КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БУДІВНИЦТВА І АРХІТЕКТУРИ

Автоматизації і інформаційних технологій
(факультет)

Кібербезпеки та комп'ютерної інженерії
(назва випускної кафедри)

КВАЛІФІКАЦІЙНА РОБОТА
ЗДОБУВАЧА СТУПЕНЯ ВИЩОЇ ОСВІТИ БАКАЛАВР

на тему:

Розробка інтерпретованої мови програмування для опису даних та імперативної логіки у безпечних програмних середовищах.

Шпак Мирослав Русланович
(прізвище, ім'я та по батькові здобувача повністю)

Київ 2026 р.

КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БУДІВНИЦТВА І АРХІТЕКТУРИ

Автоматизації і інформаційних технологій
(факультет)

Кібербезпеки та комп'ютерної інженерії
(назва кафедри)

ЗАТВЕРДЖУЮ
Завідувач кафедри

к.т.н., доцент Максим ДЕЛЕМБОВСЬКИЙ
„ ____ ” _____ 20 ___ року

КВАЛІФІКАЦІЙНА РОБОТА
ЗДОБУВАЧА СТУПЕНЯ ВИЩОЇ ОСВІТИ БАКАЛАВР

Розробка інтерпретованої мови програмування для опису даних та імперативної логіки у безпечних програмних середовищах.
(назва)

Я як здобувач вищої освіти КНУБА розумію і підтримую політику закладу з академічної доброчесності. Я не надавав (-ла) і не одержував(-ла) недозволену допомогу під час підготовки цієї роботи. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело.

Здобувач Шпак Мирослав Русланович
(прізвище, ім'я та по батькові повністю)
125 «Кібербезпека»
(спеціальність)
Безпека інформаційних і комунікаційних систем
(освітня програма)
Група БІКС-22
Керівник Шабала Є. Є.
(прізвище та ініціали)
доцент кафедри кібербезпеки та комп'ютерної інженерії, кандидат технічних наук, доцент.
(вчене звання, науковий ступінь)
Рецензент Гончаренко Т. А.
(прізвище та ініціали)

Ідентичність підтверджую

Київ 2026 р.

КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БУДІВНИЦТВА І АРХІТЕКТУРИ

Факультет: Автоматизації і інформаційних технологій
Випускова кафедра: Кібербезпеки та комп'ютерної інженерії
Ступінь вищої освіти: Бакалавр
Спеціальність: 125 «Кібербезпека»
ОПП: Безпека інформаційних і комунікативних систем

ЗАТВЕРДЖУЮ
Завідувач кафедри
к.т.н., доцент Максим ДЕЛЕМБОВСЬКИЙ
„ ____ ” _____ 20 ____ року

З А В Д А Н Н Я

ДО ВИКОНАННЯ КВАЛІФІКАЦІЙНОЇ РОБОТИ ЗДОБУВАЧА

СТУПЕНЯ ВИЩОЇ ОСВІТИ БАКАЛАВР

Шпака Мирослава Руслановича

(прізвище, ім'я та по батькові здобувача)

1. Тема роботи «Розробка інтерпретованої мови програмування для опису даних та імперативної логіки у безпечних програмних середовищах.» затверджено наказом ректора КНУБА №576/52-15/13.2/26 від «14» травня 2026 року
2. Керівник роботи к.т.н. Шабала Євгенія Євгенівна
Доцент кафедри кібербезпеки та комп'ютерної інженерії, кандидат технічних наук
(прізвище, ім'я та по батькові, науковий ступінь, вчене звання)
3. Термін подання здобувачем роботи до захисту 12 червня 2026 року.
4. Зміст пояснювальної записки за розділами:

Р. 1. Аналіз предметної області та постановка задачі дослідження.
Р. 2. Проектування скриптової мови програмування на С.
Р. 3. Огляд прототипу, оцінка переваг та недоліків.

5. Графічний матеріал за розділами:

Р. 1. _____

Р. 2. 4 Рисунки

Р. 3. 12 Рисунків

6. Консультанти розділів атестаційної випускної роботи

Розділ	Прізвище, ініціали та посада консультанта	Перевірив	
		дата	підпис
Розділ 1.	Шабала Є.Є., к.т.н, доцент	04.05.2026	
Розділ 2.	Шабала Є.Є., к.т.н, доцент	18.05.2026	
Розділ 3.	Шабала Є.Є., к.т.н, доцент	27.05.2026	

7.Календарний план виконання роботи:

Види робіт та їх зміст	Дата виконання
Розділ 1. Аналіз предметної області та постановка задачі дослідження.	Травень 2026 р.
Розділ 2. Проектування скриптової мови програмування на С.	Травень 2026 р.
Розділ 3. Огляд прототипу та оцінка переваг та недоліків	Травень 2026 р.
Остаточне оформлення роботи	Травень 2026 р.
Направлення роботи на рецензування, перевірку на плагіат	Червень 2026 р.
Попередній захист роботи на кафедрі	Червень 2026 р.

8. Дата видачі завдання: 10 лютого 2026 року.

Керівник _____ _____
(підпис) (прізвище та ініціали)

Студент _____ _____
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Шпак М.Р. «Розробка інтерпретованої мови програмування для опису даних та імперативної логіки у безпечних програмних середовищах».

Тема дипломного проекту присвячена аналізу існуючих скриптових мов програмування для інтегрування в програмне забезпечення з метою виявити маловідомі чи непопулярні методи розрахунку у програмному забезпеченні для виявлення можливих методів покращення якості роботи таких мов усередині програмного забезпечення що потребує функціонал скриптової мови програмування. У процесі виявлення недоліків було створено стандарт, архітектуру і прототип імперативної мови програмування що пропонує методи вирішення існуючих проблем безпеки, простоти використання та передбачуваності присутні у популярних інтегруємих скриптових мовах. Описано практичну реалізацію однокрокового компілятора для мови та віртуальної машини створеної для виконання розрахункої роботи. Приведені приклади як запропоновані рішення покращують безпеку та надійнсіть скриптів написаних на стовреній мові.

Ключові слова: Стек, Масив, Хеш-мапа, Сміттєзбирач, рекурсія, рекурсивний-поглиблюючий парсер, лексичний аналіз, однокроковий компілятор, віртуальна машина

SUMMARY

Shpak. M. R. “Development of interpreted programming language for data description and imperative logic within safe software environments"

Topic of this diploma project dedicates analysis of present scripting languages designed to be embedded into software with the goal of finding better, less known or popularized methods for providing safety in the process of computing within software that requires a functional scripting programming language. Alongside the process of documenting potent problems related to the topic, a new standard was described with architecture of suggested solutions and a basic working prototype of imperative programming language focused purely on safety and suggested better ways of solving problems in a simpler, more predictible way. An implementation of a single-pass compiler for the language and high-level virtual machine are developed for demonstration of a suggested computational model. Examples are provided that demonstrate benefits in safety and reliability of scripts written using developed language.

Keywords: Stack, Array, Hash-Map, Garbage Collection, Recursion, Recursive Descent Parser, Lexical analysis, Single-Pass compiler, Virtual Machine.

РЕЗЮМЕ (SUMMARY) *до кваліфікаційної випускової роботи здобувача*	ПІБ Шпак Мирослав Русланович Shpak Myroslav Ruslanovich		
ЗВО	Київський національний університет будівництва і архітектури		
Тема *(українською та англійською)*	Розробка інтерпретованої мови програмування для опису даних та імперативної логіки у безпечних програмних середовищах.		
	Development of interpreted programming language for data description and imperative logic within safe software environments		
Освітній ступінь	Бакалавр		
Факультет	Автоматизації і інформаційних технологій		
Випускова кафедра	Кібербезпеки та комп'ютерної інженерії		
Спеціальність	125 «Кібербезпека»		
Освітня програма	Безпека інформаційних і комунікаційних систем		
Керівник	Шабала Євгенія Євгенівна		
Обсяг роботи:	*Пояснювальна записка, стор.*	*Розділів*	*Презентація, кількість слайдів*
	114 (137 з додатками)	3	22
Розділ 1	Аналіз предметної області та постановка задачі дослідження.		
Розділ 2	Проектування скриптової мови програмування на С.		
Розділ 3	Огляд прототипу, оцінка переваг та недоліків.		
Висновки по роботі	Сформульовані вимоги для створення та організації безпечного розрахункового середовища, створено прототип що проявив компетентну швидкість роботи разом з виконанням поставлених вимог питань безпеки у окреструючому ПЗ.		
Ключові слова: Keywords:	Стек, Масив, Хеш-мапа, Сміттєзбирач, рекурсія, рекурсивний-поглиблюючий парсер, лексичний аналіз, рахування референсів, однокроковий компілятор, віртуальна машина. Stack, Array, Hash-Map, Garbage Collection, Recursion, Recursive Descent Parser, Lexical analysis, Reference Counting, Single-Pass compiler, Virtual Machine.		

Здобувач _____/ _____

Керівник _____/ _____

ЗМІСТ

ВСТУП 10

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ ДОСЛІДЖЕННЯ. **1**3

1.1.Особливості, застосування та вимоги від скриптових мов програмування, їх відмінність від системних (компліюмих) мов програмування.

13

1.2. Аналіз популярних скриптових мов програмування у контексті інтегрування в програмне забезпечення. 17

1.3. Виявлення ключових проблем безпеки та надійності в існуючих рішеннях. 19

1.4. Ризики безпеки при розробці та експлуатації ПЗ розробленого з переліченими проблемами присутніх в мовах програмування. 27

1.5 Формулювання вимог до високорівневої мови програмування для забезпечення максимальної безпеки. Вирішення проблем притаманних існуючим рішенням. 30

РОЗДІЛ 2. ПРОЕКТУВАННЯ СКРИПТОВОЇ МОВИ ПРОГРАМУВАННЯ НА МОВІ С

37

2.1. Опис стандарту мови, функцій, синтаксису, обмежень, інтерфейсу API для взаємодії в орекструючому ПЗ. 37

2.2. Опис обраних технологій та структур даних для реалізації мови. 47

2.3. Розробка віртуального середовища (ВМ) для роботи мови програмування у середині користувацького ПЗ. 64

2.4. Розробка компілятора з метою збірки коду для віртуального середовища.

72

РОЗДІЛ 3.ДЕМОНСТРАЦІЯ ІНСТРУМЕНТАРІЮ МОВИ, ВИЯВЛЕННЯ ПЕРЕВАГ ТА НЕДОЛІКІВ У РОЗРОБЕРНОМ РІШЕННІ, ПЕРСПЕКТИВА РОЗВИТКУ. 84

3.1. Результат розробленої розрахункової моделі мови та приклад її роботи. 84

3.2. Детальний огляд роботи скриптів на віртуальній машині, заміри швидкості їх роботи. 92

3.3. Демонстрація використання API для використання мови при встроєні в ПЗ, приклади можливого застосування. 95

3.4. Недоліки виявлені при розробці, Проблеми у парадигми С-подібних мов програмування для використання у інтегрованих середовищах при описі даних. 102

ВИСНОВКИ 107

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 109

ДОДАТОК А 114

ДОДАТОК Б 125

ДОДАТОК В 128

Вступ

При розробці програмного забезпечення частою задачаю для вирішення стає надання можливості користувачу налаштовувати чи працювати з програмним застосунком на ближчому рівні ніж на тому який може надати графічний інтерфейс чи текстовий файл, у таких випадках доцільним рішенням є створення або використання скриптової мови програмування яка пропонує більший діапазон функцій ніж ті які можливо передбачити у меню чи реалізувати самостійно як функціонал програми.

У сучасному програмному забезпеченні присутня тенденція у використанні обмеженої у виборі кількості мов які зарекомендували себе як надійні та легкі у інтеграції у програмне забезпечення, але їх підходи до вирішення проблем які характерні мовам скриптового програмування та наданий набір необхідного інструментарію мови є застарілими так як ці мови були створені від двох до чотирьох десятиліть тому і можуть вирішувати проблеми які вже не є актуальними для сучасного програмного забезпечення чи виконують під лаштунки до комп’ютерів архітектура яких відійшла у минуле.

Для вирішення проблем притаманних конкретному програмному забезпеченні доцільним є розгляд створення власної скриптової мови що повинна вирішувати саме ті проблеми які необхідно зауважити саме у розробленому програмному забезпеченні. Рішення наданій в цій роботі звертає увагу в першу чергу на надання необхідної безпеки розрахункового середовища з спробую бути доцільним рішенням для більшості програмних застосунків для настільних ПК та інших користувацьких девайсів. Запропоноване рішення - бібліотека мінімалістичної мова програмування “Duct” (Duck Tape) з високорівневою віртуальною машиною та однокроковим компілятором що будує свій стандарт виключно на ідеї відсутності ручного керування пам'яттю та вказівниками з відстурністю функцій метапрограмування та надлишкових спрощень при роботі.

Метою бакалаврської роботи є розробка мови, обгрунтування обраної архітектури як віртуальної машини так і компілятора коду та аналіз результатів створеного прототипу обох систем.

Для досягнення мети необхідно пройти відповідні процеси:

1. Виділити які проблеми має вирішувати мова програмування та який функціонал вона має мати для ефективної роботи.
2. Проаналізувати ринок існуючих мов програмування та виділити їх недоліки чи артефакти минулого які більше не є актуальними.
3. Розробити архітектуру віртуальної машини мови що має забезпечувати надійну та безпечну роботу скриптів у середині хоствої програми та компілятор якій може за один прохід збирати текст на високорівневі інструкції для віртуальної машини.
4. Спроектувати функціональний прототип що демонструє простоту реалізації інструментарію мови та характерну йому безпеку.
5. Провести підсумковий огляд проекту та зробити висновки з реалізованого прототипу.

Об’єктом дослідження виступає процес створення та принципи роботи розрахункових систем та процеси необхідні для перекладу (“збірки”) текстових програм в лінійній послідовності інструкцій які можуть бути виконані на фізичній чи віртуалізованій апаратурі.

Методами дослідження що були використані у роботі є знання з Проектування Інформаційних Систем, знання з принципів роботи Операційних Систем та Алгоритмізації разом з урахування принципів Забезпечення Безпеки Програмних Середовищ.

Новизна результатів проекту полягає у процессі створенні мови програмування без використання інструктажу чи навчальних матеріалів як шаблону для реалізації мови програмування що дає змогу виконати процес розробки з унікальною перспективою яка дозволяє вигадати інноваційні рішення при проектуванні цієї чи майбутніх мов програмування.

Значення отриманих результатів демонструє простоту запропонованої архітектури адже робочий прототип було створено в рамках можливостей однієї особи з обмеженим часом на розробку. Запропоновані ідеї можуть бути використані для створення кращої і швидшої реалізації мови Duct або проектуванні нової мови з додатковими особливостями запропонованими моделлю високорівневої віртуальної машини та однокрокового компілятора. Вказані принципи та ідеї не є обмеженими до реалізації саме C-подібної мови програмування і тому можуть бути примінені до будь-якої за стилем написання та ідеології мови програмування.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ ДОСЛІДЖЕННЯ.

1.1. Особливості, застосування та вимоги від скриптових мов програмування, їх відмінність від системних (компліюмих) мов програмування.

Системні мови програмування, розрахункова модель стеку, регістровий файл. Класичний комп'ютер на якому працює програмне забезпечення фундаментальн не відрізняється від стрічкової розрахункової машини [1] кінця 19 - початку та середини 20 століття. Головна відмінність є використання електричних методів зберігання та обробки інформації за допомогою струму та матеріалів здатних до вступу реакції з ним замість рухомих механізмів та паперових стрічок. Головним методом розрахунку в комп'ютері є логічний модуль, він є набором логічних ціпок що виконують конкретну операцію, зокрема: арифметики, переміщення даних з місця на місце, використання значення як адресу для переміщення, тощо. Дії які необхідно виконати логічному модулю та модуль мапування адрес пам'яті внесені до "Стрічки Програми" - лінійної послідовності цифрових кодів інструкцій разом з аргументами необхідними для їх виконання. Комп'ютер виконує "Программу" послідовно считуючи інструкцію за номером "Вказівника інструкції" та виконуючи її з поданими разом з нею аргументами. Інструкції можуть мати однаковий розмір або відрізнятись за розміром з метою компактного пакування інформації на стрічці, для другого виду організації інструкції потрібен модуль "Декодера Інструкцій" для коректного їх зчитування та відслідковування. Операції виконуються з використанням спеціального блоку пам'яті на який під час виконання програми завантажуються значення та інформація, такий блок пам'яті називається Стеком (Stack) [2]. Головною характеристикою Стеку є принцип "Останній на вхід, перший на вихід" (Last In, First Out - LIFO) що означає що у такій структурі даних доступ здійснюється тільки до останнього(-іх) елементів і при потребі прибирати елемент, тільки останній розміщений елемент може бути прибрано. Ця структура даних часто зображується та може бути порівняна з стопкою фізичних предметів, зокрема тарілок, бетонних плит чи книжок - де при такій організації прибрати будь-який елемент крім самого

верхнього є складною задачею. Використовуючи таку структура даних можливо виконувати розрахунки та обробку інформації що дозволяє комп'ютеру виконувати всі його функції. Додатково комп'ютер має окремий класс операцій з використанням окремих комірців що знаходяться у реєстровому файлі [3] - блоку пам'яті що знаходиться прямо на ядрі Центрального процессора. Регістри є більш зручним та простішим у розумінні методом для проведення розрахунків ніж стек з додатковою перевагою паралелізації операцій, але він потребує організації зі сторони автора програми що ускладнює сторвення автоматичних систем для генерації таких програм. Через складність написання програм напряму з використанням машинних інструкцій з урахуванням всіх особливостей кожної окремої архітектури Центрального Процессора (ЦП) та методів організації інформації в кожній окремій Операційній системі (ОС) розробниками було вирішено створити окремі програми які автоматично генерують інструкції з урахуванням усіх особливостей середовища в якому запущено программу с простого тексту який зручний для читання та написання людиною. Такою програмою є компілятор, і завдяки йому розробнику Програмного забезпечення (ПЗ) потрібно взаємодіяти тільки з абстрактною мовою що зрозуміла компілятору і через певну послідовність кроків перетворена до робочої програми для ОС та ЦП при якому вона була зібрана. Такою мовою є низькорівнева мова програмування - мова програмування що є одним додатковим шаром абстракції над інструкціями які виконує безпосередньо комп'ютер. Класифікація мов на рівні від низького до високого є характеристикою того наскільки мова програмування близька до принципів роботи комп'ютера. Низькорівневі мови програмування орієнтовані на надання максимальної свободи дій розробнику і існують як інструмент для спрощення написання швидких та компактних програм. Високорівневі мови програмування намагаються спростити всі складні аспекти написання та роботи комп'ютерних програм та надати додатковий (другий) шар абстракції в межах якого розробнику зручно здійснювати створенням прототипів, експериментами чи написання простої логіки в рамках обмежень мови чи програми у середні якої працює високорівнева мова.

Відмінність системних (низькорівневих) мов програмування від (високорівневих) скриптових. Скриптові мови програмування подуовані з метою бути виконаними у віртуальному та захищеному середовищі або віртуальної машини, або стану програми, на відміну від компільованих мов програмування що мають виконуватись на фізичному розрахунковому середовщі (комп'ютері) та потребують складного багатокрокового аналізу, скриптові мови базуються на простому синтаксисі та високорівневих абстракціях що спрощуються процес написання програм для розробника. Для скриптових мов програмування характерними рисами є:

- Відсутність системи типізації даних, натомість типи асоціюються з даними на пряму під час виконання скриптів у такій мові
- Наявність вбудованих динамічних типів (Масивів, Списків), адже на відміну від програм написаних для роботи на фізичному комп'ютері у скриптової мови немає обмежень пов'язаних з операційною системою чи архітектурою комп'ютера бо вона працює у середині віртуалізованого середовища що вирішує ці питання за мову.
- Можливість інтерактивної роботи, бо процесс сборки програми чи інтерпретації відбувається в реальному часі під час роботи програми що надає змогу взаємодіяти з скриптом без довготривалого очікування що виправдано складним процесом оптимізації компільованих програм.
- Можливості метапрограмування, адже через інтерактивну природу скриптових мов можливо створювати у середині таких мов - програми які займаються написанням інших програм або модифікують свій власний код.
- Використання автоматичних систем чи алгоритмів для процесу контролю пам'яті, такі як Garbage Collection (Сміттєзбирачі), Reference Counting (Рахування референсів) та RAII (Resource Acquisition Is Initialization)

Для роботи скриптова мова програмування може використовувати дві моделі розрахунку: Систему стану програми чи Віртуальні машину.

Стан програми - це усі дані необхідні для інтерпретатора мови щоб мати змогу виконати програму та зберегти результат виконання, такий вид

розрахункової моделі не потребує складної реалізації і може базуватися на рекурсивній природі стеку програми у якій такий інтерпретатор працює. Перевагою такого методу розрахунку є його простота, адже достатньо реалізувати структуру даних інтерпретатора та функції для виконання текстових інструкцій з аналізом на їх коректність. У такій моделі можуть бути відсутні кроки компіляції та аналітичної оптимізації тому що процес виконання програми відбувається паралельно зчитуванню її текста. Недоліком виступає найповільніша швидкість роботи таких програм разом з проблеми безпеки. Через відсутність будь-якого аналізу програми перед її виконанням не можливо провести спрощення, перестановку та оптимізацію написаних інструкцій, до того ж, програма виконується з тексту що є найповільнішим методом зчитування та порівняння даних у комп'ютері бо такий формат даних потребує постійного доступу до випадкових місць у пам'яті що є повільною операцією у порівнянні з арифметикою та чисельними співставленнями значень які знаходяться близько до один одного та проходять процес копіювання до кешу центрального процессора. Також у такій мові страждає безпека через пряму залежність стану від хостової програми, при роботі з референсами та вказівниками користувач має потенційну змогу отримати доступ до даних які йому не мають бути доступні і єдиним чинником що перешкоджає такому виду атаки є система перевірки синтаксису мови.

Віртуальна машина - це структура даних та алгоритм який оперує над ними що спрямований на симуляцію роботи фізично-існуючого або гіпотетична архітектура комп'ютера всередині програмного забезпечення. Віртуальна машина існує апаратною чи процесною. В першому випадку віртуальна машина спрямована на точну симуляцію існуючої архітектури ЦП на якій може працювати будь-яке програмне забезпечення яке було побудовано для цієї архітектури, навіть операційні системи що були створені для реального фізичного комп'ютера. В другому випадку основною ціллю машини є симуляція окремих процесів - програм, зазвичай процесна віртуальна машина створюється як досить близький аналог розрахункової системи у середині якої без потреби у зміні програми, застосунок написаний для такої машини може працювати між декількома операційними

системами чи архітектурами ЦП без будь-яких змін, незважаючи на їх суттєву відмінність.

Скриптові мови програмування мають мати характеристику яка не притаманна компільованим мовам - програминий інтерфейс для взаємодії зі станом, будь-то віртуальна машина чи простий набір мовних символів. Завдяки такому інтерфейсу розробник ПЗ може вбудовувати інтерпретатор чи віртуальну машину мови паралельно до логіки програми - що дозволяє впливати на неї з можливостями обмеження чи надання свободи доступу до стану програми та взаємодії з ним.

Застосування скриптових мов є варіативним і може виражатись у можливостях:

- Створення плагінів, модифікацій (додатків) до програми що розширюють її функціонал чи можливості поза межами того що надав розробник.
- Опис даних для налаштування як самої програми так і об’єктів над якими вона оперує
- Автоматизація процесів виконуваних програмою, опис кроків виконання рутинних дій.
- Зручне інтерфейсування для керування діями програми без потреби використовувати обмежений графічний інтерфейс чи потребу використання миші.
- Надання інтерактивного середовища для роботи з програмою чи навчання під час її використання.

1.2. Аналіз популярних скриптових мов програмування у контексті інтегрування в програмне забезпечення.

Аналіз популярних скриптових мов. Найпопулярнішими скриптовами мовами програмування виступають (виписані по спаданню популярності):

- Javascript,
- Python,

- Lua,
- Lisp.

Розрахункова модель. Із цих мов 3 мови програмування (Javascript, Python та Lua) є імперативними мовами програмування що працюють на віртуальних машинах, часто з можливістю компіляції до рідних до ЦП інструкцій на льоту (JIT - Just In Time, Compilation), натомість мові програмування Lisp (і всім її діалектам) характерна інтерпретація без використання віртуальної машини.

Одиниця даних Об'єкт. Усім переліченим мовам програмування характерно використовувати одну одиницю зображення інформації - Об'єкт (Object), або Atom (у випадку з Lisp). Об'єкт є універсальною одиницею у межах мови що може відображає усі можливі значення які присутні у такій мові. Завдяки такій особливості легко взаємодіяти з таким форматом даних як усередині мови так і поза її межами. Це зумовлено тим що кількість необхідних дій які може виконувати класична мова програмування для розрахунку чи проведення порівнянь даних з різними типами, зменшується до однієї універсальної дії так як у випадку з "Об'єктами" виконуються над для любими можливими типами базуючись на їх нинішньому занченні. Приклади:

- При додаванні цілого числа та ірраціонального числа операція додавання може динамічно під час проведення визначити домінантний (більш вагомий) тип - ірраціональне число, і перевести ціле число до такої ж форми щоб виконати рівне додавання.
- При додаванні цілого числа до масиву із строк замість помилки виконувати дію перетворення числа до текстової строки а далі додати її до масиву як новий елемент.

Методи керування пам'яттю. В усіх зазначених скриптових мовах програмування використовується метод керування пам'яті під назвою Garbage Collection ("Сміттєзбирач"), його головний принцип відслідковування створених "об'єктів" за допомогою ланцюжків, і коли до елементу більше немає доступу (тобто усі вказівники на "Об'єкт" було втрачено), алгоритм відмічає ланцюг до об'єкта і усіх дочірніх йому об'єктів як "звільнені". В залежності від реалізації

сміттєзбирача можливо накопичувати ці ланцюги до певного моменту після якого сміттєзбирач виконує “прибирання”, або цей процес може відбуватись одразу індивідуально до кожного “Об’єкту”. Існує два основних види сміттєзбирача: відслідковуючий (Tracing garbage collector), тип збирача що зазвичай і підрозумовується коли мова йде про Garbage Collection, та рахування по референсам (Reference counting). В першому випадку алгоритм сміттєзбирача пробує усі елемнти на досяжність, і ті елементи які не були помічені досяжними звільняються при наступному циклі сміттєзбирача. В другому випадку замість проходу по усім доступним “об’єктам”, під час роботи з вказівниками до пам’яті сміттєзбирач відслідковує кількість вказівників до блоку пам’яті, і коли він досягає нуля, тоді відбувається звільнення пам’яті.

Мета-програмування. Усім переліченим мовам програмування характерна наявність повної чи часткової підтримки парадигми мета-програмування, в мовах Javascript та Python за це відповідає спеціальна функція `eval`. В мові програмування Lua присутня вбудована функція `loadstring` як працює схоже до `eval`. Мова програмування Lisp по своїй природі є динамічною мета-програмованою мовою, тому мета-програмування можна проводити як звичайну алгоритмізацію через оператор “цитування”, у випадках коли потрібна розгортка кода а не тільки його перенесення можливо використовувати оператор `defmacro` що дозволяє створювати макро-символ який схожий по вигляду на функцію але замість операцій над “об’єктами” мови проводить дії над “конструкторами” та “символами”.

1.3. Виявлення ключових проблем безпеки та надійності в існуючих рішеннях.

Види вразливостей у мовах програмування. Системні мови програмування є найближчими до систем розрахунку комп’ютера класом мов, через це їх безпека напряму залежить від компетентності та знань розробника, що є проблемою через людський фактор. Існує безліч видів вразливостей чи недоліків які виникають при

дії з пам'яттю, роботі у середовищі з кількома процессами (паралелізація), роботі з метапрограмуванням чи серіалізацією інформації. Ось перелік прикладів найчастіших проблем що виникають у системних мовах програмування:

- Use after free, Double free, Dangling pointer
- Null pointer dereference
- Memory corruption
- Buffer overflow
- Context escape, privilege escalation
- String injection

Аналіз проблем пам'яті через вказівники. Вказівник в термінології мови програмування є змінна, цифрове значення якої вважається адресою на інше місце у пам'яті. Вказівники зазвичай на рівні мови мають асоціацію з типом даних до якого адреса вказівника "вказує", це виконується для того щоб під час збірки програми компілятору було можливо виконати перевірку корректності операцій над вказівниками для уникненення проблем чи вразливостей яку можуть виникнути.

Вказівники є головним джерелом [4] проблем та вразливостей пов'язаних з пам'яттю та процесом її керування, через абстракту природу "вказівника" і його віртуальну класифікацію за типом, що є дійсним тільки до кінця компіляції програми, не існує ніякої гарантії що надавала-б змогу ним керувати безпечно. Через те що вказівник фундаментально є адресою у пам'яті, він не містить ніякої інформації про тип даних на який він вказує, до того ж при роботі з вказівниками не відомо чи вказано на один елемент чи на декілька, що додатковою може викликати критичні помилки під час спроб зчитування чи написання до пам'яті.

Так як вказівник ніяк не відрізняється від беззнакового натурального числа окрім того що розрозноки вважає його значення адресою у пам'яті, у розробника в більшості системних мов програмування надана можливість виконувати арифметичні операції над вказівником. Це означає що розробник може виконувати як прості кроки назад чи вперед відносно адреси вказівника, так і будувати не стандартні алгоритми які використовують цю властивість вказівників (прикладом

є xor list). У деяких мовах програмування компілятор автоматично виконує кроки при арифметиці вказівників розміром у стільки байт, скільки має розмір типу вказівника. Фактично використовування арифметики вказівників не є проблемною дією, так як компілятор перетворює всі дії доступу до елементів масивів та структура до саме арифметики над вказівникми, практично - це є джерелом більшості проблем при роботі з вказівниками пов’язаних з людським фактором.

В половині випадків при роботі з вказівниками у низькорівневих мовах програмування не існує обмеження на перетворення типу вказівника. В мовах програмування C чи C++ користувач може перетворити вказівник одного типу на інший через “кастинг” (casting) - спеціальний механізм системи типізації мови для зміни типів виразів. Також вказівники мають набір адрес які гарантовано не є коректними (наприклад нульова адреса), тому можуть бути використані як спеціальне значення для відображення помилки. Такий спосіб зображення помилок в більшості системних мовах програмування не перевіряється під час збірки тому є особливо небезпечним у використанні бо при спробі отримати значення за недійсною для процесса адресою, операційна система відразу припиняє його роботу з кодом SEGFAULT.

Перелік сценаріїв які є вразливостями чи проблемами, що пов’язані з особливостями вказівників:

- Кастинг з меншого за розміром типу структури до більшого при спробі отримати значення поля може викликати від помилки сегменту (segfault) чи зчитуванню недопустимих (“сміттєвих”) значень до зчитування чи перезапису даних наступного елементу в масиві чи вищий за ієрархією структурі що може призвести до вразливості несанкціонованого доступу до інформації.
- Арифметика над вказівником може бути виконана не вірно або відрізнятись через особливості розмірів типів даних на нестандартних платформах (таких як 16 чи 32 бітні мікроконтроллери), що може призвести до повної поломки алгоритму чи системи.

- При отриманні значення за адресою вказівника, якщо вказівник є нулем то операційна система вимушена припинити роботу програми. Для уникнення цього, розробнику потрібно перевіряти вказівник користуючись блоками бранчування чи інструментами відладки з подальшою повною перевіркою стану програми для визначення причин виникнення помилки.
- Buffer overflow (переповнення буфера) - це проблема що виникає з масивами розмір яких не перевірено під час отримання доступу, що призводить до отримання елементів за межами масиву, що можуть належати до зовсім іншого типу інформації. Цей вид проблем часто експлуатування для несанкціонованого доступу до програмного забезпечення під час його роботи, незважаючи на простоту цієї проблеми, Buffer overflow залишається актуальною проблемою у безпеці при використання масивів.

Аналіз проблем при недоречному керуванні пам'яттю. Керування Пам'яттю - це процесс відслідковування вказівників на займані блоки пам’яті для їх подального повернення до адресного простору Операційної Системи. Потреба у цьому процесі виникає при необхідності використання великого розміру пам’яті який не відомо заздалегідь, тому у таких випадках у любій операційній системи присутній системний виклик який може надати віртуальний блок пам’яті на користування процесом який виконав запит. Цей блок пам’яті доступний через вказівник що розробник має зберегти для подальшого повернення до ОС. Процесс повернення є потрібним у випадках коли програма має функціонувати значну кількість часу і адаптуватись до обмеженої кількості пам’яті на комп’ютер де вона працює. Для спрощення процесу керування всіма вказівниками і зменшення фрагментації пам’яті існує окрема класифікація структур даних - аллокатор (Allocator). Задача аллокатора є відслідковування блоків пам’яті та надання користувачу вказівника на блок пам’яті без залежності від середовища операційної системи у якій має працювати програма. Існує багато видів аллокаторів з різними перевагами та недоліками, але деякі з них є домінантними на ринку програмного

забезпечення та інформаційних технологій на зважаючи на їх посередність у швидкості роботи, складності реалізації чи безпеці що вони надають.

Керування пам'яттю є підкласом проблем що виникають при роботі з вказівниками наданими Операційною Системою. Більшість проблем пов’язаних з керуванням пам’ятю є дочірніми до проблем вказівників, але варто визначити додаткові проблеми які виникають саме з аллокаторами:

- *Use-After-Free (Dangling pointer)* є видом проблеми який виникає тоді коли після звільнення вказівника на блоку пам’яті назад до аллоактора, програма все ще продовжує використовувати вказівник на звільнений блок пам’яті. При зчитуванні вказівника після звільнення як може не статись нічого критичного, так і можна прочитати зовсім не очікуванні (“зіпсовані”) дані. Це пов’язано з тим що аллоактор звільнений блок пам’яті може перевикористати у іншому місці, де попередні значення що залишилось після звільнення будуть перезаписані. Такі “сміттєві” значення зазвичай є симптомом саме цієї проблеми і можуть призводити до подальшого припинення роботи програми через зіпсований стан програми чи спроби вийти за межі виділеного блоку пам’яті.
- *Memory corruption* - це підклас випадку Use-After-Free коли програма виконує несанкціонований запис до пам’яті що вводить програму у недійсний чи хибний стан.
- *Memory leak* - це сценарій при якому вказівник на блок пам’яті що був отриманий від операційної системи чи аллокатора був втрачений, що означає що програма більше не може звільнити блок пам’яті до кінця виконання програми. Через відсутність вказівника, що є ключем доступу до пам’яті у аллоакторі, цей блок пам’яті залишається загубленим та не використаним, що при наявності додаткових джерел витоку пам’яті може призвести до займу усієї наявної пам’яті у комп’ютері. Така проблема має серйозні наслідки під час довгої роботи програми, і в певний момент призведе до припинення роботи або

серйозних проблем зі швидкістю роботи програмного забезпечення, в залежності від того як Операційна Система вирішує проблему використання всієї наявної фізичної пам’яті.

Недоліки популярного методу контролю пам’яті - Garbage Collection. Система контролю пам’яті “Garbage Collection” немає критичних вразливостей чи помітних недоліків [5] у безпеці для програмного забезпечення в якому вона використовується, але ця система не є кращою ніж інші існуючі рішення і тому не має вважати універсальним вирішенням питань контроль пам’яті, особливо в середовищах де безпеки чи швидкість роботи програмного забезпечення є критичним аспектом. Головний фокус цієї роботи - Максимальна безпека середовища виконання користувацької логіки усередині програмного забезпечення, то і проведено аналіз GC (Garbage Collection) буде виключно з точки зору зацікавленості у високому рівні безпеки, незважаючи на значні проблеми цієї системи керування пам'яттю з швидкістю роботи та стабільність виконання циклів “Збирання” пам’яті.

Аргументи проти “Сміттєзбирача” (Garbage Collection) при ціліспрямованим опором на безпеку:

- *Швидкість роботи.* Навіть з урахуванням відсутності у цій роботі цілі створити швидку мову програмування, питання швидкості не може бути пропущено. Швидкість алгоритму оцінюють користуючись O(f) (O-велика) нотацією, ця нотація визначає функцію яка підходить до опису залежності часу чи дій потрібних для виконання алгоритму пропорційно до розміру вхідних даних. Оптимальним варіантом є лінійний час виконання - O(n), що означає що алгоритм буде виконуватись стільки часу - скільки було йому надано елементів, такий вид залежності є мінімальним можливим значенням для алгоритмів які мають виконати дії над усіма елементами. Функція швидкості роботи будь-якого алгоритму має практично-мінімальне значення яке є працездатним для великих вхідних даних. тому коли для програмного забезпечення пропонуються алгоритм з O(n^2), чи гірше, не є доцільними для практичного застосування, адже вони зростають так швидко що в певний

момент алгоритм виконуватиметься так повільно що це не є практичним до використання. Хоча це не є проблемою у самому Сміттєзбирачі, але при його активному використанні в віртуалізонваному середовищі, де кожна інстуркція віртуальної машини чи інтерпретатора означає мінімум 2 інструкції машинного коду, додавання автоматичної системи яка буде сповільнювати роботу і так відносно повільної системи є недоречним.

- *Складність.* “Сміттєзбирач” є автоматичною системою побудованою на принципі відслідковування, цей процес по своїй природі не є складним особливо коли об’єкти в пам’яті з якими вона використовується є зручними до об’єднувальної роботи, тобто мають фіксований, однаковий розмір. Коли дані з якими розробнику чи користувачу потрібно працювати не відповідають по структурі об’єктам це створює значне сповільнення у швидкості роботи скриптів, що в результат потребує реалізації чи використання додаткового, проміжного, методу займу пам’яті. Таким чином при додавані інших вимог до методу контролю пам’яті таких як: оптимальна швидкість роботи, стабільне розміщення елементів у пам’яті, відсутність “хвиль” чи “піків” в швидкості роботи під час “Маркування” елементів, можливість роботи з циклічними зв’язками то складність такого сміттєзбирача зростає до розмірів які притаманні промисловим кодовим базам [11] якими займаються значна кількість розробників.
- *Відсутність детального контролю для користувача чи розробника.* У деяких випадках коли розробнику потрібно виконати ручну оптимізацію чи компактно упакувати інформацію, механізм GC може бути обмеженням, адже при роботі з автоматичним контроллером пам’яті часто не надається можливості власноручного займу та управління пам'яттю. Такі дії могли-б вважатися небезпечними, але тільки у тому випадку коли дії відбуваються напряму з вказівниками, при використання рефернсів чи вискогорівневих об’єктів такої небезпеки не має бути, з урахуванням надійності системи що є фундаментом контролю пам’яті у віртуальної машини мови чи інтерпретатора усередині програмного забезпечення.

Проблеми метапрограмм та прямої інтерпретації коду. У більшості інтерпретованих мов програмування присутній функціонал чи механізм що надає змогу виконувати мета-програмування - вид програмування що вважає виконувану логіку програми видом даних який можливо модифікувати та змінювати під час безпосередньої роботи програми. Незважаючи на усі переваги мета програмування для розробки функціоналу програмного забезпечення та необмежену доповнюваність яка надається мовам які мають підтримку макро-виразів чи маніпуляцій над Абстрактним Синтаксичним Деревом (AST), така парадигма програмування не є допустимою якщо метою розроблюваної мови чи середовища є забезпечення максимальної передбачуваності та безпеки при інтеграції у програмне забезпечення.

Головним недоліком мета програмування у контексті безпеки мови є непередбачуваність та складність відслідковування, це пов'язано з динамічною природою метапрограмування як призводить до зміни структури програми під час виконання і змінною. Змінність коду підрозумовує те що програма може змінити проаналізований та надійний алгоритм на іншу форму яка може не відповідати очікуванням критеріям. Також системі мета-програмування характерна повна відсутність обмежень по доступу чи правам по даних, у мові програмування Lisp не існує дійсно постійних чи прихованих даних, так як вся мова є інтерфейсом для редагування даних, де інструкції для виконання дій належать до таких даних. Саме через таку особливість мова програмування Lisp при взаємодії через інтерпретатор, зазвичай, є інтерпретована без додаткових кроків збірки коду.

Вагомим недоліком систем метапрограмування є її складність, незважаючи на мінімалізм ідеї мета-програмування, програмного забезпечення яке створюються на мета-мові потребує знань та навиків у сфері мета-програмування що потребує спеціалізованої підготовки перед потребою початку роботи, тому аргументом проти мета-програмування є вимога до простоти середовища розробки, адже простіше рішення, незважаючи на його обмеження, зменшує ризики пов’язані з людським фактором.

Системи мета-програмування через свою багатовимірність часто складно або неможливо відслідковувати та виправляти у випадках коли вони дають збій. Черезмірне використання рекурсивного мета-програмування призводить до накопичення артефактів інформації які не були описані розробником і тому не можуть бути корректно проаналізовані та виправлені без перебудови усього рішення, алгоритму, модуля чи програми.

Вразливості серіалізації текстового вводу. Для багатьох скриптових мов програмування характерна функція простого об'єднання строк за допомогою операторів "з'єднання" (string concatenation), хоч такий функціонал ефективно використовувати при розробці, він є способом за яким зловмисник може виконувати ін'єкцію зловмисного коду. Зазвичай така проблема характера саме скриптовим мовам програмування через динамічність об'єктів що дозволяє під час роботи скрипту проводити з'єднання строк. У випадках з Perl [25] у зловмисника є прямий необмежений доступ до всього синтаксису мови так як обидві мови об'єднують зміст змінних з строками одразу там де і відбувається інтерпретація коду, в таких випадках відсутня необхідність пошуку місця де відбувається динамічний чи мета-програмований запуск вводу від користувача, адже мова це робить автоматично. У мовах програмування Lua, Python чи Javascript така строка не інтерпретується, але в багатьох випадках може передаватись до аналогу eval, чи системи пошуку у базах даних, елементів XML документу, тощо - тоді, навіть у таких випадках у несанкціонованої особи є повний об'єм доступу для отримання любої інформації з ІС яку він вразив.

1.4 Ризики безпеки при розробці та експлуатації ПЗ розробленого з переліченими проблемами присутніх в мовах програмування.

Buffer overflow. Цей вид проблеми є небезпечним у контексті отримання вхідних значень від користувача та є можливою загрозою для безпеки у інших контекстах. Ця вразливість, так само як і всі інші проблеми загадні у цьому списку, є небезпечною так як у найгіршому сценарії надає змогу зловмиснику виконувати запуск власного коду віддалено якщо програма не була зібрана з GOT (Global offset table) тільки з правом читання, без перезбірки чи втручання у структуру програми.

Також ця вразливість може дозволити прочитати як певну кількість так і можливо всю інформацію до якої має доступ програма, ця вразливість є особливо небезпечною у випадку коли уся інформація програми знаходиться близько у пам’яті і зловмиснику відома структура цієї інформації.

Use-After-Free. [9] Ця вразливість схожа на наслідками до Buffer Overflow. але є більш складною до здійснення. Принцип вразливості заключається в тому що у контексті де “Об’єкти” створені і знищені при неправильній роботі системи управління пам'яттю - зловмисник може через певний час і зі значною кількістю спроб задіти цінну інформацію яка до того ж може бути не захищеною від написання. Прикладом є вразливість системі кешування запитів до бази даних Redis - у власній підверсії Lua інтерпретатор, в межах проекту, була допущена помилка у логіці Сміттєзбирача, що створювала саме такий сценарій. Потенційно це є вразливість особливого рівня так як зловмисник може отримати доступ до систем середовища Lua що може мати привілеї які дозволяють програмі зчитувати чи записувати дані на сервері де працює зламана система Redis. Важливо зазначити що на практиці ймовірність задіти цінну інформацію і її виявити як таку є низькою, але при достатній кількості спроб, може бути експлуатована, тому ця вразливість (CVE-2025-49844) отримала оцінку 10.0 - критичний рівень.

Експлуатація Стеку. Стек на якому знаходяться дані виконуваної програми та інформація про шари викликаних функцій не відрізняється від звичайної пам’яті, що означає повний контроль над інформацією потрібною до виконання програми. Якщо зловмиснику вдалося отримати доступ до вказівника не стеку, йому відсутні механізми, що перешкоджають прочитати будь-яку інформацію програми у будь-який момент. Це є особливо небезпечною вразливістю коли зловмиснику відрома структура стеку викликів, і ще гірше якщо йому відомо що одна з змінних на стеці є вказівником на функцію, у такому випадку у нього є можливість зробити заміну вказівника на свій власний який матиме змогу виконувати будь-який алгоритм зловмисника. Хоча такий сценарій має мізерний шанс відбутися, у контексті інтерпритованої мови при наявності вказівників чи схожих за суттю механізмів

мови, небезпека та тяжкість наслідків переважає будь-який вид зручності який вказівники можуть надати у мові.

GOT vulnerability. [6] Бібліотеки що використовуються розробниками усередині програм двох видів при побудові компілятором - статичні та динамічні. Статичні бібліотеки є зібраним кодом з набором мінімальних символів для ефективності з'єднання файлів при підключенні (linking) до коду програми, зазвичай компілятор мови збирає програму окремо від усього коду від якого вона залежить, а далі рекурсивно будує бібліотеки від яких залежить програма чи інші бібліотеки, під кінець з'єднуючи увесь побудований код у один файл. Динамічні бібліотеки містять більшу кількість інформації про включені у файл символи, що дозволяє програмі, користуючись механізмами знаходження і підключення файлу, прочитати та завантажити інформацію з динамічної бібліотеки під час роботи програми без потреби перебудови за допомогою компілятора. Перевагами такого методу під'єднання є:

- Відсутність потребу у перебудові усієї програми при зміні лише однієї її частини;
- Можливість в реальному часі оновлювати логіку програми без потреби її перезапускати, що є зручним для розробки складних систем які потребують час на запуск, чи відеоігор;
- Можливість додавати функціонал у головну програми методом написання "модулів" які програма може завантажити і запустити у любій кількості;
- Можливість мати "універсальний" інтерфейс для дій які по різному реалізовані на різних платформах, архітектурах ЦП чи Операційних системах, без потреби перебудовувати програму для кожного випадку. (Такий вид программ називають "Поліморфами" (Polymorphic));

Динамічність бібліотек у програмах при з'єднані динамічного коду реалізована через GOT - Global Offset Table. GOT таблиця [7] містить у собі адреси та назви функцій (процедур) які використовуються програмою під час роботи. Коли програмі потрібно запустити функцію що була приєднана динамісна, вона автоматично виконує пошук відповідного символу в Динамічній бібліотеці і далі

вносить результат у GOT таблицю для повторного перевикористання без потреби виконувати пошук символу знову.

Суть вразливості Global Offset Table заключається у тому що у випадках коли програма зібрана з динамічно під’єднаними бібліотеками (за замовчуванням LIBC під'єднана динамічно) та відсутній захист RELRO (Relocation Read-Only) [8], виду захисту що робить пам'ять цієї таблиці “тільки для читання”, зловмисник може змінити динамічну бібліотеку на яку спирається програма. Далі можливості для атаки є дві:

- Зловмисник через будь-яку з вищезгаданих вразливостей внести зміни до GOT таблиці і замінити функцію наприклад `print`, `read`, `write`, тощо на свою власну, яка може викликати механізми створення інших процесів - що є найнебезпечнішим видом атаки - віддаленого запуску коду (Remote Code Execution).
- Зловмисник може внести модифікаці до динамічною бібліотеки яку завантажує програма, що призводить до, так само як і в першому випадку, найгіршого сценарію де у зловмисника є повний доступ над програмою та середовищем поза її межами. Цей вид вразливості є важчим до виконання так як потребує доступу до комп’ютера на якому працює така програма разом з можливість внести зміни до динамічної бібліотеки, не будучи поміченим системами аудиту які швидше всього працюють на такому комп’ютерів.

1.5. Формулювання вимог до високорівневої мови програмування для забезпечення максимальної безпеки. Вирішення проблем притаманних існуючим рішенням.

Потреба у простоті дизайну системи, простота як найважливіший фактор. При розгляді наведених аргументів проти деякого функціоналу притаманним іншому програмному забезпеченні, помітна тенденція надання переваги до більш простих рішення понад будь-які інші переваги та особливості розглядаємих систем. Простота проектованих систем часто є найменш згаданим чи

взагалі проігнорованим критерієм оцінки системи, коли в дійсності цей критерій має найбільший вплив на інші критерії оцінки через метод створення і підтримки систем - через інтелектуальну та фізичну працю людської особи. Будь-яка інформаційна система або програмне забезпечення написано людиною, і підтримується після свого початкового створення теж людиною, фактор простоти системи грає ключову роль у знаходженні недоліків чи надання підтримки застосунку бо за визначенням, простіше програмне забезпечення - простіше аналізувати, супроводжувати та модифікувати, ніж складне. Складність є суб'єктивною метрикою адже вона складається з багатьох факторів, як дизайну системи так і особливостей людей які працюють з ними, але складові таких факторів є конкретними і можуть надати цільові підказки для розробки оптимального рішення. Варто розуміти і розрізняти фактор "Складності системи" та фактор "Комплексності системи" адже вони є різними метриками які по різному характеризують програмне забезпечення. Складність чи простота описують набір факторів що сприяють розумінню усіх особливостей системи яка розглядається, до таких факторів належать:

- Інформативність системи - кількість інформації що індивід може отримати з опису системи чи програми без потреби в отриманні додаткової, сторонньої інформації, конкретизація назв чи кількість використаної термінології притаманній саме цьому проекту чи взагалі сфері предметної області в якій створено систему, кількість наданої інформації разом з проектом яка намагається пояснити чи навчити нового розробника системи базовим термінами, ідеям чи принципам застосованих у середині проекту;
- Структурованість - кількість тексту, коду, графіки які описують виконавчу логіку системи разом з організацією цієї логіку, розмір структури логічних дій що впливає на легкість сприйняття людиною без потреби поглибленого аналізу виконуваних дій;
- Модульність - логічне та практичне розміщення елементів логіки системи на "Модулі" з метою інкапсуляції кожної окремої унікальної системи у проекті до окремого блоку який допомагає зрозуміти і використовувати без знань

принципів його роботи але в той самий час ефективно його використовувати при розробці.

Комплексність системи є характеристикою кількості логічних елементів та дій потрібних для роботи системи, “Складність” на відміну від “Комплексності” може бути притаманна відносно простим діям з невеликою кількістю кроків, але через відсутність необхідної інформації чи мінімальної структури зрозумілій людині концепція залишатиметься складною, коли “Комплексність” характеризує об’єм системи з великою кількістю рухомих частин які можуть залишатись як простими так і складними.

Швидкість, надійність, безпека, масштабованість, доповнюваність напряму залежать як від навиків розробника так і простоти створеної системи, тому фактор простоти є головним елементом між усіма ними. З точки погляду безпеки розрахункового середовища складність системи означає більшу кількість людського чи розрахункового ресурсу (у випадку з Штучним Інтелектом) потрібного для виявлення недоліків чи проблем. Додавання складності до системи через розширення її функціоналу експоненційно збільшує кількість варіацій стану у якому може знаходитись система, що тільки збільшує шанси існування такого стану який би привів до вразливості, дірки в безпеці, тощо. Також збільшення складності системи погіршує її доповнюваність, адже вірогідність безпечної роботи внесених дій зменшується при кожному наступному додатку.

Аргументування потреби у відсутності вказівників. Незважаючи на свою зручність у використанні - вказівники є однією з основних причин виникнення вразливостей у програмному забезпеченні і також є джерелом цілої класифікації проблем з пам'яттю, тому безпечна мова програмування для скриптового використання має уникати надання користувачу прав та можливостей їх прямого використання, і абстрагувати їх в окремі універсальні структури даних наприклад “Об’єктів”. Заміною вказівникам можуть виступати:

- Атом, константа - “Об’єкт” який не надає можливості виконання змін свого значення і тому може бути тільки прочитаним.

- Об’єкт - універсальний елемент що може відображати любі дані у мові, він є абстракцією навколо як числа так і вказівників. Також такий елементи можна сформувати у граф який зображатиме більш складну структуру даних.
- Масив - “Об’єкт” що містить вказівник на блок пам’яті та тип даних які зображено у масиві.
- Список - “Об’єкт” що має тип даних до якого належить нинішній елемент та вказівник на наступний елемент.
- Строка - “Об’єкт” що може бути зображений за допомогою масиву чи списку, потрібен для відображення саме текстової інформації.

Відсутність констант, енумерацій. У мові програмування Lua, Python та Javascript не існує констант - значень які можливо тільки прочитати та без можливості змінити. У мовах програмування Lisp такий вид значень у більшості випадків наданий, але він є не очевидним у своєму використанні через особливості функціональної парадигми програмування. Це може виступати як джерелом потенційних проблем з передбачуваністю роботи так і вразливістю до системи у випадках коли атака відбувається напряму через REPL (Read Evaluate Print Loop) інтерфейс мови. Енумерації це прості блоки (зазвичай) цифрових константових значень які асоціюють до певного типу даних який описує “Вид” даних у певному контексті. Зручність енумерацій заключається в тому що вони є зручним методом опису великої кількості цифрових констант разом з структурованим об’єднанням їх у спільні блоки що дозволяє додатково надавати контекст проект через його код без використання сторонніх методів документування проекту.

Надання прав переназначення глобальних символів (функцій, доповнення класів). Ця проблема відноситься до попередньої проблеми відсутності підтримки констант, але в той же час створює окремий підвид небезпек які можуть бути експлуатовані як випадково так і навмисно. В усіх загаданих мовах програмування комплексні значення: “Столів”, “Об’єктів” чи “Списків” мають функціонал внесення нових полів чи елементів навіть після того як об’єкт було створено. Також в усіх скриптових мовах у користувача є повна свобода перевизначати функції чи процедури без будь-яких механізмів мови щоб це обмежити. Таким чином,

наприклад, користувач може перевизначити значення функції `read` або `write` що як мінімум може змусити усю скриптову частину системи дати збій, а як максимум - бути джерелом витоку інформації яка могла-б зберігатись у інтерпритованому середовищі і не бути доступною користувачау. Відсутність механізмів що зупиняють користувача перед зміною значень існуючих функцій чи об'єктів є джерелом багатьох проблем у скриптових мовах програмування, головні проблеми які виникають з цієї особливості скриптових мов - це порушення надійності та передбачуваності системи, що в свою чергу може негативно вплинути на безпеці написаних скриптів чи логіки у програмному забезпеченні.

НІЛ значення та авто-ініціалізація змінних. В усіх скриптових мовах програмування існує значення `nil` (або `undefined` у випадку з JavaScript) що існує для зображення відсутності значення у змінної, воно існує для зображення "Пустоти", об'єкту без типу, об'єкту без значення, помилкове значення, тощо. `Nil` є аналогом вказівника з нульовою адресою (називаємий `null`), що зазвичай означає помилковий чи не знайдений результат роботи алгоритму чи логіки програми, але у випадку з скриптовими мовами ця концепція була застосована до типу даних динамічної змінної де нульових тип є недійсним, неіснуючим. Цей тип даних має широкий спектр застосувань але також є джерелом дискусій через свої взаємодії з іншими видами даних у скрипотвих мовах програмування. Реалізації цього значення притаманні Python та деяким Lisp-подібним мовам програмування адже використання `nil` у любому контексті окрім перевірочного блоку - призводить до помилки, що і є очікуваною поведінкою для "помилкового" або "неіснуючого" значення. У таких мовах до `nil` не можливо примінити арифметичні операції, перетворити його на інший тип даних (типу строки чи самиву), отримати доступ до поля, тощо. Менш вдалою є реалізація цього типу даних у Lua, де мова програмування дозволяє отримувати доступ до змінних які не існують, надаючи значення `nil` як результат, без можливості налаштування цієї поведінки що є характерною особливістю в Javascript через "суворий режим" (strict mode).

Автоматична серіалізація та перетворення даних, використання арифметичних дій над складними структурами даних. У мовах програмування

Python, Lua та Javascript присутня функція автоматичного перетворення даних з одного типу на інший без потреби розробником ручного прописування алгоритмів їх перетворення. Хоча ця функція є зручною для пришвидшення роботи, вона може призводити до неочікуваних помилок та артефактів у перетворених даних через непередбачені для системи вхідні дані чи неповні знання розробника у принципах роботи таких перетворень. Основні джерела проблем виникають коли так операції виконують:

- Додавання строк та чисел. Процес перетворення значень різних типів даних починаючи з чисел, закінчуючи константами має відбуватися вручну адже напряму залежить від намірів розробника.
- Додавання масивів чи списків. Масиви та списки є кардинально відрізняються по своїй структурі від простих атомічних значень (натуральних чи іррацональних чисел, булевів, тощо) і тому процес доповнення до масиву має бути виключно ручною дією, особливо у динамічній мові де не відбувається процесу аналізу коду програми. Додавання двох масивів чи списків може бути видом операція яка має свій власний оператор, але використання простих арифметичних операторів для опису такої дії є джерелом непередбачуваної поведінки та важки для відслідковування проблем.

Мета-програмування в середині скрипту. Як було зауважено в Розділ 1.3., мета-програмування є джерелом непередбачуваної поведінки а реалізація системи відслідковування наслідків чи передбачення проблем які можуть виникнути під час мета-програмування є складною і комплексною у реалізації так як потребує аналізу в кільковимірному середовищі (аналіз типів даних у потоці виконання програми протягом часу розгортки та виконання мета-програм), тому доцільним є виключення зазначеного функціоналу мета-програмування у скриптовій мові в будь-якій формі.

Розділ 2. ПРОЕКТУВАННЯ СКРИПТОВОЇ МОВИ ПРОГРАМУВАННЯ НА МОВІ С

2.1. Опис стандарту мови, функцій, синтаксису, обмежень, інтерфейсу API для взаємодії в орекструючому ПЗ.

Термінологічний Опис мови та її кодова назва. Duct (частковий акроним Duck Tape - англ. Клейка стрічка, скотч) - мінімалістична імперативна високорівнева процедурна скриптова мова програмування з автоматичною і обмеженою в застосуванні системою контролю пам’яті та чистими функціями без відсутності побічних ефектів, створена з метою бути придатною для інтеграції у низькорівневе програмне забезпечення і надання необхідного функціоналу скриптової мови програмування з орієнтацією на забезпечення безпеки та передбачуваності виконуваної логіки.

Специфікація. Перед описом створеного рішення важливо зауважити що мова створена мова програмування є не є єдиним можливим рішенням що може бути створено з урахуванням вимог з Розділ 1.5. тому доцільно описати специфікацію яка-б нада рекомендації щодо створення безпечної інтерпритованої мови в незалежності від парадигми та синтаксису обраного для її реалізації.

Функції та можливості мови:

- *“Натуральні функції та дані” (Pure functions and data).* У мові повинні бути відсутні вказівники та референси і не надаватися ніякий функціонал що дозволяє змінювати дані без унікального (ексклюзивного) доступу до них. Це означає що кожна проста змінна є копією, а кожен складний тип даних який потребує додаткової пам’яті існує лише до моменту виконання функції добігає кінця, за виключенням коли в кінці функції ці дані не повертаються з неї. Такі обмеження створюються “Чисте та без ефектне” середовище де дані існують допоки вони використовуються. Це створює суттєве обмеження у можливостях створення системи, однак такий підхід дозволяє уникнути всіх проблем які пов’язані з вказівниками.

- *Фіксація "Об'єктів" і протидія доповненню полів до них після їх створення.* Після створення комплексних "Об'єктів", що є заміною структурам чи інших подібним сутностям у програмуванні (Tuples, Records, Structs, Classes, Multiplex), у користувача забирається можливість додавати полі до цього "Об'єкту" після того як він покидає межі функції. Таким чином складний тип даних стає фіксованим і не може навмисно чи випадково бути доповненим додатковими полями після факту його створення. Розробник має передбачати функціонал об'єкт завчасно і додавати тільки такі поля, що завжди необхідні алгоритму що приймає такі об'єкти. Перевагою цієї зміні є потенційна можливість у майбутньому збирати скриптові програми до комп'ютерних чи більш оптимізованих "bytecode" інструкцій для пришвидшення швидкості роботи програм, що через передбачувану структуру інформації стає можливою перспективою.
- *Монолітна стратегія керування пам'яттю.* Замість використання складних автоматичних систем керування пам'яттю чи простого але небезпечного ручного методу встановленим рішенням є проста але обмежена система керування пам'ять якій в рамках проекту дано назву "Моноліт". "Моноліт" - це стратегій побудована на аллокаторі Pool та Arena. Кожен складний тип даних які потрібно зберігати у пам'яті і йому не достатньо бути місця на стеці програми, крім масиву що має виключно фіксований розмір, розміщується в маленьких блоках пам'яті які можуть вмістити найбільший елемент у мові, і все що йде перед ним. Ці елементи розміщені в "Басейні" (Pool) де вони можуть займати любе місце і створення чи видалення елементів з цього "Басейну" займає O(1) часу - постійний час. Усі змінні що мають дані у "Моноліті" та не були повернені з функції - автоматично звільняються, через повну відсутність вказівників можливість втратити блок пам'яті чи випадково запистати в звільнений блок пам'яті повністю зникає. Для типів даних які можуть мати самі себе як під-

елементи наприклад списків та структур, такі об’єкти звільняються рекурсивно якщо вони були не використані.

- Фіксований масив як інструмент оптимізацій. Замість виконання принципу “Все є списком” варто зауважити потребу розробника до використання гарантованого у свої лінійності блоку пам’яті адже це є ефективним методом забезпечення швидкості роботи деяких алгоритмів. Массив не матиме права зростати через складність до організації пам’яті у таких випадках, тому масив завжди буде обмеженим за розміром і включений у мову виключно для вирішення потреб у його існування. Усі масиви є або тимчасовими і розміщенні на динамічному стеці, або глобальними і тоді існують протягом усього життя скрипту чи програми.
- Наявність констант. Мова має мати константи для опису незмінних, постійних даних, і при спробі їх змінити має виникати помилка як б інформувала користувача.
- Наявність енумерацій. Для уникнення створення багатьох констант має існувати константний “Об’єкт” для зображення асоціативних незмінних даних.
- Фіксація змінних. Змінні не можуть змінювати свій тип даних або згадуватись коли вони не були створені (не існують). Також змінні не можуть бути “пересторвені” в залежності від блоку.

Синтаксис мови. Для забезпечення імперативної логіки синтаксис мови має мати покрокову структуру з можливістю створення блоків, для цього було обрану С-подібний синтаксис який складається з “Кроків” (Statement) всередині який може знаходитись обмежена кількість операцій які виконують дії над “символами” - функціями, змінних, модулів; та “Виразами” (Expressions) що побудовані з операторів арифметики чи доступу до під елементів комплексних типів. Скриптова мова програмування, на відміну від С, не потребує системи аналізу типів даних через свою динамічність і тому може уникнути необхідного у противному випадку

синтаксису для їх уточнення чи перенесення та спрямувати зусилля на простоту синтаксису.

Вирази, Кроки та Блоки. У С-подібних мова програмування программа описана з фіксованої кількості елементів у мові, зазвичай визначень типів, функцій, констант, тощо, у випадку з розробленою мовою “Кроки” є фіксованими (передбачуваними) у свої формі написання, тому вирази у середині них не потребують термінуючого символу, хоча він і може бути використаний за бажанням. Так само як і у мові програмування С, в Duct присутні 3 головні види виразів які можуть бути описані тільки в глобальному просторі - Функції(або Процедури), Глобальні Змінні, Константи. У мові програмування Duct синтаксис виразів є ідентично подібним до мови програмуванні С, але в більш обмеженій формі, також символ для термінації у Duct залишається таким самим як у С - `;`. Коментарі також ідентичні до написання до стандарту С89 (`/**/`)так як Duct не виконує перевірки на символ кінця стрічок, тому у мові відсутні С++ подібні (`//`) коментарі. Вирази у Duct мають звичайний - математико-подібний синтаксис, на відміну від Lisp чи Forth подібних форм які використовують Польську Нотацію (Polish notation) [23] і Обернену Польську Нотацію або постфіксну нотацію (Reverse Polish Notation) [24]. Вирази не можуть бути використані просто так у “Кроці” адже вони потребують контекст у якому їх можна застосувати. Деякі дії чи оператори очікують блок - блок описує межу дії оператора, функції, тощо. На відміну від С блок не може бути використаний просто так адже він не бути виконувати ніяку дію зі значеннями на стеку, на відміну від С, і тому не матиме ніякого ефекту у роботі програми. Блок є списком любої кількості “Кроків”, навіть нульової.

Синтаксична структура виразів у мові:

DUCT

```
function(argument) {
	statement_1;
	statement_2 /* ; is missing, its ok*/
	if true {
		variable = (1 + 2 / (5 - 3)) * 0.15 /*expression within a statement*/
```

```
            (1 + 2 / 3) /*error, expression need context.*/
        }
        /*  . . .  */
}
```

Змінні та їх типи. Через відсутність статичних типів немає потреби у використанні синтаксису що вказує типи змінних, тому достатньо використовувати виключно символи без надання додаткової інформації про змінні. Змінні завжди є глобальними до усієї функції, навіть якщо вони були згадані не на початковому рівні функції. У розробленій мові програмування присутні такі типи даних:

- Integer - 32-ох бітне число.
- Float - 32-ох бітне ірраціональне число за (IEEE-754).
- Long - 64-ох бітне число.
- Double - 64-ох бітне ірраціональне число за (IEEE-754).
- Bool - 32 бітне значення при якому 0 - false, усі інші значення - true.
- Array - фіксований у розмірі масив що зберігає елементи одного типу даних.
- List - динамічний тип даних який на відміну від масиву може змінювати свій розмір.
- Set - структура даних схожа на List але здатна зберігати структуровані дані у графо-подібній формі.

Усі атомічні (числові та булеві) значення не в залежності від їх 32 чи 64 бітної розмірності є взаємозамінними, але при цьому застосовано набір правил який виконує пропагацію при використанні різних типів чи розмірностей в відповідності до *Таблиці 2.1.1*:

Таблиці 2.1.1 - Матриця неявного перетворення типів у контексті виразів мови.

Тип правого виразу	Тип лівого виразу	Узагальнений тип виразу
BOOL	INT	INT
INT	FLOAT	INT

BOOL	FLOAT	FLOAT
INT	LONG	LONG
FLOAT	LONG	LONG
DOUBLE	LONG	LONG
BOOL	LONG	LONG
FLOAT	DOUBLE	DOUBLE
INT	DOUBLE	INT
BOOL	DOUBLE	DOUBLE

Це означає що при використанні чисел і булевів користувач може спокійно проводити перетворення з одного типу на інший, але при потребі провести ручне обмеження цих перетворень надається через використання вбудованих функціонально-подібних операторів. При переповненні 32 чи 64 бітних слотів для чисел поведінка переповнення ідентична до системних мов програмування (зокрема C, C++, Rust, etc), де число обертається назад до 0.

Тип даних визначається автоматично при збірці коду з правої частини виразу що надає значення змінній, для ручного зазначення цього типу може використовуватися вбудована функція з назвою типу. Перевірка на стикання назв відбувається окремо для кожного виду конструкцій у мові, тому функції з назвою змінних чи типів можуть спокійно співіснувати.

Приклад синтаксису змінних:

DUCT

```
main() {

        variable = 10;      /*variable is created, ok*/
        variable = 11.1;    /*ok*/
        variable = “hello!” /*error: variable:number assigned string*/
        bigvariable = long(10) /*creates 64 bit float - 10.000000*/
        bigvariable = float(10) /*all primitives can store any numeric type*/
        variable = variable + bigvariable
```

```
        print(variable)
}
```

Ключові слова, Оператори. В мові програмування надані 7 операторів : if, else, while, loop, for, break, return; У мові відсутній оператор ‘switch’ чи ‘match’ адже їх задачу виконує оператор `if`, ‘else if`, `else`. Кожен оператор, окрім break, потребує вираз з яким він буде оперувати після чого в деяких випадках, а саму при `while`, `loop`, `for`, `if`, `else if`, `else`, має бути наданий блок у якому буде знаходитись тіло відгілкованої чи циклічної логіки.

Приклад операторів у мові програмування:

DUCT

```
main() {
        if (true) {
                while(false) {
                        if(0)           { /*...*/  }
                        else if(0)      { /*...*/ }
                        else            { /*...*/ }
                }
        }
        for i, 0 to 10 {
                if i > 5 {break;} else {return 1;}
        }
        return 0;
}
```

Додатково мова включає в себе 6 зарезервованих назв символів під назви типів для вбудованих операторів перетворення: `int`, `float`, `long`, `double`, `bool`.

Функції. Функції у мові програмування Duct описуються у відповідно до того як вони описані у C, де спочатку йде тип даних який функція повертає після чого йде її назва і список параметрів окутаний округлими дужкам, з відмінністю того - що тип змінних вказувати не потрібно так само як і не потрібно вказувати тип даних який повертає функція. Якщо функція не потребує аргументів то скобки можуть бути пустими. Вхідною точкою (Entry Point) для програми чи скрипту є (за замовчуванням) функція під назвою `main`, але через інтегровану природу мови цю функцію можливо змінити.

Приклад опису та використання функцій у мові:

DUCT

```
greet(who) {
	print(“Hello”, who, ”!”);
}
main() {
	greet(“master”)
}
```

При поверненні значень з функції, дані переносяться на верх кадру стеку до виклику функції де після перенесення всі дані після даних повернення - стираються. Через атомність усіх даних у мові функції можуть повертати списки, сети, числа, булеви.

Вбудований функціонал мови, оператори. Для того щоб уникнути створення небажаного чи складного функціоналу для мови, існують складні оператори - вид операторів що схожі за використанням на функції але замість виклику об’єктів функцій - генерують байтовий код для віртуальної машини напряму. Їх суть є, по факту, заміна директиви `__asm__` з C. Це потрібно для того щоб уникнути потреби створення макросистеми, бо така система при простій реалізації є занадто не надійною для включення у розроблену мову, а складна реалізація займатиме значну частину кодової бази задля надання посереднього функціоналу, загалом створення такої макросистеми у контексті Duct є невиправданим клопітом.

Оператор print; Оператор `print` є вбудованих оператором що замінює функцію виведення інформації до консолі. Цей оператор приймає список будь-якої кількості аргументів і автоматично виконує перетворення їх на форматований текст. Вивід цього тексту напряму залежить від налаштування середовища віртуальної машини, але зазвичай напрямом виводу є потік `stdout` (Standard Output).

Оператор write; Оператор `write` э подібним до оператора `print` за виключенням того що цей оператор має право обирати напрям виводу, і формат виводу даних, цей оператор існує для можливості Duct скриптів до

контрольованого (обмеженого користувачем за привілеями) спілкування з зовнішнім світом. Цей оператор може виводити інформацію як до наданих через внутрішнє середовище програмного забезпечення - вказівників, так і до файлів і “труб” (POSIX Pipes). Оператор має вибір між форматуваннями:

- Відсутності формату, все написано напряму у вигляді байтів так як дані збережені у пам’яті
- Форматовані Строки, подібно до того як `print` виводить відформатовану текстову інформацію.
- Неформатовані Строки, виводить перетворену інформацію до тексту без використання форматування що спрощує читання, існує для роботи з іншими середовищами що проводять аналіз і обробку тексту.
- BSF (Binary Serialization Format) - формат схожий по структурі на JSON але у якому інформація зберігається у бінарному (сирому) вигляді. Структура зберігання подібна до формату RIFF що застосований у мультимедійному форматі AVI, WAV.

Оператор append; Оператор `append` виступає заміною дій з списками через арифметичний оператор `+`, для доповнення списка потрібно викликати оператор `append` який приймає два аргументи: список який потрібно доповнити та список які буде доповнювати. Оператор може бути застосований тільки до списків адже тільки списки можуть бути збільшені в розмірі і розширені новими елементами.

Оператори array та list; Замість використання складного, вузькоспеціалізованого синтаксису який ніде більше не використовуватиметься у мові, створення пустих масивів чи списків з вказаною кількістю елементів виконують оператори `array` та `list`. Оператор `array` приймає тип масиву та його розмір як аргументи, Оператор `list` приймає тільки кількість елементів які мають бути створені, через свою поліморфність. Усі елементи ініціалізовані до нуля.

Оператор assert; У бібліотеках багатьох мов програмування існує функція чи макрос для перевірки виразу на дійсність задля забезпечнення надійності даної у коді “обіцянки”, у випадку коли “обіцянка” не виконана програма зупиняє свою роботу і надає інформацію де сталося порушення. Така операція є важливою під

час розробки будь-його адже деякі дії над інформацією потребують виконання набору вимог і перевірка на ці вимоги через бранчування може займати значний об’єм тексту програми. Тому у мові Duct `assert` є стандартним включеним оператором. В Duct `assert` приймає один обов’язковий аргумент - умову яка має виконуватись, і другий, опціональний, - строку повідомлення яке має бути виведено у випадку коли `assert` спрацював.

Комплексні типи.

Масив. У мові Duct массив є фіксованим у розмірі типом даних яка може бути розміщений як локально - усередині функції, так і поза її межами - глобально. Масив є складною до організації у пам'яті структурою даних через значну фрагментацію пам’яті що виникає при спробах змінювати розмір цієї структури даних, тому у мові Duct ця структура даних не може змінювати свій розмір щоб уникнути ускладнень у дизайні системи автоматичного керування пам'яттю. Синтаксично массив описується при створенні змінної, з прямого виразу багатьох однакових елементів - квадратних дужок. Масив обирає свій тип даних автоматично з першого елементу масиву, якщо елементи масиву мають трохи різний цифровий тип - масив автоматично перетворює усі цифрові значення у відповідний до першого елементу тип. Масив також можливо створити з використання вбудованого ініціалізатору `array(type, size)` де на рівні байтового коду буде вказано створення блоку для масиву розміром `ТИП * РОЗМІР`. У випадку коли тип не є комплексним або його не можливо перетворити по інших причинам, наприклад тип даних не був завірений з функції яка його повернула, то програма пропиняє свою роботи з помилкою що вказує на проблемне створення масиву з несумісних типів.

Приклад опису масивів користуючись елементною ініціалізацією:

DUCT

```
main() {
	buffer = [byte()]
	greet(“master”)
}
```

Список. Список є універсальним типом даних для зображення динамічної у своєму розмірі інформації, список може зберігати інформацію любих типів і дозволяє поіндекскнний доступ до своїх елементів схоже до масиву. На відміну від масиву який використовує один великий блок пам’яті, список складається з багатьох елементів які з'єднані між собою вказівними ціпками. Список має схожий синтаксис до масиву, але потребує використання додаткового символу для ідентифікації набор елементів як списку: `@` оператора.

Сет. Сет - це комплексний тип даних який існує для зображення складних структурованих даних. Сет складається з полів доступ до яких надається через використання `.` (Dot) синтаксису, ідентичного синтаксису доступу до елементів до мови програмування С. На відміну від С, Сет в Duct сформовано з списку полів де кожне поле має “Ідентифікатор” за яких виконується пошук полів. Поля у Сеті не можуть повторюватись, тому спроба створити поле повторно видасть помилку. Сет має асоціативні зв'язки з усіма його елементами, і якщо сет має бути видалено чи мігрувано (в пам’яті), то ця діє відбувається рекурсивно з усіма його елементами. Сет після створення може бути доповнено допоки він не був заблокований, коли сет повернено з функції він блокується щоб запобігти випадкових чи навмисних доповнень його полів. При потребі доповнити сет, його може бути включено в інший Сет як поле. Синтаксис сету нагадує такий у мові С для структур але без використання ‘.’ перед кожним полем при створенні об’єкту.

2.2. Опис обраних технологій та структур даних для реалізації мови.

Вибір мови програмування - Мова програмування С. У межах реалізації даного проекту як основну мову програмування було обрано мову С, що обґрунтовується низкою критеріїв:

- *Поширеність.* Мова програмування С є однією з найпоширеніших мов у практичному застосуванні. Її значущість визначається не стільки обсягом програмного забезпечення, написаного безпосередньо на С, скільки часткою технологій, що базуються на ній або використовують бібліотеки, реалізовані цією мовою. Попри домінування таких мов, як JavaScript, Java та Python [16], слід зазначити, що значна частина їхньої інфраструктури побудована із

використанням компонентів, реалізованих на C. Операційні системи сучасного рівня також переважно розроблені з використанням C, що підкреслює її фундаментальну роль у сучасній обчислювальній інфраструктурі. У зв'язку з цим мову C часто називають «матір'ю мов програмування» [17], оскільки значна частина критично важливого програмного забезпечення світу базується саме на ній.

- *Простота.* Мова C характеризується відносно мінімалістичним синтаксисом та обмеженим набором мовних конструкцій. Кількість ключових слів є невеликою (приблизно 32), що свідчить про низький рівень синтаксичної надлишковості. Однак простота мови визначається не лише синтаксичними властивостями, а й концептуальною моделлю, яка базується на прямій роботі з пам'яттю та мінімальній кількості абстракцій. Основними категоріями даних є скалярні типи, вказівники та структури, які безпосередньо відображаються на рівень машинної пам'яті, організованої у вигляді байтової адресації. Керування потоком виконання реалізується за допомогою базових конструкцій розгалуження та циклів, а також умовного використання оператора безумовного переходу `goto`. Відсутність вбудованих високорівневих абстракцій сприяє прозорості виконання програм на рівні машинної моделі, однак вимагає від розробника повного контролю над структурою програми та ресурсами.
- *Варіативність, виразність.* Проста синтаксична структура мови C не обмежує її функціональних можливостей. На її основі може бути реалізовано широкий спектр програмних систем, продуктивність, надійність та безпека яких залежать переважно від якості реалізації, а не від обмежень мови. Основними факторами, що визначають складність розробки на C, є досвід розробника, доступний час та якість архітектурних рішень. Відсутність високорівневих автоматизованих механізмів призводить до необхідності ручної реалізації багатьох компонентів, які в інших мовах надаються стандартними засобами. У зв'язку з цим мову C іноді характеризують як складну, однак така характеристика є умовною, оскільки складність виникає

не на рівні окремих конструкцій, а внаслідок їхньої композиції, що фактично відповідає поняттю системної комплексності.

- *Сумісність.* Завдяки широкому поширенню мови С та значній кількості бібліотек і програмних компонентів, реалізованих на ній, більшість сучасних мов програмування та програмних систем забезпечують можливість взаємодії з кодом С через механізм Foreign Function Interface (FFI) [15]. Будь-яка мова програмування, орієнтована на практичне застосування у реальних системах, змушена підтримувати інтеграцію з бібліотеками, реалізованими на С. Таким чином, бібліотеки С фактично виступають універсальним інтерфейсом між різними мовами програмування. Прикладами мов із розвиненою підтримкою FFI до С є Rust, Python, PHP, Haskell, Java та Lua [18].
- *Пітримка.* Мова програмування С підтримується на практично всіх існуючих архітектурах процесорів та операційних системах. У більшості стандартних середовищ розробки компілятор С є доступним за замовчуванням, що забезпечує можливість створення програмного забезпечення навіть у мінімальних системних конфігураціях. Розвинена екосистема інструментів дозволяє здійснювати крос-компіляцію, тобто компіляцію програм для однієї операційної системи в середовищі іншої. Наприклад, проєкт MinGW [19] забезпечує інструментарій для компіляції програм під Windows у середовищі Linux. Навіть за відсутності спеціалізованих засобів крос-компіляції портативність програм на С може бути забезпечена шляхом використання платформних шарів абстракції або стандартної бібліотеки С (libc). У порівнянні з більшістю інших мов програмування, С демонструє один із найвищих рівнів сумісності та переносимості між системами різного типу.

Структури даних потрібні для мови програмування, недоліки вбудованих рішень. Скриптова мова програмування потребує як алгоритмів і структур даних які притаманні реальному комп’ютеру так і набір високорівневих рішень що будуть спрощувати та доповняти функціонал мови.

Динамічний массив. Самий простий і в той самий час ефективний для комп’ютера при роботі структура даних - Масив. Уся суть масиву - це вказівник за адресою якого знаходиться кілька елементів однаково розміру. У мові програмування С присутній синтаксис для опису масивів та отримання доступу до елементів, у випадку якого дії з масивом автоматично перетворюються компілятором на арифметику з вказівниками. Але стандартний масив у С має свій набір недоліків - починаючи з відсутності можливості зберігати розмір масиву разом з його елементами, що є функцією в більшості сучасних компільованих мовах програмування, закінчуючи відсутністю статичної (передчасної) перевірки на відсутність порушення меж масиву при отриманні доступу до його елементів (Bounds Checking). Головним недоліком масиву є відсутність динамічно змінювати його розмір що вносить значні обмеження на масштаб з яким розробнику потрібно створювати функціонал програмного забезпечення. Зазвичай основним методом для створення та використання динамічного масиву, актуальних для кінця 20 століття” був відповідний алгоритм дій який мав виконати С розробник:

- Виконати запит на пам’ять для масиву. Зазвичай з використанням функції `malloc` з стандартної С бібліотеки. Ця функція приймає один параметр - розмір блоку пам’яті у байтах який користувач хоче отримати. Результатом функції є вказівник на блок пам’яті який було виділено, далі потрібно перевірити чи він є корректним (у противному випадку відбулася помилка) і далі повторити його на вказівник на потрібний для нас тип даних.
- Використовувати масив допоки кількість елементів не перевищує розмір блоку.
- Використати функцію з LibC під назвою `realloc` що потребує вказівник на попередній блок і розмір нового блоку. Ця функція виконує займ нового блоку з новим розміром, виконує копіювання байт з попереднього блоку до нового з урахуванням найменшого спільного розміру, звільняє попередній блок, і повертає вказівник на новий блок.

Як можна помітити це багато кроків для простої структури даних - масиву що потребує створення нового блоку пам’яті більшого розміру у який проводиться копіювання даних з попереднього масиву при потребі у зміні розміру. Також можна помітити що така структура даних потребує два розміри: кількість елементів у масиві та місткість пам’яті масиву у елементах. Через просту суть такої структури даних та її універсальну потребу у любому програмному забезпеченні, її завжди реалізують розробники С у свої проекта щоб спростити повторювані дії ручного управління вказівниками, що як було виявлено - джерело вразливостей та проблем.

Стек. Стек - це структура даних яка розміщена всередині масиву, зазвичай фіксованого розміру, до якого додавання чи прибирання елементів дозволено тільки відносно “верху” цього масиву (його останнього елементу). Ця структура даних не надана стандартною бібліотекою С адже вона є найпростішою до реалізації з усіх перелічених структур даних. Як вже було зауважено в Розділ 1.1 Стек є основою розрахункового методу використаного в сучасних комп'ютерах тому ця структура даних обов’язкова для реалізації будь-якого розрахункового середовища що намагається провести віртуалізацію реального комп’ютера або створити своє власне віртуальне середовище.

Строка, “Зрізок” (String Slice). Строка є найпоширенішою структурою даних що не належить до базових (Атомів таких як: числа, булева, вказівники) типів даних. Вона існує для зображення текстової інформації у програмі і часто використовується для обробки вхідних та вихідних даних. В мові програмування С строки це масиви байт які всі закінчуються спеціальним числових значенням - нулем, що також називається “Термінуючим Нулем” (Null terminator). Кожен символ строки закодований або у один байт як числове значення від 0 до 127 за ASCII (American Standard Code for Information Interchange) або у кілька байт за UTF-8 (Unicode Transformation Format). Значення інформації під вказівником як тексту від будь-яких інших цифрових значень розрізняється виключно розробником і контекстом використання. Незважаючи на перевагу у 100% використання пам’яті під строку, “С-подібні” строки є відомим за своєю кількістю недоліків які ускладнюють процес редагування тексту. Через відсутність розміру строки разом з

її даними для отримання розміру потрібно виконувати прохід по всій строці допоки термінуючий символ не було знайдено. Створення підстрок потребує створення нової строки адже щоб розділити строку потрібно внести до неї зміни (що в свою чергу з “буквальними строками” - власноруч внесеними строками, не є можливим адже такі строки доступні тільки для прочитання, для створення модифікуючих строк потрібно проводити копіювання “буквальної строки” до додаткового буфера який має право на внесення змін. Рішенням для таких проблем є просте - зберігати вказівник на строку разом з розміром строки, таким чином строки можна ділити без потреби вносити невиправні зміни. Також важливо зауважити те, що строки ассоціюються з видом пам’яті до якого не можна вносити зміни (тип пам’яті “тільки для читання” - Read Only), тому для строки до якої можуть вноситься зміни зазвичай створюють окрему структуру даних схожу за принципом до масиву але з урахуванням саме текстових маніпуляцій - “Будувач строк” (або на англ. Stringbuilder).

Stringbuilder. **“**Будувач строк” - це структура даних яка зображає строку у ефективному вигляді для внесення модифікацій. На відміну від звичайних строк які використовуються тільки для читання, “Будувач” використовується для активного доповнення тексту. “Будувач строк” може бути реалізований через декілька структур даних: масив, список, дерево, їх різні комбінації. Зазвичай “Будувач строк” реалізований як масив який повинен працювати виключно з текстовою інформацією (побайтовою інформацією). У мові програмування С не існує “Будувача строк”, найближчий аналог цієї структури даних це звичайний буфер фіксованого розміру до якого застосовується функція `sprintf`. Такий вид будування строк є небезпечним адже він потребує делікатного використання `snprintf` замість `sprintf` з перевіркою на сторонні (неочікувані) випадки та перевіркою на можливі помилки.

Ціпка, Список. З’єднаний список (або на англ. Linked List), також іноді іменований “Ціпкою” - це структура даних як забезпечує ассоціативність даних у наборі із кількох елементів не за рахунок їх структурованості в пам’яті а за рахунок їх з'єднання через вказівники. У випадку з масивом інформація розміщена

поряд до один одної що забезпечує її ассоціативність при отриманні, адже для отримання сусіднього значення достатньо пройти на один розмір об’єкта вперед у пам’яті. Список використовує ідею того що кожен елемент є незалежним від розміщення і тому асоціюється до інших елементів у списку з використанням вказівників, Елемент списку зазвичай складається з вказівника на елемент до якого він належить, або саму інформацію до якого він належить, та вказівника на наступний елемент списку. Іноді для ефективності проходу по елементам список може мати два вказівники для наступного та попереднього елементів, замість одного - наступного, це надає змогу рухатись по списку в протилежному напрямі що є частою операцією у алгоритмах що виконують пошук. Ця структура даних не присутня в мові програмування С і не надається стандартною бібліотекою С, реалізація цієї структури даних різниця від джерела до джерела і немає встановленої форми для реалізації, тому ця структура даних зазвичай створена для вирішення саме тих задач які притаманні окремому проекту чи розробнику.

Дерево, Граф. Дерево (або не циклічний, орієнтований граф) - це структура даних яка побудована на ідеї “Списку” але замість використання одного вказівника на наступний елемент, граф використовує два - вказівник на наступний елемент та вказівник на наступний вимір(або шар) у дереві. Завдяки використанню кілької вказівників що зображає кілька вимірів - стає можливим зображати ієраріїчну інформацію. Дерева є способом зображення структури та залежності у комп’ютері. Застосування цієї структури даних при розробці мови програмування є фундаментальною вимогою адже структура програми, її синтаксису зображується та зберігається у пам’яті за допомогою дерева.

Аллокатор. Аллокатор - це структура даних для відслідковування та управління зайнятих від ОС блоків пам’яті, її задача виконувати розмітку та надання пам’яті користуючись механізмами що надає ОС для отримання пам’яті. Додатково аллокатор може займатися управлінням цими блоками і прийняттям рішень щодо їх звільнення. У мові програмування С через її стандартну бібліотеку надано “стандартний” Libc аллокатор, його реалізація займає 6000 строк коду [21]. Цей аллокатор немає вагомих недоліків з технічної сторони, але з точки зору

простоти використання він є складним, адже від розробника вимагається повний контроль над усіма блоками пам’яті та їх ручне звільнення. Також цей аллокатор при великій кількості займу нерівномірних блоків може страждати від фрагментації - процессу розбиття об’єму пам’яті на блоки які потім не можна зібрати докупи без руйнівних дій чи витрачання часу. Така проблема характерна не корректному використанню блок-подібного аллокатору і тому у таких випадка варто розглядати або альтерантивний тип аллоктаору, або зміну стратегії керування пам’ятью. У контексті інтерпритованої мови аллоактор який структурує дані у лінійні послідовності має кращу продуктивність ніж блокоподібний аллокатор, тому у цій роботі було використано аллокатор Арени або лінійний аллокатор.

Аргументація створення власних реалізацій структур даних замість використання існуючих рішень. При швидкому погляді на доступні реалізації на структури даних для мови програмування C, помітна тенденція у відсутності одного “найкращого” [20] рішення яке виступає залежність для багатьох проектів, натомість кожен окремий розробник реалізує часто ті самі елементи логіки і структури даних індивідуально і окремо від інших подібних реалізацій. Причин на це я декілька і вони можуть бути не очевидними на перший погляд:

- *Простота.* Більшість структур даних не є складними по своїй суті так як виконують зазвичай одну конкретну задачу або вирішують один недолік іншої структури даних. Через свою простоту більшість структур даних не потребують професійних навиків для створення та підтримки, і відсутність цих навиків не сказується погано на будь-яких аспектах структур даних окрім як кількості написаного коду та простота його використання. Припускаючи що розробник користується власно-розробленими структурами даних, ймовірність загубити серйозний недолік чи вразливість стає мізерною якщо не майже неможливою.
- *Надійність*. Сліпа довіра до стороннього коду стала джерелом багатьох вразливостей та атак. Наприклад NPM (Node Package Manager) [22] стає цілю для зловмисників так як бібліотеки та код розміщений на цій платформі завжди, в певний момент, може бути скомпрометовано, саме тому за 2025 рік

поняд 804 бібліотеки було скомпрометовано і зламано задля поширення зловмисного програмного забезпечення. Саме прості алгоритми і структури даних часто стають ціллю для атаки якщо вони мають стороннє розповсюдження і не належать до коду проекту в якому знаходяться.

- *Контроль*. Створення власного рішення убезпечує від інцидентів і надає змогу детально та цільно тестувати та розробляти рішення спеціально під поставлені вимоги. При використанні існуючого рішення процесс внесення змін напряму залежить від ліцензіювання та бажання автора проекту вносити зміни, у деяких випадках надана можливість вільно доповнювати та поширювати змінені копії рішення але це може бути більш кропітким процессом адже стороння кодова база може складатись з десятиліть змін які мають за собою історичну чи технічну причину про які користувачу може бути не повідомлено. Також зачасту “універсальне” рішення потребує компромісів на факторах зручності використання, швидкості роботи чи наявності певного функціоналу. Подібні обмеження часто є джерелом проблем з не очікуваною поведінкою, швидкістю роботи, кількістю часу необхідного для обхожу обмеження, тощо. Розробка власного рішення майже завжди є доцільним варіантом для уникнення подібних проблем.

Технічні нюанси та ідея використані при створенні структур даних. Мові програмування С характерний обмежений синтаксис через давність цієї мови програмування і потреби в сумісності старого коду з новими стандартами мови, тому для реалізації високорівневих функцій для структур даних у мові С потрібно креативно використовувати існуючий функціонал для забезпечення простої але ефективної до використання та поративної реалізації алгоритмів і структур даних.

Універсальні типи даних. У мовах програмування C++, Rust, Ada та інших мовах програмування присутні системи “Універсальної” типізації. Така система забезпечує створення типу даних (структури чи классу) який може бути застосований до будь-якого іншого типу даних. Прикладом такого функціоналу є C++ “Шаблони” (Templates) - це тип який потребує мінімум один аргумент яким

виступає тип даних. “Шаблони” вирішують проблему повторюваності операції у алгоритма чи структурах даних де природа дії над інформацією не змінюється в залежності від типу цієї інформації або потребує мінімальних змін. У С такого функціоналу немає, тому для опису структур даних які можуть оперувати над будь-якими типами С розробники використовують системи чи особливості компілятора до максимуму. Головну особливість С яку може бути використана для створення універсальних структур даних збереження інформації це свободу використання вказівників. Вказівник тільки під час збірки коду має асоційований до нього типу, адже все що тип визначає у такому контексті це розмір одного елементу в послідовності. Завдяки цій особливості вказівників все що потрібно для проведення керувальних операцій над даними (копіювання, переміщення, очищення) може виконувати на по-байтовій основі. Для маніпуляцій над типом у функціях може використовуватись шаблонний тип який описує структуру усіх видів одного і того ж універсального типу. Тип для структури даних може як зберігатися всередині самої структури після її створення, так і отримувати з інформації про вказівник у випадку коли тип було створено вручну. *Приклад реалізації експоненціального масиву з використанням цих принципів:*

C

```
. . .
#define hc_Leash(T) struct {        \
    size_t  count;                  \
    T*      items[sizeof(size_t)*8];\
    void*   (*alloc)(size_t);       \
    void    (*free)(void*);         \
}

#define lsh_typesize(leash) sizeof((leash)->items[0][0])
#define lsh_get(leash, I)\
    (leash.items[lsh_partition(I)][lsh_local_index(&leash, I)])
#define lsh_append(leash, I)\
    lsh_append_ex(leash, lsh_typesize(leash),\
            I, sizeof(*(I)) )
#define lsh_remove_unordered(leash, I)\
    lsh_remove_unordered_ex(leash, lsh_typesize(leash), (I))

typedef struct {
    size_t  count;
```

```
    void*   items[sizeof(size_t)*8];
    void*   (*alloc)(size_t);
    void    (*free)(void*);
} hc_LeashBase;
. . .
```

Експоненціальний масив має “Шаблон” до якого можуть бути перетворені усі вказівники до такого масиву, і враховуючи що структура пам’яті залишається незмінною це дає змогу функціям оперувати над будь яким типом що має відповідну репрезентацію у пам’яті. Розмір хостового типу з даного масиву можливо отримати через оператор `sizeof` що є вбудованим функціоналом мови.

Створення власної реалізації Масивів, Строк, Аллокатора пам’яті - Арени, Списків, Тощо.

Опис створеного рішення. В процесі створення мови програмування та інших проектів було створено набір структур даних з деякими функціями які притаманні високорівневим мовам. Назва проекту структур даних є “HCH” [26] - Handy C Headers (з англ. Корисні С Заголовки). В рамках цього проекту створено багато структур даних але у роботі будуть згадані тільки доцільні до теми роботи.

Динамічний Массив. Массив як структура даних є вказівником на пам’ять де зберігається інформація однакового розміру, також масиву характерний розмір в рамках якого потрібно отримувати доступ до елементів адже у випадку порушення меж наданої пам’яті є порушенням як її цілісності так і безпеки. Для масиву який може змінювати розмір потрібно відслідковувати розмір блоку пам’яті в елементах масиву від кількості існуючих елементів у масиві. Також массив має мати інформацію про розмір типу даних який він зберігає адже вказівник є всього лише адресою. Таким чином массив є структурою даних що має складатись з: *Вказівника на пам’ять* (зміст) *масиву*, *кількість елементів* у масиві, *об’єм масиву* доступний до використання. *В коді ця структура даних представлений у вигляді:*

C

```
. . .
typedef struct {
      void*  items;
      size_t count, capacity;
} hc_ArrayBase;
```

```
#define hc_Array(T) struct {\
    T* items;               \
    size_t count, capacity; \
}
#define hc_array_typed(A) (A), sizeof(((A)->items)[0])
#define hc_array_append_heap(A, I)              \
    hc_array_append_heap_ex(                    \
            hc_array_typed(A),                  \
            &I, sizeof(I))
. . .
```

Тип масиву створюється з використанням макрос `hc_Array` який описує структур з елементами типу `T` та кількістю і ємністю масиву. Розмір типу асоціюється через тип вказівника що массив використовує для свої елементів, розмір типу елемента та інформація про масив надається всім універсальним функціям в бібліотеці через макрос `hc_array_typed`. Макрос `hc_array_append_heap` як і багато інших подібних “функцій” передають і розгортають свої аргументи до функція які оперують напряму з даними про структур даних масиву.

Строка та її Будувач. Строка як було зазначено в Розділ 2.2 має складатись з вказівника на текст та розмір строки. Текст строки немає допускатись до модифікації адже вона використовується виключно для читання чи перегляду тексту в рамках меж її розміру. Бібліотека строки включає усі необхідній функції для аналізу тексту та зміни контексту погляд на текст.

Ітерфейс структура даних “Текстова строк”:

C

```
. . .
typedef struct {
       const char* items;
       size_t     count;
} hc_String;
INLINE hc_String     str_make(const char* cstr);
INLINE hc_String     str_make_sized(const char* cstr, size_t count);
INLINE hc_String     str_dup(hc_String orig);
INLINE bool     str_is_empty(hc_String s);
INLINE void     str_reset(hc_String* s);

hc_String      str_trim(hc_String s);
hc_String      str_trim_left(hc_String s);
```

```
hc_String      str_trim_right(hc_String s);
hc_String      str_substr(hc_String orig, size_t index, size_t count);
hc_String      str_split_by_chars(hc_String *s, const char* chars);
. . .
```

Для функціоналу зміни строки існує “Будувач” строки який нагадує по своїй структурі - масив, що робить його взаємозамінним з будувачем через однакову структуру у пам’яті, але на відміну від масиву його операції орієнтовані на доповнення та форматуванні інформації у текст.

Ітерфейс структура даних “Будувач строк”:

C
```
. . .
typedef struct {
      // transmutable -> DA , String
      char*  items;
      size_t count, capacity;
      const char* spacer;
} StringBuilder;

#define sb_arrlit(...)          ((const char*[]) {__VA_ARGS__})
#define sb_arrlen(arr)          (sizeof(arr) / sizeof((arr)[0]))
#define sb_arrlit_len(...)      (sb_arrlen((__VA_ARGS__)))
#define sb_append(sb, ...) \
      sb__append(\
            sb,\
            sb_arrlit(__VA_ARGS__),\
            sb_arrlen(sb_arrlit(__VA_ARGS__)))\

void sb_appendf(StringBuilder* sb, const char* fmt, ...);
. . .
```

Функціонал бібліотеки знаходиться в двох функціях: `sb__append` та `sb_appendf`, одна функція надає змогу розробнику можливість доповнювати “Будівника” строки одразу кількома строками за один виклик функції, друга є функцією для форматування числових чи текстових даних і додавання цих даних до будівника, подібно до того як C функція `snprintf` це робить для фіксованого буфера тексту.

Список та Дерево. Ці структури даних, через свою схожість у організації та принципом роботи, у HCH реалізовані як одна структура даних - “Ціпка” (Link). На відміну від інших структур даних які використовують визначення розміру елементів один раз при компіляції, “Ціпка” зберігає цю інформацію створюючи

“обгортку” яка заповнена під час взаємодії з структурою даних. Завдяки цьому принципу немає різниці де схована інформація про ціпку розміщена у структурі даних адже алгоритм її заповнення це визначить та збереже автоматично під час збірки програми, в описі “Ціпки” будуть пропущені деталі реалізації алгоритмів через об’єм інформації потрібний для згадування, детальніше можна прочитати у повному коді проекту.

Інтерфейс структура даних “Список”:

C

```
. . .
#define hc_Link(T) T*

#define ImplementLink LinkData
#define LinkHead LinkData
#define LinkData __LinkData__ __head__;
//
// Interface
//
#define hc_li_insert(list, item)            hc_li_insert_wrap(list, item)
#define hc_li_append(list, item)            hc_li_append_wrap(list, item)
#define hc_li_append_children(list, item)   hc_li_append_children_wrap(list, item)
#define hc_li_connect(list, item)           hc_li_connect_wrap(list, item)

#define hc_li_next(list)        hc_li_generic_next(list)
#define hc_li_children(list)    hc_li_generic_children(list)
#define hc_li_parent(list)      hc_li_generic_parent(list)
#define hc_li_foreach(LI, T, I) hc_li_generic_foreach(LI, T, I)
#define hc_li_defer(LI, T)      hc_li_generic_defer()
typedef struct {
    size_t typesize;
    void   *next,   *prev,
           *head,   *tail,
           *children, *parent
    ;
} __LinkData__;
. . .
```

Структура даних може бути застосована до будь-якого типу методом доповнення його необхідною мета-інформацією потрібною для роботи алгоритмів структури даних.

Приклад використання структури даних “Список”:

C

```
. . .
typedef struct Node {
            ImplementLink;
            int n;
} Node;
int main(void) {
             Link(Node) tree = 0;
             Node
                root    = make_n(1),
                child1  = make_n(2),
                child2  = make_n(3),
                child3  = make_n(4),
                child4  = make_n(5),
                child5  = make_n(6)
            ;

            li_append(tree, &root);

            li_append_children(tree, &child1);
            li_append_children(tree, &child2);
            li_append_children(tree, &child3);
            li_append_children(tree, &child4);
}
. . .
```

Аллокатор - Арена. Для реалізації альтернативної схеми керування пам'яттю проект HCH включає в себе власний аллокатор пам’яті який використовує інший підхід до керування пам’яті. Зазвичай займ пам’яті відбувається через стандартний сторінковий аллокатор `malloc`, навість у мовах програмування C++ чи Rust часто використовується саме цей аллокатор черей його відносну простоту та широку розповсюдженість. Він є функціональним і простим у використання рішенням у випадку корректного і строгого керування пам'яттю системою у якому його використано. На Жаль він має обмеження по ефективності (швидкості) своєї роботи при використанні нерівномірних за розміром даних, також через результат цієї нерівномірності - фрагментації, сповільнюється робота алгоритмів які використовують цю пам’ять для збереження своєї інформації. Тому мало-відомим та ефективним рішенням є аллокатори які вносять значне обмеження у принцип керування пам’яті змушуючи розробника системи працювати в середині

конкретного методу(стилю) організації інформації що спрощує процес керування пам’яті і допомагає по максимуму використовувати ресурси комп’ютера. Аллокатор арени є простішою формою лінійного аллокатора де єдиний, або кілька з'єднаних віртуальних, блок(-ів) пам’яті використовуються для збереження інформації. Через лінійність аллокатора уся інформація тісно упакована в пам’яті і є ефективною для кешування але натомість немає ніякого розмежування блоків і тому при потребі змінити існуючий блок пам’яті у розмірі оптимальним рішенням є створення нового займу. Перевагою такого аллокатору пам’яті є ассоціативність усієї інформації що була отримана за допомогою нього, замість потреби звільнення кожного окремого вказівника при аллокації пам’яті натомість звільняється весь блок за раз. Такий аллокатор є простий у реалізації але також при корректному використанні надає переваги у швидкості роботи над будь-якими іншими (зазвичай блоковими) видами аллокаторів. Основним компонентом є вказівник на блок віртуальної пам’яті наданий операційною системою який є початковим (або єдиним) блоком пам'яті в аллокаторі, в самому блоці є заголовочна інформація про блок: розмір блоку у байтах та вказівник на наступний блок. Також аллокатор має вказівник на нинішню адресу з якої можливо вирахувати потребу у переносу до наступного блоку.

Приклад реалізації Arena аллокатору (Інтерфейс заголовочного файлу):

C

```
. . .
// customize block size to your desire.
#ifndef ARENA_NODE_SIZE // 2,4,16,64,512kb,  (4096*1024) - linux max page size
#   define ARENA_NODE_SIZE ((1<<14)*1024) // 16 mb
#endif
#define ARENA_HEADER_SIZE sizeof(ArenaNode)
typedef unsigned char arena_bitmask8;
enum {
    ARENA_RESET_SIZE   = (1 << 0),
    ARENA_RESET_MEMORY = (1 << 1),
    ARENA_FREE_NODES   = (1 << 2),
};
typedef struct ArenaNode {
    size_t              allocated;
    struct ArenaNode*   next;
```

```
    char                data[];
} ArenaNode;
typedef struct {
    size_t      totally_allocated;
    ArenaNode*  memory;
} Arena;
// use these
void*       arena_alloc     (Arena*, size_t);
void        arena_memcpy    (Arena* a, void* data, size_t size);
void        arena_reset     (Arena*, int opt);
#define     arena_put(A, I) arena_memcpy(A, &I, sizeof(I))
#define     arena_clear(A)  arena_reset(A, ARENA_RESET_SIZE | ARENA_RESET_MEMORY)
#define     arena_rewind(A) arena_reset(A, ARENA_RESET_SIZE)
#define     arena_free(A)   arena_reset(A, ARENA_RESET_SIZE | ARENA_RESET_MEMORY |
ARENA_FREE_NODES)
. . .
```

У реалізації “Арени” для проекту HCH та мови програмування - використані блоки невеликого розміру (для розміщуваності у L1 кеш) та в кожному блоці присутня інформація необхідна для незалежності блоку від інших блоків. Швидкість роботи цього аллокатора було перевірено з `malloc` та довільною реалізацією “Сміттєзбирача” у моїй іншій роботі: *“SIMPLER AND SAFER MEMORY ALLOCATION STRATEGY - ARENA ALLOCATOR. MAKING MEMORY ALLOCATIONS EASIER AND SAFER”.*

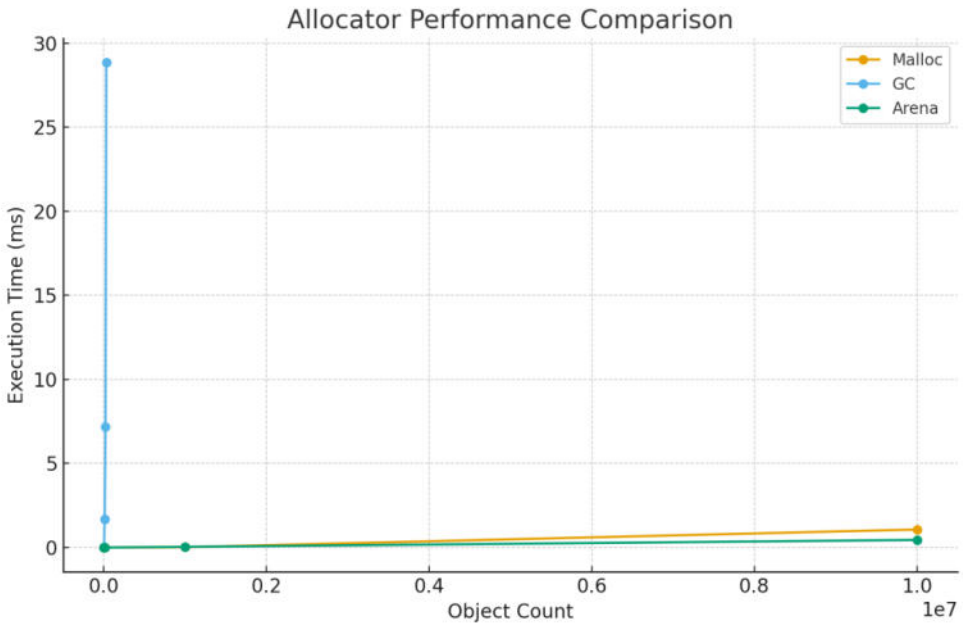

Рисунок 2.2.1 - Демонстрація Бенчмарку в публікації про переваги Arena allocator [39].

2.3. Розробка віртуального середовища (ВМ) для роботи мови програмування у середині користувацького ПЗ.

Аргументування вибору Віртуальної машини понад Інтерпретатор. Інтерпретатор за своєю суттю це лексичний аналіз тексту програми на коректність написання та виконання вказаних у ньому інструкцій відповідно до синтаксису мови. Інтерпретатор є методом створення інтерактивного середовища для взаємодії користувача з програмним забезпеченням, але він має недоліки роблять його непридатним до використання у небезпечних середовищах.

- *Повільна швидкість роботи Інтерпретатора.* Віртуальне середовища побудоване на віртуальній машині працює по принципу виконання простих та дискретних у часі виконання інструкцій, через створення системи “віртуального комп’ютера” швидкість роботи алгоритмів що були побудовані для роботи на покрокових інструкція залишається досить швидкою, хоча і в 2-3 рази повільнішою ніж у реальному комп’ютері через потребу використання кількох реальних інструкцій ЦП для виконання інструкції у Віртуальній машині. Для виконання програм на віртуальній машині потрібен крок перетворення коду користувача на інструкції ВМ, що потребує аналізу тексту перед його виконанням, такий аналіз може виконувати більш детальні і інформативну перевірку аніж їх дослівне виконання, внаслідок чого до вже досить швидких інструкцій можуть бути застосовані додаткові оптимізації. Інтерпретація на має такої переваги за дизайном адже виконання програми відбувається одночасно з процессом зчитування тексту.
- *Прямий доступ до оброблюваних даних.* Інтерпретатор працює напряму відповідно до тексту що він інтерпретує, це означає що для виконання розрахункових операцій інтерпретатору потрібно мати прямий доступ до вказівників з інформацією яка потрібна скрипту чи програмі для роботи. Через відсутність повноцінного віртуального середовища у середині якого

може відбуватися перевірка на рівні індивідуальних інструкцій, програми які проходять процес інтерпретації мають більшу вірогідність бути вразливими до експлуатації вказівників для переповнення буфера чи використання після звільнення. Звісно ці вразливості можливо передбачити і створити детальну систему перевірки на коректність програм що проходять інтерпретацію, але таке рішення є суб-оптимальним через значний об'єм роботи та складність кінцевої системи, що не підходить до вимог поставлених у Розділі 1.5.

- *Монолітність і інкапсулятивність віртуальної машини у порівнянні з інтерпретатором.* Інтерпретатор за визначенням має виконувати мінімум дві чи три алгоритма для роботи - токенизацію, парсування, інтерпретацію, тощо. Через потребу виконувати декілька задач одночасно які можуть вимагати доступ до різних об'ємів інформації, інтерпретатор не може виконувати одну з задач ефективніше можливого у такому випадку ліміту, тому тут з'являється ще одна перевага віртуальної машини - Монолітність. Монолітність є критерієм однозадачность системи. У випадку використання ВМ, мова програмування що створюється отримує два конкретні кроки які виконуються незалежно один від одного, що спрощує використання окремого функціонал бібліотеки мови Duct та покращує потенційну швидкість роботи для забезпечення мінімально-допустимого рівня. Додатковою перевагою підходу є простота кожного окремого компонента мови та його взаємозамінності у майбутньому.

Опис особливостей Віртуальної машини, аргументація вибору технічних аспектів машини. Віртуальна машина Duct є виключно-стековою віртуальною машиною без окремого адресного простору.

Стекова ВМ замість реєстрової. Зазвичай процесні віртуальні машини діляться на дві категорії: Реєстрові та Стекові.

Реєстрова віртуальна машина побудована на використанні регістрів - змінних розміром зававичай від 32 до 64 біт для яких зарезервовано окремий простір пам'яті. В контексті саме віртуальної машини перевагами регістрів є їх потенційна незалежність при виконанні інструкцій що надає змогу процессору виконувати до

кількох інструкцій одночасно завдяки “планувальнику ЦП” (CPU Scheduler), також реєстровий байт-код потребує меншу кількість інструкцій для виконання простих дій через можливість закодовування кількох операндів (значень) до однієї інструкції. Недоліком регістрів є потреба у створенні алгоритмів що могли б провести генерацію коду щоб корректно розпоряджався ними під час компіляції коду до інструкцій, адже регістри є обмеженим ресурсом який потрібно весь час пере-використовувати. Такі алгоритми не є складними самі по собі але при врахуванні всіх функцій віртуального комп’ютера: циклів та бранчування, стану бінарних операцій, роздільність типів Integer та Float (IFFF-754) то ефективе використання цих алгоритмів без створення помилкової логіки є складною задачею до вирішення і іноді є непридатним вибором для деяких парадигм мов прогармування.

Стекова віртуальна машина [29] є простішою в реалізації адже не потребує складних перетворень для наївного генерування інструкцій розрахунку, через особливість стеку що робить його зображенням графу протягом часу - результат розгортки дерева структури коду може бути необхідним мінімумом для генерації робочої програми. Також стекова віртуальна машина є простішою у аналізі через локальність усієї інформації на ній, у порівнянні перегляд усіх реєстрів на кожному окремому етапі виконання програми є складнішим для людини процессом. Недоліком стекової машини є відсутність переваги паралелізації виконуваних інструкцій через потребу у виконанні усіх дій строго у лінійному порядку для притримання правил стеку.

У реальному комп’ютері, завжди, застосовується змішаний підхід [27]. Через повільну роботу стеку для простих операцій дизайнери архітектур ЦП надаються перевагу регістрам і їх відповідних інструкцій через переваги паралелізму та економії місця інструкцій програми, коли для загального використання і організації даних та резерву пам’яті для змінних використовується стек. У віртуальних машинах надається перевага одному стилю через потребу у спрощені розрахункових операцій до одного типу мінімально швидких але коротких у виконанні дій.

Для BM Duct було обрано стековий дизайн через простоту реалізації такого дизайну і простоту роботи з ним при проведенні розрахункових операцій. Завдяки особливості структури даних дерев що забезпечує послідовність генерації інструкцій у “Глибина є першою” (Depth-First Order) порядку, створення інструкцій є операцією що може бути виконана за один прохід без потреби у використанні додаткової пам’яті чи систем відслідковування стану генеруємої програми.

Напрям зростання стеку. У стекових системах також має значення напрям зростання стеку, адже він може зростати як з нижніх адрес до верхніх, так і навпаки - з верхніх до нижніх[28]. Ця особливість існує для забезпечення простоти доступу до значення під вказівником до значень на стеку під час виконання операцій. Наприклад архітектурах ЦП “Intel Microprocessor 8008 - 8068”, ЦП що став основою x86 архітектури, головною причиною вибору низу як напряму зросту була “простота зображення стеку для передньої панелі” [30]. Від напряму зростання стеку характеристики швидкості роботи чи надійності не сильно змінюються і цей фактор є більше історичною особливістю комп’ютерів аніж якась певна перевага такого методу зображення стеку. Натомість у BM Duct причиною вибору стеку що зростає вверх є залежність напряму даних у пам’яті для програм, адже програмне забезпечення зазвичай зображає стек як структур даних що зростає вверх від початкового вказівника. Для потенційної сумісності з системами що очікують певний напрям стеку (такі програми як дизасемблери чи відладчики) мали пряму сумісність з пам'яттю стеку BM.

Технічна специфікація розробленого рішення. Інструментарій та функціонал віртуальної машини. Наведено розроблену віртуальну машину, синтаксис її мови асемблера та функціонал відладчика та інші технічні аспекти.

Специфікація віртуальної машини.

- *Структура стану BM.* Стан BM включає в себе: вказівник наверх стеку для даних операцій та вказівник наверх даних змінних, вони є окремими адже один стек використовується для виконання розрахункових операцій а інший збереження даних змінних чи для динамічної роботи з пам'яттю, додатково їх розмежування прибирає

можливість створення складних для відлаштунку ситуацій. ВМ для доступу до зовнішнього світу чи даних у програмі використовує файлові дескриптори (fd - “File Descriptor”), до списку FD які має машина входить: Файл логування, Файл відладки для ассемблера, Файл відладки стану, Файл відладки написання (до “Стандартного виходу” - stdout). Також віртуальна машина має список доступних до виконання функцій та глобальних змінних що були прочитані та можливо модифіковані під час виконання скрипту. На момент створення прототипу стек має обмежений розмір, але у майбутньому він має динамічно зростати в залежності від потреб виконуваної програми чи скрипту.

C

```
. . .
typedef struct {
    int             options_mirror; // mirror of options from interpreter.
    Arena           allocator_global;
    ObjectList      varlist_global;
    Functions       functions;
    size_t          top;
    size_t          work_sp, data_sp;
    Object          returned_object;
    Object          work_stack[DTVM_STACK_CAPACITY];
    Object          data_stack[DTVM_STACK_CAPACITY];
    FILE            *logf, *asmtracef, *tracef, *writef;
} Dtvm;
. . .
```

- *Інструкції.* На момент створення першого робочого прототипу без підтримки глобальних чи локальних об’єктів віртуальна машина має 30 інструкцій. Вона має функціонал проведення базовий функціонал арифметики та простих бінарних операцій (AND, OR, NOT), порівняння значень на стеку, опційного та безопційного скачку та інструкцію запису до FD виводу інформації. Також віртуальна машина здатна виконувати виклики функцій через інструкцію CALL та повернення значень з нинішнього кадру через RET. У машини є підтримка базових інструкцій для операцій над об’єктами на стеку:

PUSH, DUP, ROT, POP. Присутні операції отримання чи запису даних з об'єму пам’яті через інструкції: LOAD, STORE.

C

```
. . .
enum {
    DTI_NULL, // probably error
    DTI_NOP, DTI_STORE,DTI_LOAD,
    // Procedure instructions
    DTI_CALL, DTI_RET,
    // Stack mutation.
    DTI_PUSH, DTI_DUP, DTI_POP, DTI_ROT,
    DTI_ADD, DTI_SUB, DTI_DIV, DTI_MUL,
    DTI_AND, DTI_OR, DTI_NOT,
    DTI_EQ,     // ==
    DTI_LT,     // <
    DTI_GT,     // >
    DTI_LTE,    // <=
    DTI_GTE,    // >=
    // WORK ON LITERALS:
    DTI_LEQ,     // ==
    DTI_LLT,     // <
    DTI_LGT,     // >
    DTI_LLTE,    // <=
    DTI_LGTE,    // >=
    DTI_IJIF,   // INCREMENTAL JUMP IF
    DTI_JMP,
    DTI_WRT,    // writes top of the stack to vm.fstdout
    DTI_COUNT,
} EInstruction;
. . .
```

- *Інкапсуляція та пакування функцій у самостійні блоки пам’яті.* Унікальним рішенням для забезпечення модульності скриптів є інкапсуляція даних та операцій у функціях. Замість класичного рішення - коли програма (чи скрипт) є загальною одиницею яка проходить процес завантаження та роботи, у мові BM Duct такою одиницею э функція (та глобальний “Об’єкт”). Функція під час збірки з мови асемблера виконує підрахунок локальних змінних та розміру констант, разом з кількістю і розміром інструкцій та їх операндів, далі уся інформація пов'язана з інструкціями та даними у функції тісно пакується у одному лінійному блоці пам’яті і проводиться процес

індексації у заголовку. Завдяки пакуванню кожна функція може бути закешована чи збережена для повторного використання, має перевагу локальності даних що покращує швидкість читання чи запис через кеш ЦП та обмежує потенційні вразливості пов’язані з доступом до даних функції на рівні програмного забезпечення. Також через монолітність функцій їх можливо “експортувати” чи “імпортувати” через різні джерела для максимальної ефективності використання пам’яті при роботі скрипта всередині програмного забезпечення.

- *Типи даних.* Віртуальна машина Duct використовує “Об’єкт” як основну для операцій на стеці, об’єкт може зображати будь-яку структура даних над якою у мові програмування необхідно проводити операції. До типів даних які можуть бути зображені у об’єкті належать: NULL - помилкове значення, BOOL, BYTE, CHAR, INT, FLOAT, STRING, ARRAY, TYPE, COMPLEX, RESULT. COMPLEX - є зображенням списку та об’єкту адже вони мало відрізняються один від одного, STRING та ARRAY є однією і тією є структурою даних але STRING має один підтримуваний тип даних на відміну від масиву - CHAR. CHAR та BYTE не відрізняються нічим окрім того що CHAR зображується виключно у контексті символу.

Функціонал збірника (“ассемблера”). Для додаткової перевірки на наявність можливих помилок під час генерації коду і для забезпечення індивідуальності та незалежності віртуальної машини чи платформи - існують ассемблери. Ассемблер (Assembler) - це транспілятор (Transpiler). Вид компілятора, що виконує мінімальну кількість повторень та перевірок, який здебільшого потрібен тільки перекладу інформації з однієї форми на іншу. Ассемблер DUCT має простий синтаксис для мінімізації кількості коду що потрібно написати для реалізації ассемблера. Синтаксис мови ассемблера складається з тверджень які займають одну лінію (строку), кожне твердження може бути або інструкцією, або міткою, або блоком функції, або коментарем.

Приклад синтаксису мови ассемблера.

C

```
. . .
fn increment_endlessly n
	@loop0
	load n
	push 1
	add
	store n
	jmp loop0
endfn
. . .
```

У мові програмування DUCT ассемблер є функціоналом бібліотеки а не окремою програмою, хоча за потреби створення такої програми не є проблемою, бо обмеження одно-пасового компілятора можуть зменшувати потенціал віртуальної машини, тому розробнику надається свобода до написання власного байт коду для машини всередині свого програмного забезпечення користуючись бібліотекою DUCT. Цей функціонал надано через функцію в модулі VM:

C

```
. . .
LineResult dtvm_compile_instruction
	(Arena* allocator, DtSymbolMap* symbols,  String line, int* status);
. . .
```

Функціонал розбірника (“дизассемблера”). Розбірник (Disassembler) часто потрібен для розшифрування зібраних для віртуальної машини програм для відладки та перевірки генераторів інструкцій при їх розробці, визначення джерел проблем у зібраному програмному забезпеченні, знаходження повільного коду чи інших причин які потребуватимуть інтерпретацію уже зібраних інструкцій для прочитання. Зазвичай такий інструмент розроблено окремо від проекту і не надається з інструментарієм мови, наприклад C# та Lua не мають офіційного дизассемблера і потребуються використання сторонього проекту який його реалізує. DUCT надає функцію розбірки байт коду до читабельного формату у модулі VM:

C

```
. . .
const char* dtvm_decompile_instruction(Instruction i);
. . .
```

2.4. Розробка компілятора з метою збірки коду для віртуального середовища.

Причини вибору однокрокового компілятора. Непопулярним та маловідомим видом збірки у компіляторах є однокроковий прохід. У такому випадку замість використання структури даних дерева що виключно формуються під час процесу лексичного аналізу для подальшої збірки інструкцій, програма будується нальоту під час лексичного аналізу тексту, що означає одночасний лексичний аналіз та генерацію коду. Обидва види компіляторів є класичним у розробці програмного забезпечення обміном у швидкості роботи чи складності системи на збереження у пам’ять інформації, її кешування.

Цей метод не використовується в системних компільованих мовах програмування через коротку пам’ять алгоритму про інформацію у програмі що зазвичай аналізується для прийняття рішень для вибору стратегії оптимізації. Незважаючи на ресурсоефективність такого компілятора через відсутність у необхідності тримати дерево у пам’яті, недоліки швидкості роботи програм створених таким компілятором у багатьох випадках переважають позитивні аспект методу. Такий вид компілятора є одночасно простим та складним до реалізації в залежності від контексту використання. У простому випадку, створення першої мови програмування яка генерує будь який робочий результат, такий метод є куди простішим через відсутність потреби у побудові дерева, його аналізу, збереження його інформації та її організації у пам’яті, тощо. У складному випадку при наявності оптимізацій і потребі створювати швидкі програми з нелінійними залежностями напряму коду чи інформації (Code motion, Data flow - graphs) виникають обмеження по структурі синтаксису мови для якої розробляється компілятор та складнощі у багатовимірному аналізі та зберігання необхідного контексту програми.

Для мови DUCT однокроковий компілятор є доцільним вибором через простоту доповнення методики компіляціями мінімальною кількістю “очевидних” оптимзіацій до уже простішої робочої системи збору. Завдяки особливості однокрокової збірки у швидкості роботи та відсутності багатовимірної структури даних, такий вид компіляції є оптимальним для забезпечення простоти інтеграції у

програмному забезпеченні що має працювати на девайсах з обмеженими ресурсами пам’яті або коли скрипти запускатимуть в багатьох сутностях віртуальної машини одночасно.

Лексичний аналіз тексту програми. Лексичний аналіз (або альтернативно імуеєма - Токенізація) - це назва процесу що відбувається у компіляторі чи інтерпретаторі для спрощення роботи з текстовими об’єктами (словами, числами, пунктуацією, тощо) та їх перетворення до Токенів (альтернативна назва - Лексеми) що містять у собі всю, важливу до процессу аналіза синтаксису програми, інформацію. Зазвичай стан та уся інформація що відноситься до аналізуємо програми чи тексту знаходиться у структурі даних Токенайзера (Лексера), до цього стану може входить: кількість строк коду, нинішній рядок і стовпчик, загальний текст програми та текст програми що залишився до обробки алгоритмом, інформація про нинішню лексему та історія минулих лексем. Задача токенайзера це виключно групування символів та надання швидкої та простої інформації для аналізу у наступному кроці компілятора - Парсері (Parser). Процес токенізації може набувати різних форм і тому важливо уточнити як саме цей процес відбувається в розробленій мові.

В мові DUCT токенайзер є структурою даних та алгоритмом, такий вибір немає конкретної аргументації переваг чи недоліків. Токенайзер зберігає інформацію про текст який він аналізує та розмір тексту, разом з позицією у тексті. До стану токенайзера належить: тимчасовий буфер для роботи з токенами, параметр який вказує токенайзеру чи потрібно пропускати кінець лінії при розбитті на токени чи потрібно повертати цей символ як унікальний токен, інформацію про стовпчик та рядок, список унікальних лексем у мові які не входять до стандартного списку токенайзера.

Структура токенайзера у коді:

C

```
. . .
typedef struct {
    char*        scratch_buffer;
    size_t        scratch_buffer_sz;
```

```
    // config or user options
    bool                skip_newline;
    symbol_t             string_quotes[2];
    const char*         comment_block[2];
    // line tracking
    size_t              row;
    size_t              col;
    // string info
    const char*         target;
    size_t                target_length;
    size_t                 position;
    // user input table
    TokenTableEntry*    token_table;
    size_t              token_table_count;
} Tokenizer;
. . .
```

Головна функція Токенайзера в кодовій базі DUCT це надання прочитанного токену з тексту та продвинення позиції у тексті до позиції після лексеми що було знайдено. Токенайзер не вносить змін до вказівника на строку для збереження повного тексту і використовує окрему змінну для збереження позиції. *Функція яка виконує прочитання токену має таку сигнатуру у коді:*

```
. . .
Token Tokenizer_next_token(Tokenizer* t);
. . .
```

Рекурсивно-поглиблений парсер. Парсер (Parser) - це друга структура даних яка тісно ассоціюється з Токенайзером, але замість групування символів строки у легкі для обробки строки, Parser виконує аналіз токенів і подальшу обробку токенів для відображення структури програми або її безпосереднього створення . Незважаючи на їх тісне використання та роль у процесі аналізу тексту комп'ютерних програм обидві структури даних та алгоритма виконують різні задачі, саму тому їх розрізняються як два окремі кроки одного процесу. Парсери бувають різноманітні тому варто зазначити їх основні класифікації:

- *Top-Down Parser.* "Сверху вниз" є видом парсування що виконує аналіз програми відштовхуючись від ієрархії у структурі програмі де початком є найбільша одиниця - вся програма, такий парсер *"Спускається вниз"* до менших в ієрархії елементів чим в кінцевому

результаті стають прості значення чи дії які перетворюватимуться далі до інструкція для виконання у наступному кроці. Такий парсер існує у формах:

- *Recursive descent parser.* Парсер що використовує праву рекурсію для аналізу менших у ієрархії елементів синтаксису, зазвичай такий парсер відображається у коді я набір функцій що “аналізують” один вид конструкцій у синтаксисі і викликають всередині себе інші подібні функції, таким чином інформації у такому парсері зберігається на стеці програми. Такий парсер виконує постійні “повернення” до попередніх місць виконання аналізу завдяки свої рекурсивній властивості.
- *Non-recursive parser. LL(1) parser.* Це форма аналізу яка не потребує рекурсії і будується на вимозі до синтаксису мови що дозволяє точно визначити наступний крок у аналізі. Такий парсер замість стеку програми використовує або стек стану або таблицю для вибору наступного кроку. Такий парсер виконує тільки пряме просування при побудові графу чи аналізі і не потребує повернення.

- *Bottom-Up Parser.* “Снизу вверх” це вид парсування який є повною протилежністю до “Сверху вниз”, замість аналізу починаючи з найбільшого синтаксичного об’єкту у мові - програми, такий парсер будує програму починаючи з найпростіших символів у мові і далі группує їх з сусідніми символами допоки уся програма не буде прочитана. Такий парсер ділиться на два основні види:
  - *LR парсери.* Вид парсерів які виконують аналіз з ліва на право шукаючи правосторонню послідовність подібно до LL(1) парсеру тільки починаючи з найменших лексем мови.
  - *Precedence Parser.* Вид парсеру який групує символи за операторами а не за простішими лексемами. Такий оператор

групує пари об’єктів і тому часто використовується в мовах що побудовані на виразах подібних до математичних.

У мові DUCT використано Recursive Descent Parser адже він є одним з простішим парсерів вказаних у списку, він є досить швидким для етапу прототипування мови під час якого може відбуватись багато змін до синтаксису мови, та має непогану швидкість роботи що в більшості випадків ніколи не виступає обмежувальних фактором у порівнянні з генерацією коду чи процессом інтерпретації. Парсер мови DUCT складається з стану який включає в себе: аллокатор для пам’яті що потрібна для створення тимчасових об’єктів потрібних для парсування, буфер токенів, інформацію про унікальні символи у коді та рахівники токенів.

Кодова структура парсера розробленої мови:

C

```
. . .
typedef struct {
    int             options_mirror; // mirror of options from interpreter.
    StringBuilder   output;
    FILE*           tracef; // for tracing AST parsing in recursion.
    FILE*           errorf; // for logging errors from compilation.
    size_t      current;
    size_t      requested_tokens,
                parsed_tokens;
    // we buffer PTBS(size) of tokens to be able
    // to lookup tokens before and after the current one
    Tokenizer   lexer;
    Token       current_token;
    Token       token_buffer
                    [PARSER_TOKEN_BUFFER_SIZE];
    DtParserFunctions   function_symbols;
    Arena               allocator;
} DtParser;
. . .
```

В парсері DUCT використано технологію буферизації токенів, що відрізняється від класичного методу роботи з токенами. Зазвичай токени отримуються у структуру даних Queue (“Черга”) в яку токенайзер зчитує токени - що рухає загальну позицію у тексті, а при завершенні обробки токени “використовуються” що при кожному використання операції прибирає перший токен у “Черзі”. У методи буферизації використовується три змінні для рахування: кількості прочитаних токенів,

кількості токенів що були запитані парсером від токенайзера иа нинішній токен. При запиті токена змінна нинішньої кількість збільшується на один, якщо кількість за межами прочитаної запитаної кількості у токенайзера - парсер виконує процес заповнення з місця у “Черзі” розміром у половину об’єм черги через запит цієї кількості токенів у токенайзера. Якщо цей запит перейшов межу тексту, цей та подальші токени від запитів є нульовими (кінцевими). Таким чином на любому етапі парсування у буфері “Черги” присутні дані про наступні токени та історії попередніх токенів, це є не тільки ефективним методом роботи з токенами та їх історією але і ефективним методом роботи з інформацією при аналізі адже запити для прочитання чи запису кількох елементів є швидшими для кешу ЦП ніж роботи з поодинокими об’єктами. Головною перевагою цього методу є можливість на любому етапі парсування прочитати кілька токенів наперед що дає змогу використовувати складний синтаксис що потребує громісткого за об’ємом контексту, також зробити це за один виклик функції замість використання двох і більше функцій у випадку з моделлю “Запиту та використання”.

У парсері DUCT функції що відповідають за отримання нинішнього токену і оперування буфером токенів написані так:

C

```
. . .
void dtp_load_tokens_buffer(DtParser* p) {
    Tokenizer* lexer = &(p->lexer);
    // Buffer tokens in batches of set size
    // so that its easy to check previous
    if (p->parsed_tokens == 0) {
        hc_loop(i, PARSER_TOKEN_BUFFER_SIZE) {
            p->token_buffer[p->parsed_tokens++] =
                Tokenizer_next_token(lexer);
        }
    } else if (p->requested_tokens + PARSER_LOOK_AHEAD_COUNT > p->parsed_tokens) {
        hc_loop(i,PARSER_LOOK_AHEAD_COUNT)
            p->token_buffer[(p->parsed_tokens++) % PARSER_TOKEN_BUFFER_SIZE] =
                Tokenizer_next_token(lexer);
    }
}
Token dtp_step(DtParser* p) {
    dtp_load_tokens_buffer(p);
    p->current = p->requested_tokens;
```

```
    Token tk = p->token_buffer[(p->requested_tokens++) % PARSER_TOKEN_BUFFER_SIZE];
    p->current_token = tk;
    return tk;
}
. . .
```

Процесс структурованого аналізу у DUCT відбувається через набір функцій якій які виконуються парсування відповідно структури мови. Зазвичай залежність цих функцій зображують за допомогою спеціального формату BNF (Backus-Naur Form) або EBNF (Extended BNF) який описує граф синтаксису мови набором правил перетворень. *Для кращої візуалізації буде надано текстову структуру мови з використанням EBNF нотації яку зображує логіку парсера створеної мови:*

EBNF

```
program   = { function } ;
function  = identifier "(" [ params ] ")" block ;
params    = identifier { "," identifier } ;
block     = "{" { stmt } "}" ;
stmt      = assign
          | "return" expr [ ";" ]
          | "while" expr block
          | "if" expr block
              { "else" "if" expr block }
              [ "else" block ]
          | expr [ ";" ] ;
assign    = identifier "=" expr ;
expr      = cmp ;
cmp       = add { ("=="|"!="|"<"|">"|"<="|">=") add } ;
add       = mul { ("+"|"-") mul } ;
mul       = unary { ("*"|"/") unary } ;
unary     = { "-"|"!" } primary ;
primary   = number
          | string
          | boolean
          | identifier
          | call
          | list
          | array
          | set
          | "(" expr ")" ;

call      = identifier "(" [ args ] ")" ;
args      = expr { "," expr } ;
list      = "$[" [ args ] "]" ;
array     = "[" [ args ] "]" ;
```

```
set       = "{"
              [ identifier "=" expr
                { "," identifier "=" expr } ]
            "}" ;
boolean   = "true" | "false" ;
number    = integer | float ;
integer   = digit { digit } ;
float     = digit { digit } "." digit { digit } ;
string    = "\"" { character } "\"" ;
identifier = letter { letter | digit | "_" } ;
letter    = "A"…"Z" | "a"…"z" ;
digit     = "0"…"9" ;
character = ? printable except " ? ;
```

Для відслідковування помилок та надання оптимізацій у майбутньому парсер DUCT використовує систему “результатів” через яку передається інформація про попередньо згенеровані ісінтрукції. У випадку помилки персер може почати процес повернення до початково стану з кодом помилки що було зустріто, у випадку роботи з атомічними (простими) значенням, чи блоками коду що генерується, парсер може надавати додаткові інстуркції до генератора коду для видалення попередньо-згенерованого блоку якщо він не буде використовуватись чи у випадках коли дії саморуйнівні (такі як проведення у мінус негативного значення) і їх кількість чи присутність не грають ролі.

Генерація байт-коду для віртуальної машини. Генерація коду відбувається одночасно до процесу аналізу та токенізації. Генератор коду записує інструкції в поле у стані парсера яке надалі буде надано ассемблеру. Генерація інструкцій у текстовому вигляді дозволяє виконати не тільки перевірку компілятора а і справність роботи ассемблера і пакувальника для функцій мови, завдяки цій унікальній у проекті ускладненості можливо виконувати перевірку одночасно кількох систем для забезпечення кращої вірогідності зустріти та виправити проблеми та недоліки під час активної розробки. За генерацію коду у компіляторі відповідає набір макросів та функцій що використовують функціонал будівника строки `sb_append` разом з функціоналом системи стану парсування. При генерації інструкції результат генерації коду у середині конкретного елементу аналізу

генератор `dtp_emmit_asm` змінює стан результату генерації. У випадку отримання цього результату з іншого, глибшого, рівня рекурсії він перевіряється, і у випадку помилки повертає її одразу, таким чином при виникненні помилки під час аналізу чи генерації коду для скрипту парсер одразу відступає до самого початку де користувач може отримати помилку та місце де відбулася проблема. Парсер не виконує багаторівневий аналіз і не збирає більше однієї помилки, це зроблено для зменшення шуму під час зустрічі з ціпкою залежних помилок та для покращення циклу ітерації під час розробки скрипту. Функціонал парсера який виконує перевірку аналізованого тексту знаходиться у макросах з назвами що починаються `dtp_expect_` - ці макроси використовують механізм перевірки токена відповідно структури мови і при зустрічі з проблемними токенами створюють код помилки і архівують місце де вона відбулась, таки чином можливо отримати повідомлення помилки з додатковою інформацією про контекст її виникнення.

Набір макросів для повідомлення про помилки.

C

```
. . .
DtParseResult dtp_error_token(FILE* out, Token t, const char* message) {
    dtp_error_message(out, message, t.row, t.col);
    return dtp_error();
}
#define dtp_expect_str(P,T,S,E) (!dtp_match_str(T, S)) ? \
    dtp_error_token((P)->errorf, T, E) : dtp_ok();
#define dtp_expect_kind(P,T,K,E) if (!dtp_match_kind(T, K)) {\
    assert(0);\
    dtp_error_token((P)->errorf, T, E);\
    return dtp_error();\
}
#define dtp_expect_sym(P,T,S,E) if (!dtp_match_sym(T, S)) {\
    assert(0);\
    dtp_error_token((P)->errorf, T, E);\
    return dtp_error();\
}
. . .
```

Тимчасово для відладки під час розробки макроси репортування помилок припиняють роботу усієї програми через `assert` але це не є функціональним

рішенням мови, адже система надаватиме можливість спробувати зібрати один і той же чи багато різних скриптів без припинення роботи.

Генерація виразів. Вирази зображені у синтаксисі мови при розгортанні та “сплющені” є прямим відображенням стекових дій які потрібно виконати для отримання результату. При проходження рекурсивного шляху під час токенізації методом рекурсії Depth-First результат згенерованого коду не відрізняється від проходження існуючого дерева адже шлях пройдений генерації такого дерева прямо-відповідний до і після створення структури даних, тому і не присутня потреба будувати дерево для їх генерації.

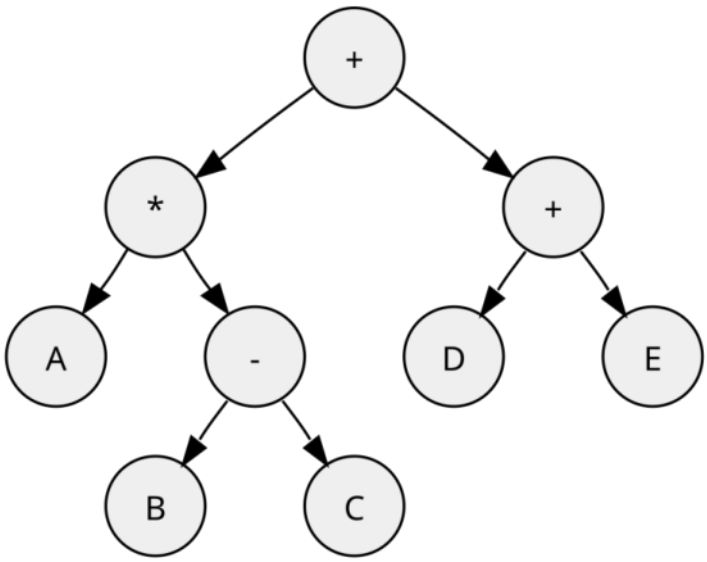

*Рисунок 2.4.1 При проході Depth-First зліва на право при генерації інструкцій отримаємо вираз: A B C - * D E + +, який дійсно є вірною послідовністю дій для стекового розрахунку.*

Генерація бранчування. Бранчування може відбуватись по різному, для створення простішого бранчування у ВМ мови DUCT було використано концепцію “відносних адрес”. Замість виконання переходу через інструкцію `JMP` (JUMP) яка б виконувала перехід до конкретної адреси у програмі, у ВМ DUCT відповідна інструкція виконує перехід відносно нинішнього місця у коді, що дозволяє виконувати перевірку на значення у гілках та виконувати перехід на кількість інструкцій що знаходяться в під гілкою якщо перевіка не відбулась успішно. Таке

“перестрибування” дозволяє створювати легкий до прочитання людиною та лінійного виконання комп’ютером програмний байт-код. У випадках коли гілка успішно виконала перевірку та пройша виконання всіх інструкцій, кінцевою інструкцією виступає безумовний перехід до кінця блоку, який завжди генерується у кінці крім випадків коли гілка тільки одна. У ВМ назва інструкції для відносного переходу є `IJIF` (Incremental Jump IF).

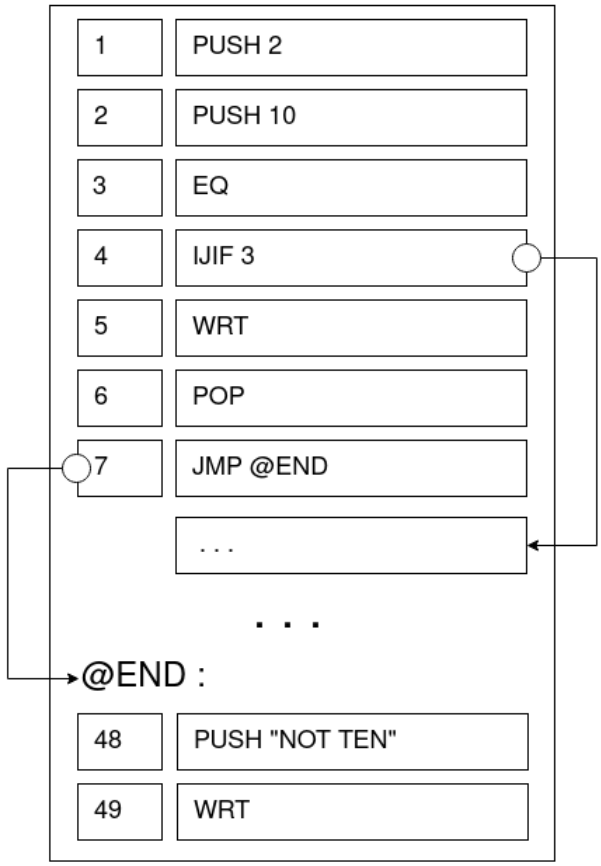

Рисунок 2.4.2 - Структура (її приклад) стекової операції порівняння для розгілкування виконуваної логіки програмою.

Генерація циклів. Генерація циклів відбувається з використанням того-ж механізму що і бранчування, але замість адреси стрибку гілок у кінці блоку що веде до позиції після блоку, цей стрибок відбуватиметься до початку, що створює цикл, якій тільки припиниться коли вираз який перевіряється у циклі не підійде кінця.

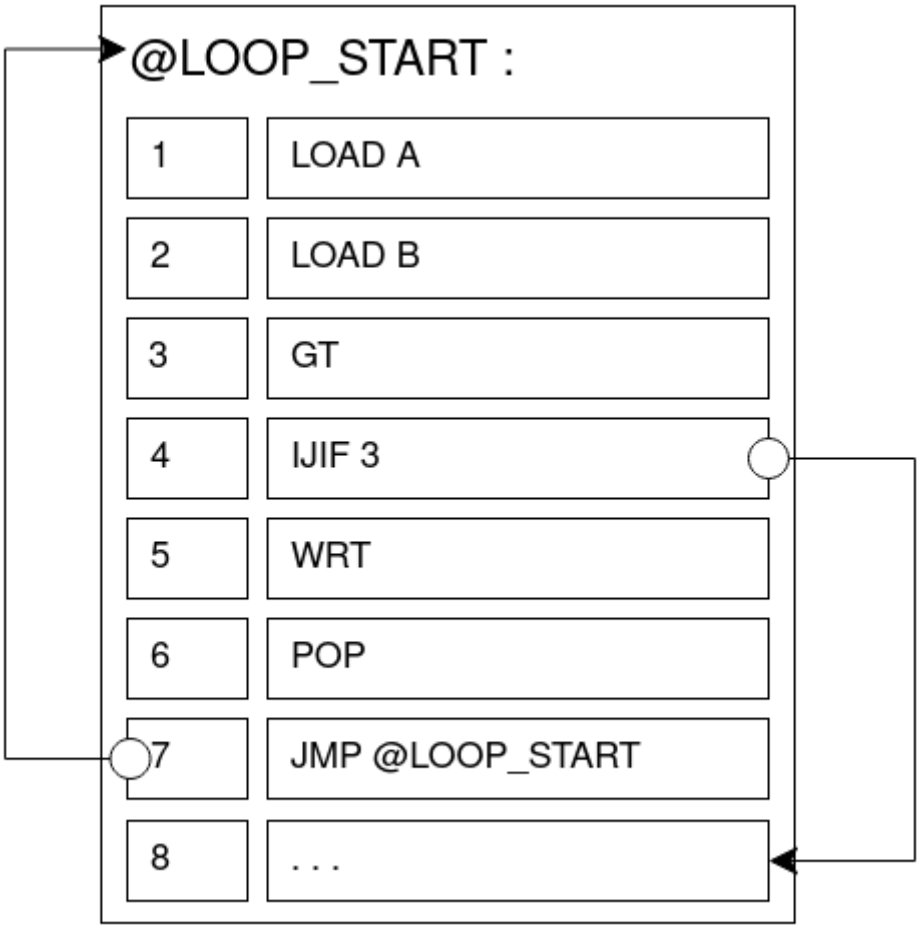

Рисунок 2.4.3 - Структура (приклад) стекової операції аргументного стрибку що реалізує параметричний цикл.

РОЗДІЛ 3. ДЕМОНСТРАЦІЯ ІНСТРУМЕНТАРІЮ МОВИ, ВИЯВЛЕННЯ ПЕРЕВАГ ТА НЕДОЛІКІВ У РОЗРОБЕРНОМ РІШЕННІ, ПЕРСПЕКТИВА РОЗВИТКУ.

3.1. **Результат розробленої розрахункової моделі мови та приклад її роботи.**

Опис розробленого рішення. В рамках зазначених вимог був розроблений прототип мови програмування [33] що відповідає вимогам зазначеними у Розділ 2.1. Код мови програмування та її інструментарію поділений на два основних модулі - компілятор та віртуальна машина. Програмний код який використовується в обох модулях знаходиться під третім модулем з назвою “Спільне” (Shared), цей модуль не розглядається як окремий модуль адже його наповнення може бути дубльовано для кожного окремого модуля і при збірці отримати ідентичний результат але для уникнення дублювання коду це рішення не застосовувалося. Проект займає 4389 строк коду суммарно в обох модулях (не рахуючи спільний модуль), спільний модуль включає в себе код усіх структур даних, що були розроблені в рамках цього і кількох інших проектів, і знаходиться в одному файлі. Усі структури даних займають 1768 строк коду, увесь спільний код займає загалом 1917 строк коду. Увесь проект має 6306 строк коду.

В кожному модулі присутній файл “опису” (declarations.h) що має у собі усі структури даних використані кожним файлом у модулі, також деякі з функцій описані наперед з метою спрощення в документуванні їх використання. Усередині модулю компілятор розміщена уся логіка пов’язана з збиранням програми, в ньому реалізований одно-пасовий аналізатор (Парсер), Токенайзер та генератор інструкцій у відповідності до опису у Розділ 2. В модулі ВМ (vm) знаходиться такий ще файл `declarations.h` в якому вже описуються структури даних та API для роботи з Віртуальною машиною. Віртуальна машина має стан через який проводиться розрахункові операції, і з якого витягується інформація про їх результат. Також модуль ВМ має у собі функціонал ассемблера для байт-коду віртуальної машини та дизассемблера для відновлення байт-коду інструкцій до читабельного людиною тексту. Інтерпретатор мови для інтерактивної взаємодії не було реалізовано через обмеження у часі, тому його функції виконує файл що і є

основним демонстаційним об’єктом роботи - `dt.c`. Він є прикладом використання API компілятора та машини для взаємодії з мовою та перевірок її можливостей. Проект структуровано саме так для можливості об’єднання всіх файлів в один для спрощення розповсюдження бібліотеки мови, проект будується користуючись методикою “Цілісної побудови” (Unity Build) так як цей метод характеризується високою швидкістю побудови на сучасних комп’ютерах і є доцільним для цього проекту через його не великий розмір. При потребі можливість “покрокової побудови” (Incremental Build) може бути надана шляхом побудови кожного файлу окерма вручну і подальше їх злиття через лінкер (Linker) мови C.

Слід зазначити що цей прототип є незакінченим і не реалізує багато згаданих у Розділ 2 систем. На момент написання роботи мова є функціональною для роботи з натуральними та раціональними числами (Integer, Float) має підтримку бранчування та циклів і здатна працювати з змінними, виразами та функціями. В мові не закінчено чи не реалізовано:

- Глобальний список/стіл змінних.
- Сети, Списки, Масиви - комплексні структури даних.
- Усі оператори окрім `print`.
- Підтримка інтеграції C функцій.

Варто зазначити що проект незважаючи на свій обмежений функціонал може бути використаний для розрахункових операцій у безпечному від частих вразливостей середовищі та надати необхідні функції простого математичного інтерпретатора.

Демонстрація вітальної програми. Першою програмою яку зазвичай створюють при ознайомлені з мовою це тестова програма виведення повідомлення - “Hello world”, така програма традиційно використовується це виводить у консоль привітання “Привіт світ”. Такай программа зазвичай використовується для перевірки інструментарію мови (компілятора, інтерпретатора, ассемблера, тощо) на справну роботу. В мові DUCT така програма має просту структуру і може бути написана в одну строчку коду.

```
  [~/c/c/duct] › ./ducti 'main() {print("Hello World")}' --quiet
Hello World
  [~/c/c/duct] ›
```

Рисунок 3.1.1 - Приклад програми “Привіт Світ” на мові Duct.

Після запуску програми спостерігається при запуску скрипту через прототип інтерпретатора - програма успішно виводить повідомлення. На рівні мови ассемблера для віртуальної машини мови використовуються спеціальні інструкція - WRT (Write), через надане джерело виводу тексту ця інструкція виконує серіалізацію тексту та виписує створений текст інформації. Ця програма має тільки 3 строчки коду ассемблера так як не виконує складних обчислень.

```
  [~/c/c/duct] › ./ducti 'main() {print("Hello World")}'

### COMPILING ASSEMBLY ###
# Emmiting bytecode for 'write':
000000  load v@0
000001  wrt
000002  pop
        ARGS: [ v ]
> ROOT:
 > function:main
  > block:
   > statement:
    > fncall 'print'
     > expr
            > value: "Hello World"
GATHERED ASSEMBLY: ----
fn main
        push "Hello World"
        wrt
        pop
endfn

----------

### COMPILING ASSEMBLY ###
# Emmiting bytecode for 'main':
000000  push s(Hello World)
000001  wrt
000002  pop
        ARGS: [ ]
        Data stack pointer : 0
Hello World
  [~/c/c/duct] ›
```

Рисунок 3.1.2 - Вихідна інформація інтерпретатора про збірку скрипту та проаналізований синтаксис.

Демонстрація програм з бранчуванням. У більшості програм використовуються гілки для розмежування логіки від порівняних значень, у мові

DUCT для цього присутня тільки одна конструкція бранчування - IF блоки. Як було описано в Розділ 2. DUCT створює інструкції інкрементних стрибків в залежності від виконання умови, якщо хоча б одна умова виконується, то усі інструкції у ній виконуються до моменту зустрічі з кінцевою інструкцією блоку - стрибка за межі всіх асоційованих IF блоків.

Наступна програма написана на мові DUCT демонструє роботу інтерпретатора з скриптом що визначає належність значення до якої з трьох гілок належить задане число.

```
  [~/c/c/duct] › cat examples/branching.dt
main() {
    number = 10
    print("Number is: ", number, "\n", "")
    if number > 10 {
        print("number is bigger than 10")
    } else if number == 10 {
        print("number is equal to 10")
    } else {
        print("number is less than 10")
    }
}
  [~/c/c/duct] › cat examples/branching.dt | ./ducti --quiet
Number is: 10
number is equal to 10
  [~/c/c/duct] ›
```

Рисунок 3.1.3 - Демонстрація програми з “гілками”.

У згенеровані інструкція компілятором структура відповідає опису в *Розділ 2.4* та *Рисунок 2.4.2* що описує логіку при створенні бранчування. Через відсутність оптимізацій та одно-крокову компіляцію згенеровані інструкцій мови ассемблера відповідають повністю до написаного коду і є прямолінійним (“наївними”) адже компілятор не виконує оптимізацій пов’язаних з перестановкою, виключенням, об’єднанням чи перевикористанням значень. Перевагою підходу є простота у перевірці кожної інструкції що покращує потенційний аналіз програм з метою забезпеченням безпеки, але недоліком підходу є проблема використання багатьох

гілок та циклів може призвести до неконтрольованого збільшення об’єму необхідних до виконання інструкцій.

Демонстрація програм з циклами. У мові DUCT присутній тільки один метод лінійного циклу - WHILE блок. Наведено програму яка виконує цикл та демонструє результат своєї роботи - програма що виконує задану кількість ітерацій та виводить значення ітератора пока цикл продовжує свою роботу.

```
  [~/c/c/duct] › cat ./examples/loop.dt
main() {
    i = 0; n = 10
    print("Printing numbers from ", i, " to ", n - 1, "\n", "")
    while i < n {
        print("i = ", i, "\n", "")
        i = i + 1
    }
}
  [~/c/c/duct] › cat ./examples/loop.dt | ./ducti --quiet
Printing numbers from 0 to 9
i = 0
i = 1
i = 2
i = 3
i = 4
i = 5
i = 6
i = 7
i = 8
i = 9
  [~/c/c/duct] ›
```

Рисунок 3.1.4 - Программа циклу

```
### COMPILING ASSEMBLY ###
# Emmiting bytecode for 'main':
        > found label: 'label000000 : 24'
000000  push i.0
000001  store i@0
000002  push i.10
000003  store n@1
000004  push s(Printing numbers from )
000005  wrt
000006  pop
000007  load i@0
000008  wrt
000009  pop
000010  push s( to )
000011  wrt
000012  pop
000013  load n@1
000014  push i.1
000015  sub
000016  wrt
000017  pop
000018  push s(
)
000019  wrt
000020  pop
000021  push s()
000022  wrt
000023  pop
000024  nop
000025  load i@0
000026  load n@1
000027  lt
000028  ijif 19
000029  pop
000030  push s(i = )
000031  wrt
000032  pop
000033  load i@0
000034  wrt
000035  pop
000036  push s(
)
000037  wrt
000038  pop
000039  push s()
000040  wrt
000041  pop
000042  load i@0
000043  push i.1
000044  add
000045  store i@0
000046  jmp 24
```

Рисунок 3.1.5 - Інструкції згенеровані компілятором.

Результати виконання відповідають очікуваним, під час аналізу на вихідні значення для ассемблера, помітимо що конструкція циклу не відрізняється від гілки

бранчування нічим, окрім адреси куди відбувається стрибок у кінці блоку. Коли гілка IF блоку виконує стрибок до кінця усієї конструкцій IF, з метою покинути лінійний процес перевірки блоків один за одним, WHILE виконує стрибок на початок з метою виконання перевірки знову і у випадку невдачі - виконати стрибок за межі циклу щоб його припинити. Згенеровані інструкції асемблера відповідають *Рисунку 2.4.3.* в *Розділі 2.4*.

Демонстрація функцій. В розробленій мові програмування присутні функції як основний метод опису повторюваної логіки та операцій. Функції приймають аргументи які розміщені на стеку програми та реорганізовують їх для роботи на наступному рівні - викликаної функції. Для спрощення візуалізації стеку і розрізнення даних у DUCT дані змінних (їх пам’ять) знаходяться на окремому стеці що зберігає свій початок (data stack pointer) на стеці ВМ що захищено хостовим ПЗ. Написання функції у DUCT вже було описано в *Розділ 2.* та продемонстровано в попередніх прикладах через потребу у “початковій функції” для кожного скрипту, за замовчуванням така функція називається “main”. В прикладі проведено послідовність розрахунків користуючись функціями замість бінарних операцій у виразах.

```
  [~/c/c/duct] › cat examples/functions.dt
add(l,r) {return l + r}
sub(l,r) {return l - r}
mul(l,r) {return l * r}

main() {
    print("Result of add(sub(10,2), mul(17,3)) = ",
            add(sub(10,2), mul(17,3))
    )
}
  [~/c/c/duct] › cat examples/functions.dt | ./ducti --quiet
Result of add(sub(10,2), mul(17,3)) = 59
  [~/c/c/duct] ›
```

Рисунок 3.1.6 - Демонстрація функцій у мові програмування.

Демонстрація рекурсивних функцій. У функцій є рекурсивна властивість, так як вони заповнюються аргументи на стеці та зберігають всю інформацію що

відповідає контексту виконання інструкцій, то функція може рекурсивно викликати сама себе. Таким чином можна виконувати циклічні дії або вести побудову графів, багато алгоритмів базуються на рекурсії. У прикладі наведено приклад рекурсивного алгоритму де функція виводить інформацію про числове значення свого аргументу та виконує сама себе зменшуючи значення на 1, таким чином виконуючи реверсивний прохід.

```
  [~/c/c/duct] › cat examples/recursion.dt
recursive_print(n) {
    if n > 0 {
        print(n, "\n")
            recursive_print(n - 1)
    }
    return 0
}

main() {
    n = 10
        recursive_print(n)
}
  [~/c/c/duct] › cat examples/recursion.dt | ./ducti --quiet
10
9
8
7
6
5
4
3
2
1
  [~/c/c/duct] ›
```

Рисунок 3.1.7 - Приклад рекурсивної програми написаної на DUCT.

```
### COMPILING ASSEMBLY ###
# Emmiting bytecode for 'recursive_print':
        > found label: 'label000000 : 17'
000000  load n@0
000001  push i.0
000002  gt
000003  ijif 13
000004  pop
000005  load n@0
000006  wrt
000007  pop
000008  push s(
)
000009  wrt
000010  pop
000011  load n@0
000012  push i.1
000013  sub
000014  call recursive_print
000015  pop
000016  jmp 17
000017  pop
000018  push i.0
000019  ret
        ARGS: [ n ]
# Emmiting bytecode for 'main':
000000  push i.10
000001  store n@0
000002  load n@0
000003  call recursive_print
000004  pop
        ARGS: [ ]
```

Рисунок 3.1.8 - Згенерований байт-код програм

Демонстрація опису даних. В мові програмування DUCT надана можливість описувати дані, синтаксис опису даних подібний до синтаксису інших C-подібних скриптових мов програмування, описані дані можуть бути витягнуті користуючись API мови для подальшої десеріалізації до низькорівневих значень які може використовувати оркеструюча программа. Хоча цей API не було повністю реалізовано до демонстрації у роботі, системи для коректного зчитування синтаксису на рівні мови були реалізовані та протестовані.

```
  [~/c/c/duct] › cat examples/values.dt

main() {
    TABLE = {
        string = "hello world",
        number = 10,
        floater = 10.1,
        set = {
            values = [1,2],
            list = $[3,{}]
        }
    }

}
  [~/c/c/duct] › cat examples/values.dt | ./ducti --norun
> ROOT:
 > function:main
  > block:
   > statement:
    > variable 'TABLE'
     > object:
     > variable 'string'
      > expr
            > value: "hello world"
     > variable 'number'
      > expr
            > value: 10
      > variable 'floater'
       > expr
             > value: 10.100000
       > variable 'set'
        > object:
        > variable 'values'
         > array
          > expr
                > value: 1
          > expr
                > value: 2
         > variable 'list'
          > list
           > array
            > expr
                  > value: 3
             > object:
```

Рисунок 3.1.9. Приклад прочитанного синтаксичного дерева для значень в сеті “TABLE”

3.2. **Детальний огляд роботи скриптів на віртуальній машині, заміри швидкості їх роботи.**

Для подальшого висновку та додаткової оцінки розробленого рішення виконаємо детальний огляд роботи інтерпретатора мови, проведемо аналіз алгоритмів рахувального циклу разом з двома версіями алгоритму для розрахунку чисел Фібоначчі та виконаємо заміри швидкості роботи цих скриптів з додаткових порівнянням у *Таблицях 3.2.1, 3.2.2 та 3.2.3 відповідно.* До кожного наведеного DUCT скрипту створено еквівалентну реалізацію у мовах Lua, Javascript та Python для порівняння швидкості роботи прототипу з існуючими рішеннями. Кожен замір відбувається 5 разів і додатково приводиться до середнього арифметичного для зменшення вірогідності флуктуацій значень пов'язаних з кешуванням байт-коду та мапуванням блоків пам’яті при першому запуску, що може спотворювати результати вимірювань для їх коректної оцінки. Одиниці виміру швидкості роботи - мілісекунди, для заміру використано POSIX інструмент - `time`, що надає інформацію про виконання скрипту у реальному часі який складається з часу витраченого в ядрі системи та в користувацькому просторі (sys + userspace). Код усіх скриптів з порівнюваних мов наведено у блоці Додатку - Додаток б).

Рахувальний цикл. Для демонстрації наведено ітераційний алгоритм що збільшує лічильник `i` на 1 допоки не дійде ліміту `n-1`, він виконує вивід текстової стрічки з форматуванням числа починаючи з нової лінії для кожного `i`. Зображення програми та результату її збірки надано у - *Рисунку 3.1.4*, що демонструє программу, та *Рисунку 3.1.5*, що є зображенням байт-коду програми для віртуальної машини який згенерований компілятором мови.

Перевірка ітеративного алгоритму Fibonaccі. Для додаткової демонстрації швидкості запису та зчитування даних створено ітеративний алгоритм Фібоначчі. Алгоритм використовує три змінні для розрахунку та одну для ітерації N чисел Фібоначчі. Надано *Рисунку 3.2.1* коду програми, знімку байт-коду не надано в цьому розділі через його значний розмір в обох випадках демонстрації цього алгоритму, але ця інформація присутня в додатку - Додаток в).

```
  [~/c/c/duct] › cat examples/fib_iterative.dt
fibi(n) {
    last = 1
    prev = 0
    i = 0
    while i < n {
        print(last, "\n")

        next = prev + last
        prev = last
        last = next

        i = i+1
    }

}

main() {
    fibi(20)
}
```

Рисунок 3.2.1 - Код інтерактивної програми Фібоначчі.

Перевірка рекурсивного алгоритму Fibonaccі. для оцінювання підтримки рекурсії у віртуальній машині, також реалізовано стандартний рекурсивний алгоритм Фібоначчі. Його код та роботу наведено на Рисунку:

```
  [~/c/c/duct] › cat examples/fib.dt
fib(number) {
    if number <= 1 { return number }
    return fib(number-1) + fib(number-2)
}

main() {
    i = 1
    while i < 20 {
        res = fib(i)
        print(res, "\n")
        i = i + 1
    }
}
  [~/c/c/duct] › cat examples/fib.dt | ./ducti --quiet
1
1
2
3
5
8
13
21
34
55
89
144
233
377
610
987
1597
2584
4181
```

Рисунок 3.2.2 - Результат роботи рекурсивної програми Фібоначчі.

Таблиця 3.2.1 - Порівняння замірів швидкості роботи програми рахувальника.

Заміри тестів в одиницях мілісекунд.

Мова програмування	Тест 1 (ms)	Тест 2 (ms)	Тест 3 (ms)	Тест 4 (ms)	Тест 5 (ms)	Середнє арифметичне (ms)
DUCT	25.87	29.28	29.53	28.94	29.19	28.56
PYTHON	44.02	44.01	49.13	44.27	45.64	45.41
JAVASCRIPT	101.01	96.32	109.25	100.06	96.26	100.58
LUA	31.33	30.05	29.57	24.57	30.27	29.16

Таблиця 3.2.2 - Порівняння замірів швидкості роботи програми Фібоначчі *(ітеративний алгоритм).*

Мова програмування	Тест 1 (ms)	Тест 2 (ms)	Тест 3 (ms)	Тест 4 (ms)	Тест 5 (ms)	Середнє арифметичне (ms)
DUCT	2.81	2.75	2.52	2.33	2.62	2.61
PYTHON	16.62	26.11	13.06	16.47	16.87	17.83
JAVASCRIPT	37.73	38.81	33.42	32.50	31.55	34.80
LUA	2.11	2.16	1.90	1.97	2.04	2.04

Таблиця 3.2.3 - Порівняння замірів швидкості роботи програми Фібоначчі *(рекурсивний алгоритм).*

Мова програмування	Тест 1 (ms)	Тест 2 (ms)	Тест 3 (ms)	Тест 4 (ms)	Тест 5 (ms)	Середнє арифметичне (ms)
DUCT	19.73	19.69	15.76	20.38	19.85	19.08
PYTHON	17.95	14.77	17.64	18.38	17.52	17.25
JAVASCRIPT	31.61	30.73	33.21	34.05	32.39	32.40
LUA	3.19	3.04	2.93	3.29	3.15	3.12

3.3. Демонстрація використання API для використання мови при встроєні в ПЗ, приклади можливого застосування.

Існуюче API мови. Розроблена бібліотека мови має базовий необхідний програмний інтерфейс для взаємодії з інтерпретатором, від наданий у вигляді полів структури інтерпретатора які користувач може вказати для налаштування поведінки інтерпретатора. *Структура інтерпретатора надає відповідні поля:*

- Поля для виводу інформації про збірку та роботу скриптів до файлових дескритпорів: `trace_vm_execution_f`, `trace_vm_assembly_f`, `trace_compiler_ast_f`.
- Поле для виводу надрукованого програмою тексту: `outputf`
- Поля з бітовою маскою налаштування машини та компіляції: `options`.

  Наведено кодове зображення розробленого інтерперпретатора що складається з ВМ та компілятора з набором параметрів для налаштування поведінки обох систем:

  C

```
. . .
typedef struct {
    DtParser        parser;
    Dtvm            vm;
    bool            have_set_trace_file;
    FILE            *tracef,
                    *errorf;
    FILE    *trace_vm_execution_f,
            *trace_vm_assembly_f,
            *trace_compiler_ast_f;

    int options;
    FILE *outputf;
} DtInterpreter;
. . .
```

Для виконання скриптів надано API що складається з найпростіших операцій: “збірки”, “очистки стану” та “запуску”, а також їх багатьох комбінацій для різних випадків застосування. Також програмний інтерфейс надає змогу доповнювати стани інтерпретатора інструкціями мови ассемблера машини для реалізації ручних оптимізаційних алгоритмів при виникнені такої потреби.

Надано перелік API функцій інтерпретатора для взаємодії зі станом інтерпретатора:

C

```
void dt_prepare(DtInterpreter* interpreter);
bool dt_compile_assembly_from_string(DtInterpreter* interpreter, const char*
source, size_t length);
void dt_append_assembly(DtInterpreter* inter,  const char* assembly);
void dt_run(DtInterpreter* interpreter, const char* entry_point);
void dt_reset(DtInterpreter* interpreter);
void dt_run_reset(DtInterpreter* interpreter, const char* entry_point);
void dt_compile(DtInterpreter* interpreter, const char* source);
void dt_compile_run_reset(DtInterpreter* interpreter, const char* source);
```

Параметричне API для C99+. Завдяки функціоналу стандарту мови С після 1999 року є можливість застосування багато-аргументного інтерфейсу мови користуючись особливостями макросів у мові С. Завдяки “композитним літералам” (Compound literals) [34] в стандартах мови С після STD 99 є можливість створення функцій з опціонними аргументами що можуть додавати додаткові параметри та забезпечувати функціонал подібний до таких в більш сучасних мовах програмування. Для забезпечення зручності API мови DUCT має функцію `dt_execute` яка доступна до використання в усіх стандартах С після та включно з С99.

Наведено кодове зображення API для використання в нових стандартах мови С разом з прикладом його використання:

C

```
. . .
#if __STDC_VERSION__ >= 199901L
typedef enum {
    DT_EXEC_SOURCE_CODE,
    DT_EXEC_SOURCE_ASSEMBLY,
} DtExecuteSourceKind;
typedef struct {
    const char*         override_entrypoint;
    const char*         source;
    DtExecuteSourceKind source_kind;
    bool                do_not_reset;
    bool                do_not_run;
} DtExecuteOptions;
```

```
#define dt_execute(interpreter, ...) \
    dt_exectue_ex(interpreter, (DtExecuteOptions){__VA_ARGS__})
void dt_exectue_ex(DtInterpreter* interpreter, DtExecuteOptions opt);
#endif
. . .

int main(void) {
    dtinterpreter interp = {0};
    dt_execute(&iterp,
            .override_entrypoint = "start",
            .source = str_multiline(
                start() {
                    print("hello world!")
                }
            ));
}
```

Сфери застосування мови, легкість використання. Завдяки простому API та компактній реалізації з використанням C99, мова може бути застосована на широкому діапазоні девайсів та програмного пакунку, допоки девайс чи платформу було створено та підтримано протягом останніх 30 років - допоки у DUCT є можливість працювати у даному середовищі. Завдяки простоті інтерпретатор може бути зібраним з монолітною побудовою та використанням однострокової комманди C компліятора. Що дозволяє легко почати роботу з інструментарієм за менше ніж 1 хв.

Приклад необхідних кроків для створення проекту та використання бібліотеки мови програмування у середині окерструючого ПЗ:

C

```
$ ls
dt.h example.c
$ cat example.c
       #include “dt.h”
       int main(void) {
              DtInterpreter In = {0};
              dt_compile_run_reset(&In, str_multiline(main() {print(“Hi”)}))
       }
$ cc -o main main.c        # build
$ ./main            # run
Hi
```

До прикладів можливого застосування мови входять:

- *Інтерактивне середовище мови - REPL.* Мову програмування Python вважають доречною мовою програмування для навчання алгоритмізації та програмування, частково це так через можливість взаємодії з мової інтерактивно. Read Eval Print Loop (REPL) - це заснована сімейством мово LISP ідея використання текстової строки для взаємодії з функціоналом мови через прості блоки коди які виконують операції у контексті працюючого інтерпретатора. Завдяки інтерактивності REPL надає рішення для прототипування ідей, експериментування, навчання, чи швидкого вирішення актуальних одноразових проблем. Мова DUCT так само як і всі інші скриптові мови - може бути застосована у REPL для роботи у реальному часу і взаємодією з користувачем.
- *Зчитування файлів та серіалізації інформації.* Для збереження та серіалізації інформації за останні 20 років набули популярності текстові формати JSON, YAML, INI, тощо. Ці формати існують виключно для організації інформації у читабельному для користувача вигляді, вони є доречними до використання для забезпечення можливості людині їх читати та модифікувати з додатковою перевагою у свободі вільного доповнення без порушення існуючої інформації, що не є доцільним при збереженні бінарної (сирої) інформації програми як при зміні структури у майбутньому повністю порушує її порядок і потребує створення конвертаторів для процесу міграції до нової версії формати даних. Проблемою таких форматів є їх фактична напів-готовність бути мовою програмування, ефективні та надійні парсери JSON та YAML за об’ємом коду близькі до простих скриптових мов програмування з додатковими функціями обчислення виразів що можуть бути доречними у файлах налаштування чи набору інструкцій для виконання дій. В таких задачах інформаційні файлові формати можуть бути повністю замінені мінімалістичними скриптовими мовами, що можуть функціонувати на половину свого функціоналу, замість створення чи використання окремого рішення що вирішує одну і й ту саму проблему.

- Модифікації та розширене налаштування програмного забезпечення. У багатьох програмах кількість можливих модифікацій є настільки різноманітною що створення відповідної системи налаштувань є складною або іноді не раціональною по ресурсам (часу, фінансування, кількість робочої сили, тощо), тому скриптова мова разом з розробленим інтерфейсом до функціоналу програмного забезпечення може забезпечити необхідну свободу до створення плагінів, скриптів і інших видів користувацької логіки що надає функції що розробник міг навіть не мати в планах при створенні застосунку. Прикладом таких систем є плагіни Графічних редакторів - GIMP, KRITA, де користувач може створювати програмовану поведінку для робочого аркуша чи інструментів редагування, 3D програм для моделювання - Adobe Illustrator, 3Ds Max, Blender, чи програми для аудиту чи сканування безпеки наприклад NMAP що використовує LUA скрипти для розширеного налаштування джерел інформації в мережі для яких повинно бути виконано сканування.

Реалізація простого консольного калькулятора. Для надання реального робочого прикладу застосування DUCT в оркеструючому програмному забезпеченні - створено найпростіший термінальний калькулятор з використанням розробленої мови. Головною задачею бути прийняти вираз користувача в терміналі як одну строку і далі використати API мови для розрахунку та виведення результату на екран з відповідним до типу даних форматуванням. Програма має перевірити наявність аргументів щоб не роботи спроб доступу до неіснуючої пам’яті, далі прочитати аргумент у випадку якщо він є, після чого виконати форматування виразу до строки повноцінної DUCT програми - користуючись C функцією `snprintf`, та надати цю строку до функції інтерпретатора що виконає програму та очистить стан віртуальної машини та компілятора, далі результат буде виведено на екран з допомогою `printf` та формуючої функції `const char* object_to_string(Object)`.

Для інтерпретації використано описану в цьому розділі функцію `dt_compile_run_reset(DtInterpreter*, const char*)` адже вона є найзручнішою до

простої демонстрації. *Далі наведено приклад кодової реалізації калькулятора, після чого наведено Рисунок 3.1.2, що демонструє роботу цієї програми, разом з Блок-Схемою калькулятора на Рисунку 3.3.1.*

Програмний код простого консольного калькулятора з використанням DUCT, вся складна логіка інтерпретації виразу знаходиться всередині однієї функції - ‘dt_compile_run_reset`:

C

```
#include <stdio.h>
#include <stdlib.h>
#include "../../src/dt.h"

int main(int argc, char** argv) {
    static char buffer[1<<10];
    const char* source = 0;
    DtInterpreter interp = {0};

    if(argc>1 && argv[1]) {
        source = argv[1];
    } else {
        fprintf(stderr, "USAGE: %s <EXPR>", argv[1]);
       exit(1);
    }

    snprintf(buffer, sizeof(buffer)-1, "main() {return (%s)}", source);
    dt_compile_run_reset(&interp, buffer);

    Object result = interp.vm.returned_object;
    printf("> %s", object_to_string(result));
}
```

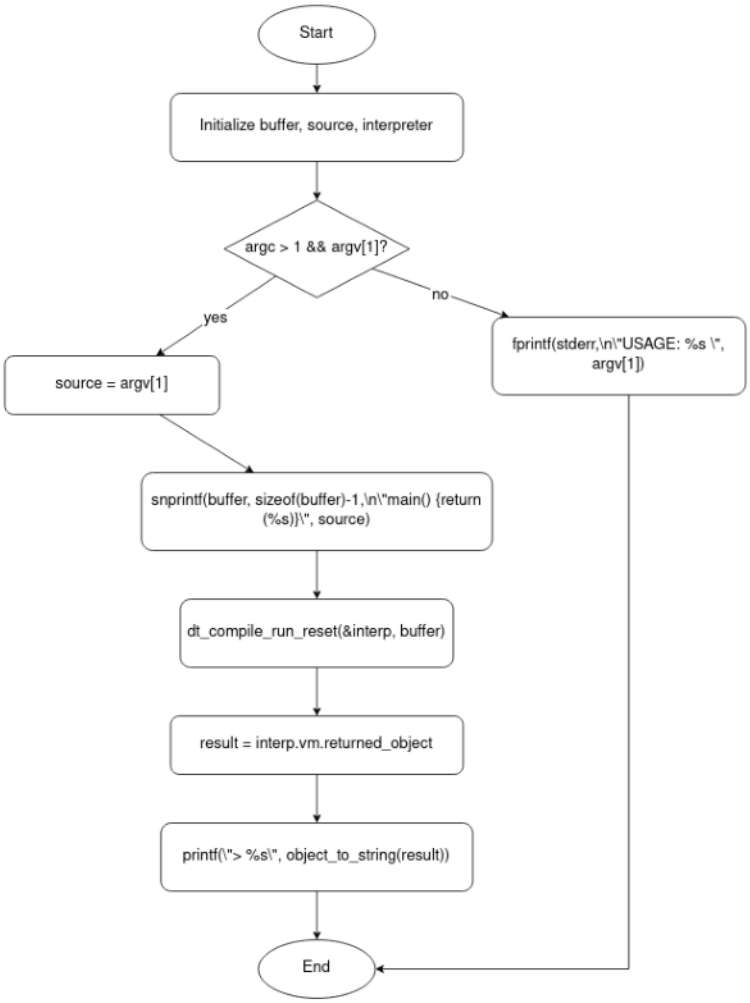

Рисунок 3.3.1 - Блок схема програми калькулятора що є прикладом використання мови Duct в середині ПЗ що потребує функції динамічного розрахунку.

```
  [~/c/c/d/e/usage_example] › ./main '(1024*1024*(1 + 2 + 3 + 4)) + 12.0'
> 10485772
  [~/c/c/d/e/usage_example] › ./main 'true'
> true
  [~/c/c/d/e/usage_example] › ./main 'true + true'
> true
  [~/c/c/d/e/usage_example] › ./main 'true + 2'
> 2
  [~/c/c/d/e/usage_example] › ./main 'false + 1'
> 1
  [~/c/c/d/e/usage_example] › ./main 'false + 0'
> 0
  [~/c/c/d/e/usage_example] › ./main 'false + false'
> false
  [~/c/c/d/e/usage_example] › ./main '1 + 1.0'
> 2
  [~/c/c/d/e/usage_example] › ./main '1 + 1.5'
> 2
  [~/c/c/d/e/usage_example] › ./main '1 + 2.5'
> 3
  [~/c/c/d/e/usage_example] › ./main '1.25 + 2.5'
> 3.750000
  [~/c/c/d/e/usage_example] › ./main '0.1+0.2'
> 0.300000
  [~/c/c/d/e/usage_example] ›
```

Рисунок 3.3.2 - Демонстрація роботи програми калькулятора.

3.4. **Недоліки виявлені при розробці, Проблеми у парадигми С-подібних мов програмування для використання у інтегрованих середовищах при описі даних.**

Під час розробки мови програмування DUCT та її експлуатації під час тестування та прототипування було визначено ряд недоліків які проявили себе після написання першого напів-завершеного прототипу. Недоліки мови у випадку DUCT притаманні більше синтаксису мови аніж головним принципам і архітектурним вимогам визначених в Розділ 2, варто зазначити головні недоліки С-подібного синтаксису та аргументувати потребу в альтернативі з описом вагомих переваг які можуть бути отримані при зміні парадигми мови.

Недоліки парадигми мови програмування на інструкціях. Мови програмування в міру обмеженості розрахункових систем зазвичай утворюють набори правил та концепцій яких має притримуватись розробник щоб його програми корректно зчитував компілятор або інтерпретатор. Стиль написання програм та ідеологія яка встановлена мовою є основним роздільним фактором між мовами програмування, адже кожен стиль написання може намагатись покращити один із аспектів мови за рахунок погіршення іншого. Стиль написання та ідеї мови називають парадигмою мови програмування. Деякі парадигми мають кращі, для меншої кількості коду та швидкості роботи програми, властивості синтаксису - що робить цю мову кращою для роботи з комп'ютером, але такі мови зазвичай мають складнішу для розуміння людини структуру. Мови програмування що спрощують синтаксис для розмінням людиною зазвичай потребують багатьох абстракцій що в свою чергу є фоновими перетвореннями коду які користувач міг не мати на увазі під час написанні алгоритму.

Всього існує 3 основних синтаксичні парадигми програмування:

- Функціональна парадигма (Functional paradigm)
- Стверджувальна парадигма (Statement paradigm)
- Послідовна парадигма (Concatenative programming paradigm) [32]

Мов програмування С належить до “Стверджувальних” мов програмування не адже кожен крок у коді відображається твердженням, але незважаючи на класифікацію мова люба С-подібна програмування має конструкції тверджень що притаманні функціональним мовам. С не є виключно “Стверджувальних” мовою через складність написання складних дій до якої належить: бранчування, цикли, арифметика, індексування, тощо. У комп’ютер такі дії є складними операціями і потребують кілької строчок з твердженнями для вирішення задачі, чистою формою тверджень мов є мови ассемблера адже вони є прямим інтересом до включно стверджувальної моделі розрахунку на стеці комп'ютера. Для балансу швидкості обробки тексту, ефективності обробки у обмеженому в ресурсах комп’ютері та для збереження всіх відомих методів оптимізації при розробці мови програмування С було обрано “Стверджувальну”[31] парадигму з елементами функціональної для максимального балансу між читаємостю людиною та практичності при збірці програм комп'ютером. Альтернативою до стверджувального виконання дій є функціональна парадигма яка замінює одне конкретне твердження на їх послідовність виразів які складаються або з кінцевих значень та операцій, або інших виразів. У такій мові програмування концепція дій та інформації змивається в єдину структуру написання і таким чином дані стають кодом, а код стає даними над якими можна проводити операції. Хоча така мова програмування в кінцевому результаті потребує розгортання до імперативних інструкцій - її абстрактний у написанні синтаксис та властивості можуть спростити і скоротити написання алгоритмів чи опису даних, хоч і погіршуючи контроль розробника над кожною індивідуальною дією

Переваги функціональної парадигми програмування, приклади.

- *Експресивність та однотонність.* Функціональні (декларативні) мови програмування через свою рекурсивну природу є більш експресивними ніж імперативна послідовність інструкцій , що дозволяє створювати логічні ціпки та блоки які можуть бути замінені, навіть при встановленому в умовах роботи обмеження на повну відсутність головного наслідку такої парадигми програмування - наявність першокласcової підтримки метапрограмування,

функціональна парадигма все ще дозволяє описувати логіку програми так само як і дані, що робить її простою в прочитанні та виконанні. Наприклад мова Common Lisp на відміну від мови програмування C має лише одну структурну одиницю синтаксису - список, коля мові C потрібні унікальні формації токенів що кардинально відрізняються один від одного.

- *Простота роботи та обробки.* При обробці коду в незалежності від методу обраного для розрахунку - Віртуальної машини чи інтерпретації через рекурсію, синтаксис мова має бути перетворено до десеріалізованих даних у пам’яті програми що далі використані в розрахунку та логістиці. В скриптових мовах програмування основним кроком окрім ВМ є аналіз через Парсер, через особливість виразів та їх рекурсивну природу, декларативний синтаксис є простішим до аналізу та прочитання інтерпретатором порівняно до тверджень, адже кожне унікальне твердження потребує унікальної послідовності прочитання, коли вирази подібні до таких у Lisp завжди пишуться однаково.

Для демонстрації переваги простоти аналізу C подібних мов до Lisp подібних достатньо поглянути на мінімальний приклад синтаксичної структури що можливо описати за допомогою EBNF, адже розмір такого зображення синтаксису напряму вказує на кількість необхідних дій для отримання ідентичного результату - виконання розрахункових операцій. Далі надано EBNF (Extended Backus-Naur Form) для простих, мінімальних, прикладів мов обох парадигм - *C-подібні* та *Lisp-подібні.* Обидві мови з відповідним функціоналом можуть бути однаково використанні в існуючій машині, і менший за об’ємом синтаксис для аналізу є об’єктивно кращим до застосування.

Приклад структури синтаксису Lisp-подібної декларативної мови описаної через Extended Backus-Naur Form:

EBNF

```
program = { expression } ;
expression = atom | list;
list = "(" { expression } ")" ;
atom = number | string | symbol | boolean;
```

```
number = integer | float ;
integer = [ "-" ] digit { digit } ;
float = [ "-" ] digit { digit } "." digit { digit } ;
string = "\"" { character } "\"" ;
boolean = "true" | "false" ;
symbol = initial { subsequent } ;
initial = letter | "+" | "-" | "*" | "/" | "=" | "<" | ">" | "?" | "!";
subsequent = initial | digit | "_";
letter = "A"…"Z" | "a"…"z" ;
digit = "0"…"9" ;
character = ? any printable character except " ? ;
```

Приклад структури синтаксису С-подібної імперативної мови описаної через Extended Backus-Naur Form:

EBNF

```
program = { declaration } ;
declaration = function_decl | variable_decl;
function_decl = type identifier "(" [ parameter_list ] ")" block ;
parameter_list = parameter { "," parameter } ;
parameter = type identifier ;
variable_decl = type identifier [ "=" expression ] ";" ;
type = "int"| "float"| "bool"| "string"| "void";
block = "{" { statement } "}" ;
statement = variable_decl| assignment ";"| if_statement| while_statement|
return_statement ";"| expression ";"| block;

assignment = identifier "=" expression ;
if_statement = "if" "(" expression ")" statement [ "else" statement ] ;
while_statement = "while" "(" expression ")" statement ;
return_statement = "return" [ expression ] ;
expression = equality ;
equality = comparison { ( "==" | "!=" ) comparison } ;
comparison = term { ( "<" | ">" | "<=" | ">=" ) term } ;
term = factor { ( "+" | "-" ) factor } ;
factor = unary { ( "*" | "/" ) unary } ;
unary = [ "!" | "-" ] primary ;
primary = integer| float| string| "true"| "false"| identifier| function_call| "("
expression ")";

function_call = identifier "(" [ argument_list ] ")" ;
argument_list = expression { "," expression } ;
identifier = letter { letter | digit | "_" } ;
integer = digit { digit } ;
float = digit { digit } "." digit { digit } ;
```

```
string = "\"" { character } "\"" ;
letter = "A"…"Z" | "a"…"z" ;
digit = "0"…"9" ;
character = ? any printable character except " ? ;
```

Для проведення порівняння наведено *Таблицю 3.4.1* яка співставляє синтаксис обох парадигм разом з числовою інформацією визначену з обох наведених прикладів EBNF.

Таблиця. 3.4.1 - Порівняння складності та комплексності синтаксичного аналізу імперативної мови (C-подібної) та декларативної (Lisp-подібної).

Критерій порівняння	C подібний синтаксис та імперативна парадигма програмування.	Lisp-подібний синтаксис та декларативна парадигма програмування
Неформальных правил синтаксису (Non-terminal rules)	27	13
Кількість рівнів пріоритету у виразах	5	0 (Всі рівні пріоритету напряму залежить від вказаної розробником послідовності розгортання виразів.
Ключових слів	35	~10
Кількість форм тверджень	7	0 (Вся мова - вирази)
Складність токенізації	Потребує огляду на кілька токенів в перед, потребує нелінійного вирішення деяких конструкцій чи при роботі з типами.	LL(1) парсер не потребує більше одного символу на контекст для аналізу.
Різноманітність розмежувальних токенів	35 - `{}`, `()`, `[]`, `;`, `,`, unary, binary, assignment, keyword, declarations.	2 - `()`.
Розмір парсера	1000-10000+ (duct,umka[35],teak[36])	100-300 (fe[37], tinylisp[38])

ВИСНОВКИ

У дипломній роботі розглянута проблема забезпечення безпеки програмного середовища що використовує скриптову мові мови програмування для надання користувачу свободи налаштування, організації чи програмування молиживостей розробленого програмного забезпечення. Під час аналізу було визначено ряд проблем які характерні як системним мовам програмування так і високорівневим - скриптовим мовам, визначені аспекти та функції мов як є джерелом проблем надійності, передбачуваності та безпеки що проявляють себе існуючому програмному забезпеченні та ідеї реалізація яких погіршує якість забезпечення надійності та цілісності розробленого програмного забезпечення.

В рамках дипломного проектування було розроблено набір правил який орієнтований на забезпечення досить швидкого і найнебезпечнішого вирішення розрахункових операцій у середині оркеструючого програмного забезпечення з спробами мінімізувати людський фактор при напианних виконуючих програм - скриптів, зменшити кількість ускладнень та абстракцій у процесі керування пам’ятью разом з процессом організації даних у ній та надати необхідну інформацію для створення власної інфраструктури мови що здатна вирішувати проблему імперативного опису інформації та даних, разом з підтримкою, за потреби, декларативного підходу до запису та розрахунку інформації.

Під час проектування було розроблено прототип імперативної С-подібної скриптової мови програмування, з назвою DUCT, що незважаючи на свій не завершений стан була використана для демонстрації розробленої архітектури з дотриманням поставлених вимог до дизайну системи. Мова була застосована для виконання простих математичних скриптів з додатковими замірами якості їх роботи. Ці скрипти бути виконані на Віртуальній Машині розробленої для роботи з саме цією мовою і зібрані одно-кроковим Компілятором мови що має базовий функціонал перевірки на помилки які не виконують половина скриптових мов

використаних в комерційному та відкритому програмному забезпеченні. По результатам замірів нинішній стан машини показав себе компетентним у швидкості роботи порівняно з іншими скриптовами мовами програмування незважаючи на свій прототипний стан, що робить рішення не тільки функціональним для забезпечення надійного розрахункового середовища а і досить швидкого до застосування як альтернативу для LUA, що вважається досить швидкою скриптовою мовою програмування.

Під час і після розробки прототипу було виявлено недоліки які по великому рахунку пов’язані з вибором імперативної форми синтаксису для мови, через невиправдані переваги подібності до C за синтаксисом мова потребувала більше часу на реалізацію і не отримала багатьох функцій які було очікувано продемонструвати до терміну дипломного проектування. Навіть у напів-заверешному стані прототип продемонстрував потенціал напряму для розвитку мови орієнтованої для безпечного розрахунку в орекструючому програмного забезпечення як не вирішену сферу в якій є простір до покращення чи інновацій.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Paper Tape Storage Systems. RI Computer Museum : вебсайт. URL: https://www.ricomputermuseum.org/data-storage/paper-tape (дата звернення: 24.05.2026).
2. Cormen T. H., Leiserson C. E., Rivest R. L., Stein C. *Introduction to Algorithms*. 3rd ed. Cambridge, MA : MIT Press, 2009. P. 232–233. ISBN 978-0-262-03384-8.
3. Patterson D. A., Hennessy J. L. Computer Organization and Design: The Hardware/Software Interface. 3rd ed. Amsterdam ; Boston : Morgan Kaufmann Publishers, 2004. 656 p. ISBN 1-55860-604-1.
4. Robert A., Kern C. Secure by Design: Google's Perspective on Memory Safety. Google Security Engineering Technical Report. Mountain View : Google, 2024. 32 p. URL: https://storage.googleapis.com/gweb-research2023-media/pubtools/pdf/70477b1d77462cfffc909ca7d7d46d8f749d5642.pdf (дата звернення: 24.05.2026).
5. Bastien J. F., Meredith A. P2186R2: Removing Garbage Collection Support. WG21: ISO/IEC JTC1/SC22, 2021. URL: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2186r2.html (дата звернення: 24.05.2026).
6. MaskRay. All about Global Offset Table. URL: https://maskray.me/blog/2021-08-29-all-about-global-offset-table (дата звернення: 24.05.2026).
7. Linux Foundation. zSeries ELF Application Binary Interface Supplement. Chapter 3. Program Loading and Dynamic Linking. Section 3.2.2 Global Offset Table. URL: https://refspecs.linuxfoundation.org/ELF/zSeries/lzsabi0_zSeries/x2251.html (дата звернення: 25.05.2026).
8. Relocation Read-Only (RELRO). CTF101. URL: https://ctf101.org/binary-exploitation/relocation-read-only/ (дата звернення: 24.05.2026).

9. RunSafe Security. Memory Safety Vulnerabilities Explained. 2024. URL: https://runsafesecurity.com/blog/memory-safety-vulnerabilities/ (дата звернення: 04.06.2026).
10. Dhurjati D., Kowshik S., Adve V., Lattner C. Memory Safety without Runtime Checks or Garbage Collection. Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems. New York : ACM, 2003. P. 69–80. DOI: 10.1145/780732.780743. [Electronic resource].
11. OpenJDK. Z Garbage Collector (ZGC) project / OpenJDK Source Code Repository [Electronic resource]. URL: https://github.com/openjdk/zgc (дата звернення: 25.05.2026).
12. Muratori C. The Thirty Million Line Problem [Video]. YouTube. 2023. URL: https://www.youtube.com/watch?v=kZRE7HIO3vk (дата звернення: 24.05.2026).
13. Muratori C. Simple Code, High Performance [Video]. YouTube. 2024. URL: https://www.youtube.com/watch?v=Ge3aKEmZcqY&list=PLEMXAbCVnmY4JbNByvpgEzWsLRKVaF_pk (дата звернення: 24.05.2026).
14. Muratori C. Thirty Million Line Problem. 2005. URL: https://caseymuratori.com/blog_0031 (дата звернення: 04.06.2026).
15. ANSI INCITS 226-1994 (formerly ANSI X3.226-1994). Information Technology—Programming Language—Common Lisp (15.17R ed.). 12 Aug. 1994, p. 243.
16. Statista. Most Used Programming Languages Among Developers Worldwide. 2024. URL: https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/ (дата звернення: 04.06.2026).
17. Rinz P., Crawford T. C in a Nutshell. Sebastopol : O'Reilly Media, 2005. 504 p. ISBN 978-0-596-55071-4.
18. IMPRESSUM. FFI Library Documentation. URL: https://luajit.org/ext_ffi.html (дата звернення: 24.05.2026).
19. Wikipedia contributors. (n.d.). MinGW. Wikipedia. URL: https://en.wikipedia.org/wiki/MinGW (дата звернення: 04.06.2026).

20. Barrett S. stb Single-File Public Domain Libraries. GitHub Repository. URL: https://github.com/nothings/stb (дата звернення: 04.06.2026).
21. Free Software Foundation. GNU C Library malloc implementation. URL: https://codebrowser.dev/glibc/glibc/malloc/malloc.c.html#3841 (дата звернення: 04.06.2026).
22. Snyk. SHA-1 Hulud NPM Supply Chain Incident Analysis. 2025. URL: https://security.snyk.io/sha1-hulud-npm-supply-chain-incident-nov-2025 (дата звернення: 04.06.2026).
23. OXY Occidental College. Polish Notation. URL: https://sites.oxy.edu/traiger/logic/primer/chapter2/polish-notation.html (дата звернення: 02.05.2026).
24. Łukasiewicz, Jan. (1951). Aristotle's Syllogistic from the Standpoint of Modern Formal Logic. Oxford: Clarendon Press, p. 78.
25. GitHub Blog. Now You C Me, Now You Don't: Vulnerability Research in C Projects. 2025. URL: https://github.blog/security/vulnerability-research/now-you-c-me-now-you-dont/ (дата звернення: 04.06.2026).
26. hch Handy C Headers data structures. GitHub. URL: https://github.com/amuerta/hch/tree/master/src (дата звернення: 04.06.2026).
27. Shi Y., Engelberg S., Kaashoek M. F. Virtual Execution Environments for Dynamic Binary Translators // Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2005). New York : ACM, 2005. P. 153–163. URL: https://static.usenix.org/events/vee05/full_papers/p153-yunhe.pdf (дата звернення: 25.05.2026).
28. TechRadar. Does the Stack Go Up or Down? Memory Architecture Explained. 2023. URL: https://techradar.info/does-the-stack-go-up-or-down-memory-architecture-explained/ (дата звернення: 19.05.2026).
29. Koopman P. J., Jr. Stack Computers: The New Wave. Chapter 3. Multiple-Stack, 0-Operand Machines [Electronic resource]. URL:

https://users.ece.cmu.edu/~koopman/stack_computers/sec3_2.html (дата звернення: 24.05.2026).

30. Computer History Museum. Intel CPU Architecture and the Stack [Electronic resource]. URL: https://tcm.computerhistory.org/ComputerTimeline/Chap37_intel_CS2.pdf (дата звернення: 05.06.2026).
31. Recurse Center. Statements vs Expressions. 2023. URL: https://recurse.se/2023/11/statements-vs-expressions/ (дата звернення: 24.05.2026).
32. Purdy J. Why Concatenative Programming Matters [Electronic resource] // The Big Mud Puddle. 2012. URL: https://web.archive.org/web/20241008135442/https://concatenative.org/wiki/view/Concatenative%20language (дата звернення: 12.08.2025).
33. Amuerta. Duct Programming Language Source Code. GitHub. URL: https://github.com/amuerta/duct (дата звернення: 24.05.2026).
34. Standard C++ Foundation. Compound Literals in C. URL: https://en.cppreference.com/c/language/compound_literal (дата звернення: 24.05.2026).
35. Tereshkov V. Umka Lexer Implementation. GitHub. URL: https://github.com/vtereshkov/umka-lang/blob/master/src/umka_lexer.c (дата звернення: 24.05.2026).
36. Nakst. Teak Programming Language Source Code. GitHub. URL: https://github.com/nakst/teak/blob/master/teak.c (дата звернення: 24.05.2026).
37. RXI. Fe Programming Language Source Code. GitHub. URL: https://github.com/rxi/fe (дата звернення: 24.05.2026).
38. Daneel San. TinyLisp: Building a Minimal Lisp Interpreter. 2025. URL: https://daneelsan.github.io/2025/03/20/tinylisp.html (дата звернення: 24.05.2026).
39. Шпак М., Курченко О., Щебланін Ю. Зменшення рівня вразливості Garbage Collection за допомогою Arena Allocator. 2025. URL:

https://csecurity.kubg.edu.ua/index.php/journal/article/view/1096 (дата звернення: 24.05.2026).

40. Backus J. Can Programming Be Liberated from the von Neumann Style? Communications of the ACM. 1978. Vol. 21, no. 8. P. 613–641.
41. McCarthy J. Recursive Functions of Symbolic Expressions… Communications of the ACM. 1960. Vol. 3, no. 4. P. 184–195.
42. McIlroy M. D. Mass Produced Software Components. NATO Software Engineering Conference, 1968.
43. ISO/IEC 9899:2018. Programming Languages — C. Geneva: International Organization for Standardization, 2018.
44. Ousterhout J. K. Scripting: Higher-Level Programming for the 21st Century. Computer. 1998. Vol. 31, no. 3. P. 23–30.
45. Gosling J., Joy B., Steele G., Bracha G., Buckley A. The Java Language Specification. Oracle, 2014.

ДОДАТОК А

Надано заголовковий код Віртуальної Машини та Компілятора.

ВМ:

```
#ifndef __DTVM_DEFINITIONS_H
#define __DTVM_DEFINITIONS_H

#include "../config.h"
#include "../shared/shared.h"
#include "../shared/hc_plug.h"

/*  DEFINES OR CONSTANTS    */
#define         DT_SIZE_T_INVALID                   ((size_t)-1)

/*  ENUMS   */

typedef enum {
    OT_NULL = 0, // should be an error if encountered. NULL's are bad.
    OT_BOOL     = 1,
    OT_BYTE     = 2,
    OT_CHAR     = 3,
    OT_INT      = 4,
    OT_FLOAT    = 5,
    OT_CHARACTER= 6,
    OT_STRING   = 7,
    OT_TYPE     = 8,
    OT_RESULT   = 10,

    OT_IS_ARRAY     = (1<<4),
    OT_IS_LIST      = (1<<5),
    OT_IS_OBJECT    = (1<<6),
} EObjectType;

// VM
enum {
    DTI_NULL, // probably error
    DTI_NOP,
    DTI_STORE,
    DTI_LOAD,

    DTI_CALL,
    DTI_RET,

    DTI_PUSH,
    DTI_DUP,
    DTI_POP,
```

```
    DTI_ROT,
    DTI_ADD,
    DTI_SUB,
    DTI_DIV,
    DTI_MUL,
    DTI_AND,
    DTI_OR,
    DTI_NOT,


    DTI_EQ,     // ==
    DTI_LT,     // <
    DTI_GT,     // >
    DTI_LTE,    // <=
    DTI_GTE,    // >=

    // WORK ON LITERALS:
    DTI_LEQ,     // ==
    DTI_LLT,     // <
    DTI_LGT,     // >
    DTI_LLTE,    // <=
    DTI_LGTE,    // >=

    DTI_IJIF,   // INCREMENTAL JUMP IF
    DTI_JMP,

    DTI_WRT,    // writes top of the stack to vm.fstdout

    DTI_COUNT,
} EInstruction;

enum {
    DTSTAT_OK       = 0,
    DTSTAT_IGNORE,
    DTSTAT_START_FUNCTION,
    DTSTAT_END_FUNCTION,
    DTSTAT_NOT_ERROR,

    DTSTAT_INVALID_OPERAND_TYPE,
    DTSTAT_NO_OPERAND,

    DTSTAT_COUNT,
} ECompileStatus;

enum {
    ELINE_INSTRUCTION,
    ELINE_LABEL,
```

```
    ELINE_FUNCTION,
} ELineResult;

/* TYPEDEFS */
static String  STRING_NULL = {0};

// STRUCTS //

struct DtList  ;
struct DtArray ;
struct Object  ;

typedef struct {
    String  name;
    long    address;
} Label;

typedef struct {
    Label* items;
    size_t count, capacity, typesize;
} Labels;

typedef struct DtArray {
    void*  items;
    size_t count, capacity, typesize;
} DtArray;

typedef struct DtComplex {
    struct Object
        *next, *prev, *tail,
        *children
    ;
} DtComplex;

// OBJECT(s)
typedef int Result;
typedef struct Object {
    String   id;
    byte    properties;
    int     type;
    union {
        bool            B;
        unsigned char   b;
        int             i;
        float           f;
        Result          r;
        String          s; // read only string
```

```
        DtArray         A;
        DtComplex       C;
        EObjectType     T; // object represents "Type"
    } as;
} Object;

typedef struct ObjectMap {
    Arena   local_allocator;  // temporary allocator for dynamic objects
    Object* items;
    MapHead map_head;
} ObjectMap;

typedef struct {
    size_t prev_pointer, count;
} ScopeStackFrame;

typedef struct {
    ObjectMap       buf             [DTVM_SCOPES];
    ScopeStackFrame object_frames   [DTVM_SCOPES];
    size_t      top, count;
} Scopes;

//  INSTRUCTIONS
typedef struct Instruction {
    int kind;
    union {
        struct {
            String ident;
            size_t id;
        } load, store, call;

        struct {
            Object object;
        } push, leq, lgt, llt, lgte, llte;

        struct {
            size_t jumpto;
            String label; // never used in the actual vm,
                          // exists strictly for second assembler pass.
        } ijif, jmp;
    } as;
} Instruction;

typedef struct {
    int kind;
    union {
        Instruction     instruction;
```

```
        Label           label;
    } as;
} LineResult;

typedef struct Instructions {
    Instruction* items;
    size_t count, capacity;
} Instructions;

//
// FUNCTIONS
//


// Duct Function is a handle with data on
// function identity and memory pointer - under which
// all of the function data is stored.
//
// This way each function is self contained "piece" that
// can be treated as single unit which either is valid, or not.
typedef struct {
    String      id;
    String*     argv; size_t argc;

    // owned memory, it has everything that function needs,
    // packed tightly in linear block of memory.
    void* memory;
    // data section
    size_t  symbol_count; // total=(args + instruction ids) symbols
    size_t  variable_symbols;
    // code section
    Instructions code;
} Function;

typedef struct {
    Function* items;
    size_t count, capacity;
} Functions;

typedef struct {
    String *items;
    size_t count, capacity, typesize;
} Strings;

// VM
typedef struct {
    int             options_mirror; // mirror of options from interpreter.
```

```
    // TODO: use arena allocator per stack frame
    //Arena           allocator;
    Functions       functions;

    size_t          top;
    size_t          work_sp, data_sp;
    Object          returned_object;
    Object          work_stack[DTVM_STACK_CAPACITY];
    Object          data_stack[DTVM_STACK_CAPACITY];

    Scopes          scopes;
    FILE            *logf, *asmtracef, *tracef, *writef;
} Dtvm;

/*  FUNCTION DECLARATIONS */

// UTILITY
int string_to_integer       (String s);
float string_to_float       (String s);
bool string_is_text         (String s);
bool string_is_identifier   (String s);

String   string(const char* str);
bool     string_cmp(String l, String r);

// DT OBJECTS
static inline Object object_byte(char c);
static inline Object object_int(int i);
static inline Object object_float(float f);
static inline Object object_string(String s);

Object*     object_get  (Dtvm* map, size_t id);
Object      object_store(Object* ref, Object value);
Object*     object_reserve_or_get(ObjectMap* map, String id);

// DT SCOPE(S)
ObjectMap*  dtvm_get_scope(Dtvm* vm);
void        dtvm_free_scope(ObjectMap* scope);

// VM INTERNALS
//
void        dtvm_trace_execution(Dtvm* vm, Instruction inst, size_t ip);
Function*   dtvm_get_function(Dtvm* vm, String fnid);
Object      dtvm_call(Dtvm* vm, String fnid, Object* argv, int argc, size_t datasp);
void        dtvm_if(Dtvm* vm, Function* fn);
void        dtvm_push(Dtvm* vm, Object value);
```

```
// ASSEMBLER

#define dta_trace(tracef, ...) if(tracef) {\
    fprintf(tracef, __VA_ARGS__);\
}


// VM API
void dtvm_compile_from_asm(Dtvm* vm, String source);

#endif//__DTVM_DEFINITIONS_H
```

Компілятор:

```
#ifndef __DT_COMPILER_DECLARATIONS
#define __DT_COMPILER_DECLARATIONS

#include "../config.h"
#include "../shared/shared.h"

/*
 * Handy C Header glue
 */
#include "../shared/hc_plug.h"

/*
 *  TOKENIZER
 */

#define INVALID_INDEX (size_t)(-1)

#ifndef HC_DEFINITION_TYPE_INDEX_T
#define HC_DEFINITION_TYPE_INDEX_T
    typedef size_t              index_t;
#endif//HC_DEFINITION_TYPE_INDEX_T


typedef unsigned char      symbol_t;
typedef unsigned int       wsymbol_t;
typedef const char*        cstr_t;
typedef unsigned int       tokenid_t;

#define MUTCSTR(s) ((char*)(s))


// TODO: remove this nonsense
```

```
#define TOKENIZER_TOKEN_FMT_CONTENT "$"

#ifndef TOKENIZER_TOKEN_FMT_CONTENT
#      define TOKENIZER_TOKEN_FMT_CONTENT "text: "
#endif
#ifndef TOKENIZER_TOKEN_FMT_KIND
#      define TOKENIZER_TOKEN_FMT_KIND "kind: "
#endif
#ifndef TOKENIZER_TOKEN_FMT_ID
#      define TOKENIZER_TOKEN_FMT_ID "id: "
#endif

#define TEMP_CSTR_LENGTH DT_SCRATCH_BUFFER_SIZE
#ifndef TOKENIZER_TOKEN_INITIAL_COUNT
#      define TOKENIZER_TOKEN_INITIAL_COUNT 32
#endif


typedef struct {
       const char* data;
       size_t            length;
} TknSlice;

typedef struct {
       const char*   txt;
       tokenid_t           id;
} TokenTableEntry;

typedef enum {
       TOKEN_KIND_NULL,
       TOKEN_KIND_SYMBOL,
       TOKEN_KIND_WORD,
       TOKEN_KIND_LITERALL_INTEGER,
       TOKEN_KIND_LITERALL_FLOAT,
       TOKEN_KIND_LITERALL_STRING,
    TOKEN_KIND_COMMENT_BLOCK,
       TOKEN_KIND_EOL,
       TOKEN_KIND_EOF,
} TokenKind;

typedef struct {
    size_t
        row,
        col;

       TokenKind    kind;
       tokenid_t    id;
```

```
        union {
                symbol_t      as_symbol;
                TknSlice      as_word;
                int                   as_int;
                float         as_float;
                bool          as_bool;
        }                     data;
} Token;

typedef struct {
    Token*  items;
    size_t  count;
    size_t  capacity;
} Tokens;

typedef struct {
        char*         scratch_buffer;
    size_t           scratch_buffer_sz;

    // config or user options
    bool                 skip_newline;
    symbol_t                string_quotes[2];
    const char*          comment_block[2];

    // line tracking
    size_t               row;
    size_t               col;

    // string info
    const char*             target;
        size_t                      target_length;
        size_t                             position;

    // user input table
        TokenTableEntry*    token_table;
        size_t                      token_table_count;

    // memory for linear allocation of tokens
    // arena kind of
        Tokens                tokens;
} Tokenizer;



//
// PARSER
//
```

```
typedef enum {
    TI_NULL,
    TI_RANGE,
    TI_DOT,
    TI_COMMA,
    TI_PAREN_O,
    TI_PAREN_C,
    TI_SBRACK_O,
    TI_SBRACK_C,
    TI_CURLY_O,
    TI_CURLY_C,
    TI_COLON,
    TI_SEMICOLON,
    TI_QMARK,
    TI_EMARK,
    TI_MINUS,
    TI_PLUS,
    TI_MUL,
    TI_DIV,
    TI_EQUAL,
    TI_COMPARISON_EQUAL,
    TI_COMPARISON_NOT_EQUAL,
    TI_COMPARISON_GREATER,
    TI_COMPARISON_LESS,
    TI_COMPARISON_GREATER_EQUAL,
    TI_COMPARISON_LESS_EQUAL,
    TI_LIST_OPERATOR,
} TokenId;

typedef enum {
    NKP_IS_ADD      = 1,
    NKP_IS_MUL      = 2,
    NKP_HAS_UNARY   = 4,
    NKP_IS_EQALITY  = 8,
    NKP_IS_CMP_EQ   = 16,
    NKP_IS_CMP_GT   = 32,
} DtNodeKindProperties;

typedef char            byte;
typedef unsigned char   ubyte;

// TODO: make this slice one and only slice in this project
// TL;DR: remove TknSlice struct
typedef struct {
    const char*   data;
```

```
    size_t  length;
} Slice;

//
// Compile-time analysis.
//

enum {
    DT_PS_NONE, // error, probably.
    DT_PS_FUNCTION_SIGNATURE,
    DT_PS_VARIABLE,
};

typedef struct {
    String  name;
    size_t  argc;
    bool    has_return_value;
    bool    already_seen;
    bool    used_in_expression;
} DtParserFunctionSymbol;

typedef struct {
    DtParserFunctionSymbol *items;
    MapHead                 map_head;
} DtParserFunctions;

typedef enum {
    DT_PNKIND_ERROR,
    DT_PNKIND_OK,
    DT_PNKIND_VALUE,
} DtParseNodeKind;

typedef struct {
    bool negative_sign;
    bool foldable_constant;
} DtParseValue;

typedef struct {
    DtParseNodeKind     kind;
    size_t              instructions_generated;
    union {
        DtParseValue    value;
    } as;
} DtParseResult;


bool dtp_validate_function(DtParserFunctions* map, DtParserFunctionSymbol s);
```

```
void dtp_free_function_symbols(DtParserFunctions* s);
DtParserFunctions dtp_init_function_symbols(void);



typedef struct {
    int             options_mirror; // mirror of options from interpreter.
    StringBuilder   output;
    FILE*           tracef; // for tracing AST parsing in recursion.
    FILE*           errorf; // for logging errors from compilation.

    size_t      current;
    size_t      requested_tokens,
                parsed_tokens;
    // we buffer PTBS(size) of tokens to be able
    // to lookup tokens before and after the current one
    Tokenizer   lexer;
    Token       current_token;
    Token       token_buffer
                    [PARSER_TOKEN_BUFFER_SIZE];

    DtParserFunctions   function_symbols;
    Arena               allocator;
} DtParser;

bool            dtp_is_ok(DtParseResult r);
DtParseResult   dtp_ok(void);
DtParseResult   dtp_error(void);

#endif  //_DT_COMPILER_DECLARATIONS
```

ДОДАТОК Б

```
. . . FIB ITERATIVE JAVACRIPT . . .
function fibi(n) {
    last = 1
    prev = 0
    i = 0
    while (i < n) {
        console.log(last)
        next = prev + last
        prev = last
        last = next
        i = i+1
    }
}
fibi(40)
```

```
. . . FIB ITERATIVE LUA . . .
function fibi(n)
    last = 1
    prev = 0
    i = 0
    while i < n do
        print(last)
        next = prev + last
        prev = last
        last = next
        i = i + 1
    end
end
fibi(40)

. . . FIB ITERATIVE PYTHON . . .
def fibi(n):
    last = 1
    prev = 0
    i = 0
    while i < n:
        print(last)
        nxt = prev + last
        prev = last
        last = nxt
        i = i + 1
fibi(40)

. . . FIB RECURSIVE JAVASCRIPT . . .
function fib(n) {
    if(n <= 1) return n;
    return fib(n-1) + fib(n-2);
}

i = 1
while(i < 20) {
    res = fib(i)
    console.log(res)
    i = i + 1
}

. . . FIB RECURSIVE LUA . . .
function fib(n)
    if n <= 1 then return n end
    return fib(n-1)+ fib(n-2)
end
```

```
i = 1
while i<20 do
    res = fib(i)
    print(res)
    i = i + 1
end

. . . FIB RECURSIVE PYTHON . . .
def fib(n):
    if n <= 1: return n
    return fib(n-1) + fib(n-2)

i = 1
while i < 20:
    res = fib(i)
    print(res)
    i = i + 1

. . . COUNTER JAVASCRIPT . . .
var i = 0;
var n = 10000;
console.log("Printing numbers from ", i, " to ", n);

while (i < n) {
    console.log("i = ", i);
    i = i + 1;
}

. . . COUNTER LUA . . .
i = 0
n = 10000
print("Printing number from ", i, " to ", n-1)
while i < n do
    print("i = ", i)
    i = i + 1
end

. . . COUNTER PYTHON . . .
#!/usr/bin/python
i = 0
n = 10000
print("Printing numbers from ", i, " to ", n-1)
while i < n:
    print("i = ", i)
    i = i + 1
```

ДОДАТОК В

```
  [~/c/c/duct] › cat ./examples/fib_iterative.dt | ./ducti

### COMPILING ASSEMBLY ###
# Emmiting bytecode for 'write':
000000    load v@0
000001    wrt
000002    pop
    ARGS: [ v ]
> ROOT:
 > function:fibi
  > block:
   > statement:
    > variable 'last'
     > expr
            > value: 1
   > statement:
    > variable 'prev'
     > expr
            > value: 0
   > statement:
    > variable 'i'
     > expr
            > value: 0
   > statement:
    > while:
     > expr
            > value: i
       > compare |
            > value: n
     > block:
      > statement:
       > fncall 'print'
        > expr
              > value: last
        > expr
              > value: "\n"
      > statement:
       > variable 'next'
        > expr
              > value: prev
          > term |
              > value: last
```

```
      > statement:
       > variable 'prev'
        > expr
               > value: last
      > statement:
       > variable 'last'
        > expr
               > value: next
      > statement:
       > variable 'i'
        > expr
               > value: i
           > term |
               > value: 1
 > function:main
  > block:
   > statement:
    > fncall 'fibi'
     > expr
            > value: 40
GATHERED ASSEMBLY: ----
fn fibi n
    push 1
    store last
    push 0
    store prev
    push 0
    store i
    nop
    @label000000
    load i
    load n
    lt
    ijif 24
    pop
    load last
    wrt
    pop
    push "\n"
    wrt
    pop
    load prev
    load last
    add
    store next
    load last
    store prev
```

```
    load next
    store last
    load i
    push 1
    add
    store i
    jmp label000000
endfn
fn main
    push 40
    call fibi
    pop
endfn

----------

### COMPILING ASSEMBLY ###
# Emmiting bytecode for 'fibi':
    > found label: 'label000000 : 6'
000000    push i.1
000001    store last@1
000002    push i.0
000003    store prev@2
000004    push i.0
000005    store i@3
000006    nop
000007    load i@3
000008    load n@0
000009    lt
000010    ijif 24
000011    pop
000012    load last@1
000013    wrt
000014    pop
000015    push s(
)
000016    wrt
000017    pop
000018    load prev@2
000019    load last@1
000020    add
000021    store next@4
000022    load last@1
000023    store prev@2
000024    load next@4
000025    store last@1
000026    load i@3
```

```
000027    push i.1
000028    add
000029    store i@3
000030    jmp 6
    ARGS: [ n ]
# Emmiting bytecode for 'main':
000000    push i.40
000001    call fibi
000002    pop
    ARGS: [ ]
1
1
2
3
5
8
13
21
34
55
89
144
233
377
610
987
1597
2584
4181
6765
10946
17711
28657
46368
75025
121393
196418
317811
514229
832040
1346269
2178309
3524578
5702887
9227465
14930352
24157817
```

```
39088169
63245986
102334155
    Data stack pointer : 0

. . .
 [~/c/c/duct] › cat ./examples/fib.dt | ./ducti

### COMPILING ASSEMBLY ###
# Emmiting bytecode for 'write':
000000    load v@0
000001    wrt
000002    pop
    ARGS: [ v ]
> ROOT:
 > function:fib
  > block:
   > statement:
    > if:
     > expr
            > value: number
       > compare |
            > value: 1
     > block:
      > statement:
       > expr
              > value: number
   > statement:
    > expr
            > fncall 'fib'
             > expr
                    > value: number
                > term |
                    > value: 1
          > value: fib
       > term |
            > fncall 'fib'
             > expr
                    > value: number
                > term |
                    > value: 2
          > value: fib
 > function:main
  > block:
   > statement:
    > variable 'i'
     > expr
```

```
            > value: 1
    > statement:
     > while:
      > expr
            > value: i
        > compare |
            > value: 20
      > block:
       > statement:
        > variable 'res'
         > expr
                > fncall 'fib'
                 > expr
                        > value: i
               > value: fib
        > statement:
         > fncall 'print'
          > expr
                > value: res
          > expr
                > value: "\n"
        > statement:
         > variable 'i'
          > expr
                > value: i
             > term |
                > value: 1
GATHERED ASSEMBLY: ----
fn fib number
    load number
    push 1
    lte
    ijif 4
    pop
    load number
    ret
    jmp label000000
    pop
    @label000000
    load number
    push 1
    sub
    call fib
    load number
    push 2
    sub
    call fib
```

```
    add
    ret
endfn
fn main
    push 1
    store i
    nop
    @label000001
    load i
    push 20
    lt
    ijif 17
    pop
    load i
    call fib
    store res
    load res
    wrt
    pop
    push "\n"
    wrt
    pop
    load i
    push 1
    add
    store i
    jmp label000001
endfn

----------

### COMPILING ASSEMBLY ###
# Emmiting bytecode for 'fib':
    > found label: 'label000000 : 8'
000000    load number@0
000001    push i.1
000002    gt
000003    ijif 4
000004    pop
000005    load number@0
000006    ret
000007    jmp 8
000008    pop
000009    load number@0
000010    push i.1
000011    sub
000012    call fib
```

```
000013    load number@0
000014    push i.2
000015    sub
000016    call fib
000017    add
000018    ret
    ARGS: [ number ]
# Emmiting bytecode for 'main':
    > found label: 'label000001 : 2'
000000    push i.1
000001    store i@0
000002    nop
000003    load i@0
000004    push i.20
000005    lt
000006    ijif 17
000007    pop
000008    load i@0
000009    call fib
000010    store res@1
000011    load res@1
000012    wrt
000013    pop
000014    push s(
)
000015    wrt
000016    pop
000017    load i@0
000018    push i.1
000019    add
000020    store i@0
000021    jmp 2
    ARGS: [ ]
1
1
2
3
5
8
13
21
34
55
89
144
233
377
```

```
610
987
1597
2584
4181
```