слів async та await. Але через те, що це мова більш низькорівнева аніж попередні, то треба самостійно налаштовувати асинхронне виконання через бібліотеки tokio або async-std. Окрім того є проблеми з блокуванням або взаємодією з синхронним кодом. З боку системи керування життєвим циклом всередині Actix є складність для проектування супервізорів, бо треба конкретно вказувати які актори контролюють інших, що ускладнює логіку програми. Це за собою веде іншу проблему, таку як обробка складних сценаріїв помилок, що не передбачається при запуску звичайного імперативного коду. Також, супервізори ведуть за собою різні циклічні проблеми, наприклад є актор A та актор B, якщо один контролює іншого і навпаки, то можуть виникнути проблеми витоку пам’яті, або завершення обох акторів одночасно, якщо один з них не працює. На додачу може виникнути зациклення перезапусків акторів, проблема взаємного блокування та визначення відповідальності за помилку.

2 ПОСТАНОВКА ЗАДАЧІ

2.1 Визначення вимог до функціоналу

З огляду на переваги та недоліки поточних рішень, які розглянуті у першому розділі, можна визначити вимоги до реалізації мови програмування. Насамперед мова буде скриптовою, динамічно типізованою на базі парадигм імперативного стилю програмування, акторного та функціонального. Розроблена на базі віртуальної машини, але без прив’язки до BEAM, як це робить Elixir. Передача повідомлень між акторами буде відбуватися виключно через ключове слово .send(), що дозволяє повністю ізолювати обчислення всередині конкретного актора. Взаємодія між акторами відбувається лише через асинхронні повідомлення. Особливою перевагою є імплементація акторної моделі на рівні мови, що дає перевагу над Akka і Actix тим, бо немає накладних витрат у вигляді реалізації зовнішнього фрейморку над мовою. Окрім того синтаксис буде C-подібним, до якого набагато легше звикнути ніж до Erlang та Elixir. Синтаксис мови є доволі простим, на відміну від розглянутих мов та фреймворків. Також планується надання механізму вбудованих у мову асинхронних повідомлень, без блокування основного потоку виконання програми. Для облегшення роботи з пам’яттю зі сторони розробників на даній мові, буде зроблено автоматичний збірник сміття.

У багатьох мовах програмування ООП реалізовано на зразок того, що ми знаємо у Страуструпа та його мови C++, яка з’явилася у 1983 році. Основну концепцію ООП сформував Алан Кей у 1960-х роках і вона полягала у симуляції абсолютно незалежних сутностей які взаємодіють між собою за допомогою повідомлень. Сутності у цьому випадку інкапсульовані, з власним станом та поведінкою. Страуструп же спростив цю ідею, через що фокус перейшов більше на класи, успадкування, поліморфізм та віртуальні методи. Інкапсуляція теж була порушена, так як

методи і поля могли бути публічними. Насамперед це легше реалізувати та використовувати розробникам, окрім того це досить практично для використання. C++ надавав класи та інтеграцію з C, при тому нічого не порушуючи в початковій мові, надаючи високу продуктивність і контроль над пам’яттю. Ключовою фразою Алана Кея було те, що при створенні терміну «об’єктно орієнтовний» він не мав на увазі C++, підтверджуючи цим, що ідея була порушена.

Через це виникають проблеми з тісним з’єднанням, коли об’єкти мають посилання одне на одного, що знижує гнучкість і можливість легко змінювати компоненти, так як вони зв’язані. Через інкапсуляцію легко змінити код у будь-якому місці програми, через що проблему буде дуже складно шукати, бо відомо, що це могло статися не тільки всередині класу, а будь-де. Ще одна проблема з успадкуванням та повторним використанням коду, наприклад множинне наслідування, що призводить до ромбовидної проблеми або роздування та копіювання коду. Окрім того виникає велика проблема з паралелізмом, так як об’єкти легко можуть використовувати спільну пам’ять, що призводить до гонок даних, або дедлоків. Отже у підсумку було вирішено прибрати класи та поточну популярну ідею ООП з мови програмування і реалізувати акторну модель базуючись на деяких суміжних принципах ідеї ООП Алана Кея.

Тепер перейдемо до більш детального розгляду синтаксису мови програмування. Команда print … полягає у тому, що при виклику відповідного токену на рівні віртуальної машини, воно буде зразу відпрацьовувати та видавати результат, ще до повного виконання усіх інших інструкцій.

У мові програмування будуть передбачені такі стандартні типи даних, як булеві значення, цілі числа та з плаваючою точкою, строки, та nil тип, яким будуть ініціалізовуватися усі пусті змінні. Їх можна побачити у лістингу 2.1.

Лістинг 2.1 – Стандартні типи мови програмування

```
true
false

"Some string!"
""
"173"

993 // 1000 - 7
993.58

nil
```

Далі перейдемо до виразів, а саме арифметичних, синтаксис можна побачити у лістингу 2.2. У мові програмування буде 4 види арифметичних виразів, це оператори +, -, * та /, які є інфіксними, тобто йдуть після операндів, до того ж оператор - є також і префіксним, за допомогою чого можемо змінити знак операнду.

Лістинг 2.2 – Арифметичні оператори

```
first + second;
another - one;
mult * third;
divide / even;

-variable
```

Далі на черзі йдуть оператори порівняння та рівності, зображені у лістингу 2.3. Будь-які порівняння з різними типами, навіть якщо візуально вони ідентичні, будуть розглянуті як абсолютно різні значення.

Лістинг 2.3 – Оператори порівняння

```
first < second;
```

Продовження лістингу 2.3

```
another > one;
mult <= third;
last >= value;

18 == 32;
"var" != "val";
```

Логічні оператори. У мові програмування будуть реалізовані префіксний оператор not за допомогою знаку оклику, що зображає інверсію, а також два бінарних оператора and та or, перший повертає true, тільки якщо обидва є true, в іншому випадку false, а or повертає false, тільки якщо обидва операнди є false та в іншому випадку true. Приклад можна побачити у лістингу 2.4. При наявності бінарних операторів можна було зробити назви аналогічними до C для консистентності.

Лістинг 2.4 – Логічні оператори

```
!true; // false
!false; // true

true and false; // false
false or false; // false

true and true; // true
false or true; // true
```

Останній вид виразів, які будуть доступні в контексті виразів це дужки, вони дозволять групувати операції і підвищувати їх пріоритет. Пріоритетність усіх операцій співпадає з тим, що реалізовано в мові програмування C. Також, як і в будь-якій іншій мові програмування будуть доступні стейтменти. Їх також можна об’єднувати у блоки за допомогою

фігурних дужок, які у свою чергу будуть визначати область видимості, лістинг 2.5.

Лістинг 2.5 – Групування, пріоритетність, стейтменти та блоки

```
{
  print "One statement.";
  print (182 - 37) / 2 + 37;
}
```

Далі йдуть змінні та значення, лістинг 2.6. Перші оголошуються за допомогою ключового слова var і можуть змінюватися під час виконання програми, інші за допомогою val і їх вміст призведе до помилки виконання програми.

Лістинг 2.6 – Змінні та значення

```
val some_value = "here is my value";
var nil_variable;
var department = "AI";
print department; // AI
department = "not AI";
print department;
```

Важливим компонентом мови програмування також є контроль над потоком виконання. Цього можна досягти за допомогою ключових слів if, else, while та for, лістинг 2.7. усі ці ключові слова працюють аналогічно до того, як це реалізовано в C, для того щоб уникнути надлишкового пояснення.

Лістинг 2.7 – Контроль потоку виконання, ключові слова if, else, while, for

```
var a = 1;
while (a <= 10) {
```

Продовження лістингу 2.7

```
  if (a > 8) {
    print "a more than 8";
  } else if (a < 4) {
    print "a less than 4";
  } else {
    print "another value of the variable a";
  }
}

var res = 1;
for (var i = 2; i < 6; i = i + 1) {
  res = res * i;
}
```

Для виклику різного набору виразів декілька та для акторів в подальшому передбачається розроблення функцій, а також механізму closure, тобто замикання. Основною ідеєю замикання є те, що вона «пам’ятає» змінні з контексту, в якому вона була створена, навіть після завершення контексту. Кожна функція в майбутній мові програмування є функцією вищого порядку, тобто її можна присвоїти до змінної або значення, можна передати як аргумент до іншої функції, повернути всередині іншої функції. Тут додається новий оператор кома, який дозволяє розділяти аргументи між собою при передачі у функцію, лістинг 2.8.

Лістинг 2.8 – Функції та замикання

```
fun add_or_mul(add_op) {
  val text = "inner function";

  fun add(a, b) {
    print text;
    return a + b;
  }
```

Продовження лістингу 2.8

```
  fun mul(a, b) {
    print text;
    return a * b;
  }

  if add_op {
    return add;
  } else {
    return mul;
  }
}

val func = add_or_mul(true);
print func(2, 4);
```

Наостанок передбачається створення самих акторів, як незалежних та інкапсульованих одне від одних сутностей, лістинг 2.9. Актори оголошуються за допомогою ключового слова actor, їх можна ініціалізувати за допомогою назви та передачі аргументів до актора, як показано на рисунку. Актора також можна реалізувати без визначення повідомлення init, тоді він не буде мати аргументів при ініціалізація за замовчуванням. Дані можуть змінюватися лише всередині актора, а за його межами лише через передачу повідомлень. Першим неявним параметром у повідомленні є поточний актор, тобто this, за допомогою якого і можна взаємодіяти з даними. Після виконання програма виведе результат 9.5.

Лістинг 2.9 – Реалізація актору Fraction та його використання

```
actor Fraction {
  init(num, den) {
    this.num = num;
    this.den = den;
```

Продовження лістингу 2.9

```
  }
  get_and_divide_by(divisor, sender) {
    val result = this.num / divisor;

    // Send result back
    sender.send(reply, result);
  }

  reply(_value) {
    print 5 + _value; // expect: 5 + (9 / 2) = 9.5
  }
}

val frac = Fraction(9, 4);
frac.send(get_and_divide_by, 2, frac);
```

2.2 Архітектура акторної моделі

Процес виконання програмного коду відбувається таким чином. Спочатку вхідний код, це може бути файл будь-якого розширення передається у сканер, який посимвольно по ньому проходиться, далі генеруються токени, тобто вирази, ключові слова та інші одиниці коду, після цього компілятор оброблює ці токени та генерує байткод, які в кінці виконує віртуальна машина і розробник отримує результат.

У свою чергу актори повинні будуть спроектовані таким чином, щоб повністю інкапсулювати свій внутрішній стан. Через це токен «.» оброблюється тільки якщо знаходиться всередині якогось актора, варто зауважити що вкладені актори призведуть до помилки компіляції. За межами актора доступне тільки ключове слово «.send», яке дозволяє лише відправляти повідомлення, без будь-якої взаємодії з внутрішнім станом актора.

Кожен актор буде мати свій стек для віртуальної машини, чергу повідомлень яка буде постійно прослуховуватися для отримання нових повідомлень, та легковажний потік, який буде діставати повідомлення з черги та виконувати їх якщо черга не пуста. Легковажний потік у свою чергу це потік, який займає набагато менше пам’яті ніж звичайний і має менші витрати на перемикання контексту, у приклад можна привести горутини у мові програмування Go.

Повідомлення в акторах будуть викликатися за допомогою send(message, …args) і тільки. Операція send у свою чергу не блокує інші операції і не чекає на відповідь. Повідомлення не будуть повертати результат за допомогою ключового слова return, замість цього використовується трюк з перенаправленням результату у інше повідомлення, для імітації поведінки з повернення результати. Це корисно тим, що тепер актор може передавати результат лише через аргументи, тому легше буде відслідкувати який параметр всередині функції буде змінено, без необхідності шукати повернене значення. Також, це полегшує реалізацію віртуальної машини, так як стек після виконання повідомлення від актора ніяк не буде розширюватися, бо ми завжди знаємо, що функція виконається і розробник ніколи не отримає з неї результату, все буде відбуватися всередині актора. Окрім цього, не потрібно буде відслідковувати поки всередині повідомлення розробник отримає Future значення, бо його просто не існує, все через повідомлення, інший актор зможе отримати повідомлення та обробити його не раніше ніж виконається потрібний програмний код.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ

3.1 Загальний опис

У цьому розділі буде більш детально розглянуто програмну реалізацію додатку та його код. Даний підрозділ передбачає короткий опис наступних підрозділів, для більш зрозумілого заглиблення у деталі реалізації.

У виконаному додатку було реалізовано основну частину функціоналу, включаючи увесь перерахований функціонал з розділу 2, окрім імплементації асинхронності, через брак часу на виконання роботи. Але основні компоненти та синтаксис реалізовані в повному обсязі.

Тепер перейдемо до розгляду основного функціоналу. Потік виконання програми від файлу до виконаного результату можна побачити на рисунку 3.1.

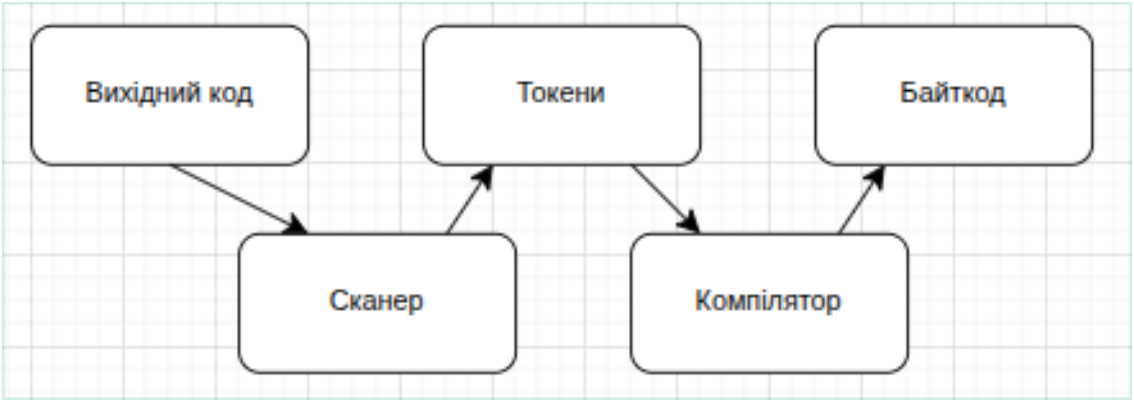

Рисунок 3.1 – Потік обробки вихідного коду

Так як мова програмування є скриптовою, то є можливість запускати її через REPL, або передаючи у виконуваний файл шлях до іншого файлу, який потрібно виконати. Приклад роботи REPL зображено на рисунку 3.2.

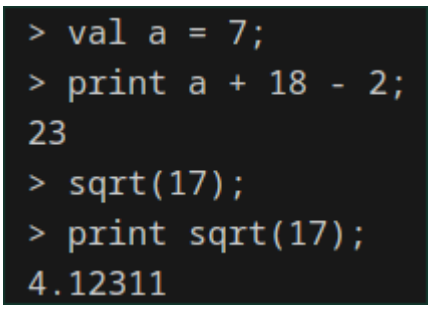

Рисунок 3.2 – Приклад виконання коду за допомогою REPL

Цей код працює в циклі, всередині запущеної функції, яка зображена у лістингу 3.1. Код виконується всередині виклику interpret, а до цього ми зчитуємо строчку коду, до символу нового рядка і передаємо туди, як вхідні дані. Як зазначено у коментарі, якщо було виявлено натискання клавіш Ctrl+D або Ctrl+X, то програма завершає свою роботу, окрім того працює комбінація клавіш Ctrl+C.

Лістинг 3.1 – Функція для виконання коду через термінал

```
// Read-Eval-Print Loop
static void repl() {
  // Some restriction for input line of code
  char line[1024];
  for (;;) {
    printf("> ");

    // Exit with Ctrl+D on Linux/Mac or Ctrl+X on windows
    if (!fgets(line, sizeof(line), stdin)) {
      break;
    }

    line[strcspn(line, "\n")] = '\0';
    if (strcmp(line, "exit") == 0) {
      break;
    }
```

Продовження лістингу 3.1

```
    interpret(line);
  }
}
```

Інша функція run_file, у лістингу 3.2, спочатку підвантажує файл у масив символів, після чого виконується код за допомогою аналогічної функції interpret. Далі ми очищуємо виділену пам’ять під масив, та в залежності від результату повертаємо код помилки, яка описана в специфікації FreeBSD.

Лістинг 3.2 – Функція run_file

```
static void run_file(const char *path) {
  char *source = read_file(path);
  const InterpretResult result = interpret(source);
  free(source);

  if (result == INTERPRET_COMPILE_ERROR) exit(65);
  if (result == INTERPRET_RUNTIME_ERROR) exit(70);
}
```

Перейдемо до функції interpret, показаною у лістингу 3.3. Мова не має стартової точки виконання програми, як наприклад у C, C++ чи Java, з певної функції main, тому ми робимо функцію обгортку для цього всередині компілятора. Якщо ми отримали помилку, тоді функція не створюється, та повертається помилка. В іншому випадку виконуються дивні операції на перший погляд, поміщаємо у віртуальну машину функцію, потім створюємо замикання, витягаємо функцію, поміщаємо замикання, та викликаємо його, наприкінці запускаємо віртуальну машину командою run.

Чому не можна було прибрати зайві виклики push та pop для змінної function? Основна проблема полягає в збірнику сміття. При створенні нового об’єкту є ймовірність виклику збірнку, який проходиться по пам’яті

віртуальної машини, та видаляє усі посилання, до яких не може дістатися. Якщо змінна function не буде розміщена на стеку віртуальної машини, тоді є ймовірність, що її пам’ять звільниться ще до виклику, через що ми отримаємо помилку виконання у C.

Лістинг 3.3 – Функція interpret

```
InterpretResult interpret(const char *source) {
  ObjFunction *function = compile(source);
  if (function == NULL) return INTERPRET_COMPILE_ERROR;

  push(OBJ_VAL((Obj*)function));
  ObjClosure *closure = new_closure(function);
  pop();
  push(OBJ_VAL((Obj*)closure));
  call(closure, 0);

  return run();
}
```

Наразі буде опущено пояснення основних функцій compile, яка буде розглянута у підрозділі 3.2, та push, pop, у підрозділі 3.4.

Перейдемо до розгляду функції віртуальної машини call, лістинг 3.4. Спочатку перевіряємо що кількість аргументів замикання співпадає з очікуваним, якщо не співпадає, тоді повертаємо помилку виконання. Далі перевіряємо вкладеність функцій, наразі значення ліміту є 256. Наприкінці в програмі виконується ініціалізація структури CallFrame, яка визначає функцію, з якої буде запускатися віртуальна машина, про що буде розказано у відповідному підрозділі. Це якраз і є основна ідея поточної функції, встановити стартову точку виконання для віртуальної машини, щоб run надалі змогла побачити цю функцію та виконати код.

Лістинг 3.4 – Функція віртуальної машини call

```
static bool call(ObjClosure *closure, const int arg_count)
{
  if (arg_count != closure->function->arity) {
    runtime_error("Expect %d arguments but got %d.",
      closure->function->arity, arg_count);
    return false;
  }

  if (vm.frame_count == FRAMES_MAX) {
    runtime_error("Stack overflow.");
    return false;
  }

  // TODO Maybe function for actor call frame
  CallFrame *frame = &vm.frames[vm.frame_count++];
  frame->closure = closure;
  frame->ip = closure->function->chunk.code;

  // Maybe bug here, when we expand memory, then we drop
vm.stack_top and here we have invalid...
  frame->slots = vm.stack_top - arg_count - 1;
  return true;
}
```

Тепер перейдемо до розгляду останньої функції у цьому підрозділі, run, лістинг 3.5. На цьому рисунку буде показана лише частина виконання, атк як це доволі велика функція і буде розглянута детально у підрозділі віртуальної машини.

Отже, більшість часу програма виконує код саме в цій функції, компілятор під час виконання випускає байткод, який поміщається всередину віртуальної машини, що і є компіляцією всієї програми. На цьому етапі ми у нескінченному циклі виконуємо інструкції, тобто опкоди, які взаємодіють зі стеком віртуальної машини, який має тип даних Value.

Опкод – це інструкція, яка визначає операцію, яка повинна бути виконана процесором.

Лістинг 3.5 – Частина основного циклу функції run

```
for (;;) {
#ifdef DEBUG_TRACE_EXECUTION
  printf("          ");
  for (const Value *slot = vm.stack; slot < vm.stack_top;
++slot) {
    printf("[ ");
    print_value(*slot);
    printf(" ]");
  }
  printf("\n");
  disassemble_instruction(&frame->closure->function-
>chunk,
    (int)(frame->ip - frame->closure->function-
>chunk.code));
#endif
    uint16_t instruction;
    switch (instruction = READ_WORD()) {
      case OP_CONSTANT: {
        const Value constant = READ_CONSTANT();
        push(constant);
        break;
      }
      case OP_NIL:   push(NIL_VAL); break;
      case OP_TRUE:  push(BOOL_VAL(true)); break;
      case OP_FALSE: push(BOOL_VAL(false)); break;
      case OP_POP:   pop(); break;
      case OP_SET_LOCAL: {
        const uint16_t slot = READ_WORD();
        frame->slots[slot] = peek(0);
        break;
      }
```

Для кращого розуміння, розглянемо простий код в режимі відладки на рисунку 3.3. Ініціалізуємо дві змінні, після чого виводимо результат їх порівняння. Бачимо, що під капотом компілятор випромінює інструкцію OP_CONSTANT, після чого ініціалізуємо змінну з ім’ям ‘a’, яка знаходиться на вершині стеку, індекс 0. Аналогічним чином виконуємо із змінною b, тільки в цьому випадку змінна ініціалізована як var, а не константа, тому резервується комірка з індексом 3 під це значення. На завершення у останньому стейтмені з ключовим словом print виконується декілька дій одночасно, а саме отримання глобальних змінних a та b, після чого їх порівняння за допомогою опкода OP_EQUAL, та вивід. Далі виконуються стандартні операції виходу з поточної функції верхнього рівня, тобто повернення пустого результату.

```
> val a = false; var b = true; print a == b;
== <script> ==
0000    1 OP_FALSE
0001    | OP_CONSTANT         1 'true'
0003    | OP_DEFINE_GLOBAL    0 'a'
0005    | OP_TRUE
0006    | OP_CONSTANT         3 'false'
0008    | OP_DEFINE_GLOBAL    2 'b'
0010    | OP_GET_GLOBAL       0 'a'
0012    | OP_GET_GLOBAL       2 'b'
0014    | OP_EQUAL
0015    | OP_PRINT
0016    | OP_NIL
0017    | OP_RETURN
false
```

Рисунок 3.3 – Виконання простої команди з виводом опкодів

3.2 Опис сканеру

Розглянемо сканер більш детально, його основною роботою є формування токенів з вихідного коду, та передачу їх до компілятора, що було показано у лістингу 3.1. У лістингу 3.7 зображено функцію compile. Спочатку ініціалізуються сканер, парсер та компілятор, після чого

виконується функція advance, яка сканує токени, і доходимо до основної частини, де коміплятор ітерується по токенам та оброблює код. В кінці закриваємо функцію, очищуємо компілятор, та повертаємо результат.

Лістинг 3.7 – Функція compile

```
ObjFunction *compile(const char *source) {
  init_scanner(source);
  Compiler compiler;
  init_compiler(&compiler, TYPE_SCRIPT);

  parser.had_error = parser.panic_mode = false;
  advance();

  while (!match(TOKEN_EOF)) {
    declaration();
  }

  ObjFunction *function = end_compiler();
  free_compiler(&compiler);
  return parser.had_error ? NULL : function;
}
```

Опустимо розгляд функцій компілятора до наступного підрозділа і перейдемо до ініціалізації сканеру, лістинг 3.8. Так як код виконується однопоточно, то маємо лише 1 сканер на всю програму, отже можемо створити його глобальною змінною в C. Він містить поле строчки коду та вказівники на початок токену та кінець. Наприклад ми маємо стейтмент print grade; в цьому випадку токенами є print, grade та ; тому нам треба визначити вказівник на строку початок цього токену та кінець, що структура Scanner і зберігає у свою чергу.

Щодо функції ініціалізації, то в програмі ставимо на початок зчитаного програмного коду з файлу або командного рядку.

Лістинг 3.8 – Структура Scanner, та функція init_scanner

```
typedef struct {
  const char *start;
  const char *current;
  int line;
} Scanner;

Scanner scanner;

void init_scanner(const char *source) {
  scanner.start = scanner.current = source;
  scanner.line = 1;
}
```

Далі розглянемо структуру Parser, яка є ключовою для роботи компілятора, лістинг 3.9. На всю програму також створюється 1 парсер, який використовує компілятор. Основна ідея полягає в тому, що ми одночасно і скануємо і компілюємо код, бо компілятору достатньо знати поточний та попередній токен для компіляції коду. Змінна had_error показує чи є помилки в коді, якщо є хоча б одна помилка, то значення стає true і код відповідно ніколи не буде виконаним. Щодо panic_mode, то ця змінна цікавіша, так як вона дозволяє компілятору не виводити цілий стек помилок із вкладенностями. Наприклад програміст не поставив дужку в кінці вкладеної функції, тоді немає сенсу виводити помилки у всіх зовнішніх функціях, так як проблема лише у поточній. Тому флаг panic_mode виставляється в 0, коли виходить за поточну область оголошення функції.

Лістинг 3.9 – Структура Parser

```
typedef struct {
  Token current;
  Token previous;
  bool had_error;
```

Продовження лістингу 3.9

```
  bool panic_mode;
} Parser;
```

Тепер розглянемо як компілятор сканує токени, за допомогою функції advance зображеної у лістингу 3.10. Поточний токен тепер стає попереднім, далі викликається основна функція сканеру, яка повертає структуру Token. Якщо отримали помилку, то виводимо помилку поточного токена, та продовжуємо сканувати, допоки не отримаємо валідний токен.

Лістинг 3.10 – Функція advance

```
static void advance() {
  parser.previous = parser.current;

  for (;;) {
    parser.current = scan_token();
    if (parser.current.type != TOKEN_ERROR)
      break;

    error_at_current(parser.current.start);
  }
}
```

Структура Token зображена у лістингу 3.11, який містить тип токену, вказівник на початок токену, його довжина та рядок коду, де він використовується.

Лістинг 3.11 – Структура Token та перечислення TokenType

```
typedef enum {
  // Single-character tokens
  TOKEN_LEFT_PAREN, TOKEN_RIGHT_PAREN,
  TOKEN_LEFT_BRACE, TOKEN_RIGHT_BRACE,
  TOKEN_COMMA, TOKEN_DOT, TOKEN_MINUS, TOKEN_PLUS,
```

Продовження лістингу 3.11

```
  TOKEN_SEMICOLON, TOKEN_SLASH, TOKEN_STAR,

  // One or two character tokens
  TOKEN_BANG, TOKEN_BANG_EQUAL,
  TOKEN_EQUAL, TOKEN_EQUAL_EQUAL,
  TOKEN_GREATER, TOKEN_GREATER_EQUAL,
  TOKEN_LESS, TOKEN_LESS_EQUAL,

  // Literals
  TOKEN_IDENTIFIER, TOKEN_STRING, TOKEN_NUMBER,

  // Keywords
  TOKEN_OR, TOKEN_AND, TOKEN_IF, TOKEN_ELSE, TOKEN_PRINT,
  TOKEN_WHILE, TOKEN_FOR, TOKEN_FUN, TOKEN_NIL,
  TOKEN_RETURN, TOKEN_SEND, TOKEN_THIS, TOKEN_ACTOR,
  TOKEN_TRUE, TOKEN_FALSE, TOKEN_VAL, TOKEN_VAR,

  TOKEN_ERROR, TOKEN_EOF
} TokenType;

typedef struct {
  TokenType type;
  const char *start;
  int length;
  int line;
} Token;
```

Розглянемо основну функцію сканера, а саме scan_token, показаного у лістингу 3.12. Після ініціалізації сканеру, можемо ітеруватися по символам та створювати токени. Спочатку пропускаємо будь-які символи пробілу, табів, повернення каретки, та коментарі, які визначаються ідентично до мов програмування C або C++. Далі виконується перевірка на символ кінця, а саме нуль-символ, в цьому випадку генеруємо TOKEN_EOF, що означає

кінець файлу. Після цього отримуємо наступний символ за допомогою функції advance, яка у свою чергу інкрементує вказівник current змінної у сканеру та повертає символ. Якщо змінна є літерою, тоді повертаємо токен ідентифікатору, якщо числом, то токен числа. В усіх інших випадках можна побачити розгалуження для кожного символа та відповідний токен для цього.

Лістинг 3.12 – Функція scan_token

```
Token scan_token() {
  skip_whitespace();
  scanner.start = scanner.current;

  if (is_at_end()) return make_token(TOKEN_EOF);

  const char c = advance();
  if (is_alpha(c)) return identifier();
  if (is_digit(c)) return number();

  switch (c) {
    case '(': return make_token(TOKEN_LEFT_PAREN);
    case ')': return make_token(TOKEN_RIGHT_PAREN);
    case '{': return make_token(TOKEN_LEFT_BRACE);
    case '}': return make_token(TOKEN_RIGHT_BRACE);
    case ';': return make_token(TOKEN_SEMICOLON);
    case ',': return make_token(TOKEN_COMMA);
    case '-': return make_token(TOKEN_MINUS);
    case '+': return make_token(TOKEN_PLUS);
    case '/': return make_token(TOKEN_SLASH);
    case '*': return make_token(TOKEN_STAR);
    case '.':
      return make_token(is_send() ? TOKEN_SEND :
TOKEN_DOT);
    case '!':
```

Продовження лістингу 3.12

```
      return make_token(match('=') ? TOKEN_BANG_EQUAL :
TOKEN_BANG);
    case '=':
      return make_token(match('=') ? TOKEN_EQUAL_EQUAL :
TOKEN_EQUAL);
    case '<':
      return make_token(match('=') ? TOKEN_LESS_EQUAL :
TOKEN_LESS);
    case '>':
      return make_token(match('=') ? TOKEN_GREATER_EQUAL :
TOKEN_GREATER);
    case '"': return string();
    default : return error_token("Unexpected character.");
  }
}
```

Розглянемо функції make_token, у лістингу 3.13. Створюємо структуру токену та ініціалізуємо поля від сканеру, так як це відображення його поточного стану. Як і було розглянуто раніше, поточний символ йде до закінчення будь-якого токену, тому різниця між ним та стартовим символом і є довжиною.

Лістинг 3.13 – Функція make_token

```
static Token make_token(const TokenType type) {
  Token token;
  token.type = type;
  token.start = scanner.start;
  token.length = (int)(scanner.current - scanner.start);
  token.line = scanner.line;
  return token;
}
```

Перейдемо до розгляду функції number, зображеною у лістингу 3.14. Зчитуємо значення до того моменту, поки поточний символ є цифрою, далі якщо маємо справу з числом із плаваючою точкою, то також поглинаємо усі символи після крапки, після чого випускаємо TOKEN_NUMBER.

Лістинг 3.14 – Функція number

```
static Token number() {
  while (is_digit(peek())) advance();

  // Look for a fractional part
  if (peek() == '.' && is_digit(peek_next())) {
    // Consume the "."
    advance();

    while (is_digit(peek())) advance();
  }

  return make_token(TOKEN_NUMBER);
}
```

Тепер розглянемо код функцій identifier та identifier_type показані у лістингу 3.15. Так як перший символ вже було оброблено у лістингу 3.12, то далі проводимо ітерацію допоки зустрічаємо літеру або цифру. Після цього визначаємо тип ідентифікатору. Це у свою чергу може бути ключове слово, змінна або значення.

Лістинг 3.15 – Функції identifier та identifier_type

```
static Token identifier() {
  while (is_alpha(peek()) || is_digit(peek())) advance();
  return make_token(identifier_type());
}
```

У лістингу 3.15 створюється токен на основі функції identifier_type, зображеною у лістингу 3.16, тип токену визначається за допомогою префіксного дерева у вигляді switch-case блоку. Trie працює таким чином, що пошук відбувається не довше ніж за O(n) часу, де n – кількість символів у рядку, що є досить зручним, так як не потрібні накладні витрати на збереження усіх строк, або пошуку по масиву і постійному порівнянні. У цьому коді можна знайти всі ключові слова, які використовуються у поточній мові програмування, такі як if, else, actor, nil, or, print і так далі.

Лістинг 3.16 – Функція identifier_type

```
static TokenType identifier_type() {
  // Or maybe *scanner.start
  switch (scanner.start[0]) {
    case 'a':
      if (scanner.current - scanner.start > 1) {
        switch (scanner.start[1]) {
          case 'c': return check_keyword(2, 3, "tor",
TOKEN_ACTOR);
          case 'n': return check_keyword(2, 1, "d",
TOKEN_AND);
        }
      }
      break;
    case 'e': return check_keyword(1, 3, "lse",
TOKEN_ELSE);
    case 'f':
      if (scanner.current - scanner.start > 1) {
        switch (scanner.start[1]) {
          case 'a': return check_keyword(2, 3, "lse",
TOKEN_FALSE);
          case 'o': return check_keyword(2, 1, "r",
TOKEN_FOR);
          case 'u': return check_keyword(2, 1, "n",
TOKEN_FUN);
```

Продовження лістингу 3.16

```
        }
      }
      break;
    case 'i': return check_keyword(1, 1, "f", TOKEN_IF);
    case 'n': return check_keyword(1, 2, "il", TOKEN_NIL);
    case 'o': return check_keyword(1, 1, "r", TOKEN_OR);
    case 'p': return check_keyword(1, 4, "rint",
TOKEN_PRINT);
    case 'r': return check_keyword(1, 5, "eturn",
TOKEN_RETURN);
    case 's': return check_keyword(1, 3, "end",
TOKEN_SEND); // associated with the is_send function
    case 't':
      if (scanner.current - scanner.start > 1) {
        switch (scanner.start[1]) {
          case 'h': return check_keyword(2, 2, "is",
TOKEN_THIS);
          case 'r': return check_keyword(2, 2, "ue",
TOKEN_TRUE);
        }
      }
      break;
    case 'v':
      if (scanner.current - scanner.start > 2 &&
scanner.start[1] == 'a') {
        switch (scanner.start[2]) {
          case 'l': return TOKEN_VAL;
          case 'r': return TOKEN_VAR;
        }
      }
      break;
    case 'w': return check_keyword(1, 4, "hile",
TOKEN_WHILE);
  }
```

Продовження лістингу 3.16

```
  return TOKEN_IDENTIFIER;
}
```

Остання функція, яка буде розглянута у сканері це check_keyword, показана у лістингу 3.17. Вона приймає індекс початку та довжину строки після за допомогою вбудованої С функції memcmp порівнює дві строки на рівність і повертає очікуваний тип, якщо він вірний, інакше TOKEN_IDENTIFIER.

Лістинг 3.17 – Функція check_keyword

```
static TokenType check_keyword(const int start, const int
length,
    const char *rest, const TokenType type) {
  if (scanner.current - scanner.start == start + length &&
      memcmp(scanner.start + start, rest, length) == 0) {
    return type;
  }
  return TOKEN_IDENTIFIER;
}
```

3.3 Опис компілятору

У цьому підрозділі повністю розглянемо компілятор та як він працює. Згадаємо лістинг 3.7 та функцію compile, там залишилося 4 не розглянутих функцій, а саме init_compiler, declaration, end_compiler та free_compiler. Розпочнемо розгляд з init_compiler зображеною у лістингу 3.18. Щодо параметрів, то в якості аргументів приймаються компілятор та тип функції, FunctionType у свою чергу це перелік з варіантами TYPE_SCRIPT, TYPE_FUNCTION та TYPE_MESSAGE. Так як це фукція верхнього рівня, то це TYPE_SCRIPT. Далі видно ініціалізацію всіх полів компілятора та його вкладених змінних. Наприкінці штучно вводимо додаткову змінну this,

це потрібно для повідомлень всередині акторів, щоб вони могли отримувати доступ до свого стану.

Лістинг 3.18 – Функція init_compiler

```
static void init_compiler(Compiler *compiler, const
FunctionType type) {
  compiler->enclosing = current;
  compiler->function = NULL;
  compiler->type = type;
  compiler->local_count = 0;
  compiler->scope_depth = 0;

  compiler->local_capacity = GROW_CAPACITY(0);
  compiler->upvalue_capacity = GROW_CAPACITY(0);
  compiler->locals = NULL;  // To prevent UB from compiler
  compiler->upvalues = NULL; // Same reason
  compiler->locals = GROW_ARRAY(Local, compiler->locals, 0,
compiler->local_capacity);
  compiler->upvalues = GROW_ARRAY(Upvalue, compiler-
>upvalues, 0, compiler->upvalue_capacity);

  compiler->function = new_function();
  current = compiler;

  // Function live like global vars, so need own copy
  if (type != TYPE_SCRIPT) {
    current->function->name =
copy_string(parser.previous.start,
parser.previous.length);
  }

  Local *local = &current->locals[current->local_count++];
  local->depth = local->name.length = 0;
  local->constant = true;
  local->is_captured = false;
```

Продовження лістингу 3.18

```
  if (type == TYPE_MESSAGE) {
    local->name.start = "this";
    local->name.length = 4;
  } else {
    local->name.start = "";
    local->name.length = 0;
  }
}
```

Далі розглянемо структуру Compiler, лістинг 3.19. Сам Compiler є однозв’язним списком, де змінна enclosing виступає у вигляді next вказівника. Основна ідея полягає у тому, що для кожної функції створюється свій компілятор, який зберігає функцію, яка компілюється, її тип, усі локальні зміни, та копії змінних на рівні вище, як частина реалізації замикання, а також їх метадані.

Лістинг 3.19 – Структура Compiler

```
typedef struct Compiler {
  struct Compiler *enclosing;
  ObjFunction *function;
  FunctionType type;

  Local *locals;  // Array
  int local_capacity;
  int local_count;

  Upvalue *upvalues;  // Array
  int upvalue_capacity;
  int scope_depth;
} Compiler;
```

Перейдемо до структур Local та Upvalue у лістингу 3.20. Розглянемо структуру Local, вона містить свій токен, це може бути наприклад тип даних, булеве значення чи є ця змінна константою чи ні, depth визначає рівень вкладеності, щодо is_captured, то ця змінна визначає, чи захоплює замикання її чи ні, щоб вона не пропала із стеку віртуальної машини. Структура Upvalue у свою чергу містить булеве значення is_local, яке визначає це поточна локальна змінна, чи інше значення на рівень вище, index показує відступ у стеку віртуальної машини або всередині масиву upvalues і співпадає з індексом у масиві Local.

Лістинг 3.20 – Структири Local та Upvalue

```
typedef struct {
  Token name;
  int depth;
  bool is_captured;
  bool constant;
} Local;

typedef struct {
  uint16_t index;
  bool is_local;
} Upvalue;
```

Далі розглянемо функції free_compiler та end_compiler, перша достатньо проста, так як все що вона робить, то це видаляє динамічно виділені масиви locals та upvalues. Функція end_compiler зображена у лістингу 3.21. Ми робимо повернення в байткод, за допомогою emit_return та повертаємо поточну функцію, а також закриваємо current компілятор.

Лістинг 3.21 – Функція end_compiler

```
static ObjFunction *end_compiler() {
  emit_return();
```

Продовження лістингу 3.21

```
  ObjFunction *function = current->function;
#ifdef DEBUG_PRINT_CODE
  if (!parser.had_error) {
    disassemble_chunk(current_chunk(), function->name != 
NULL
        ? function->name->chars : "<script>");
  }
#endif

  current = current->enclosing;
  return function;
}
```

Далі розглянемо основну функції declaration у лістингу 3.22, яка компілює основний код. Тут зображені загальні та більш високорівневі токени для обробки, усі інші обробляються всередині statement функції і будуть розглянуті пізніше.

Лістинг 3.22 – Функція declaration

```
static void declaration() {
  if (match(TOKEN_ACTOR)) {
    actor_declaration();
  } else if (match(TOKEN_FUN)) {
    fun_declaration();
  } else if (match(TOKEN_VAL)) {
    var_declaration(true);  // constant
  } else if (match(TOKEN_VAR)) {
    var_declaration(false); // variable
  } else {
    statement();
  }
  if (parser.panic_mode) synchronize();
}
```

Розглянемо функцію fun_declaration, показаною у лістингу 3.23. Тут оголошено змінну constant як заглушку, це потрібно для функцій parse_variable та define_variable, які у свою чергу ініціалізують змінні і їм потрібно заздалегідь знати можна її змінювати чи ні. Після цього створюємо відповідно функцію та оголошуємо її як константу з ім’ям global.

Лістинг 3.23 – Функція fun_declaration

```
static void fun_declaration() {
  const bool constant = true; // plug
  const uint16_t global = parse_variable("Expect function
name.", constant);
  mark_initialized();
  function(TYPE_FUNCTION);
  define_variable(global, constant);
}
```

Детальніше розглянемо основну функцію function, лістинг 3.24. Ініціалізуємо вкладений компілятор і збільшуємо глибину області видимості на одиницю, всередині begin_scope функції. Далі оброблюються аргументи функції в круглих дужках і зчитуємо допоки зустрічаємо TOKEN_COMMA. Після цього очікується закриття круглих дужок та відкриття нової області видимості через фігурні дужки. Основна компіляція коду функції виконується всередині block функції. В кінці отримуємо поточну скомпільовану функцію, випускаємо її опкод як обгортку OP_CLOSURE, так як усі функції є замиканням, після цього поміщаємо усі upvalue в стек, для їх подальшого можливого використання, очищаємо компілятор. У function виклику end_scope, так як він не потрібен, всередині begin_scope збільшується змінна scope_depth, але немає потреби її змінювати, якщо поточний компілятор буде очищено.

Лістинг 3.24 – Функція fucntion

```
static void function(const FunctionType type) {
  Compiler compiler;
  init_compiler(&compiler, type);
  begin_scope();

  consume(TOKEN_LEFT_PAREN, "Expect '(' after function
name.");
  // Pass params here
  if (!check(TOKEN_RIGHT_PAREN)) {
    do {
      current->function->arity++;
      if (current->function->arity > 65535) {
        error_at_current("Can't have more than 65535
parameters.");
      }

      const uint16_t constant = parse_variable("Expect
parameter name.", true);
      define_variable(constant, true);
    } while (match(TOKEN_COMMA));
  }
  consume(TOKEN_RIGHT_PAREN, "Expect ')' after
parameters.");
  consume(TOKEN_LEFT_BRACE, "Expect '{' before function
body.");
  block();

  ObjFunction *function = end_compiler();
  emit_bytes(OP_CLOSURE,
make_constant(OBJ_VAL((Obj*)function)));

  for (int i = 0; i < function->upvalue_count; ++i) {
    emit_byte(compiler.upvalues[i].is_local ? 1 : 0);
    emit_byte(compiler.upvalues[i].index);
```

Продовження лістингу 3.24

```
  }

  free_compiler(&compiler);
}
```

Заглибимося у функцію block, лістинг 3.25. Тут компілюються токени за допомогою функції declaration, лістинг 3.22, допоки не зустріли кінець програми або дужку закриття. В кінці зчитується закриваюча фігурна дужка і видається помилку, якщо її не отримано.

Лістинг 3.25 – Функція block

```
static void block() {
  while (!check(TOKEN_RIGHT_BRACE) && !check(TOKEN_EOF)) {
    declaration();
  }

  consume(TOKEN_RIGHT_BRACE, "Expect '}' after block.");
}
```

Тепер перейдемо до розгляду функції actor_declaration, показаного у лістингу 3.26. Спочатку отримуємо ім’я актора та створюємо константу, після чого повертаємо індекс константи всередині вкладеного масиву значень. Далі за допомогою функції declare_variable з аргументом true, який означає константність змінної, записуємо значення в кінець в кінець динамічного масиву Local, функції яких будуть пропущені, бо мають багато низькорівневих деталей. Далі додаємо опкод актора, після чого за допомогою функції define_variable оголошуємо актора як глобальну зміну або поміщаємо в масив Local, якщо оголошений всередині функції. Актор також може бути оголошений всередині повідомлення іншого актора, але не на їх рівні. Через цю вкладеність як раз і створюється ActorCompiler екземпляр класу. Далі за допомогою функції named_variable створюємо

опкод для отримання актора на вершині стеку для того, щоб оголосити всі наступні повідомлення і прив’язати їх до актора відповідно до закриття фігурної дужки. В кінці видаляємо цей опкод та повертаємося на рівень нижче в списку акторів, якщо такі існують.

Лістинг 3.26 – Функція actor_declaration

```
static void actor_declaration() {
  consume(TOKEN_IDENTIFIER, "Expect actor name.");
  Token actor_name = parser.previous;
  uint16_t                name_constant                =
identifier_constant(&parser.previous);
  declare_variable(true);
  emit_bytes(OP_ACTOR, name_constant);
  define_variable(name_constant, true);

  ActorCompiler actor_compiler;
  actor_compiler.enclosing = current_actor;
  current_actor = &actor_compiler;

  named_variable(actor_name, false);
  consume(TOKEN_LEFT_BRACE,   "Expect   '{'   before   actor
body.");
  while (!check(TOKEN_RIGHT_BRACE) && !check(TOKEN_EOF)) {
    message();
  }
  consume(TOKEN_RIGHT_BRACE,   "Expect   '}'   after   actor
body.");
  emit_byte(OP_POP);

  current_actor = current_actor->enclosing;
}
```

Оголошення повідомлень відбуваються досить просто на базі function, зображені у лістингу 3.27. По суті це так ж сама функція описана у лістингу 3.23, але з певними обмеженнями, які описані у розділі 2, тож у

коді спочатку отримується назва функції, потім передається тип і викликається функція з лістингу 3.24, після чого випускається відповідний сигнал для віртуальної машини.

Лістинг 3.27 – Функція message

```
static void message() {
  consume(TOKEN_IDENTIFIER, "Expect message name.");
  const uint16_t constant =
identifier_constant(&parser.previous);

  function(TYPE_MESSAGE);
  emit_bytes(OP_MESSAGE, constant);
}
```

Розглянемо останню невідому функцію з actor_declaration, це named_variable, Лістинг 3.28. Всередині неї визначається що буде віртуальна машина робити зі змінною, змінювати чи отримувати до неї доступ. Також за допомогою функцій resolve_local та resolve_upvalue визначається індекс значення у відповідних масивах, якщо такі існують, у протилежному випадку створюється нова константа в масиві глобальних змінних і повертається індекс на неї. В кінці перевіряється чи можна взагалі присвоїти цю змінну і чи є вона константою і в залежності від цього випускається відповідний опкод.

Лістинг 3.28 – Функція named_variable

```
static void named_variable(const Token name, const bool
can_assign) {
  uint16_t get_op, set_op;
  bool constant = false;
  int arg = resolve_local(current, &name, &constant);

  if (arg != -1) {
```

Продовження лістингу 3.28

```
    get_op = OP_GET_LOCAL;
    set_op = OP_SET_LOCAL;
  } else if ((arg = resolve_upvalue(current, &name)) != -
1) {
    get_op = OP_GET_UPVALUE;
    set_op = OP_SET_UPVALUE;
  } else {
    arg = identifier_constant(&name);
    get_op = OP_GET_GLOBAL;
    set_op = OP_SET_GLOBAL;
  }

  if (can_assign && match(TOKEN_EQUAL)) {
    expression();
    if (constant) {
      error_at(&name, "Can't reassign a constant");
      return;
    }
    emit_bytes(set_op, (uint16_t)arg);
  } else {
    emit_bytes(get_op, (uint16_t)arg);
  }
}
```

Для подальшого пояснення компілятора потрібно розглянути те, яким чином працює парсер коду. Було розібрано та реалізовано парсер Пратта, згідно навчальних матеріалів Crafting Compilers.

Отже, основну структуру парсера зображено на рисунку 3.4. Тут можна побачити основні операції, з яких складається компілятор, та які обробляє.

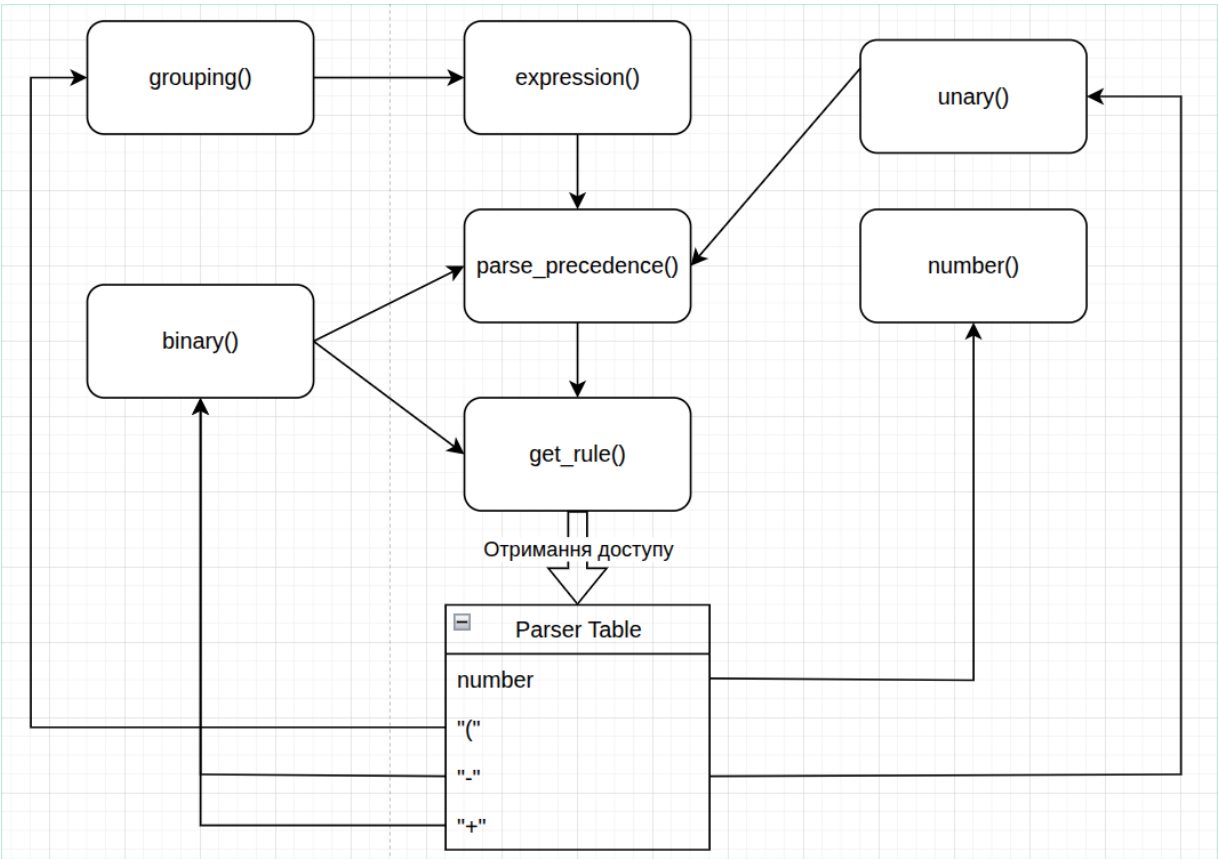

Рисунок 3.4 – Структура викликів функцій в парсері Пратта

Тепер перейдемо до лістингу 3.29, на якому зображена частина правил для парсингу усіх токенів. Це зручно тим, що в одній функції на основі токена і його індекса, який ініціалізовано за допомогою нотації C99, можна отримати заздалегідь визначено структуру, яка містить вказівники на функції. Тому під час компіляції на основі токена можна легко отримати функцію, яка буде його оброблювати.

Лістинг 3.29 – Масив правил парсингу rules

```
ParseRule rules[] = {
  [TOKEN_LEFT_PAREN]     = {grouping, call,   PREC_CALL},
  [TOKEN_RIGHT_PAREN]    = {NULL,     NULL,   PREC_NONE},
  [TOKEN_LEFT_BRACE]     = {NULL,     NULL,   PREC_NONE},
  [TOKEN_RIGHT_BRACE]    = {NULL,     NULL,   PREC_NONE},
  [TOKEN_COMMA]          = {NULL,     NULL,   PREC_NONE},
```

Продовження лістингу 3.29

```
  [TOKEN_DOT]           = {NULL,     dot,    PREC_CALL},
  …
}
```

Тепер перейдемо до типу структури ParseRule та переліку Precedence, лістинг 3.30, яка зберігає вказівники на префіксну та інфіксну функцію, а також пріоритетність, порядок яких важливий, бо чим вище індекс, тим вища пріоритетність. Наприклад кругла дужка виступає як оператор групування і є префіксним, або у випадку з функцією є початком її виклику, бо далі йдуть аргументи до цієї функції. Також, щодо пріоритетності, то немає потріби зберігати для префіксної функції, так як там вони однакові.

Лістинг 3.30 – Структура ParseRule

```
typedef struct {
  ParseFn prefix;
  ParseFn infix;
  Precedence precedence;
} ParseRule;

typedef void (*ParseFn)(bool can_assign);

typedef enum {
  PREC_NONE,
  PREC_ASSIGNMENT,  // =
  PREC_OR,          // or
  PREC_AND,         // and
  PREC_EQUALITY,    // == !=
  PREC_COMPARISON,  // < > <= >=
  PREC_TERM,        // + -
  PREC_FACTOR,      // * /
  PREC_UNARY,       // ! -
  PREC_CALL,        // . () send
```

Продовження лістингу 3.30

```
  PREC_PRIMARY
} Precedence;
```

Тепер розглянемо ядро самого парсеру всередині функції parse_precedence, лістинг 3.31. Основна його задача парсити усі вирази, які мають таку ж пріоритетність як у precedence параметру або вищу. Перевіряємо, чи існує для поточного виразу правило для парсингу, і якщо так, то викликаємо. Єдине що, для оголошення змінних, потрібно знати чи можемо ми їх присвоювати чи ні, тому все що має пріоритет нижче за PREC_ASSIGNMENT можна присвоїти, для уникнення виразу a * b = c * d. Але проблема з присвоєнням не вирішена, так як цей токен просто ігнорується і буде оброблено пізніше, хоча він повинен видати помилку, тому останній if-блок запобігає цьому і виводить помилку компіляці. Після цього компілятор проходиться по всім іншим операціям, що видно з циклу і викликає інфіксний оператор. Код достатньо простий, а в результаті виконується парсинг усього коду.

Лістинг 3.31 – Функція parse_precedence

```
static void parse_precedence(const Precedence precedence)
{
  advance();
  const           ParseFn           prefix_rule           =
get_rule(parser.previous.type)->prefix;
  if (prefix_rule == NULL) {
    error("Expect expression.");
    return;
  }

  const bool can_assign = precedence <= PREC_ASSIGNMENT;
  prefix_rule(can_assign);
```

Продовження лістингу 3.31

```
  while (precedence <= get_rule(parser.current.type)-
>precedence) {
    advance();
    const ParseFn infix_rule = 
get_rule(parser.previous.type)->infix;
    infix_rule(can_assign);
  }

  if (can_assign && match(TOKEN_EQUAL)) {
    error("Invalid assignment operator.");
  }
}
```

Тепер повернемося до функцій var_declaration та statement, розглянемо першу, у лістингу 3.32. За допомогою функції parse_variable отримуємо індекс змінної в масиві Value. Далі перевіряємо чи є присвоєння, якщо є тоді expression функція робить виклик parse_precedence з Precedence рівним PREC_ASSIGNMENT, інакше повертає випускає OP_NIL. В кінці отримуємо крапку з комою як закриття стейтменту та визначаємо змінну.

Лістинг 3.32 – Функція var_declaration

```
static void var_declaration(const bool constant) {
  const uint16_t global = parse_variable("Expect variable
name", constant);
  if (match(TOKEN_EQUAL)) {
    expression();
  } else {
    emit_byte(OP_NIL);
  }
  consume(TOKEN_SEMICOLON, "Expect ';' after variable
declaration");
  define_variable(global, constant);
}
```

Тепер перейдемо до функції statement, лістинг 3.33, в якій обробляються усі інші вирази. Розглянемо лише деякі з цих функцій, так як інші мають схожу структуру. Буде опущено вираз з block, так як усі ці функції були розглянуті раніше, if_statement, for_statement та expression_statement.

Лістинг 3.33 – Функція statement

```
static void statement() {
  if (match(TOKEN_PRINT)) {
    print_statement();
  } else if (match(TOKEN_FOR)) {
    for_statement();
  } else if (match(TOKEN_IF)) {
    if_statement();
  } else if (match(TOKEN_RETURN)) {
    return_statement();
  } else if (match(TOKEN_WHILE)) {
    while_statement();
  } else if (match(TOKEN_LEFT_BRACE)) {
    begin_scope();
    block();
    end_scope();
  } else {
    expression_statement();
  }
}
```

Почнемо з print_statement, лістинг 3.34. Парсимо вираз за допомогою відповідної функції, та очікуємо його закриття, після випускаємо опкод OP_PRINT. Функція expression_statement відрізняється лише тим, що в кінці випускаємо опкод OP_POP, який означає витягування останнього значення із стеку викликів.

Лістинг 3.34 – Функція print_statement

```
static void print_statement() {
  expression();
  consume(TOKEN_SEMICOLON, "Expect ';' after value.");
  emit_byte(OP_PRINT);
}
```

Функцію return_statement зображено у лістингу 3.35. При умові якщо це функція верхнього рівня, або повідомлення всередині актора, то буде помилка компіляції. Якщо немає типу повернення, то за замовчуванням повертається OP_NIL, в іншому випадку виконуємо вираз та повертаємо OP_RETURN.

Лістинг 3.35 – Функція return_statement

```
static void return_statement() {
  if (current->type == TYPE_SCRIPT) {
    error("Can't return from top-level code.");
  }

  if (current->type == TYPE_MESSAGE) {
    error("Can't return a value from a message.");
  }

  if (match(TOKEN_SEMICOLON)) {
    emit_return();
    return;
  }

  expression();
  consume(TOKEN_SEMICOLON,   "Expect   ';'   after   return
value.");
  emit_byte(OP_RETURN);
}
```

Останньою функцією буде while_statement, вказаною у лістингу 3.36. Спочатку отримуємо довжину коду всередині чанку, тобто самого while циклу. Далі отримуємо умову всередині круглих дужок. Далі виникає проблема, якщо умова не виконується, тоді треба зразу вийти з циклу, але ми ще не дійшли до кінця циклу і невідомо якої довжини вона буде. У цьому випадку ми записуємо у стек фіктивні значення за допомогою інструкції emit_jump і повертаємо індекс всередині віртуальної машини на це значення. Інструкція OP_POP потрібна для виходу з циклу, а emit_loop функція потрібна для визначення старту циклу. Після statement компілятор вже дійшов до кінця циклу, тому знає адресу виходи, і можна за індексом exit_jump перезаписати значення для виходу з циклу.

Лістинг 3.36 – Функція while_statement

```
static void while_statement() {
  const int loop_start = current_chunk()->length;
  consume(TOKEN_LEFT_PAREN, "Expect '(' after 'while'.");
  expression();
  consume(TOKEN_RIGHT_PAREN,     "Expect     ')'     after
condition.");

  const int exit_jump = emit_jump(OP_JUMP_IF_FALSE);
  emit_byte(OP_POP);
  statement();
  emit_loop(loop_start);
  patch_jump(exit_jump);
  emit_byte(OP_POP);
}
```

3.4 Опис віртуальної машини

У цьому розділі буде розглянута остання компонента мови програмування, а саме віртуальна машина. Основне виконання виконується

у функції run, яка була частково розглянута у лістингу 3.5, який знаходиться у підрозділі 3.1. Спочатку розглянемо допоміжну структуру CallFrame, лістинг 3.37. Вона зберігає closure, яке містить компільовану функцію з виконуваним кодом, далі йде ip, покажчик інструкції, який є вказівником на інструкцію розміром в 2 байти і slots, масив змінних, які доступні віртуальній машині. Ця структура дозволяє переходити до конкретної функції і виконувати її код, після чого повертатися на те місце, звідки було призупинене виконання інших інструкції віртуальної машини.

Лістинг 3.37 – Допоміжна структура CallFrame

```
typedef struct {
  ObjClosure *closure;

  uint16_t *ip;
  Value *slots;
} CallFrame;
```

Тепер розглянемо основну структуру віртуальної машини, її можна побачити у лістингу 3.38. У свою чергу frames зберігає масив вкладених функцій, а також розмір цього масиву, змінна frame_count. Далі йде сам стек змінних віртуальної машини, це масив значень, а також його ємність, окрім того, щоб не відслідковувати індекс поточного елемента, то ми напряму підтримуємо вказівник на вершину стека, за допомогою змінної stack_top. Далі маємо масив для зберігання глобальних змін, та хеш-таблицю строк, друге має сенс підтримувати, щоб не створювати додаткові строки, для назв функцій, або змінних, а просто створювати їх один раз, та використовувати. init_string строка ініціалізації, а open_upvalues зберігає зв’язний список значень від замикання, тому якщо не буде знайдемо змінну у масиві globals, тоді буде проведено пошук по open_upvalues, як по зв’язному списку.

Далі знаходяться поля для збірника сміття, він відслідковує скільки байтів було виділено, коли наступного разу краще викликати його

перевірку, а також інші поля. Змінна objects також є зв’язним список, який зберігає абсолютно всі виділені коли-небудь об’єкти всередині програмного коду, це допомагає переглянути їх всі в одному циклі. Після цього використовується динамічний масив на стек сірих об’єктів, це дозволяє створити абстракцію трьох кольорів, де білі об’єкти виступають в якості необроблених елементів, сірі, які будуть оброблені наступним кроком, та чорні, вже оброблені. Даний підхід дозволяє уникнути ситуації, коли ми двічі проходимо на один і той самий елемент, якщо на нього є посилання. Принцип роботи полягає у тому, що під час будь-якого виділення пам’яті для об’єкту. він поміщається у початок списку, та може запуститися збирач сміття. Якщо після проходу по всьому стеку і обробки всіх елементів залишаться необроблені, значить їх немає в стеку, вже використані і більше не потрібні, тому їх можна вільно очистити і звільнити пам’ять.

Лістинг 3.38 – Структура VM

```
typedef struct {
  CallFrame frames[FRAMES_MAX];
  int frame_count;

  int capacity;
  Value *stack;

  Value *stack_top;
  GlobalVarArray globals;

  Table strings;

  ObjString *init_string;  // init keyword for actors
  ObjUpvalue *open_upvalues;

  size_t bytes_allocated;
  size_t next_gc; // some threshold
```

Продовження лістингу 3.38

```
  Obj *objects;  // head of the list
  int gray_count;
  int gray_capacity;
  Obj **gray_stack;
} VM;
```

Перед тим як приступити до функції run, розглянемо як організовані об’єкти у мові програмування. Розпочнемо з оголошень макросів та переліку, лістинг 3.39. Всього надано 7 різних об’єктів, OBJ_NATIVE у свою чергу це вбудована в мову функція, яка викликається так само як і функція користувача, всі інші були згадані раніше або розглянуті. Для кожного значення переліку існує макрос, опис було зменшено для наглядності. Функція is_obj_type перевіряє чи є value об’єктом, а потім порівнює чи є саме об’єктом наданого типу, AS_ACTOR у свою чергу проводить цю конвертацію.

Лістинг 3.39 – Оголошення макросів та перечислення для об’єктів

```
#define OBJ_TYPE(value)         (AS_OBJ(value)->type)
#define IS_ACTOR(value)                 is_obj_type(value,
OBJ_ACTOR)
#define AS_ACTOR(value)         ((ObjActor*)AS_OBJ(value))

typedef enum {
  OBJ_ACTOR,
  OBJ_CLOSURE,
  OBJ_FUNCTION,
  OBJ_INSTANCE,
  OBJ_NATIVE,
  OBJ_STRING,
  OBJ_UPVALUE,
} ObjType;
```

Перейдемо до розгляду конкретних структур деяких з цих об’єктів, почнемо з Obj та ObjFunction, лістинг 3.40. Як можна бачити структура Obj є базовою, а всі інші похідними, тому треба імітувати динамічний поліморфізм, якого немає в C. Це можна виконати за допомогою простого трюку, що перший елемент структури це базовий об’єкт, так як об’єкт структури це вказівник на його перший елемент. Тому неважливо який це об’єкт, завжди можна отримати доступ до Obj, після чого до type, де і буде визначено тип поточного об’єкта.

ObjFunction у свою чергу зберігає лічильник змінних верхнього рівня, Chunk це структура для збереження інструкцій коду поточної функції, а також її ім’я.

Лістинг 3.40 – Структури Obj та ObjFunction

```
struct Obj {
  ObjType type;
  bool is_marked;
  Obj *next;
};

typedef struct {
  Obj obj;
  int arity;
  int upvalue_count;
  Chunk chunk;
  ObjString *name;
} ObjFunction;
```

Перейдемо до ObjClosure та ObjUpvalue, лістинг 3.41. Перша структура визначає значення, яке зберігається у замиканні, основна суть у тому, що вони можуть бути видалені зі стеку, як тільки буде виконано опкод у віртуальній машині, тому варто його зберегти у відповідній структурі. Змінна location визначає де у стеку віртуальної машини знаходиться

значення, closed це копія цього значення, воно необхідно для того, щоб при виході з функції ми не втратили його, тому після виходу з функції обов’язково закриваємо всі значення, і location буде вказувати на closed. Для простішої вставки значень було реалізовано зв’язний список замість динамічного масиву. ObjClosure зберігає функцію виконання, динамічних масив структур ObjUpvalue та кількість цих змін, копія яких потрібна і тут, окрім ObjFunction, для зручного використання у збірнику сміття.

Лістинг 3.41 – Струкутри ObjClosure та ObjUpvalue

```
typedef struct ObjUpvalue {
  Obj obj;
  Value *location;
  Value closed;
  struct ObjUpvalue *next;
} ObjUpvalue;

typedef struct {
  Obj obj;
  ObjFunction *function;
  ObjUpvalue **upvalues;
  int upvalue_count;
} ObjClosure;
```

Далі розглянемо ObjString, лістинг 3.42. У ньому міститься довжина строки, її хеш, для зручного пошуку у хеш-таблиці, так як строки можуть бути різної довжини. Та масив символів, але не динамічний, так як тут використовується певна оптимізація гнучного елементу масиву, коли можливо динамічно виділити пам’ять під цю структуру, але не виділяти всередині неї ще додатково, щоб позбутися вкладеностей.

Лістинг 3.42 – Структура ObjString

```
struct ObjString {
```

Продовження лістингу 3.42

```
  Obj obj;
  int length;
  uint32_t hash;
  char chars[];
};
```

Останніми розглянутими структурами будуть ObjActor та ObjInstance, лістинг 3.43. Актор зберігає масив хеш-таблицю своїх повідомлень, а конкретний створений екземпляр містить цього актора, а також усі його поля, які також зберігаються у хеш-таблиці.

Лістинг 3.43 – Структури ObjActor та ObjInstance

```
typedef struct {
  Obj obj;
  ObjString *name;
  Table messages;
} ObjActor;

typedef struct {
  Obj obj;
  ObjActor *actor;
  Table fields;
} ObjInstance;
```

Тепер детально розглянемо функцію run, поступово розбираючи її частини, так як вона завелика для розгляду одним блоком. Почнемо з лістингу 3.44, де зображено макроси, які використовуються для зручної взаємодії з віртуальною машиною. Функція повертає значення OK, COMPILE_ERROR або COMPILE_RUN. На початку створюється функція для глобального коду. Володіючи знаннями про адресну арифметику у C, та тим що ip це вказівник на певну область у масиві, можемо виконати

READ_WORD розіменувавши вказівник, та просунувши його далі. READ_INT спочатку інкрементує вказівник на 4 байти вперед, після чого збирає значення int з двох слів використовуючи логічне АБО. READ_STRING та READ_CONSTANT будуть пропущені через їх декларативність. Останньою йде BINARY_OP, яка приймає тип конвертації та бінарну операцію. Макроси по своїй природі є підстановкою, тому у коді можливо таке зробити, також щоб обмежити код в область видимості, знаючи що в кінці макроса може бути виставлено крапку з комою, обгортаємо його у do while цикл, який компілятор C в подальшому оптимізує. Спочатку перевіряємо чи 2 значення є числами, якщо є витягаємо їх зі стеку та поміщаємо результат їх виконання назад у стек.

Лістинг 3.44 – Макроси функції run

```
static InterpretResult run() {
  CallFrame *frame = &vm.frames[vm.frame_count - 1];
#define READ_WORD() (*frame->ip++)
#define READ_INT() (frame->ip += 2, (uint32_t)((frame->ip[-
2] << 16) | frame->ip[-1]))
#define     READ_CONSTANT()     (frame->closure->function-
>chunk.constants.values[READ_WORD()])
#define READ_STRING() AS_STRING(READ_CONSTANT())
#define BINARY_OP(ValueType, op) \
  do { \
    if (!IS_NUMBER(peek(0)) || !IS_NUMBER(peek(1))) { \
      runtime_error("Operands must be numbers."); \
      return INTERPRET_RUNTIME_ERROR; \
    } \
    double b = AS_NUMBER(pop()); \
    double a = AS_NUMBER(pop()); \
    *vm.stack_top++ = ValueType(a op b); \
  } while (false)
```

Раніше вже було пояснено як працює віртуальна машина, тому надалі розглянемо використані опкоди. Почнемо з локальних змінних, лістинг 3.45. З локальними змінними простіше, бо спочатку зчитуємо місцезнаходження змінної у функції, після або зразу штовхаємо значення в стек, або перезаписуємо значенням із нього.

Лістинг 3.45 – Опкоди для локальних змінних

```
case OP_SET_LOCAL: {
  const uint16_t slot = READ_WORD();
  frame->slots[slot] = peek(0);
  break;
}
case OP_GET_LOCAL: {
  const uint16_t slot = READ_WORD();
  push(frame->slots[slot]);
  break;
}
```

Перейдемо до глобальних змінних, лістинг 3.46. Відмінність заключається в тому, що тут пошук відбувається за іменем, так як ця змінна не належить функції, на відміну від локальних, тому після його отримання проводиться пошук по масиву globals і виконуються операції global_find та global_set_at. Щодо OP_DEFINE_GLOBAL, то у стек додається зразу 3 значення, а саме Value, потім булеве значення, яке визначає константа це змінна чи ні, та її назва, після чого ми її зчитуємо як показано у лістингу та ініціалізуємо за допомогою функції global_set.

Лістинг 3.46 – Опкоди для глобальних змінних

```
case OP_GET_GLOBAL: {
  const ObjString *name = READ_STRING();
  uint16_t ind; // plug
```

Продовження лістингу 3.46

```
  const GlobalVar *var = global_find(&vm.globals, name,
&ind);
  if (var == NULL) {
    runtime_error("Undefined variable '%s'", name->chars);
    return INTERPRET_RUNTIME_ERROR;
  }
  push(var->value);
  break;
}
case OP_DEFINE_GLOBAL: {
  const bool constant = AS_BOOL(pop());
  const ObjString *name = READ_STRING();
  global_set(&vm.globals, name, peek(0), constant);
  pop();
  break;
}
case OP_SET_GLOBAL: {
  const ObjString *name = READ_STRING();
  uint16_t ind;
  const GlobalVar *var = global_find(&vm.globals, name,
&ind);

  if (var == NULL) {
    runtime_error("Undefined variable '%s'", name->chars);
    return INTERPRET_RUNTIME_ERROR;
  }
  if (var->constant) {
    runtime_error("Can't reassign a constant '%s'", name-
>chars);
    return INTERPRET_RUNTIME_ERROR;
  }
  global_set_at(&vm.globals, peek(0), ind);
  break;
}
```

Перейдемо до розгляду отримання та виставлення полів у повідомленнях показаних у лістингу 3.47. За допомогою додаткових макросів розглянутих раніше. Логіка досить проста, отримуємо з хеш-таблиці конкретне значення, видаляємо зі стеку отриманого актора та записуємо на його місце значення. Виставлення значення працює схожим чином, але в стеку зберігаються актор, а потім значення, тому дістаємо актора за допомогою функції peek із зсувом в одну інструкцію, а потім за допомогою pop дістаємо значення, актора, та штовхаємо отримане значення, для можливості використання ланцюгового присвоєння.

Лістинг 3.47 – Отримання та виставлення полів у повідомленнях

```
case OP_GET_PROPERTY: {
  if (!IS_INSTANCE(peek(0))) {
    runtime_error("Only instances have properties.");
    return INTERPRET_RUNTIME_ERROR;
  }

  ObjInstance *instance = AS_INSTANCE(peek(0));
  ObjString *name = READ_STRING();

  Value value;
  if (table_get(&instance->fields, name, &value)) {
    pop(); // Instance
    push(value);
  }
  break;
}
case OP_SET_PROPERTY: {
  if (!IS_INSTANCE(peek(1))) {
    runtime_error("Only instances have fields.");
    return INTERPRET_RUNTIME_ERROR;
  }

  ObjInstance *instance = AS_INSTANCE(peek(1));
```

Продовження лістингу 3.47

```
  table_set(&instance->fields, READ_STRING(), peek(0));
  Value value = pop();
  pop();
  push(value);
  break;
}
```

Надалі багато інструкцій є досить схожими, наприклад OP_GREATER та OP_LESS, які відрізняються лише тим, що додають різне значення у макрос BINARY_OP, тому деякі з них будуть пропущені. Розглянемо унікальні опкоди, показані на рисунку 3.48. Почнемо з OP_EQUAL, стек влаштовано таким чином, що спочатку на його вершину поміщаються значення, а потім вже операція, яку потрібно виконати, тому з нього витягаються останні два значення, після чого додається їх результат. OP_ADD працює схожим чином, за відмінністю того, що можна виконувати таку операцію із строками, де інструкція видозмінена, бо дістає зі стеку значення за допомогою AS_STRING та низькорівнево працює з хеш-таблицею strings. Інші операції є декларативними, тому їх пояснення буде опущено

Лістинг 3.48 – Прості, унікальні опкоди

```
case OP_EQUAL: {
  const Value b = pop();
  const Value a = pop();
  *vm.stack_top++ = BOOL_VAL(values_equal(a, b));
  break;
}
case OP_GREATER:       BINARY_OP(BOOL_VAL, >);  break;
case OP_ADD: {
  if (IS_STRING(peek(0)) && IS_STRING(peek(1))) {
    concatenate();
  } else if (IS_NUMBER(peek(0)) && IS_NUMBER(peek(1))) {
```

Продовження лістингу 3.48

```
    const double b = AS_NUMBER(pop());
    const double a = AS_NUMBER(pop());
    *vm.stack_top++ = NUMBER_VAL(a + b);
  } else {
    runtime_error(
    "Operands must be two numbers or two strings");
    return INTERPRET_RUNTIME_ERROR;
  }
  break;
}
case OP_NOT:
  push(BOOL_VAL(is_falsey(pop())));
  break;
case OP_NEGATE:
  if (!IS_NUMBER(peek(0))) {
    runtime_error("Operand must be a number.");
    return INTERPRET_RUNTIME_ERROR;
  }
  negate();
  break;
case OP_PRINT: {
  print_value(pop());
  printf("\n");
  break;
}
```

Наступними опкодами будуть OP_JUMP, OP_CALL, та OP_INVOKE, лістинг 3.49. Перша інструкція досить проста, так як вона просто виконує стрибок на певну кількість інструкцій, щодо схожого опкоду OP_LOOP, то в цьому випадку виконується повернення на початок циклу віднімаючи змінну offset. Далі виконується основний опкод для виклику функцій та замикань, у коді отримуємо кількість аргументів та функцію, яка викликала, так як перед аргументами у віртуальній машині знаходиться об’єкт самої

функції. Функція call_value зводиться до функції call, у лістингу 3.4, але перед цим вона визначає який об’єкт її викликав, після чого передає відповідні параметри. Останній опкод OP_INVOKE є оптимізацією і поєднанням двох раніше розглянутих опкодів OP_GET_PROPERTY та OP_CALL і виконується лише для акторів.

Лістинг 3.49 – Опкоди OP_JUMP, OP_CALL та OP_INVOKE

```
case OP_JUMP: {
  const uint32_t offset = READ_INT();
  frame->ip += offset;
  break;
}
case OP_CALL: {
  const int arg_count = READ_WORD();
  if (!call_value(peek(arg_count), arg_count)) {
    return INTERPRET_RUNTIME_ERROR;
  }
  frame = &vm.frames[vm.frame_count - 1];
  break;
}
case OP_INVOKE: {
  const ObjString *message = READ_STRING();
  const int arg_count = READ_WORD();
  if (!invoke(message, arg_count)) {
    return INTERPRET_RUNTIME_ERROR;
  }
  frame = &vm.frames[vm.frame_count - 1];
  break;
}
```

Далі, у лістингу 3.50 розглянемо опкод OP_CLOSURE. Так як у коді на стек поміщається лише, функція, то на основі неї створюємо замикання, після чого поміщаємо у стек, для подальшого використання опкодами

OP_CALL чи іншими. Після зчитування двох опкодів, визначаємо як отримувати конкретну змінну. Функція capture_upvalue знаходить це значення всередині відповідних open_upvalues, всередині структури віртуальної машини і повертає його, або якщо не існує, то вставляє у поточне місце і повертає. В іншому випадку, якщо значення не локальне, то так як OP_CLOSURE викликається в кінці функції, наразі у стеку знаходиться поточний CallFrame, значення якого і можна використати у коді.

Лістинг 3.50 – Опкод OP_CLOSURE

```
case OP_CLOSURE: {
  ObjFunction *function = AS_FUNCTION(READ_CONSTANT());
  ObjClosure *closure = new_closure(function);
  push(OBJ_VAL((Obj*)closure));

  for (int i = 0; i < closure->upvalue_count; ++i) {
    const uint16_t is_local = READ_WORD();
    const uint16_t index = READ_WORD();

    if (is_local) {
      closure->upvalues[i] = capture_upvalue(frame->slots
+ index);
    } else {
      closure->upvalues[i]        =        frame->closure-
>upvalues[index];
    }
  }
  break;
}
```

Перейдемо до останніх опкодів OP_ACTOR, OP_MESSAGE та OP_RETURN віртуальної машини, лістинг 3.51. OP_ACTOR просто зчитує відповідне значення із стеку та створює актора, OP_MESSAGE у свою чергу

спочатку зчитує повідомлення, потім актора, який викликає це повідомлення, після чого ініціалізує у своїй таблиці повідомлень його. При виході з функції у коді отримується результат, замикаються змінні, це те, що було пояснено у лістингу 3.41, у коді ітеруємося по всім значенням поточного фрейму і закриваємо їх, щоб вони не пропали зі стеку. Якщо ми дійшли до нульового фрейму, тоді програма завершилася, в іншому випадку маємо завершення поточної функції, тому потрібно повернутися до виконання функції на один рівень вище і помістити результат на вершину стеку.

Лістинг 3.51 – Опкоди OP_ACTOR, OP_MESSAGE та OP_RETURN

```
case OP_ACTOR:
  push(OBJ_VAL((Obj*)new_actor(READ_STRING())));
  break;
case OP_MESSAGE:
  define_message(READ_STRING());
  break;
case OP_RETURN: {
  const Value result = pop();
  close_upvalues(frame->slots);
  if (--vm.frame_count == 0) {
    pop();
    return INTERPRET_OK;
  }

  vm.stack_top = frame->slots;
  push(result);
  frame = &vm.frames[vm.frame_count - 1];
  break;
}
```

У цьому розділі було розглянуто основний код та ідеї, які передували його розробці.

4 РЕЗУЛЬТАТИ

У цьому розділі буде розглянуто деякі з програм, та функціонал, який можна написати за допомогою розробленої мови програмування. Розглянемо простий алгоритм пошуку у глибину для дерева, зображеного на рисунку 4.1.

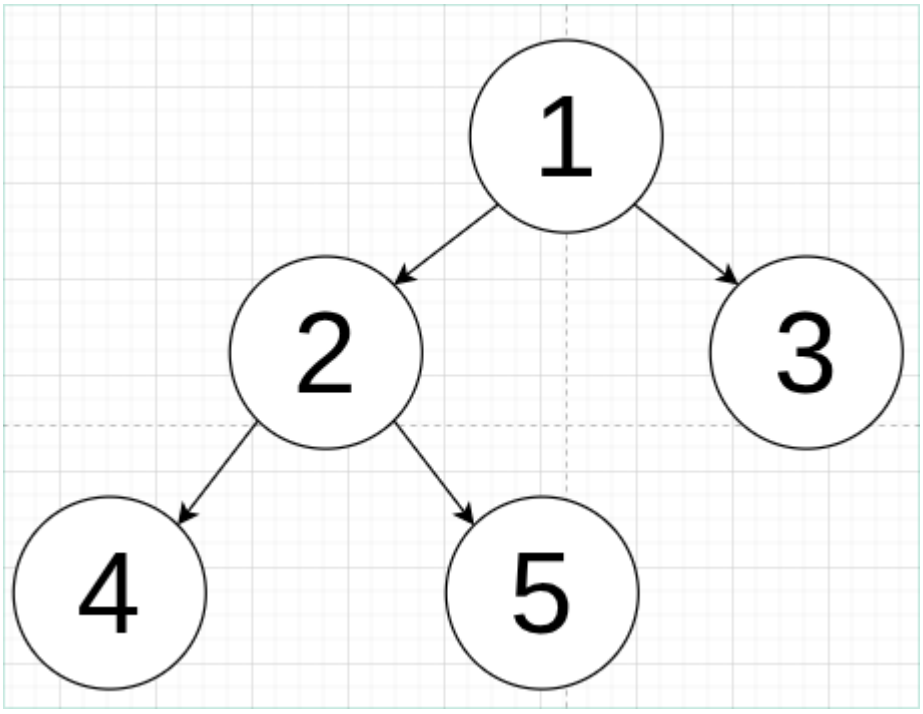

Рисунок 4.1 – Дерево для представлення

Тепер перейдемо до коду, лістинг 4.1. Кожна гілка містить поточне значення та ліве і праве піддерево. За допомогою механізму надсилання повідомлень, рекурсивно викликаємо dfs та виконуємо алгоритм і виводимо значення. В результаті отримуємо послідовність 1, 2, 4, 5, 3.

Лістинг 4.1 – Реалізація dfs для дерева, зображеного на рисунку 4.1

```
actor TreeNode {
  init(curr, left, right) {
```

Продовження лістингу 4.1

```
    this.curr = curr;
    this.left = left;
    this.right = right;
  }

  dfs() {
    print this.curr;
    if (this.left != nil) {
      this.left.send(dfs);
    }

    if (this.right != nil) {
      this.right.send(dfs);
    }
  }
}

val third = TreeNode(3, nil, nil);
val fourth = TreeNode(4, nil, nil);
val fifth = TreeNode(5, nil, nil);
val second = TreeNode(2, fourth, fifth);
val first = TreeNode(1, second, third);
first.send(dfs);
```

Наступною програмою буде створення функції Point та імітація поведінки екземплярів актора за допомогою замикань, лістинг 4.2. Маємо 3 рівня вкладеностей функцій, на першому рівні визначаємо отримані змінні, на другому функції get та додавання, а також повернення функції за отриманою назвою повідомлення і на третьому рівні виконується логіка програми. Тут також можна побачити використання циклу for з усіма його параметрами. У виводі буде 5 значень по 2, а саме 1, -1, потім 2, 0, далі 3, 1; і наостанок 4, 2.

Лістинг 4.2 – Функція Point та імітація поведінки екземплярів акторів

```
fun Point(_x, _y) {
  val x = _x;
  val y = _y;

  fun messages(message) {
    fun get_x() {
      return x;
    }

    fun get_y() {
      return y;
    }

    fun add_point(point) {
      return    Point(x    +    point("get_x")(),    y    +
point("get_y")());
    }

    if (message == "get_x") {
      return get_x;
    } else if (message == "get_y") {
      return get_y;
    } else if (message == "add_point") {
      return add_point;
    } else {
      return nil;
    }
  }
  return messages;
}

for (var i = 0; i < 4; i = i+1) {
  val point1 = Point(i, i+2);
```

Продовження лістингу 4.2

```
  val point2 = Point(1, -3);
  val result = point1("add_point")(point2);
  print result("get_x")();
  print result("get_y")();
  print "";
}
```

Як підсумок до розділу, можна сказати, що було показано та реалізовано більшість синтаксису мови та показано його правильне виконання.

ВИСНОВКИ

В результаті виконання цієї роботи було досліджено методи реалізації різних мов програмування загального призначення, а також мови програмування та фреймворки, що використовують акторну модель. В ході дослідження було виявлено, що більшість наявних рішень реалізують акторну модель як надбудову це Actix та Akka, а інші розглянуті рішення мають проблеми з оптимізацією та навантаженням на процесор, це Erlang та Elixir.

Тому, було сформульовано вимоги до нової мови програмування на базі віртуальної машини, яка містить повноцінну вбудовану підтримку акторної моделі. У процесі реалізації було створено власний синтаксис мови програмування, включаючи реалізацію ізольованої системи акторів. Сканер для перетворення вхідного коду у послідовність токенів. Зі сканером працює компілятор, який на основі токенів від сканеру компілює їх у байткод для віртуальної машини. Віртуальна машина у свою чергу підтримує виконання базових арифметичних операцій, функцій, акторів та інших необхідних опкодів для них. Окрім того, реалізовано ізольовану передачу повідомлень між акторами, без повернення значень чи будь-якою впливу зі сторони зовнішньої області видимості, окрім як через повідомлення.

В ході виконання роботи не було реалізовано асинхронну передачу повідомлень через брак часу на виконання, але це ніяк не вплинуло на загальний синтаксис мови. В кінці було показано результати та приклади використання мови програмування та її команд.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Hewitt, C. Actor Model of Computation: Scalable Robust Information Systems. 2010.

2. Actor | actix web. *Actix Web*. URL: https://actix.rs/docs/actix/actor/ (дата звернення: 05.06.2025).

3. A tour of go. *The Go Programming Language*. URL: https://go.dev/tour/concurrency/1 (дата звернення: 05.06.2025).

4. Dr. Alan Kay On the meaning of "object-oriented programming". *PURL Administration*. URL: https://www.purl.org/stefan_ram/pub/doc_kay_oop_en (дата звернення: 05.06.2025).

5. Erlang/OTP 28.0 – erlang/otp v28.0. *Index - Erlang/OTP*. URL: https://www.erlang.org/doc/readme.html (дата звернення: 05.06.2025).

6. Zhang, J. Characterizing the Scalability of Erlang VM on Many-core Processors (Trita-ICT-EX, Vol. 5). KTH, School of Information and Communication Technology (ICT). 2011.

7. Mix. *HexDocs*. URL: https://hexdocs.pm/mix/Mix.html (дата звернення: 14.05.2025).

8. Reselman B. How the Actor Model works by example. *TheServerSide | Your Java Community discussing server side development.* URL: https://www.theserverside.com/blog/Coffee-Talk-Java-News-Stories-and-Opinions/How-the-Actor-Model-works-by-example (дата звернення: 05.06.2025).

9. The lox language · crafting interpreters. *Crafting Interpreters*. URL: https://craftinginterpreters.com/the-lox-language.html (дата звернення: 05.06.2025).

10. Welcome to akka :: akka documentation. *Akka Documentation*. URL: https://doc.akka.io/ (дата звернення: 05.06.2025).

11. Contributors to Wikimedia projects. Flexible array member - Wikipedia. *Wikipedia, the free encyclopedia*. URL: https://en.wikipedia.org/wiki/Flexible_array_member (дата звернення: 01.06.2025).

12. Depth first search or DFS for a graph - geeksforgeeks. *GeeksforGeeks*. URL: https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/ (дата звернення: 01.06.2025).

13. GeeksforGeeks. Mark-and-Sweep: garbage collection algorithm - geeksforgeeks. *GeeksforGeeks*. URL: https://www.geeksforgeeks.org/mark-and-sweep-garbage-collection-algorithm/ (дата звернення: 01.06.2025).

14. Simple but powerful pratt parsing. *matklad*. URL: https://matklad.github.io/2020/04/13/simple-but-powerful-pratt-parsing.html (дата звернення: 01.06.2025).

15. Sysexits. *FreeBSD Manual Pages*. URL: https://man.freebsd.org/cgi/man.cgi?query=sysexits&apropos=0&section=0&manpath=FreeBSD+4.3-RELEASE&format=html (дата звернення: 01.06.2025).

16. Trie data structure - geeksforgeeks. *GeeksforGeeks*. URL: https://www.geeksforgeeks.org/trie-insert-and-search/ (дата звернення: 01.06.2025).