МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

на тему: **«РОЗРОБКА ІНТЕРПРЕТАТОРА ВБУДОВУВАНОЇ СКРИПТОВОЇ МОВИ ПРОГРАМУВАННЯ З МОЖЛИВІСТЮ ВИЗНАЧЕННЯ ДОВІЛЬНОЇ ПРЕДМЕТНО-ОРІЄНТОВАНОЇ ГРАМАТИКИ»**

Виконав: студент 2 курсу, групи 8.1212-іпз-1
спеціальності 121 інженерія програмного забезпечення
(шифр і назва спеціальності)
освітньої програми інженерія програмного забезпечення
(назва освітньої програми)
К.В. Стахов
(ініціали та прізвище)
Керівник професор кафедри програмної інженерії, професор, д.т.н. Гоменюк С.І.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)
Рецензент завідувач кафедри фундаментальної та прикладної математики, професор, д.т.н. Гребенюк С.М.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя – 2023

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра програмної інженерії

Рівень вищої освіти магістр

Спеціальність 121 інженерія програмного забезпечення
(шифр і назва)

Освітня програма інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Завідувач кафедри програмної інженерії, к.ф.-м.н., доцент

_____ Лісняк А.О.
(підпис)

“ ____ ” _____ 2023 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Стахову Кирилу Віталійовичу
(прізвище, ім’я та по-батькові)

1. Тема роботи Розробка інтерпретатора вбудовуваної скриптової мови програмування з можливістю визначення довільної предметно-орієнтованої граматики

керівник роботи Гоменюк Сергій Іванович, д.т.н., професор
(прізвище, ім’я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 01 » травня 2023 року № 642-с

2. Строк подання студентом роботи 27.11.2023 р.

3. Вихідні дані до роботи 1. Постановка задачі.
2. Перелік літератури.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Постановка задачі.
2. Основні теоретичні відомості.
3. Розробка інтерпретатора вбудовуваної скриптової мови програмування з можливістю визначення довільної предметно-орієнтованої граматики

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)
презентація

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання 03.05.2023 р.

К А Л Е Н Д А Р Н И Й П Л А Н

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	15.05.2023	
2.	Збір вихідних даних.	09.06.2023	
3.	Обробка методичних та теоретичних	23.08.2023	
	джерел.		
4.	Розробка першого та другого розділу.	18.10.2023	
5.	Розробка третього розділу.	14.11.2023	
6.	Оформлення та нормоконтроль	20.11.2023	
	кваліфікаційної роботи магістра.		
7.	Захист кваліфікаційної роботи.	15.12.2023	

Студент _____ К.В. Стахов
(підпис) (ініціали та прізвище)

Керівник роботи _____ С.І. Гоменюк
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

Нормоконтролер _____ А.В. Столярова
(підпис) (ініціали та прізвище)

РЕФЕРАТ

Кваліфікаційна робота магістра «Розробка інтерпретатора вбудовуваної скриптової мови програмування з можливістю визначення довільної предметно-орієнтованої граматики»: 111 с., 106 рис., 7 табл., 31 джерело, 11 додатків.

АБСТРАКТНЕ ДЕРЕВО ВИРАЗІВ, ВБУДОВУВАНА МОВА ПРОГРАМУВАННЯ, ІНТЕРПРЕТАТОР, ПРЕДМЕТНО-ОРІЄНТОВАНА МОВА ПРОГРАМУВАННЯ, СИНТАКСИЧНИЙ АНАЛІЗ.

Об’єкт дослідження – механізми та техніки синтаксичного аналізу, проєктування мов програмування, розробки інтерпретаторів та їх вбудовування в існуючі програмні продукти.

Мета роботи: спроєктувати портативну скриптову мову програмування та реалізувати її інтерпретатор з можливістю визначення довільної граматики та вбудовування у існуючі програмні продукти.

Метод дослідження – аналіз літературних джерел.

У кваліфікаційній роботі розглядається процес проєктування скриптової мови програмування та процес розробки її інтерпретатора з можливістю розширення граматики та вбудовування до сторонніх програмних продуктів. Для синтаксичного аналізу коду інтерпретатором застосовано техніку Pratt-parsing. З використанням API прив’язок інтерпретатора реалізовано CLI для його зручного використання як самостійної середи виконання програм, написаних спроєктованою у даній роботі мовою програмування. Продемонстровано використання API вбудовування та прив’язки C++ функцій та класів на прикладі реалізації системи комп’ютерної алгебри та вбудованих класів інтерпретатора. Результати роботи можуть бути використані у будь-якому програмному продукті, розробленому мовою C++ для реалізації динамічного розширення або змінення його функціональності та контенту.

SUMMARY

Master’s qualifying paper «Development of the Embeddable Scripting Programming Language Interpreter with the Ability to Define Arbitrary Domain-Specific Grammars»: 111 pages, 106 figures, 7 tables, 31 references, 11 supplements.

ABSTRACT SYNTAX TREE, EMBEDDABLE PROGRAMMING LANGUAGE, INTERPRETATOR, DOMAIN-SPECIFIC PROGRAMMING LANGUAGE, EXPRESSION PARSING.

Object of the study – methods and techniques of expression parsing, programing language design, interpreter development and embedding into existing software products.

Aim of the study: to design a portable scripting programming language and implement an interpreter for it with ability to define arbitrary grammar and to embed it into existing software products.

Method of research – literature analysis.

This study examines the process of designing a scripting programming language and the process of developing its interpreter with the possibility of defining arbitrary grammars and embedding the interpreter into third-party software products. The interpreter uses the Pratt-parsing technique to parse the source code. With the use of the binding API of the interpreter, the CLI is implemented for its convenient use as a standalone execution environment for programs written in the programming language designed in this study. The use of the embedding API and C++ function and class binding APIs is demonstrated on the example of the implementation of a minimal computer algebra system and the built-in classes of the interpreter. The results of the study can be used in any software product developed in the C++ language to expand or change its functionality and content dynamically.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

Methan0l	Мова програмування, створена в результаті даної роботи
AST	Абстрактне синтаксичне дерево
Токен	Лексична одиниця внутрішнього представлення програми.
Лексер	Компонент інтерпретатора, що проводить лексичний аналіз вихідного коду, формуючи токени з символів або їх послідовностей
Парсер	Компонент інтерпретатора, що відповідає за синтаксичний аналіз токенів
Пунктуатор	Деякий символ вихідного коду, що може використовуватися для розділення між собою токенів
expr	Скорочення від «expression», тобто вираз
API	Прикладний програмний інтерфейс
CLI	Інтерфейс командного рядка
REPL	Цикл читання-обчислення-друку
stdout	Стандартний потік виводу
stdin	Стандартний потік вводу

ВСТУП

Вбудовувані інтерпретовані мови програмування мають широку сферу використання у розробці програмного забезпечення різноманітних типів та призначень, серед яких: системи комп'ютерної алгебри, ігрові движки, утиліти для автоматизації дій користувача, текстові редактори тощо. Необхідність таких мов програмування полягає в тому, що у наведених вище типах програмних продуктів часто виникає потреба у динамічному розширенні, зміненні або оновленні функціональності чи контенту без втручання у вихідний код.

Варто зазначити, що реалізація конкретного інтерпретатора деякої вузькоспеціалізованої предметно-специфічної мови програмування може вимагати значних матеріальних та часових витрат на первинну розробку та подальшу підтримку, тож цілком доцільним є використання загальної середи для реалізації інтерпретаторів довільних граматик, або використання вже існуючої вбудовуваної скриптової мови, просто прив'язуючи компоненти існуючого продукту до runtime-середовища інтерпретатора.

Таким чином, актуальність кваліфікаційної роботи обумовлена необхідністю у подібних інструментах у сфері розробки розширюваного та динамічно оновлюваного програмного забезпечення.

Робота складається з трьох розділів, в першому з яких розглядається граматика та процес проєктування скриптової мови програмування.

В другому розділі описані деталі реалізації інтерпретатора та детально розглянуто весь ланцюг операцій, що застосовуються до вихідного коду інтерпретатором для його виконання: від первинного читання тексту з файлу до лексичного та синтаксичного аналізу з формуванням абстрактного синтаксичного дерева з лексичних одиниць з подальшим обходом отриманого дерева для виконання виразів. Разом з цим було створено API для розробки динамічно завантажуваних модулів та бібліотек, прив'язок C++ функцій, функторів та класів разом з їх конструкторами, методами та перевантаженими

операторами.

В третьому розділі як демонстрацію використання отриманого інтерпретатора та його API з розширення лексики та граматики було створено CLI для виконання Methan0l-програм, мінімалістичну реалізацію REPL системи комп’ютерної алгебри та прив’язку сторонньої зовнішньої бібліотеки в якості динамічно завантажуваного модуля інтерпретатора. В якості прикладу вбудовування інтерпретатора в існуючі програмні продукти Methan0l було інтегровано у сторонній графічний редактор, написаний мовою програмування C++.

Інтерпретатор разом зі всіма його API реалізовано мовою програмування C++20 з використанням бібліотеки Boost. Для зборки було використано компілятор GCC 13 (та його порт від проєкту MinGW для зборки під ОС Windows) та систему зборки GNU Make 4.2.

1 МОВА ПРОГРАМУВАННЯ METHAN0L

Так як кінцевою метою роботи є розробка інтерпретатора мови Methan0l, спочатку буде розглянуто її граматику та раціоналізацію тих чи інших рішень, прийнятих під час її проєктування.

1.1 Загальний опис

Methan0l – це динамічно-типізована скриптова інтерпретована мова програмування загального призначення, націлена на вбудовуваність та портативність. Вона є мультипарадигмальною та дозволяє писати код у процедурній, структурній, функціональній та об’єктно-орієнтовній формах.

Елементарна виконавча одиниця мови Methan0l, що представлена деякою послідовністю токенів – це вираз. В даній мові відсутнє граматичне розділення на вирази та інструкції. Замість цієї концепції використовується розділення за наявністю батьківського виразу:

а) вирази верхнього рівня виконуються;

б) вирази, що мають батьківські вирази – обчислюються.

Таким чином, про вирази типу *а* можна сказати, що вони не виробляють ніяких результатів після виконання, а замість цього залишають деякий побічний ефект (вивід у stdout для оператору виводу, виконання одної з двох гілок виразів для операторів умовного переходу тощо).

З виразів типу *б* інтерпретатор обчислює деякі значення, які використовуються у батьківському виразі.

Цей аспект мови програмування Methan0l приводить до того, що всі функції в ній є функціями першого класу [1]. Це означає, що функції в Methan0l – це об'єкти, які можна передавати як аргументи, повертати як результати виклику інших функцій та зберігати в змінних.

В розроблюваній мові програмування також присутні традиційні імперативні оператори умовного переходу, циклів, механізми кидання та відловлювання винятків.

Розглянемо основні принципи проєктування, що були використані при створенні Methan0l.

Логічність та послідовність синтаксису. Якщо деяка послідовність токенів у одному контексті має значення *N*, то у будь-якому іншому контексті її граматичний сенс зберігається. В Methan0l відсутні контекстно-залежні синтаксичні конструкції, тобто всі вирази однозначні та послідовні. Як приклад контекстно-залежних виразів можна навести ключове слово *auto* мови програмування C++ [2], що виконує різну граматичну роль у різних виразах.

C++-центричність. Methan0l повинен бути вбудовуваним у C++ програми, що має вплив і на синтаксис, через що він проєктується в більшості як C-подібний [3], і на внутрішню реалізацію вбудованих типів та класів інтерпретатора.

Простота. Хоча Methan0l і є C-подібною мовою програмування, було прийнято рішення надати її синтаксису більш сучасного вигляду. Наприклад, кінці рядків не потрібно помічати крапками з комою, замість цього використовуються переноси рядків. Мову спроєктовано так, що технічно можливо записувати цілі програми навіть без переносів рядків. Також у традиційних виразах умовного переходу, циклів, обробки винятків, визначенні функцій дужки навколо дочірніх виразів є опціональними, як і операторні дужки навколо тіл цих виразів, якщо вони складаються тільки з єдиного виразу. Наявні й різні форми запису лямбда-функцій, що значно скорочують обсяг коду. Вибір такого мінімалістичного підходу до розробки мови програмування має свої очевидні переваги: полегшує навчання та розуміння коду, дозволяє швидше розробляти та відлагоджувати програми, а також зменшує ризик виникнення помилок через простоту синтаксису.

Безпечність відносно пам’яті. Так як Methan0l – інтерпретована мова програмування, було прийнято рішення не надавати користувачеві засобів для

прямого доступу до пам’яті. Такий принцип дозволяє ізолювати код, що виконується у runtime-середовищі інтерпретатора від ОС, під керуванням якої він працює. Це також дозволяє надійніше обробляти помилки, тому що з вільним доступом до пам’яті були б можливі випадки, коли стан самого інтерпретатора міг бути пошкоджений, що призвело б до його некоректної роботи.

1.2 Типи даних

До архітектури системи типів даних було обрано вже існуючий підхід. Для інтерпретованих мов програмування, особливо для AST-інтерпретаторів [4], яким є Methan0l, має сенс реалізувати дві категорії типів даних:

а) примітивні типи;

б) типи, значення яких знаходяться у купі.

До типів категорії *а* відноситимо цілі числа, числа з плаваючою точкою, булеві значення, символьний тип та відсутність значення. Це такі типи даних, значення яких можна напряму зберігати у таблицях даних, пов’язаних з зонами видимості або об’єктами.

До типів категорії *б* будемо відносити всі інші типи даних, значення яких занадто великі, щоб зберігати їх напряму. Замість цього змінні таких типів містять вказівник на дані з підрахунком посилань.

Одною з основних переваг цього підходу є те, що робота з примітивними значеннями значно швидша за роботу з об’єктами. Тому у коді, в якому ефективність критично важлива, треба якомога більше користуватися примітивами.

Однак, також існує і фундаментальний недолік такого підходу. Окрім того факту, що швидкість взаємодії інтерпретатора з об’єктами нижче, ніж з примітивами, у Methan0l відсутня можливість передавати примітивні значення як аргументи функцій за посиланням. Замість цього можна використовувати класи-обгортки, що мають єдине поле, в якому і буде зберігатися примітивне

значення. В стандартній бібліотеці Methan0l присутній такий клас і має назву *Mutable*. Також існують і окремі його спеціалізації для кожного примітивного типу даних – *MutableInteger*, *MutableFloat*, *MutableBoolean* тощо.

При вбудовуванні або прив’язуванні примітивних значень до C++ коду вони повністю сумісні з нативними типами для полегшення інтероперабельності між C++ та Methan0l.

В таблиці 1.1 наведено всі типи даних розроблюваної мови програмування.

Таблиця 1.1 – Типи даних

Назва	Опис	Синтаксис визначення
Nil	Тип, який має єдине можливе значення – nil. Описує відсутність значення.	nil
Int	64-бітне ціле число.	1234, 0xABC, 0b1010.
Float	64-бітне число з плаваючою точкою.	123.456
Boolean	Булеве значення.	true, false.
Character	Символьний тип.	‘a’
Reference	Посилання на інше значення. Не має ідентифікатора типу.	Немає
Unit	Звичайний юніт або бокс.	{expr1, expr2, ...}
Function	Будь-яка функція.	fun: a, b, ... {expr1, expr2, ...}
Object	Об’єкт деякого класу. Не має ідентифікатора типу.	new: ClassName(arg1, arg2, ...)
Expression	Не обчислений вираз (див. додаток А).	noeval: expr
Fallback	Будь-яке значення не вбудованого типу.	Немає

Тип *Reference* є внутрішнім типом інтерпретатора та повертається

обмеженою кількістю операторів, наприклад, операторами присвоєння або оператором індексування. Створити посилання на інше значення зсередини runtime-середовища Methan0l неможливо через вимоги до безпечності мови відносно доступу до пам’яті, тому що може виникнути ситуація завислого вказівника, після якої роботу інтерпретатора відновити буде вже неможливо.

Тип даних *Object* створено для внутрішнього розрізнення між об’єктами та не класовими значеннями. Сам по собі він не має ідентифікатора типу і тому невидимий з runtime-середовища інтерпретатора. Ідентифікатор типу будь-якого об’єкта – це ідентифікатор його класу.

Тип *Fallback* використовується для зберігання нативної частини об’єктів (якщо клас об’єкта не повністю визначений у runtime-середовищі інтерпретатора, а має нативну прив’язку) та для взаємодії Methan0l-коду з зовнішніми модулями.

Окрім розділення на примітиви та типи, що зберігаються у купі, типи даних також поділяються на:

а) класові;

б) не класові.

Всі типи, наведені у таблиці 1.1 є не класовими, тобто їх значення не є об’єктами деяких класів.

Methan0l також має декілька стандартних класів. Ці типи відносяться до категорії *б* і кожному з них відповідає деякий клас у runtime-середовищі Methan0l. У таблиці 1.2 наведено всі стандартні класи, наявні у проєктованій мові програмування.

Таблиця 1.2 – Стандартні класи

Назва	Опис	Синтаксис створення
String	Рядковий тип.	“abc” new: String(string) $”format”, arg1, ...
List	Динамічно-розширюваний масив.	[expr1, expr2, ...] new: List(list)
Set	Множина елементів без дублікатів.	set: expr1, expr2, ... new: Set(list)

Продовження табл. 1.2

Назва	Опис	Синтаксис створення
Map	Асоціативний контейнер з парами ключ-значення.	map: key1 => value1, ... new: Map(map)
Buffer	Байтовий буфер.	new: Buffer()
Pair	Пара значень.	new: Pair(expr1, expr2)
IntRange, FloatRange	Деякий числовий проміжок, виражений арифметичною прогресією.	start..end..step start..end
File	Файловий клас.	new: File(path)
Random	Генератор псевдовипадкових чисел.	new: Random([seed])

1.3 Базовий синтаксис

Синтаксис розроблюваної мови програмування Methan0l включає в себе всі традиційні конструкції С-подібних мов програмування. Розглянемо їх види.

Символьні оператори. До цього виду операторів в основному відносяться арифметичні та логічні оператори, які мають ідентичний іншим С-подібним мовам сенс та поведінку. Вони поділяються на префіксні, інфіксні та постфіксні.

Префіксні оператори записуються перед єдиним виразом-операндом. Інфіксні накладають деяку операцію на два операнди і записуються між ними, а постфіксні застосовуються до єдиного операнду та записуються після нього.

З кожним оператором асоційовано деяке значення пріоритетності (див. додаток Б), яке описує ступінь його зв’язаності з операндом.

Для класів існує підтримка перевантаження майже всіх символьних операторів. Даний механізм буде розглянуто пізніше.

У таблиці 1.3 детально описано всі символьні оператори в Methan0l.

Таблиця 1.3 – Символьні оператори

Категорія	Позначення	Опис
Інфіксні	+, -, *, /, %.	Базові арифметичні оператори. Результат обчислення розширюється до числа з плаваючою точкою, якщо хоча б один з операндів є значенням цього типу.
Інфіксні	<, >, <=, >=.	Арифметичні оператори порівняння.
Інфіксні	==, !=.	Оператори порівняння рівності.
Інфіксні	<<, >>, &, |, ^.	Оператори бітової арифметики: зсув ліворуч, праворуч, бітові AND, OR та XOR.
Інфіксні	&&, ||, ^^.	Булеві оператори AND, OR та XOR.
Префіксний	!	Оператор булевого заперечення.
Префіксний	~	Оператор бітового заперечення.
Префіксні або інфіксні	++, --.	Оператори збільшення / зменшення на один.
Інфіксний	=	Оператор присвоювання. Обчислюється справа наліво. Ліва частина виразу повинна бути ідентифікатором або мати тип посилання.
Інфіксний	<-	Інвертований оператор присвоювання. Обчислюється зліва направо.
Інфіксний	:=	Конвертуючий оператор присвоювання. Конвертує правий операнд у тип лівого перед присвоєнням.
Інфіксний	::	Оператор рядкової конкатенації. Якщо хоч один з операндів не є рядком, виконується конвертація, тому результат обчислення цього оператора завжди рядкового типу.

Продовження табл. 1.3

Категорія	Позначення	Опис
Інфіксний	.	Оператор доступу. Використовується для посилання на поля та методи об’єктів або на вміст боксів.
Інфіксний	@	Оператор статичного доступу. Використовується для посилання на статичні методи класів.
Префіксний	%%	Оператор виводу. Конвертує результат обчислення правого операнду у рядкове представлення та виводить його у stdout.
Префіксний	<%	Оператор виводу з перенесенням рядка.
Префіксний	%>	Оператор вводу. Визначає тип введеного значення та конвертує його у ціле число, число з плаваючою точкою або рядок перед присвоюванням до правого операнду.
Постфіксний	!	Оператор повертання значення.
Префіксний	$$	Оператор конвертації у рядкове представлення.

Ключові слова-оператори. Для випадків, коли є потреба описати деяку типову дію, але її неможливо дескриптивно асоціювати з послідовністю пунктуаторних токенів, в Methan0l використано концепцію ключових слів-операторів. При цьому, на відміну від, наприклад, ключових слів C++ [5], в Methan0l вони не є резервованими, тобто токен, що використовується для створення виразу ключового слова-оператора, може також бути використаним в якості ідентифікатора в іншому контексті. Це досягається шляхом введення вимоги наявності токена «:» після ключового слова.

Загальний синтаксис ключових слів-операторів в Methan0l можна описати як *operator: expr*. Окрім префіксних виразів цього типу, існують також і інфіксні, що виглядають як *expr1 operator: expr2*. У таблиці 1.4 наведено всі ключові слова-оператори, наявні в розроблюваній мові програмування.

Таблиця 1.4 – Ключові слова-оператори

Категорія	Позначення	Опис
Префіксний	typeid	Повертає числовий унікальний ідентифікатор типу значення правого операнду.
Префіксний	typename	Повертає назву типу правого операнду у вигляді рядка.
Префіксний	identity	Повертає адрес значення правого операнду у пам’яті.
Префіксний	return	Оператор повернення значення з юнітів та функцій.
Префіксний	copy	Оператор копіювання об’єктів. Повертає новий об’єкт того ж типу, що й правий операнд.
Префіксний	hash_code	Хешує значення правого операнду. Значення, що повертає цей оператор, не унікальні для кожного значення, але в загальному випадку можна сказати, що якщо *obj1 == obj2*, то *(hash_code: obj1) == hash_code: obj2*. Зворотна умова також істинна.
Префіксний	noeval	Повертає правий операнд у вигляді виразу, не обчислюючи його.
Префіксний	import	Імпортує зовнішні модулі, як нативні, так і написані мовою Methan0l.
Префіксний	new	Оператор створення об'єктів. Права частина повинна мати синтаксис *new: ClassName(arg1, arg2, ...)*, де *ClassName* – ім’я класу, об’єкт якого буде створено, а *arg1*, *arg2, ...*, аргументи, з якими буде викликано конструктор.
Префіксний	global	Імпортує посилання на ідентифікатор, який знаходиться поза поточною зоною видимості.

Продовження табл. 1.4

Категорія	Позначення	Опис
Інфіксний	assert	Кидає виняток, якщо правий операнд, конвертований у булеве значення дорівнює *false*. Повідомлення винятка береться з лівого операнду, конвертованого у рядок.
Інфіксний	is	Аналог *instanceof* мови Java. Повертає *true*, якщо значення лівої частини того ж типу, що й правої.
Інфіксний	to	Оператор конвертування. Працює тільки для примітивних типів.
Префіксний	var	Оператор явного визначення локальної змінної.
Інфіксний	require	Аналогічний оператору *is*, але кидає виняток замість повертання *false*.

Оператор умовного переходу. Даний оператор в Methan0l має традиційний вигляд, схожий на будь-який інший варіант цього оператора з будь-якої С-подібної мови.

Важливою відмінною рисою цього оператора є можливість обчислювати значення всього умовного блоку, фактично використовуючи його як більш потужну версію тернарного оператора.

Загальний синтаксис оператора умовного переходу зображено на рисунку 1.1.

```
if: condition_expr {
        ...
} else: {
        ...
}
```

Рисунок 1.1

Гілка *else* не обов’язково повинна бути присутня у *if-else* виразі.

При цьому якщо в *if* або *else* гілці наявний тільки один вираз, то операторні

дужки вказувати не треба.

Цикли. В Methan0l реалізовано три види циклів: *while*, *for* та *for-each*.

While цикли, аналогічно цьому типу циклів з C++ [6], виконують блок виразів, поки вказана умова при обчисленні має значення *true* (див. рис. 1.2).

```
while: condition_expr {
        ...
}
```

Рисунок 1.2

For цикли також повністю повторюють C++ *for* цикли [7] за поведінкою. Вони вимагають наявності виразу ініціалізації, де визначаються змінні, за якими будуть проводитися ітерації, вираз умови закінчення циклу та вираз, що модифікує змінні циклу після кожної ітерації (див. рис. 1.3).

```
for: init_expr, condition_expr, step_expr {
        ...
}
```

Рисунок 1.3

Для *for-each* циклів використано підхід, що дещо відрізняється від *range-for* циклів мови C++ [8]. В основі цього типу циклів у C++ є вимога до контейнерного типу, щоб для нього існували або вільні перевантаження функцій *begin()* та *end()*, сумісні з потрібним контейнером, або вони були визначені як члени контейнерного класу [9]. Ці функції повинні повертати ітератори однакового типу, що й дозволятимуть послідовно отримувати доступ до елементів.

В Methan0l же було прийнято рішення висунути більш конкретну вимогу до класів, що можуть бути використані у *for-each* циклі, а саме – клас повинен реалізовувати інтерфейс *Iterable*. Цей інтерфейс має єдиний метод, який треба визначити у класі-реалізаторі – *iterator()*. Значення, яке повертатиме цей метод повинне бути класового типу і реалізовувати інтерфейс *Iterator*. Ітератор, який

буде повернено даним методом, буде використовуватися *for-each* циклом для послідовної ітерації всіх елементів контейнеру.

Синтаксис *for-each* циклу зображено на рисунку 1.4.

```
for: name, iterable_expr {
        ...
}
```

Рисунок 1.4

Тут *name* – ім’я, під яким буде доступний поточний елемент ітерації, а *iterable_expr* – вираз, з якого має обчислюватися об’єкт, що реалізує інтерфейс *Iterable*.

Для переривання роботи циклу існує зарезервований ідентифікатор *break*, який використовується як операнд оператора *return*. Окрім *break*, в Methan0l існує набір інших зарезервованих ідентифікаторів (див. додаток В).

Обробка винятків. Як і в C++, в Methan0l наявна система винятків. При цьому вони використовують той самий механізм, що і нативні винятки – оператор *throw* мови програмування C++. Цей аспект прибирає потребу окремої логіки для обробки винятків runtime-середовища Methan0l та його нативних компонентів.

Синтаксис виразу обробки винятків зображено на рисунку 1.5.

```
try: {
        ...
}
catch: exception_name {
        ...
}
```

Рисунок 1.5

Якщо в блоці *try* буде кинуто виняток, потік виконання повернеться до *try-catch* виразу та перейде у гілку *catch*, присвоюючи *exception_name* об’єкт винятка, що був кинутий.

Зовнішні модулі. Окрім можливості вбудовування інтерпретатора Methan0l в існуючі програмні продукти, було також реалізовано систему розширення набору доступних класів та функцій за допомогою динамічно завантажуваних модулів, які можуть бути як нативними, так і написаними мовою Methan0l.

Після завантаження модуля він представляється бокс-юнітом, а його вільні функції та модульно-глобальні змінні додаються до вмісту цього юніта. При цьому класи, які реєструє модуль реєструються у менеджері класів (див. додаток Г) глобально для всього runtime-середовища інтерпретатора.

Існує два способи завантажити зовнішній модуль.

Перший з них – це використати функцію *load()*, вказавши в якості аргументу шлях до модуля або його назву. Ця функція поверне значення типу *Unit*, до вмісту якого можна отримати доступ за допомогою вже розглянутого синтаксису доступу до елементів боксів.

Другий спосіб – використати оператор *import*. На відміну від функції *load()*, цей оператор завантажує та одразу імпортує весь вміст отриманого модуля у зону видимості, де його було викликано.

1.4 Юніти

Другим ступенем виконавчої архітектури над окремими виразами розроблюваної мови програмування є юніти, що представляють з себе композитний вираз.

Юніт – це загальне поняття, що інкапсулює деяку сукупність декількох виразів разом з зоною видимості, до якої вони мають доступ. В загальному випадку вони поводяться аналогічно зонам видимості будь-якої іншої C-подібної мови програмування: при залишенні потоком керування юніту всі змінні, що були визначені в його зоні видимості, знищуються, або, у випадку об’єктів, знищується конкретне посилання на їх зміст, зменшуючи підрахунок посилань

на один.

Така загальність та універсальність юнітів обрана саме через принцип мінімалізму. При проєктуванні виконавчої архітектури мови найпріоритетнішим було добитися того, щоб якомога більше варіацій результуючої поведінки виконуваних програм можна було описати якомога меншою кількістю концепцій та синтаксичних конструкцій.

Зона видимості юнітів описується структурою, яку називатимемо таблицею даних. В реалізації інтерпретатора, наведеній у цій роботі, вона реалізована як хеш-таблиця (обгортка над *std::unordered_map*), де ключами є імена ідентифікаторів, а значеннями – їх вміст.

Будь-яка програма, що виконується інтерпретатором, також є юнітом. В такому випадку можна розглядати сукупність всіх виразів програми як корінь її абстрактного синтаксичного дерева.

Таким чином, юніти присутні у великій кількості синтаксичних конструкцій мови – від самостійного використання як окремого виразу до блочних виразів (оператор умовного переходу, цикли тощо), функцій та боксів.

Однак, юніти описують не тільки таке елементарне поняття, як зона видимості. Вони також можуть повертати деяке значення, спричиняти повертання з батьківських юнітів, а також мати розширений доступ до інших зон видимостей. Розглянемо ці аспекти детальніше.

За зміною напрямку потоку виконання після повертання значення юніти поділяються на:

– не несучі – повернене значення стає результатом обчислення юніт-виразу;
– несучі – повернене значення переноситься до батьківських юнітів як їх результат обчислення вгору за стеком виконання до тих пір, поки не буде зустрінуто перший не несучий юніт.

За доступом до зон видимості юніти поділяються на:

– сильні – можливий доступ лише до локальних ідентифікаторів; для того, щоб посилатися на змінну за межами локальної видимості, треба

використовувати спеціальний префікс перед її ім’ям – «#»;

– слабкі – можливий доступ як до локальних ідентифікаторів, так і до тих, що знаходяться за межами зони видимості вгору за стеком виконання до першого сильного юніта.

За строком життя зони видимості юніти можна поділити на:

а) звичайні – змінні у зоні видимості знищуються після покидання її потоком виконання;

б) стійкі – знищення змінних, визначених у зоні видимості, не відбувається.

Юніти типу *б* також називатимемо боксами. Для них виділено окрему категорію у граматиці Methan0l. Бокси представляють з себе аналог агрегатних структур мови C++ [10].

Даний тип юнітів визначається за допомогою оператору *box: {...}*, що робить юніт стійким та виконує його, після чого до вмісту його таблиці даних можна отримувати доступ за допомогою оператора «.». Наприклад, вираз *my_box.some_value* обчислюється інтерпретатором у значення поля *some_value* бокса *my_box*.

Бокси є важливою частиною мови Methan0l, тому що забезпечують можливість агрегувати набори довільних даних в одну структуру для її збереження, передавання та обробки. Також вони використовуються для надання доступу до вмісту динамічно завантажуваних модулів.

Синтаксис визначення юнітів. Юніти можуть бути використані і як частина блокового виразу (такого як оператор умовного переходу, цикл, функція тощо), і як самостійний вираз.

Розглянемо останній випадок. В Methan0l існує спосіб визначити сильні та слабкі юніти. Вміст сильних юнітів визначається всередині фігурних дужок (див. рис. 1.6).

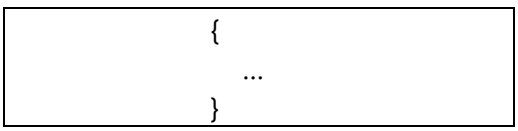

Рисунок 1.6

Слабкі юніти визначаються додаванням спеціального токена до синтаксису визначення сильних юнітів (див. рис. 1.7).

```
->{
   ...
}
```

Рисунок 1.7

При цьому контекст використання самостійного визначення юнітів має значення. Якщо юніт визначено синтаксисом виконання (як вираз верхнього рівня), то його буде виконано одразу. Таке використання цього виразу аналогічно виразу зони видимості з C++ [11], тобто воно дозволяє вибірково обмежувати час життя значень, що визначаються всередині зони видимості юніта (див. рис. 1.8).

```
...
{
        tmp_file = new: File("$:/tmp.dat")
        tmp_file.open()
        tmp_file.write_line("some data")
        tmp_file.close()
}
...
```

Рисунок 1.8

Цей код використовує підхід обмеження часу життя для змінної *tmp_file*, тобто вона буде знищена в момент покидання потоком виконання цього юніта.

Якщо ж юніт визначено синтаксисом обчислення (як частина іншого виразу, наприклад, присвоєння), то виконання не відбудеться, а з виразу буде обчислено значення типу *Unit*. Його можна викликати як будь-який інший викликуваний об'єкт (див. рис. 1.9).

Цей код визначає сильний юніт *print_info*, що отримує деяку інформацію та виводить її з заданим форматуванням (див. додаток Д). Цей юніт викликається у циклі кожну секунду.

```
print_info = {
        <% $"[{}] Info: {}", now(), get_some_info()
}
while: true {
        print_info()
        sleep(1000)
}
```

Рисунок 1.9

Також існує можливість викликати анонімний юніт одразу при визначенні, зберігаючи повернуте значення. Такий механізм може бути корисним для реалізації багатокрокових операцій, що мають проміжні значення та єдиний результат (див. рис. 1.10).

```
numbers = list: 1, 2, 3, 4, 5
sum = ->{
        result = 0
        for: (num, numbers) result += num
        return: result
}()
```

Рисунок 1.10

Ця програма визначить список деяких чисел та використає техніку, описану вище – створить юніт, що має тимчасову змінну *result*, в яку буде акумулюватися результат обчислення та яка буде повернута з юніта після закінчення роботи циклу. У кінці визначення юніта присутній оператор виклику без аргументів – «()», що й виконає описану логіку, після чого результат обчислення буде присвоєно змінній *sum*. При цьому і сам юніт, і тимчасову змінну *result* буде знищено після повернення значення у операції присвоєння.

Зони видимості. Всі змінні в Methan0l локальні за замовчуванням. Як приклад мови програмування зі зворотною архітектурою зон видимості можна привести Lua. В цій мові всі визначені змінні глобальні за замовчуванням поки не використано ключове слово *local* [12].

Для будь-якого сильного юніта в Methan0l, змінні, що визначені всередині його зони видимості – локальні для нього, а всі змінні поза зоною видимості –

глобальні. Для того, щоб отримати доступ до глобальних змінних можна використати один з двох механізмів:

– зазначити префікс «#» перед ім’ям ідентифікатора;
– використати оператор *global*, в якому вказати один або декілька глобальних змінних, посилання на які треба імпортувати у поточну зону видимості.

Для слабких же юнітів такого обмеження не існує – вони мають доступ і до змінних в середині зони видимості і поза нею.

Для експліцитного визначення локальних змінних (наприклад, для створення змінних у *if-else* виразах або циклах) існує оператор *var*. Він завжди створює змінні як локальні для поточної зони видимості, навіть якщо юніт є слабким і за його межами вже визначено змінну з ідентичним ім’ям.

1.5 Функції

Функції в Methan0l є викликуваними об’єктами, тобто їх можна віднести до категорії функцій першого класу. Реалізовано два типи функцій:

а) звичайні функції;
б) лямбда-функції.

Ці дві категорії мають лише один розрізняючий фактор – тип юніта-тіла функції. Для функцій категорії *а* тіло є сильним юнітом, а для категорії *б* – слабким. Це означає, що лямбда-функції мають доступ до значень поза їх зоною видимості без використання засобів експліцитного посилання на глобальні змінні, а звичайні функції – ні.

Для визначення звичайних функцій використовується ключове слово *fun* (див. рис. 1.11), аналогічно мові програмування Kotlin [13], а для лямбда-функцій існує слово *f*, що своєю довжиною символізує те, що частіше за все цей тип функцій використовується анонімно (без надання їм назви).

```
my_function = fun: x, y, ... {
        ...
}
```

Рисунок 1.11

У випадку коли потрібен порожній список аргументів, замість нього треба вказати порожні дужки: «()».

Також підтримуються аргументи за замовчуванням. Синтаксис їх визначення зображено на рисунку 1.12.

```
my_function = fun: x, y => 123, ... {
        ...
}
```

Рисунок 1.12

В цій функції аргумент *y* – опціональний. Якщо його не буде передано у списку аргументів під час виклику, йому буде присвоєно вказане значення за замовчуванням.

Також для функцій працює правило опціональності операторних дужок коли тілом функції є тільки один вираз.

Синтаксис визначення лямбда-функцій зображено на рисунку 1.13.

```
f: x, y, ... -> expr
```

Рисунок 1.13

Тут права частина виразу буде розглядатися так, наче вона є операндом оператора *return*, тобто результат обчислення лямбда-функції – це результат цього виразу і експліцитно вказувати цей оператор не треба.

Існує також і складений лямбда-синтаксис (див. рис. 1.14). Його було створено для випадків, коли лямбда-функція виконує деяку складну дію, для якої потрібно виконати декілька кроків.

В такому випадку оператор *return* не буде додано автоматично, тому якщо потрібно повертати значення, треба вказувати його експліцитно.

```
f: x, y, ... -> expr1, expr2, ...
```

Рисунок 1.14

Незважаючи на те, що імена функцій – це такі ж звичайні ідентифікатори, як і імена змінних, для виклику функцій використовується додатковий крок пошуку імені у таблицях даних, а саме – для виразу виклику функцій немає обмеження за зоною видимості, тобто для функцій, які знаходяться поза зоною видимості сильного юніта, префікс «#» або використання оператора *global* не потрібні.

1.6 Класи

В Methan0l наявна система класів з наслідуванням, інтерфейсами та перевантаженням операторів. Загальний синтаксис визначення класів зображено на рисунку 1.15.

```
class: ClassName {
        member_name1 => expr1
        member_name2 => expr2
        ...
}
```

Рисунок 1.15

Тут *member_name* – ім’я будь-якого члена класу (методу або поля), а після токену «=>» записується значення, яке відповідний член повинен мати.

В класах Methan0l відсутня інкапсуляція, тому поля доступні з будь-якого контексту: як зсередини методів класу, так і з зовнішнього коду.

Методи. Методи визначаються вже розглянутим раніше синтаксисом визначення функцій, але замість токену *fun* використовується токен *method* (див. рис. 1.16).

Для доступу до членів класу зсередини його методів потрібно

використовувати ідентифікатор *this*. Аналогічно механізму виклику методів в C++ [14, 15], в розроблюваній мові програмування *this* – це імпліцитний аргумент, який під час виклику передається першим перед усіма іншими визначеними аргументами.

```
class: Printer {
        text
        set_text => method: text {
                this.text = text
        }
        print => method: () {
                <% this.text
        }
}
printer = new: Printer()
printer.print()
printer.set_text("Hello, World!")
printer.print()
```

Рисунок 1.16

Вивід програми зображено на рисунку 1.17.

```
nil
Hello, World!
```

Рисунок 1.17

Даний код визначить клас *Printer*, що містить єдине поле *text*, яке буде дорівнювати *nil* за замовчуванням після створення об'єкта. Клас *Printer* також містить два методи, один з яких встановлює значення поля *text*, а інший виводить його у stdout. Після визначення класу створюється його об'єкт та робиться виклик методу *print()*, що виведе «nil» тому що поле *text* не було ініціалізовано. Після цього викликається метод *set_text* з деяким рядком та відбудеться вивід у stdout за допомогою методу *print*. Цього разу буде виведено «Hello, World!».

Конструктори. Конструктори класів визначаються як метод з ім'ям *construct* (див. рис. 1.18). При створенні об'єкта класу через оператор *new*, цей метод буде викликаний з тими аргументами, які були присутні у виразі *new*.

```
class: Point {
        x, y
        construct => method: x, y {
                this.x = x
                this.y = y
        }
}
```

Рисунок 1.18

Таким чином, в результаті, наприклад, виразу *new: Point(1, 2)* буде створено об’єкт вищеописаного класу, полям *x* та *y* якого у конструкторі буде присвоєно значення 1 та 2.

Перевантаження операторів. Для використання даного механізму необхідно задати токен оператора замість імені методу, після чого визначити перевантаження як звичайний метод. При цьому для операторів індексування існує окремий механізм перевантаження (див. додаток Е). Наприклад, перевантаження оператора додавання наведено на рисунку 1.19.

```
class: Complex {
        ...
        `+` => method: rhs {
                this.re += rhs.re
                this.im += rhs.im
                return: this
        }
        ...
}
```

Рисунок 1.19

Інтерфейси. При створенні моделі ООП в розроблюваній мові програмування, враховуючи принцип мінімалістичного дизайну, було вирішено не вводити дистинкцію між абстрактними класами та інтерфейсами, як це, наприклад, зроблено у Java [16]. Замість цього в Methan0l існує єдина концепція інтерфейсів, яку можна описати як клас, що не має стану. При цьому інтерфейси можуть визначати методи, які будуть наслідувані класом-реалізатором по аналогії з методами за замовчуванням у Java 8 [17].

Синтаксис визначення інтерфейсу відрізняється від визначення класів тільки ключовим словом (див. рис. 1.20).

```
interface: InterfaceName {
        ...
}
```

Рисунок 1.20

Абстрактні методи інтерфейсів визначаються через вказування їх імен та прирівнювання їх значення до *nil* (див. рис. 1.21).

```
interface: Clickable {
        click => nil
}
```

Рисунок 1.21

Приклад використання методу за замовчуванням наведено на рисунку 1.22.

```
interface: Clickable {
        click => nil
        click_n_times => method: n {
                for: (i, 1..n) this.click()
        }
}
```

Рисунок 1.22

Нехай інтерфейс *Clickable* описує можливість натискання на об’єкт класу-реалізатора. Абстрактний метод *click* тоді повинен бути реалізований в конкретному класі, а метод *click_n_times* буде їм наслідувано. При виклику цього метода конкретну реалізацію *Clickable@click* буде викликано *n* разів.

Наслідування. Модель наслідування в Methan0l натхнена наслідуванням у мові Java [18], тобто кожним класом може бути наслідувано тільки один суперклас, але при цьому може бути реалізовано довільну кількість інтерфейсів.

Синтаксис визначення суперкласу для деякого дочірнього класу зображено на рисунку 1.23.

```
class: Derived base: Base {
        ...
}
```

Рисунок 1.23

Тут *Derived* – дочірній клас суперкласу *Base*.

Дочірні класи наслідують всі методи та стан батьківських класів. При цьому конструктори також наслідуються, що значить, що у випадку коли конструктор дочірнього класу не перевизначено, при створенні об’єкта буде викликано конструктор суперкласу.

Для того, щоб отримати доступ до батьківського конструктора при перевизначенні у дочірньому, створено спеціальне поле *super*, яке посилається на супер-конструктор (див. рис. 1.24). Виклик батьківського конструктора при перевизначенні є обов’язковим, якщо потрібно, щоб батьківська частина стану об’єкта була ініціалізована. Інакше інтерпретатор не викликає супер-конструктор автоматично.

```
class: Derived base: Base {
        data
        construct => method: data {
                this.super(data.size())
                this.data = data
        }
}
```

Рисунок 1.24

Реалізація інтерфейсів має схожий на наслідування синтаксис, але дозволяє вказувати декілька супер-інтерфейсів після ключового слова *interface*.

Наприклад, визначення класу деякого віджета смуги прокрутки виглядала б наступним чином (див. рис. 1.25).

```
class ScrollBar base: Widget implements: Clickable, Draggable {
        ...
}
```

Рисунок 1.25

В цьому випадку смуга прокрутки *ScrollBar* наслідує клас *Widget*, що може описувати довільний елемент графічного інтерфейсу. В якості інтерфейсів цей клас реалізує *Clickable* та *Draggable*, що відображає можливість перетягування смуги прокрутки на натискання на неї.

Анонімні об’єкти. Окрім боксів в Methan0l також було реалізовано систему анонімних об’єктів, тобто об’єктів, що не мають класу. Вони, як і бокс-юніти, дозволяють агрегувати декілька значень в один об’єкт, але на відміну від них, також дозволяють визначати методи, що дає можливість додавати і стан, і поведінку у тимчасові агрегатні значення.

Синтаксис створення анонімного об’єкта зображено на рисунку 1.26.

```
object = new: @{
        ...
}
```

Рисунок 1.26

Анонімні об’єкти також можуть бути використані в якості прототипів [19], тобто при вказуванні анонімного об’єкта у правій частині оператора *new* буде створено його копію, після чого буде викликано конструктор.

2 РЕАЛІЗАЦІЯ ІНТЕРПРЕТАТОРА

Загальний обсяг коду реалізації інтерпретатора Methan0l складає близько 20 000 рядків. Тому в цьому розділі розглянемо найголовніші елементи архітектури інтерпретатора, засобів для прив'язок зовнішніх нативних класів та вбудовування у існуючі C++ проєкти. Повний вихідний код інтерпретатора, стандартної бібліотеки мови програмування Methan0l та зовнішніх модулів доступні у GitHub репозиторії проєкту [20].

Найголовнішими компонентами інтерпретатора є лексер та парсер, завдяки яким відбувається первинна обробка вихідного коду перед виконанням програми. На рисунку 2.1 зображена загальна архітектура цього процесу.

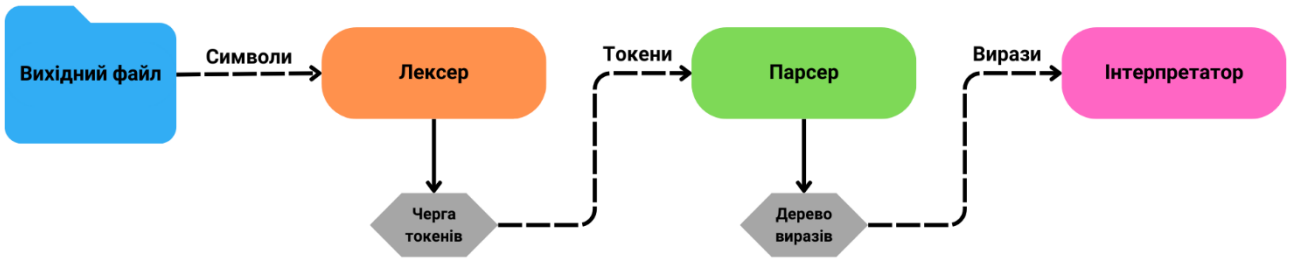

Рисунок 2.1

Розглянемо більш детально реалізацію окремих компонентів інтерпретатора, зображених на цій схемі.

2.1 Лексичний аналіз

Першим етапом обробки вихідного коду програми перед її виконанням є лексичний аналіз, або лексинг. В ході цього процесу лексер за заданими правилами відокремлює лексеми з вихідного коду, перетворюючи їх на лексичні токени, які один за одним додаються в чергу, з якої на наступному етапі їх можна буде отримувати для подальшої обробки.

Лексемами називають послідовності конкретних символів, що складають деяку лексичну одиницю. Токеном називатимемо лексему, асоційовану з її лексичним типом таким як число, ідентифікатор, рядок тощо.

Структура лексичної одиниці. Клас, що описує один токен, назвемо *Token* та визначимо його у просторі імен *mtl*, як і всі подальші компоненти інтерпретатора.

Розглянемо приватні члени класу *Token* (див. рис. 2.2).

```
class Token
{
    private:
        TokenType type = Tokens::NONE;
        uint32_t line = 0;
        uint32_t column = 0;
        std::string value;
        Separator sep = Separator::NONE;
    public:
            ...
}
```

Рисунок 2.2

Першим членом класу визначено його тип, описаний класом *TokenType*. Замість підходу зі статичним визначенням видів лексичних одиниць через, наприклад, *enum*, в Methan0l використано більш гнучку систему, завдяки якій існує можливість повністю перевизначити набір лексичних токенів.

Структура *TokenType* містить всього два публічних члени: об’єкт *std::string_view name*, що відображає ім’я типу та примітив *uint64_t id*, що при створенні набуває значення хешу назви. Конструктор цієї структури визначено як *constexpr*, тому її значення можна використовувати у compile-time контексті. Як приклад такого використання можна привести можливість застосування об’єктів *TokenType* як гілок у інструкції *switch* через доступ до членів *id*, що не можна робити з runtime-типами.

Другий та третій члени *line* та *column* описують місце у вихідному файлі, де починається описуваний токен.

Четвертий член *value* містить конкретну лексему лексичного типу *type*. Наприклад, для чисел це набір цифр, для ідентифікаторів – їх назва, для рядків – їх вміст тощо.

Останній член цього класу *sep* має тип перелічення та описує спосіб відділення цього токену від попереднього у черзі. Токен може не відділятися від попереднього (*Separator::NONE*), відділятися пробілом (*Separator::SPACE*), або перенесенням рядка (*Separator::NEWLINE*). Ця інформація потрібна для уникнення неоднозначності синтаксису у деяких випадках.

Методи класу *Token* в основному тільки надають доступ до його членів та містять логіку переведення у рядкове представлення.

Лексер. Загальну реалізацію лексера описано класом *Lexer*. Він оброблює рядкове представлення вихідного файлу перетворюючи його на токени, які послідовно додаються в чергу для подальшого використання парсером.

Розглянемо категорії токенів, які розрізняє лексер:

– ціле число;
– число з плаваючою точкою;
– булеве значення;
– ідентифікатор;
– рядок;
– символ;
– пунктуаторний оператор;
– ключове слово.

Розглянемо тепер деякі з наведених категорій більш детально.

Цілі числа можуть бути записані у трьох різних форматах – у десятковій, шістнадцятковій (*0xABC*) та двійковій (*0b1010*).

Ідентифікаторами вважаються абетково-цифрові комбінації символів, які не починаються з цифри. Вони також можуть містити не абетковий символ «_».

Рядки задаються між двома двійними лапками та підтримують багаторядковий текст. Також підтримуються всі керуючі послідовності та екранування лапок таким же чином, як в C++ [21].

Символи задаються між двома одинарними лапками та також мають підтримку керуючих послідовностей.

Пунктуаторними операторами називатимемо оператори, які повністю складаються з не абетково-цифрових символів.

Перед тим, як розглянути використаний алгоритм лексичного аналізу, введемо декілька важливих понять.

Пунктуатором називатимемо деякий символ, який відділяє лексичні одиниці одну від одної. Найчастіше це будь-який не абетково-цифровий символ (за винятком «_»).

Введемо також дистинкцію між слабкими та сильними ключовими словами. Слабкими ключовими словами називатимемо такі ідентифікатори, які є зарезервованими тільки у деякому контексті. Сильні ж ключові слова є зарезервованими у будь-якому контексті і не можуть бути ім’ям змінної, класу тощо.

Сепаратором називатимемо тип роздільника токена (див 2.1.1).

Точкою входу у процес лексингу є головний метод класу *Lexer – lex(const std::string &code, bool preserve_state = false)*. Він приймає посилання на рядок, що містить код програми, та опціональний булевий параметр, що вказує, чи потрібно проводити очищення стану лексера перед початком процесу (див. рис. 2.3).

```
void Lexer::lex(std::string &code, bool preserve_state)
{
    cur_chr = code.begin();
    input_end = code.end();
    reset(!preserve_state);
    while (has_next())
        consume_and_deduce();
    push();
    if (!preserve_state)
        tokens.push(Token::EOF_TOKEN);
}
```

Рисунок 2.3

Метод *consume_and_deduce* відповідає за акумуляцію, аналіз чергової лексеми та створення токена з додаванням його до черги (див. рис. 2.4). Токен вважається завершеним, якщо лексер натрапляє на пунктуаторний символ.

```
void Lexer::consume_and_deduce()
{
    if (toktype == Tokens::NONE)
        begin(*cur_chr);
    else
        consume();
    deduce_separator();
    next_char();
}
```

Рисунок 2.4

Метод *deduce_separator* перевіряє, яким сепаратором було відокремлено поточний символ від попереднього, і оновлює на основі цього сепаратор всього токена.

Метод *next_char* обирає наступний символ вхідного рядка коду як поточний.

Формування чергового токена методом *consume_and_deduce* відбувається у три етапи:

а) первинне припущення;

б) коректування припущення згідно з наступними символами;

в) фіналізація токена.

На першому етапі лексер не має жодної інформації про поточний токен для аналізу, тому поточний символ, оброблюваний лексером, вважається першим символом нового токену. На основі цього символу у методі *begin* робиться початкове припущення про тип лексичної одиниці, яка аналізується. Розглянемо цей метод.

Спочатку символ аналізується на предмет того, чи є він початком рядка або форматованого рядка (див. рис. 2.5).

Метод *save* тут додає символ до поточної лексеми.

```
void Lexer::begin(char chr)
{
    cur_int_literal = IntLiteral::NONE;
    if (chr == Tokens::LIST && match_next(Tokens::QUOTE)) {
        toktype = Tokens::FORMAT_STRING;
        next_char();
        begin(*cur_chr);
    }
    else if (chr == Tokens::QUOTE || chr == Tokens::SINGLE_QUOTE) {
        if (toktype != Tokens::FORMAT_STRING)
            toktype = (chr == Tokens::QUOTE)
                                    ? Tokens::STRING : Tokens::CHAR;
        save(chr);
    }
    ...
```

Рисунок 2.5

Далі робляться припущення щодо того, чи є поточний токен пунктуаторним оператором (див. рис. 2.6).

```
else if (!std::isalnum(chr) && is_punctuator(chr)) {
        char next = look_ahead();
        if (is_punctuator(next))
            if (try_save_multichar_op(chr, next))
                return;
        push(chr);
        clear();
        return;
    }
    ...
```

Рисунок 2.6

Метод *try_save_multichar_op* перевіряє, чи є послідовність символів мульти-символьним пунктуаторним оператором, а метод *push* одразу фіналізує токен та додає його до черги, якщо аналізований пунктуаторний оператор складається тільки з одного символу. Після фіналізації токена відбувається очищення поточного стану аналізу за допомогою метода *clear*, після чого лексер може перейти до аналізу наступного токена починаючи з першого етапу.

Після цього йде припущення, чи є поточний токен числом. На першому кроці ми ще не можемо знати, яке саме число описує цей токен – ціле чи з

плаваючою точкою, записане у десятковій формі або у шістнадцятковій тощо. Тому для числових токенів на першому етапі лексер завжди вважає, що перед ним ціле число.

Останнім припущенням є перевірка на відповідність поточного символу на дозволеність в іменах ідентифікаторів (див. рис. 2.7).

```
    else if (std::isalpha(chr) || chr == Tokens::UNDERSCORE) {
        toktype = Tokens::IDENTIFIER;
        save(chr);
    }
}
```

Рисунок 2.7

Якщо символ задовольняє цю умову, лексер вважає, що поточний токен це ідентифікатор.

Перейдемо до другого етапу аналізу лексем, що відбувається у методі *consume*. Логіка, за якою коректується тип поточного токена, схожа на логіку початкового припущення, тож розглянемо тільки найменш тривіальні випадки.

Під час аналізу цілих чисел спочатку перевіряється умова, чи відомий в даний момент формат, в якому записане число. Якщо вона задовольняється, то акумулювання символів відбувається звичайним чином без подальшого аналізу. Якщо ж формат числа ще не відомий, то відбувається його визначення. Формат запису чисел визначено типом перелічення *IntLiteral*, який визначає всі підтримувані формати запису цілих чисел, а саме: *NONE* (невідомий), *HEX* (шістнадцятковий), *BIN* (двійковий) та *DEC* (десятковий).

Пробіл виконує роль пунктуатора (викликає фіналізацію поточного токена) для будь-яких токенів окрім чисел та рядків. При цьому звичайні пунктуаторні символи завжди приводять до фіналізації поточного токена.

При збереженні рядкових токенів також аналізуються контрольні послідовності (наприклад \n, \t, \0). Через те, що в C та C++ вони не існують у вигляді двох символів під час роботи програми, а одразу перетворюються на сам контрольний символ під час компіляції, виконується заміна двосимвольних контрольних комбінацій на контрольні байти.

Ідентифікатори аналізуються простим акумулюванням абетко-цифрових символів поки не буде зустрінуто пунктуатор.

На етапі фіналізації токенів лексер ігнорує символи точки з комою та перенесення рядка. Таким чином, з точки зору лексичного аналізу вони ідентичні.

Розглянутий механізм застосовується до кожного символу вихідного коду поки не буде проаналізовано всі з них. В результаті роботи лексера буде отримано чергу лексичних токенів в порядку їхньої появи у вихідному коді.

Незважаючи на те, що в класі *Lexer* реалізовано набір правил, за якими з наборів символів створюються лексичні одиниці, в цьому класі не визначені безпосередньо конкретні лексичні ролі комбінацій символів. Замість цього клас *Lexer* надає набір методів для реєстрації лексем за заданими лексичними категоріями (див. рис. 2.8).

```
void register_punctuator(TokenType punctuator);
void register_keyword(TokenType keyword, bool hard = false);
void register_operator(TokenType op);
void register_block_begin_token(TokenType tok);
void register_block_end_token(TokenType tok);
```

Рисунок 2.8

Реєстрація лексичних категорій повинна бути проведена перед початком лексингу, тому найкращим підходом є наслідувати клас Lexer та реєструвати всю потрібну для конкретної мови лексику у конструкторі. Саме такий підхід використовує і мова Methan0l, лексичні категорії токенів якої визначено у класі *Methan0lLexer*.

2.3 Синтаксичний аналіз

Наступним кроком після лексингу є синтаксичний аналіз, або парсинг. В загальному випадку синтаксичний аналіз представляє з себе групування

декількох токенів з черги лексера в окремі вирази, які залежать від деяких інших виразів, утворюючи абстрактне синтаксичне дерево програми. На рисунку 2.9 продемонстровано приклад дерева виразів деякої програми.

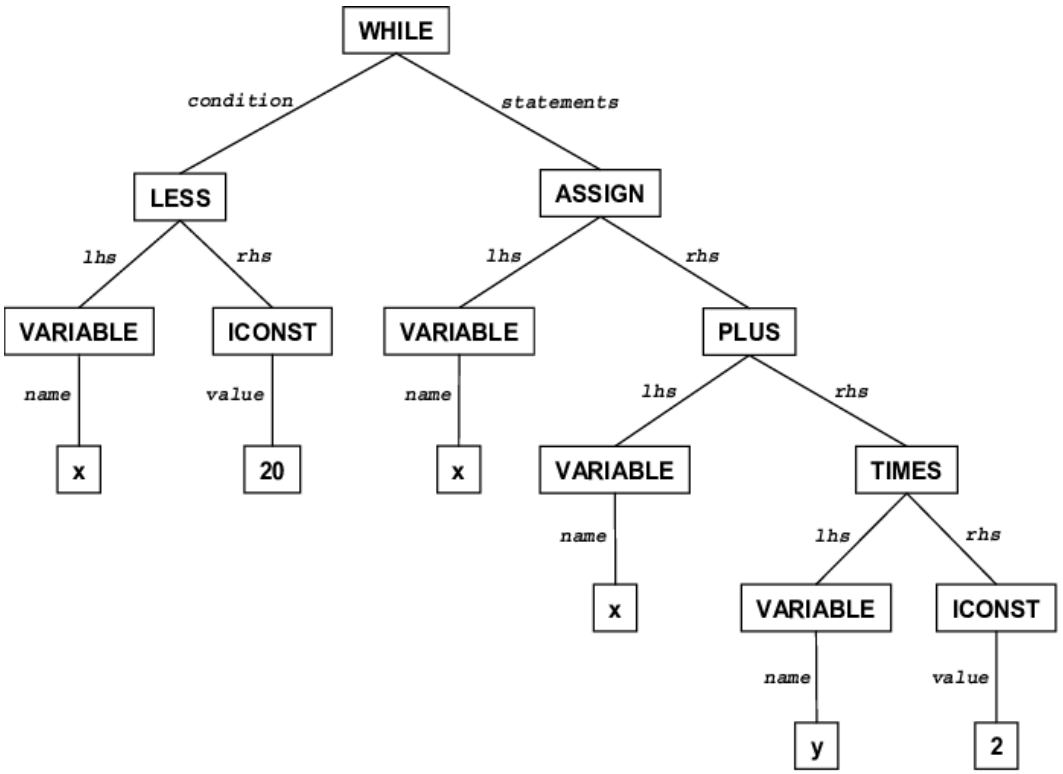

Рисунок 2.9

На даному рисунку зображено синтаксичне дерево, кореневим виразом якого є цикл *while*. Дочірніми для цього виразу є умова виконання циклу ліворуч від батьківського виразу та тіло циклу праворуч від нього. Як можна побачити, умова в даному випадку може бути записана як $x < 20$, а тіло циклу – як $x = x + y * 2$.

Вирази. Перш ніж перейти до реалізації парсера, ознайомимося із основною структурною одиницею внутрішнього представлення програм в Methan0l – виразами та їх базовим класом *Expression*.

Основні члени класу *Expression* зображено на рисунку 2.10.

Даний клас має всього одне поле *line*, яке описує номер рядка вихідного файлу, де було зустрінуто вираз. В основному випадку це потрібно для звітування про помилки.

```
class Expression
{
    private:
        uint32_t line = 0;
    public:
        virtual Value evaluate(Interpreter &context) = 0;
        virtual void execute(Interpreter &context);
    ...
};
```

Рисунок 2.10

Метод *evaluate* є чистим віртуальним методом, тобто конкретні класи виразів повинні визначити його поведінку. Як слідує з назви, цей метод викликається під час обчислення значення виразу. Метод *execute* же викликається під час виконання виразу і має реалізацію за замовчуванням, що полягає у виклику метода *evaluate* з ігноруванням повернутого значення. Обидва з цих методів приймають посилання на контекст інтерпретатора, що має тип *Interpreter*. Це необхідно для будь-якої взаємодії з runtime-середовищем.

Приклад реалізації одного з стандартних виразів наведено у додатку Ж.

Парсер. Для реалізації парсера було обрано та дещо модифіковано алгоритм синтаксичного аналізу, що має колоквіальну назву «Pratt-parsing», але також відомий як «Top Down Operator Precedence Parsing» (парсинг операторів зверху вниз за пріоритетом) [22].

Однією з ключових особливостей «Pratt-parsing» є те, що він дозволяє обробляти різні рівні пріоритетів виразів без необхідності використання таблиць пріоритетів. Кожний вираз має свій власний обробник, який визначає його пріоритет і виконує необхідні дії при знаходженні цього виразу у черзі токенів. Називатимемо такі класи-обробники, що відповідають за парсинг конкретного виразу, суб-парсерами.

Операторами в термінології «Pratt-parsing» називаються будь-які «якірні» токени, при виявленні яких парсер викликає відповідний суб-парсер.

Робота цього алгоритму базується на рекурсивному спуску і відбувається зверху вниз, починаючи з кореневого виразу. Даний алгоритм враховує пріоритет операторів під час аналізу виразу. Ключовим аспектом є те, що парсер

обробляє токени виразу один за одним, доки не зустрічає токен, який має пріоритет оператора, більший або рівний поточному пріоритету. В такому випадку поточний вираз вважається закінченим.

Кожний оператор може мати два різних пріоритети: лівий та правий. Лівий пріоритет використовується при обробці префіксних виразів, тоді як правий пріоритет використовується при обробці інфіксних та постфіксних виразів.

Завдяки цьому методу можливо ефективно обробляти вирази довільної складності з різними рівнями вкладеності та різними типами операторів.

Реалізація парсера. Механізм синтаксичного аналізу методом «Pratt-parsing» та перетворення черги токенів на абстрактне синтаксичне дерево, що складається з виразів різних типів реалізовано у класі *Parser*, який, як і клас *Lexer* реалізує лише логіку синтаксичного аналізу, але не визначає жодних синтаксичних конструкцій. Як і в попередньому випадку, це повинно бути зроблено в конструкторі класу, що реалізує парсер конкретної мови програмування та наслідує *Parser*. Також клас *Parser* потребує створення об’єкта *Lexer* та передачі власності над ним через конструктор.

Результатом роботи парсера є кореневий юніт, який містить всі проаналізовані вирази. Цей юніт після закінчення аналізу може бути виконаний інтерпретатором.

Розглянемо основні члени класу *Parser* (див. рис. 2.11).

```
class Parser
{
   private:
      std::unordered_map<TokenType, Shared<InfixParser>> infix_parsers;
      std::unordered_map<TokenType, Shared<PrefixParser>> prefix_parsers;
      int32_t nesting_lvl = 0;
      std::deque<Token> read_queue;
      Unit root_unit;
      std::stack<PeekPos> peek_stack;
      int32_t peek_pos = 0;
      Interpreter *context = nullptr;
      Unique<Unit> const_scope;
      ...
```

Рисунок 2.11

Тут словники *prefix_parsers* та *infix_parsers* встановлюють відповідність між операторами та їх суб-парсерами лівого та правого пріоритетів відповідно.

Число *nesting_lvl* позначає поточний рівень вкладеності парсингу. Це значення використовується для механізму парсингу наперед разом з *peek_stack* та *peek_pos*. Цей механізм дозволяє парсити вирази, не видаляючи токенів з черги. Він може бути корисним при аналізі особливо складних синтаксичних конструкцій.

Член *root_unit* – це кореневий юніт, до якого будуть додані всі сформовані вирази.

Контекст інтерпретатора *context* та юніт *const_scope* використовуються для const-обчислень (див. додаток И) на етапі парсингу.

Const-обчислення виконується методом *evaluate_const*, що повертає буквальний вираз, який містить результат обчислення або *nil*, якщо const-вираз не повертає значення.

Методи реєстрації суб-парсерів виразів наведено на рисунку 2.12.

```
void register_parser(TokenType token, InfixParser *parser);
void register_parser(TokenType token, PrefixParser *parser);
void alias_infix(TokenType registered_tok, TokenType alias);
void alias_prefix(TokenType registered_tok, TokenType alias);
```

Рисунок 2.12

Методи *alias_infix* та *alias_prefix* дозволяють призначити декілька різних токенів як оператори для одного суб-парсера.

Реалізовано також і окремі спеціалізовані методи для реєстрації операторних виразів (див. рис. 2.13).

Разом з API класу *Lexer* ці методи забезпечують можливість визначення довільної граматики та поведінки під час виконання заданих виразів.

Розглянемо тепер реалізацію алгоритму «Pratt-parsing» у класі *Parser*.

Парсинг Methan0l-програми починається з виклику методу *Parser::load*, який викликає метод *Lexer::lex* для проведення лексичного аналізу вихідного коду, після чого викликає метод *Parser::parse_all*, який відповідає за

синтаксичний аналіз всієї черги токенів лексера.

```
void register_prefix_opr(TokenType token,
   Precedence precedence = Precedence::PREFIX);
void register_infix_opr(TokenType token, Precedence precedence,
   BinOprType type = BinOprType::LEFT_ASSOC);
void register_postfix_opr(TokenType token,
   Precedence precedence = Precedence::POSTFIX);
void register_literal_parser(TokenType token, TypeID val_type);
void register_word(TokenType wordop,
   Precedence prec = Precedence::PREFIX,
   bool multiarg = false);
void register_infix_word(TokenType wordop, Precedence prec,
   BinOprType type = BinOprType::LEFT_ASSOC);
```

Рисунок 2.13

Розглянемо реалізацію методу *Parser::parse_all* (див. рис. 2.14).

```
void Parser::parse_all()
{
   ...
   ExprList &expression_queue = root_unit.expressions();
   while (!lexer->empty()) {
         reset();
         auto expression = parse();
         if (expression != Expression::NOOP)
            expression_queue.push_back(expression);
   }
   ...
}
```

Рисунок 2.14

Як можна побачити, процес парсингу відбувається поки черга токенів лексера не порожня. Перед початком аналізу нового виразу викликається метод *Parser::reset*, який скидає тимчасовий внутрішній стан парсера. Для отримання кожного виразу верхнього рівня викликається метод *Parser::parse*, який приймає єдиний числовий аргумент, що визначає поточний рівень пріоритетності відносно якого відбувається аналіз. За замовчуванням значення цього аргументу дорівнює 0, що й приводить до парсингу виразу верхнього рівня. Отриманий від

Parser::parse вираз додається у список виразів кореневого юніта тільки якщо він не дорівнює константі *Expression::NOOP*, що означає відсутність дії. Це значення повертають константні вирази, які не мають результату обчислення.

Перейдемо тепер до методу *Parser::parse*, який є точкою входу у алгоритм «Pratt-parsing» (див. рис. 2.15).

```
ExprPtr Parser::parse(int precedence, bool prefix_only)
{
    ++nesting_lvl;
    Token token = consume();
    ExprPtr lhs = parse_prefix(token);
    ...
```

Рисунок 2.15

Спочатку збільшуємо значення рівня вкладеності, тому що цей виклик майже гарантовано буде відбуватися рекурсивно. Далі, використовуючи метод *Parser::consume*, отримуємо один токен з черги лексера, після чого за допомогою методу *Parser::parse_prefix* аналізуємо поточний вираз як префіксний, вважаючи отриманий токен його початком. В результаті цього виклику буде зроблено пошук відповідного заданому токену суб-парсера та викликано його, якщо він існує. Суб-парсери майже завжди роблять виклики до *Parser::parse* для отримання дочірніх виразів, що й забезпечує рекурсивність алгоритму.

Далі перевіряється, чи викликано парсер в режимі парсингу тільки префіксних виразів, або чи наявний у черзі токенів токен вимушеного кінця виразу (див. рис. 2.16).

```
if (prefix_only || match(Tokens::EXPR_END)) {
    --nesting_lvl;
    return lhs;
}
...
```

Рисунок 2.16

Ці два механізми потрібні для деяких виразів з особливо складною

структурою.

Після цього починається етап алгоритму «Pratt-parsing», який називають «precedence climbing». Його суть в тому, що відбувається послідовний аналіз виразів правого пріоритету (інфіксних та постфіксних) з встановленням попереднього отриманого виразу в якості лівої частини нового до тих пір, поки не буде зустрінуто токен, пріоритет якого більший або рівний тому, з яким було викликано метод *Parser::parse* (див. рис. 2.17).

```
while (precedence < get_lookahead_precedence()) {
    token = consume();
    auto &infix = infix_parsers.at(token.get_type());
    if (!infix->is_compatible(token)) {
        emplace(token);
        --nesting_lvl;
        return lhs;
    }
    lhs = infix->parse(*this, lhs, token);
    if (is_parsing_access_opr() && nesting_lvl == access_opr_lvl) {
        access_opr_lvl = -1;
        break;
    }
}
--nesting_lvl;
return lhs;
}
```

Рисунок 2.17

Тут метод *InfixParser::is_compatible* перевіряє сумісність оператора з конкретним виразом, на наявність якого відповідний суб-парсер аналізує токени. Цю логіку реалізовано для уникнення прив'язування деяких операторів до невірних виразів. Перевірка результату методу *Parser::is_parsing_access_opr* разом з рівнем вкладеності потрібна щоб забезпечити правильний аналіз оператору доступу («.»).

Метод *Parser::parse* з аргументом за замовчуванням виконується до тих пір, поки черга токенів лексера не стане порожньою. В кінці аналізу маємо абстрактне дерево виразів у вигляді кореневого юніта зі всіма проаналізованими виразами. Даний юніт готовий до виконання інтерпретатором.

2.4 Інтерпретатор

Основним та найголовнішим компонентом Methan0l є клас *Interpreter*, що реалізує інтерпретатор даної мови, а також надає API для створення runtime-об’єктів, реєстрації класів, прив’язки функцій та операторів, а також керування виконанням Methan0l-програм.

Реалізований в даній роботі інтерпретатор відноситься до класу AST або Tree Walk інтерпретаторів [4], що означає, що внутрішнє представлення програм не компілюється у байт-код деякої віртуальної машини перед виконанням, а виконується безпосередньо після синтаксичного аналізу. Такий підхід є менш ефективним, ніж компіляція вихідного коду у байт-код, але потребує менше часу на реалізацію та підходить для невеликих скриптових мов програмування.

Клас *Interpreter*, як і багато інших компонентів інтерпретатора, користуються набором деяких специфічних структур даних, якими описуються значення довільних типів, об’єкти класів та таблиці даних (див. додаток К).

2.4.1 Виконання юнітів

Завантаження програми в Methan0l може бути виконано декількома способами:

– з файлу;
– з рядка;
– виконання вже існуючого об’єкта Unit або Function.

Для завантаження Methan0l-програми, що знаходиться у файлі на диску реалізовано метод *Interpreter::load_program* (див. рис. 2.18).

Метод *load_file* завантажує весь зміст файлу за наданим шляхом та викликає метод *Parser::load* з отриманим рядком. Результуючий юніт встановлюється як юніт *Interpreter::main*. Якщо цей ланцюг операцій завершується успіхом, то метод *Interpreter::load* поверне *true*, буде проведено

ініціалізацію оточення програми та зміну робочої директорії на директорію програми (якщо значення *change_cwd – true*).

```
bool Interpreter::load_program(const std::string &path, bool change_cwd)
{
    bool loaded = load(load_file(path));
    if (loaded) {
        auto program_path = core::path(*this, path);
        auto program_path_abs = core::absolute_path(*this, path);
        set_program_globals(std::filesystem::path(program_path_abs)
                                                                .string());
        if (change_cwd) {
            std::filesystem::current_path(
                std::filesystem::path(program_path).parent_path()
            );
        }
    }
    return loaded;
}
```

Рисунок 2.18

В результаті виконання методу *Interpreter::load_program* стан інтерпретатора повністю ініціалізований, а сам інтерпретатор готовий до виконання програми. Якщо існує потреба передавати аргументи до завантаженої програми (в тому числі аргументи командного рядка), то для цього реалізовано метод *Interpreter::load_args*, який має сигнатуру, схожу на сигнатуру точки входу за замовчуванням для C та C++ програм *main*, але також в якості третього аргументу має індекс, з якого повинне починатися runtime-представлення цього списку для випадків, коли передається оригінальний масив аргументів командного рядка, що містить шлях до нативного виконуваного файлу в якості першого аргументу.

Після завантаження програми та ініціалізації стану інтерпретатора запуск програми можна виконати методом *Interpreter::run* (див. рис. 2.19).

При виклику будь-якого юніта (включаючи функції та методи) інтерпретатор додає його до стеку виконання, тому метод *Interpreter::current_unit* повертає найперший юніт з верхівки стеку. На момент

завантаження програми з файлу у стек виконання додається юніт *Interpreter::main*, проаналізований парсером, тому виконання програми починається саме з цього кореневого юніта. Юніти виконуються методом *Interpreter::execute*.

```
Value Interpreter::run()
{
    if (main.empty()) return Value::NO_VALUE;
    Value ret;
    auto &handler = get_exception_handler();
    do {
        try {
            ret = execute(*current_unit());
        } catch (const std::exception &e) {
            if (!handle_exception(e)) return -1;
        } catch (Value &e) {
            if (!handle_exception(e)) return -1;
        } catch (...) { handle_unknown_exception(); }
    } while (handler.is_handling());
    return ret;
}
```

Рисунок 2.19

Головний юніт може повертати значення, як і будь-який інший. Це значення можна отримати з результату виконання методу *Interpreter::run*, що робить юніти в Methan0l схожими на чанки мови програмування Lua [23].

Так як Methan0l не має власної системи обробки винятків, а замість цього використовує C++ винятки, то виконання програми знаходиться всередині *try-catch* блоку, який, в свою чергу, знаходиться у *do-while* циклі, що використовується для runtime-обробки винятків.

Перед початком виконання також отримується поточний обробник винятків, який реалізовано класом *ExceptionHandler*. Це допоміжний клас, який містить стек вказівників на активні на даний момент *try-catch* вирази, асоційовані з їх рівнями вкладеності відносно головного юніта. Ці дані потрібні, щоб мати можливість переключати потік виконання з місця, де стався виняток на блок його обробки.

Метод *Interpreter::handle_exception* також використовує об’єкт *ExceptionHandler*. Механізм його роботи полягає в тому, що зі стеку *try-catch* виразів отримується верхній елемент, з якого отримується блок обробки винятка, на який переключається потік виконання. Після обробки винятка виконання програми продовжується з першого виразу після *try-catch* блоку.

Якщо ж Methan0l-код, який спричинив виняток, не оточений *try-catch* блоком, то обробник винятку поверне *false*, що викликає завершення програми.

Розглянемо тепер реалізацію методу *Interpreter::execute*, який є основним засобом виконання внутрішнього представлення Methan0l-програм (див. рис. 2.20).

```
Value Interpreter::execute(Unit &unit, const bool use_own_scope)
{
    if (use_own_scope) enter_scope(unit);
    if (exception_handler.is_handling())
        exception_handler.stop_handling();
    else
        unit.reset_execution_state();
    ...
```

Рисунок 2.20

Виконання юніта починається з входу у його зону видимості, якщо значення *use_own_scope* дорівнює *true* (що є його значенням за замовчуванням). Інтерпретатор має окремий стек для зон видимості для полегшення знаходження посилань на ідентифікатори.

Саме в цей стек додає зону видимості поточного юніта метод *Interpreter::enter_scope*.

Після цього виконується частина логіки обробки винятків – якщо на момент початку виконання юніта відбувається обробка деякого винятка, то це означає, що юніт – його обробник, тому викликається метод припинення обробки винятків *ExceptionHandler::stop_handling*. В протилежному випадку інтерпретатор скидає стан виконання отриманого юніта, який представляє з себе вказівник на поточний виконуваний вираз.

Далі починається безпосереднє послідовне виконання всіх виразів юніта (див. рис. 2.21).

```
while (unit.has_next_expr() && !unit.execution_finished())
      exec(*(current_expr = unit.next_expression()));
  if (execution_stopped())
      return Value::NO_VALUE;
  ...
```

Рисунок 2.21

Після циклу виконання всіх виразів юніта знаходиться логіка завершення роботи одного юніта зі стеку.

Якщо виконання програми було передчасно завершено, результатом виконання стає константа *Value::NO_VALUE*, що означає відсутність значення.

Далі виконується обробка оператору *break* (див. рис. 2.22).

```
Unit *parent = exec_stack.size() > 1
                                ? *std::prev(exec_stack.end(), 2)
                                : current_unit();
Value returned_val = unit.result();
bool carry_return = unit.carries_return();
if (unit.break_performed()) {
   if (carry_return)
      for (auto it = std::prev(exec_stack.end(), 2);
            it != std::prev(exec_stack.begin()); --it) {
         if ((*it)->carries_return())
            (*it)->stop(true);
      }
}
...
```

Рисунок 2.22

Ця логіка описує обхід всіх несучих юнітів та зупинку їх виконання без повернення значення.

Після чого інтерпретатор покидає зону видимості юніта та обробляє перенесення поверненого значення за логікою несучих юнітів (див. 1.4) на один юніт вниз за стеком (див. рис. 2.23).

```
    if (use_own_scope)
        leave_scope();
    if (carry_return && !returned_val.empty()) {
        parent->save_return(returned_val);
    }
    return returned_val;
}
```

Рисунок 2.23

На даний момент було розглянуто всі основні кроки, які робить інтерпретатор при отриманні файлу вихідного коду: лексичний та синтаксичний аналіз, const-обчислення та безпосереднє виконання програми. Але для того, щоб виконання коду мало результат, потрібно здійснити прив'язування логіки операторів, функцій та методів runtime-класів до інтерпретатора, тому що реалізацій поведінки жодних з перелічених синтаксичних структур за замовчуванням в ядрі Methan0l немає.

Динамічні бібліотеки. Для зручності організації наборів прив'язок та інших модифікацій стану інтерпретатора (включаючи реєстрацію довільних граматик) в Methan0l створено концепцію динамічних бібліотек, базова логіка яких описана у класі *Library*. Цей клас приймає вказівник на контекст інтерпретатора у конструкторі та має набір методів для зручної взаємодії з ним. Клас *Library* також визначає абстрактний метод *load*, у перевантаженні якого бібліотека-реалізатор повинна виконати всі операції прив'язування функцій, класів та інших runtime-сутностей до інтерпретатора.

Кожна бібліотека має збиратися як окремий динамічно-завантажуваний модуль. Для того, щоб забезпечити можливість інтерпретатора завантажити та ініціалізувати бібліотеку, треба додати до її файлу вихідного коду макрос *METHAN0L_LIBRARY()*, в якості аргументу якого вказати назву класу бібліотеки, що наслідує клас *Library*. Даний макрос визначає в динамічній бібліотеці вільну функцію з ім'ям *load_methan0l_library*, яка створює та повертає об'єкт заданого класу бібліотеки. Цю функцію шукає інтерпретатор під час завантаження файлу бібліотеки та викликає її, після чого викликає метод *Library::load* отриманого об'єкта та зберігає вказівник на отриманий з функції об'єкт бібліотеки.

Якщо вищевказаний макрос застосовано коректно, то після копіювання файлу бібліотеки до папки «libraries» в робочій директорії інтерпретатора та його запуску її буде завантажено, а всі визначені прив'язки будуть визначені та доступні до використання в runtime-середовищі. Саме цей механізм використовує набір стандартних бібліотек мови Methan0l.

2.4.2 Прив'язування операторів

Розглянемо механізм прив'язування операторів з використанням API, які надає клас *Library*.

Внутрішнє оператори в Methan0l поділяються на дві категорії та три типи, які присутні в кожній з них.

До категорій операторів можна віднести:

а) оператори, що застосовуються до значень;

б) оператори, що застосовуються до виразів.

Оператори категорії *а* також називатимемо value-операторами. Як слідує з назви, їх операнди спочатку обчислюються, а потім передаються на обробку.

Оператори категорії *б*, які називатимемо лінивими, отримують в якості операндів не обчислені вирази. Це може бути корисним для селективного обчислення (наприклад, для реалізації логічних OR та AND операторів).

Типи операторів обумовлені кількістю операндів та поділяються на:

– бінарні;

– унарні.

Оператори визначаються як спеціалізації класу *std::function*<...> (див. рис. 2.24).

```
using BinaryOpr = std::function<Value(Value&, Value&)>;
using UnaryOpr = std::function<Value(Value&)>;
using LazyUnaryOpr = std::function<Value(const ExprPtr&)>;
using LazyBinaryOpr = std::function<Value(
                                const ExprPtr&, const ExprPtr&)>;
```

Рисунок 2.24

Клас *Interpreter* має чотири окремих словника з ключами, що мають тип *TokenType* та з типами значень у вигляді кожного з вищевказаних типів. При обчисленні операторного виразу інтерпретатор спочатку перевіряє, чи існує прив'язка лінивого оператору вказаного типу. Якщо її немає, то виконується обчислення всіх операндів та виклик value-оператора.

Операція прив'язування просто додає нову пару значень до одного з операторних словників. Прив'язування операторів зсередини бібліотеки, що наслідує клас *Library* здійснюється за допомогою одного з трьох методів, що мають по два перевантаження кожний відповідно до категорій операторів (див. рис. 2.25).

```
void prefix_operator(TokenType, const LazyUnaryOpr&);
void prefix_operator(TokenType, const UnaryOpr&);
void infix_operator(TokenType, const LazyBinaryOpr&);
void infix_operator(TokenType, const BinaryOpr&);
void postfix_operator(TokenType, const LazyUnaryOpr&);
void postfix_operator(TokenType, const UnaryOpr&);
```

Рисунок 2.25

Приклад прив'язування оператора до runtime-середовища інтерпретатора наведено у додатку Л.

2.4.3 Прив'язування функцій

Окрім операторів невід'ємною частиною будь-якої мови програмування є функції та методи. Їх можна визначати повністю за допомогою Methan0l-коду, але також існує можливість прив'язувати нативні функції, функтори та методи до runtime-середовища Methan0l.

Механізм, що відповідає за прив'язування будь-яких нативних викликуваних об'єктів до runtime-середовища Methan0l є одним з найважливіших у всій базі коду інтерпретатора. Він здійснює відображення

нативного викликуваного об’єкта (вільної функції, функтору або методу) з довільною сигнатурою у функціональний об’єкт виду *std::function<Value(const ExprList&)>*, тобто відбувається відображення типу, що повертається викликуваним об’єктом у значення типу *Value*, а весь формальний список параметрів відображається у runtime-список аргументів Methan0l-функції. Цю логіку реалізовано завдяки шаблонній мета-структурі *Interpreter::call_helper* та її методу *call*. Розглянемо їх реалізацію більш детально.

Прив’язка викликуваних об’єктів починається з шаблонного методу *Interpreter::bind_func*, сигнатуру якого зображено на рисунку 2.26.

```
template<unsigned default_argc = 0, typename F>
NativeFunc bind_func(F &&f, Value default_args = Value::NO_VALUE)
```

Рисунок 2.26

В цьому шаблоні *F* – тип будь-якої функції, функтору або методу, не типовий шаблонний параметр *default_argc* – кількість аргументів за замовчуванням, а *default_args* – runtime-об’єкт списку аргументів за замовчуванням.

Цей метод перевіряє різні характеристики викликуваного об’єкта (наприклад, кількість аргументів, тип, що повертається, та інші) за допомогою мета-структури *function_traits*, після чого повертає об’єкт типу *NativeFunc*, який представляє собою runtime-відображення нативної функції. Зсередини C++ коду сигнатура функціональних об’єктів *NativeFunc* виглядає як *Value(const ExprList&)*. При цьому, вміст цих об’єктів завжди містить виклик до методу *Interpreter::call*, який і забезпечує зв’язок між нативними викликуваними об’єктами та їх runtime-відображеннями (див. рис. 2.27).

Як можна побачити, в основному цей метод розділяє функції, які не мають типу, що повертається від тих, які повертають значення деякого типу. При цьому сама внутрішня логіка безпосереднього зв’язування runtime-відображення функції з її нативною сигнатурою знаходиться у методі Interpreter::*call_helper<...>::call*.

```
template<typename F, typename Container>
auto call(F &&f, const Container &c)
{
   constexpr unsigned argc = function_traits<decltype(f)>::arity;
   using caller = call_helper<argc == 0, decltype(f), Container, argc>;
   if constexpr (std::is_void_v<typename
                  function_traits<F>::return_type>) {
      caller::call(*this, f, c);
      return Value::NO_VALUE;
   } else return caller::call(*this, f, c);
}
```

Рисунок 2.27

Розглянемо реалізацію його перевантажень. Перше з них використовується для формування compile-time списку індексів параметрів функції, поки шаблонний не типовий параметр *I* не стане рівним арності *N* функції *Functor* (див. рис. 2.28).

```
template<bool Done, typename Functor, typename Container,
   unsigned N, unsigned ...I>
struct call_helper
{
   static auto call(Interpreter &context, Functor &&f,
      const Container &c)
   {
      return call_helper<sizeof...(I) + 1 == N, Functor, Container,
            N, I..., sizeof...(I)>::call(context, f, c);
   }
};
```

Рисунок 2.28

Після того, як *I* стає рівним *N*, виконується головна мета-функція виклику прив'язаних нативних функцій (див. рис. 2.29).

Тут виконання нативної функції здійснюється за допомогою стандартної функції *std::invoke*, в якості аргументів якої використано розпакування шаблонного списку аргументів, обробленого методом *Interpreter::eval*.

Цей метод є ключовим у даній операції, так як він зіставляє тип та категорію значення нативного параметру функції з її runtime-відображенням та отримує від runtime-середовища потрібне представлення відповідного об’єкта зі

списку формальних параметрів прив’язуваної функції.

```
template<typename Functor, typename Container, unsigned N, unsigned ...I>
struct call_helper<true, Functor, Container, N, I...>
{
   ...
  static auto call(Interpreter &context, Functor &&f, const Container &c)
  {
     if constexpr (N > 0) {
        inject_callarg<Context>(c, context);
        inject_callarg<CallArgs>(c, context, unconst(c));
     }
     if (c.size() < N) throw std::runtime_error("Too few arguments");
     return std::invoke(f, context.eval<ArgType<I>>(*c.at(I))...);
  }
};
```

Рисунок 2.29

В результаті маємо однозначну відповідність runtime-функції її нативній версії, що забезпечує можливість прив’язування зовнішніх модулів до інтерпретатора, а також його вбудовування у існуючі програмні продукти.

Також перед безпосередньо викликом функції можна побачити використання методу *call_helper::inject_callarg*.

Цей метод дозволяє захоплювати стан інтерпретатора або список всіх аргументів функції (включаючи ті, що не було визначено у її сигнатурі) за допомогою спеціального механізму, який називатимемо фіктивними аргументами.

Суть їх полягає в тому, що при реалізації нативної функції, яку буде прив’язано до runtime-середовища інтерпретатора, до її списку аргументів можна додати значення спеціальних типів *Context* та *CallArgs*, які при виклику прив’язаної функції будуть набувати значення посилання на контекст інтерпретатора та списку переданих аргументів.

При цьому ці аргументи будуть відсутні у сигнатурі runtime-відображення даної функції.

2.4.4 Прив’язування класів

Невід’ємною частиною будь-якої інтерпретованої мови програмування є використання вже існуючих нативних класів у вигляді їх відображення у runtime-середовище. В Methan0l такі відображення можна створювати двома способами.

Першим з них є просте створення класу, який наслідує клас *Class* та визначає всі прив’язки нативних методів вручну за допомогою методу *Class::register_method*. Такий спосіб підходить для прив’язування або реалізації невеликих класів або для випадків, коли необхідна реалізація деякої додаткової логіки методів, якої немає у прив’язуваному класі. Однак, прив’язування існуючих класів у такий спосіб не є зручним і значно збільшує об’єм коду.

Другим способом є використання API прив’язок класів. Дане API надає спеціальний клас *NativeClass<T>*, який повинен наслідувати клас-прив’язник, встановлюючи в якості типу *T* клас, який повинен бути прив’язаний до runtime-середовища. Логіка ініціалізації з прив’язуванням конструктору та всіх методів класу повинна бути реалізована у віртуальному методі *NativeClass<T>::initialize*, який викликається з оператору виклику без аргументів, що перевантажений у самому класі *NativeClass<T>*.

Не зважаючи на можливість виконувати наслідування та реалізацію методу *NativeClass<T>::initialize* вручну, це також може приводити до того, що об’єм коду прив’язок класів швидко зростає та стає складним для підтримки та модифікації. З цієї причини API прив’язок класів повністю складається зі спеціальних макросів. Основними з них є макрос визначення зв’язку між нативним та runtime-класом, який повинен використовуватися у заголовковому файлі (див. рис. 2.30).

```
NATIVE_CLASS(RuntimeClass, NativeClass)
```

Рисунок 2.30

В якості імені *RuntimeClass* необхідно вказати ім’я, під яким прив’язаний

клас буде доступний у runtime-середовищі інтерпретатора. Разом з цим, клас із цим же ім'ям буде визначено і в нативному C++ коді, тому хорошою практикою є визначати прив'язки у окремому просторі імен. В якості імені *NativeClass* необхідно вказати ім'я прив'язаного нативного класу. При цьому підтримуються як звичайні, так і шаблонні класи.

Для визначення набору всіх прив'язок також існує окремий макрос (див. рис. 2.31), який має бути використаний у файлі вихідного коду (на відміну від попереднього макросу, який вказується у заголовковому файлі).

```
NATIVE_CLASS_BINDING(RuntimeClass, {
    ...
})
```

Рисунок 2.31

Замість імені *RuntimeClass* необхідно вказати ім'я runtime-класу з попереднього макросу.

Всередині блоку виразів треба визначити прив'язки конструктору, методів та операторів, для яких повинна бути можливість використання зсередини runtime-середовища.

Існує два способи прив'язки методів:

– явна прив'язка (визначення вмісту методу вручну);
– прив'язка за вказівником на член класу.

Далі розглянемо їх реалізацію та особливості використання.

Після того, як повний опис прив'язки реалізовано, треба зареєструвати її у менеджері типів. Це можна зробити під час ініціалізації бібліотеки (див. 2.4.1) або модуля (див. 2.5).

Явна прив'язка методів. Спосіб явної прив'язки методів добре підходить для невеликих класів, модифікації поведінки прив'язаних методів «на льоту» або визначення нових невеликих методів як членів прив'язаного класу. Скористатися ним можна за допомогою наступного макросу, який повинен використовуватися всередині блоку *NATIVE_CLASS_BINDING* (див. рис. 2.32).

```
METHOD(method_name) (OBJ, Type arg1, Type arg2, ...) {
    auto &_this = THIS;
    ...
};
```

Рисунок 2.32

Замість *method_name* необхідно вказати ім'я, під яким метод буде доступний зсередини runtime-середовища. Після цього вказується список аргументів методу. В ньому на першому місці обов'язково повинен бути макрос *OBJ*, який приймає значення типу *Object*, яке вказує на об'єкт, для якого було викликано метод. Для того, щоб отримати доступ до нативного вмісту цього об'єкта як до об'єкта прив'язаного класу *T* зсередини тіла методу, треба використати макрос *THIS* (див. рис. 2.33).

```
#define OBJECT(binder, obj) \
    std::remove_reference<decltype(binder)>::type::as_native(obj)
#define THIS OBJECT(class_binder, this_obj)
```

Рисунок 2.33

Доступ до контексту інтерпретатора зсередини таких прив'язок можна отримати за допомогою макросу *CONTEXT*.

Реалізацію продемонстрованого вище макросу *METHOD(name)* зображено на рисунку 2.34.

```
#define STANDARD_METHOD(name) class_binder.register_method(name) = []
#define METHOD(name) STANDARD_METHOD(STR(name))
```

Рисунок 2.34

Як можна побачити, механізм прив'язки методу у такий спосіб зводиться до використання методу *ClassBinder<T>::register_method*, який приймає тільки ім'я майбутнього методу та повертає об'єкт спеціальної структури з перевантаженим оператором присвоювання. Коли воно відбувається, цей об'єкт виконує прив'язування вказаної лямбда-функції до вказаного класу в якості

методу з заданим ім’ям. Такий підхід обрано для наближення зовнішнього вигляду синтаксису використання макросів прив’язок до того, як виглядає традиційний синтаксис визначення функцій.

Таким чином можна визначити будь-який метод, але для простої прив’язки без будь-яких модифікацій поведінки методів використання цього способу буде потребувати надто великої кількості зайвого коду. Для уникнення цього було реалізовано прив’язування за вказівником на член класу.

Прив'язка за вказівником на член класу. Прив’язування нативних методів даним способом робить код більш лаконічним та дозволяє повністю розділити нативну реалізацію від безпосередньої логіки її прив’язування до runtime-відображення.

Всі макроси прив’язки за вказівником на член класу також використовуються всередині блоку *NATIVE_CLASS_BINDING*.

Найперший та найменш тривіальний у реалізації макрос – *BIND_CONSTRUCTOR(...)*, який реалізує операцію прив’язування нативного конструктора за його сигнатурою. Так як конструктори в C++ насправді не існують у результуючому бінарному файлі, і тому не мають адреси [24], то при його прив’язуванні доводиться створювати спеціальну прив’язану до runtime-середовища лямбда-функцію, яка має таку ж сигнатуру, як і конструктор та делегує отриманий виклик до справжнього нативного конструктора об’єкта.

Розглянемо реалізацію макросу *BIND_CONSTRUCTOR(...)* та всіх методів, від яких він залежить. Цей макрос розгортається у виклик методу *ClassBinder<T>::bind_constructor<...Sig>*, який реалізує логіку прив’язування (див. рис. 2.35).

```
template<typename ...Sig>
inline void bind_constructor()
{
    clazz->register_method(
        Methods::Constructor, wrap_constructor<Sig...>()
    );
}
```

Рисунок 2.35

Цей метод реєструє runtime-метод з ім’ям *construct*, яке визначено константою *Methods::Constructor* за допомогою методу *Class::register_method*. Метод *ClassBinder<T>::wrap_constructor<...Sig>* виконує основну роботу із зв’язування виклику до конструктора з його runtime-відображенням (див. рис. 2.36).

```
template<typename ...Sig>
inline NativeFunc wrap_constructor()
{
    return context.bind_func([](Object &obj, Sig ...args) {
        obj.def(mtl::str(Fields::NATIVE_OBJ))
            = Factory<C, Sig...>::make(std::forward<Sig>(args)...);
    });
}
```

Рисунок 2.36

Тут використовується метод *Object::def*, який надає доступ до полів об’єктів, або, якщо вказаного поля не існує, то створює його. За допомогою цього методу встановлюється значення нативного вмісту об’єкта. Нативний конструктор викликається за допомогою методу *Factory<C, Sig...>::make*, який створює об’єкт та повертає вказівник з підрахуванням посилань на нього.

Тепер розглянемо прив’язування методів. Воно виконується за допомогою макросу *BIND_METHOD(name)*, де *name* – ім’я методу C++-класу. Даний макрос розгортається у виклик методу *ClassBinder<T>::bind_method*, який приймає рядкове представлення імені методу та вказівник на нього (див. рис. 2.37).

```
template<typename F>
inline void bind_method(std::string_view name, F &&method)
{
    clazz->register_method(name, context.bind_func(
        mtl::polymorphic_method<bound_class>(method)));
}
```

Рисунок 2.37

Для реєстрації прив’язки методу використовуються вже розглянуті

механізми *Class::register_method* та *Interpreter::bind_func*, але також перед прив’язуванням заданого методу він передається функції *polymorphic_method*. Вона забезпечує встановлення коректного типу імпліцитного аргументу *this* для прив’язаного методу, тобто приводить його тип до конкретного класу *T*, що прив’язується до runtime-середовища. Це необхідно через особливість реалізації механізму зберігання нативного вмісту об’єктів, а саме через те, що він описується значенням типу *std::any*. Цей клас при отриманні доступу до об’єкту, що в ньому зберігається, вимагає вказування конкретного класу значення без підтримки поліморфізму. Наприклад, якщо існує нативний об’єкт класу *Derived*, який наслідує клас *Base*, і в *std::any* зберігається саме об’єкт класу *Derived*, то до нього неможливо отримати доступ як до *Base*.

2.5 Зовнішні модулі

Для будь-якої мови програмування завжди виникає необхідність підключення та використання сторонніх модулів під час виконання програми, тому цю можливість було реалізовано і в Methan0l. На відміну від бібліотек, модулі не завантажуються інтерпретатором автоматично під час його ініціалізації, а можуть бути завантажені тільки користувачем одним з двох способів: оператором *import* або функцією *load* (див. 1.3). Модулі можуть бути реалізовані як мовою Methan0l, так і мовою C++ з використанням відповідного API. У випадку нативної реалізації модуля, його повинно бути зібрано як самостійну динамічну бібліотеку.

Розглянемо деталі реалізації API модулів.

Модуль може бути визначено одним з двох способів:

– без точки входу;

– з точкою входу.

Під точкою входу розуміємо функцію, яка буде виконуватися під час його ініціалізації.

Модуль без точки входу. Якщо модуль має тільки прив'язки вільних функцій, можна використати перший підхід. В такому випадку все, що потрібно зробити для індикації того, що динамічна бібліотека є Methan0l-модулем, це використати макрос *INIT_MODULE* з заголовкового файлу «methan0l.h» у файлі вихідного коду модуля. Визначення даного макросу наведено на рисунку 2.38.

```
#define INIT_MODULE                                    \
mtl::Interpreter *context = nullptr;                   \
mtl::DataTable *local_scope = nullptr;                 \
API void init_methan0l_module(mtl::Interpreter* context) { \
    ::context = context;                               \
    local_scope = context->local_scope();              \
}
```

Рисунок 2.38

Як можна побачити, цей макрос визначає у глобальній зоні видимості модуля змінні *context* та *local_scope*, які вказують відповідно на контекст інтерпретатора, в якому було завантажено модуль, та зону видимості модуля. Всі прив'язки вільних функцій та модульних змінних будуть додаватися саме до *local_scope*. Функцію *init_methan0l_module* буде знайдено та викликано інтерпретатором після завантаження модуля. Якщо вона відсутня, то буде кинуто виняток.

На цьому ініціалізація модуля без точки входу закінчено. Для того, щоб прив'язувати вільні функції у такому випадку у файлі вихідного коду модуля треба використовувати макрос *FUNCTION(name)*, де *name* – ім'я нативної функції, що вже визначена. Цей макрос визначає спеціальну вільну функцію з префіксом «_register_methan0l_func_», після якого вказується ім'я прив'язаної функції. В тілі цієї спеціальної функції відбувається прив'язування вказаної нативної функції до зони видимості модуля за допомогою розглянутого раніше методу *Interpreter::register_func*. При завантаженні модуля інтерпретатор шукатиме всі функції, що починаються з вищевказаного префіксу та виконуватиме їх, створюючи прив'язки як результат.

Модуль з точкою входу. Для випадків, коли модуль має виконувати більш

складні операції, ніж просте прив’язування вже реалізованих вільних функцій, було реалізовано спеціальний макрос для позначення його вхідної точки (див. рис. 2.39).

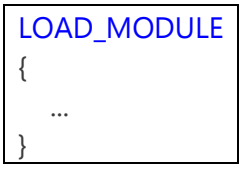

Рисунок 2.39

Вся логіка, описана всередині даного блоку, буде виконана під час ініціалізації модуля. Визначення цього макросу наведено на рисунку 2.40.

```
#define LOAD_MODULE INIT_MODULE API void load()
```

Рисунок 2.40

Тобто при його використанні він розгортається у розглянутий раніше макрос *INIT_MODULE* з додаванням ще одної вільної функції *load*, яка виконається інтерпретатором під час завантаження модуля за умови її присутності.

Для виконання прив’язок вже існуючих функцій та змінних з тіла блоку *LOAD_MODULE* потрібно використовувати макроси *REG_FUNC(name)* та *REG_VAR(name)*.

3 ДЕМОНСТРАЦІЯ ВИКОРИСТАННЯ ТА ВБУДОВУВАННЯ METHAN0L

В цьому розділі буде продемонстровано використання реалізованого інтерпретатора мови програмування Methan0l на прикладі реалізації інтерфейсу командного рядка для взаємодії з інтерпретатором та можливістю самостійного виконання програм мовою Methan0l, створенні нативних динамічно завантажуваних модулів та інтеграції інтерпретатора у існуючу програму, написану мовою програмування C++ з використанням фреймворку Qt.

3.1 Реалізація CLI для інтерпретатора

Так як інтерпретатор Methan0l розраховано на вбудовування та реалізовано у вигляді динамічної бібліотеки, а не деякої виконуваної програми, то для звичайного користувача не існує способу взаємодії з ним. Для вирішення цієї проблеми реалізуємо CLI, що використовуватиме бібліотеку інтерпретатора та надаватиме користувачеві можливість виконувати Methan0l програми з файлів вихідного коду з можливістю передачі аргументів командного рядка до них.

Почнемо реалізацію зі створення класу *cli*, який буде містити всю логіку взаємодії користувача з інтерпретатором (див. рис. 3.1).

```
class cli
{
    private:
        bool no_exit = false;
        int parse_args(int, char**);
    public:
        int run(int argc, char **argv);
};
```

Рисунок 3.1

В залежності від значення поля *no_exit* інтерпретатор буде завершувати роботу після закінчення виконання програми або чекатиме на ввід з клавіатури перед цим.

Метод *parse_args* відповідатиме за обробку аргументів самого CLI. Для того, щоб мати можливість передавати аргументи командного рядка і до CLI, і до скрипту, який буде запускатися, визначимо формат аргументів виклику розроблюваного CLI (див. рис. 3.2).

```
methan0l --cli-arg1 ... path/to/script.mt0 script-arg1 ...
```

Рисунок 3.2

Тобто аргументи, які призначені для CLI повинні вказуватися перед шляхом до Methan0l скрипту, а аргументи для скрипту – після нього.

Реалізація *mtl::parse_args* зводиться до перевірки всього списку аргументів командного рядка (див. додаток М) та застосування вказаних в них параметрів до інтерпретатора. Також будемо зберігати індекс в списку аргументів, з якого починаються аргументи, які необхідно передавати скрипту. Це значення буде повертатися з даного методу.

Для безпосереднього виконання програм створимо окремий клас *Runner*, який прийматиме посилання на об’єкт інтерпретатора в конструкторі та використовуватиме його API для запуску скриптів (див. 2.4.1). Головним публічним методом цього класу буде *Runner::run_file* (див. рис. 3.3), який отримуватиме список аргументів командного рядка, їх кількість та індекс початку аргументів для скрипту, а повертатиме приведене до *int* значення, яке повертає головний юніт виконуваного скрипту (аналогічно коду повернення в C та C++ програмах).

Метод *load_file* завантажує скрипт засобами інтерпретатора. Якщо на якомусь етапі відбудеться помилка, цей метод повідомить про неї. Після цього виконується завантаження аргументів командного рядка у стан інтерпретатора та викликається метод *Runner::run*, який виконує метод *Interpreter::run*

(див. 2.4.1) та повертає значення, повернуте головним юнітом, приводячи його до типу *int*.

```
int Runner::run_file(int argc, char **argv, int start_from)
{
    std::string src_name(argv[start_from]);
    if (load_file(src_name)) {
        methan0l.load_args(argc, argv, start_from);
        return run();
    }
    return -1;
}
```

Рисунок 3.3

Повернемося до реалізації CLI та визначимо головний метод *cli::run*, який будемо викликати з точки входу програми, передаючи до нього отримані аргументи командного рядка та їх кількість (див. рис. 3.4).

```
int cli::run(int argc, char **argv)
{
    auto arg_start = parse_args(argc, argv);
    if (arg_start < 0)
        return 0;
    int return_value = 0;
    {
        Methan0l methan0l(argv[0]);
        return_value = Runner(methan0l)
            .run_file(argc, argv, arg_start);
    }
    if (no_exit) {
        std::string _;
        std::getline(std::cin, _);
    }
    return return_value;
}
```

Рисунок 3.4

Додамо створення об’єкта *Runner* та виклик цього методу у точці входу програми та скомпілюємо її, після чого напишемо наступну тестову Methan0l-

програму у файлі «hello.mt0», в якій продемонструємо декілька основних аспектів мови Methan0l (див. рис. 3.5).

```
<% $"Hello, Methan0l! It's {}.", {
    pad = fun: val {
        str = $$val
        if: str.size() < 2 str = "0" + str
        return: str
    }
    time = now() / 1000
    seconds = time % 60
    minutes = (time % 3600) / 60
    hours = (time % 86400) / 3600
    return: pad(hours)   :: ":" :: pad(minutes) :: ":"
         :: pad(seconds)
}()
<% $"Command line arguments: {}", {
    str = "{\n"
    get_launch_args().for_each(
        f: arg -> str += "\t" :: arg :: "\n"
    )
    str += "}"
    return: str
}()
```

Рисунок 3.5

Даний скрипт виведе повідомлення «Hello, Methan0l!», поточний час та список всіх аргументів командного рядка, які було передано скрипту у стандартний потік виводу, використовуючи техніку приховування тимчасових значень за допомогою виклику анонімних юнітів (див. 1.4). Для отримання поточного часу в цій програмі використовується стандартна функція *now*, яка повертає кількість мілісекунд з початку епохи Unix.

Виконаємо створену програму за допомогою CLI інтерпретатора (див. рис. 3.6).

Переконавшись у коректності роботи реалізованої оболонки командного рядка для запуску програм, можемо перейти до розширення її функціоналу.

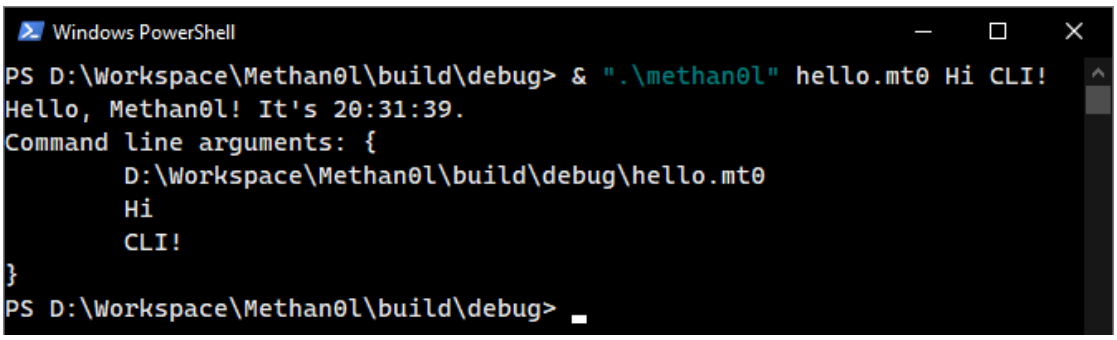

Рисунок 3.6

Інтерактивний режим CLI. Реалізуємо тепер інтерактивний режим командної оболонки аналогічно інтерактивній консолі мови програмування Python [25]. Для цього створимо новий клас *InteractiveRunner*, в якому реалізуємо логіку інтерактивності. Даний режим буде активуватися, якщо командній оболонці інтерпретатора не передано жодного аргументу. Додамо також окремий режим системи комп'ютерної алгебри (CAS), в якому результат кожного введеного виразу буде одразу виводитися та зберігатися для подальшого використання.

В інтерактивному режимі буде реалізовано наступний набір функцій:

– порядкове виконання виразів;
– можливість багаторядкового вводу;
– наявність спеціальних команд оболонки, за допомогою яких можна керувати деякими аспектами інтерпретатора або виконувати типові дії;
– розширення граматики новим оператором для доступу до результатів попередніх обчислень у режимі CAS.

Головним методом нового класу *InteractiveRunner* буде метод *InteractiveRunner::run*, який реалізовуватиме REPL механізм обробки виразів. Цей метод буде отримувати та аналізувати введені користувачем дані за наступною логікою:

– якщо рядок містить префікс «/», то це команда інтерактивного режиму і її буде виконано;
– якщо рядок – не команда, але містить не закриті блокові вирази, це переводить CLI в багаторядковий режим, і запит на введення буде

продовжуватися до тих пір, поки не залишиться не закритих блоків або дужок;

– якщо ні одна з попередніх умов не виконалася, то рядок – коректний Methan0l-код і його буде виконано інтерпретатором.

Після запуску реалізованої командної оболонки для інтерпретатора Methan0l без аргументів побачимо інтерфейс інтерактивного режиму (див. рис. 3.7).

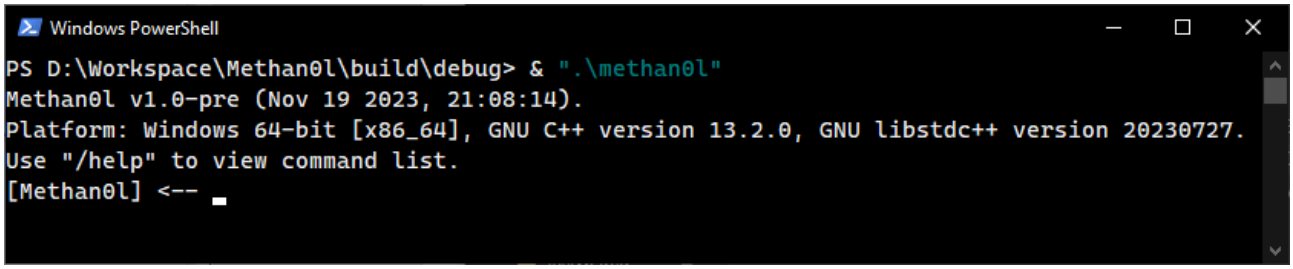

Рисунок 3.7

Протестуємо одразу два аспекти інтерактивного режиму: порядкове виконання виразів та багаторядковий ввід, під час якого введені рядки аналізуються тільки лексером до тих пір, поки акумульована черга токенів не буде описувати коректну послідовність виразів, після чого відбудеться синтаксичний аналіз та виконання програми.

Реалізуємо тестову програму, яка виводить список встановлених бібліотек Methan0l, використовуючи стандартний клас *File* та його метод *for_each*, який розглядає вказаний у конструкторі *File* шлях як шлях до директорії та викликає передану функцію для кожного з файлів у ній (див. рис. 3.8).

```
(new: File("$:\\libraries")).for_each(
        f: file -> <% "- " :: file
)
```

Рисунок 3.8

Введемо тестову програму у CLI та виконаємо її щоб переконатися у правильності реалізації описаних механізмів (див. рис. 3.9).

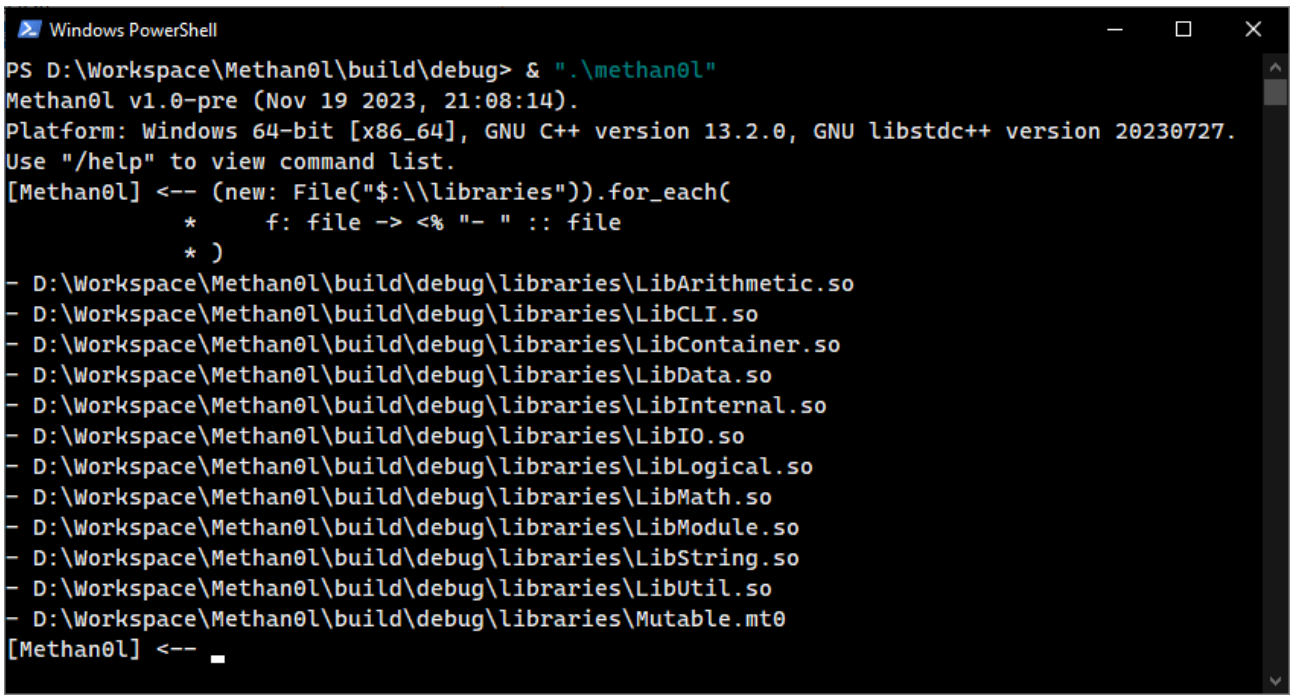

Рисунок 3.9

Інтеграція режиму CAS з інтерпретатором. Для можливості доступу до попередніх результатів обчислень зсередини runtime-середовища інтерпретатора, зареєструємо новий префіксний оператор «@», після якого необхідно вказувати номер збереженого значення. При цьому для того, щоб отримати останнє збережене значення, можна вказати 0 або символ «_» після токену оператору.

Виконаємо визначення та реєстрацію цього оператору у перевантаженні методу *load* нової бібліотеки *LibCLI* (див. рис. 3.10).

```
prefix_operator(Tokens::AT, LazyUnaryOpr([&](auto &expr) {
    auto idx = val(expr);
    auto &runner = interactive_runner();
    return runner.get_saved_value(
        IdentifierExpr::get_name_or_default(expr) == "_"
        ? 0 : idx.template as<size_t>());
}));
```

Рисунок 3.10

Метод *LibCLI::interactive_runner*, використаний у тілі реалізації оператора «@», є ключовим для взаємодії інтерпретатора та реалізованої оболонки

командного рядка та використовується для отримання посилання на об’єкт *InteractiveRunner* (див. рис. 3.11).

```
inline InteractiveRunner& interactive_runner()
{
    return context->get_env_hook<InteractiveRunner>(
        CLIHooks::INTERACTIVE_RUNNER
    );
}
```

Рисунок 3.11

Даний метод використовує механізм хуків інтерпретатора, який дозволяє додавати до таблиці змінних оточення вказівники на нативні об’єкти, забезпечуючи взаємодію компонентів інтерпретатора з зовнішніми програмними продуктами. В цьому випадку вказівник на поточний *InteractiveRunner* зареєстровано під назвою, визначеною у константі *CLIHooks::INTERACTIVE_RUNNER* («.interactive»). Структуру зі списком назв хуків до CLI визначено у головному заголовковому файлі командної оболонки.

Реєстрація хуку відбувається при ініціалізації *InteractiveRunner* (див. рис. 3.12).

```
methan0l.get_env_table().set(CLIHooks::INTERACTIVE_RUNNER, this);
```

Рисунок 3.12

Окрім модифікації граматики, також було додано новий аргумент CLI інтерпретатора – «--cas», при використанні якого командна оболонка запускається одразу у режимі системи комп’ютерної алгебри.

Розглянемо також поведінку реалізованих раніше методів *InteractiveRunner* коли активний режим CAS.

Для методу *parse* поведінку під час режиму системи комп’ютерної алгебри відрізняється від звичайного інтерактивного режиму тим, що кореневий вираз введеного рядка коду стає дочірнім виразом оператору *return*, завдяки чому CLI може отримувати доступ до результатів обчислень.

Метод *run* в даному режимі зберігає результат виконання виразу для можливості його подальшого використання та виводить його у stdout.

Для демонстрації роботи реалізованого режиму системи комп'ютерної алгебри командної оболонки Methan0l обчислимо відому формулу Лейбніца для апроксимації числа π [26]:

$$\int_0^1 \frac{4}{1+x^2}\,dx \approx \pi.$$

Для численного наближення цього інтегралу використаємо формулу середніх трапецій Рімана [27]:

$$\int_a^b f(x)\,dx \approx h\sum_{i=a}^{b-h} f\left(\frac{x_i + x_{i+1}}{2}\right),$$

де крок $h = \frac{b-a}{n}$, а n – кількість елементарних відрізків інтервалу інтегрування.

Застосуємо дану формулу для апроксимації числа π за вищевказаною формулою у системі комп'ютерної алгебри Methan0l (див. рис. 3.13).

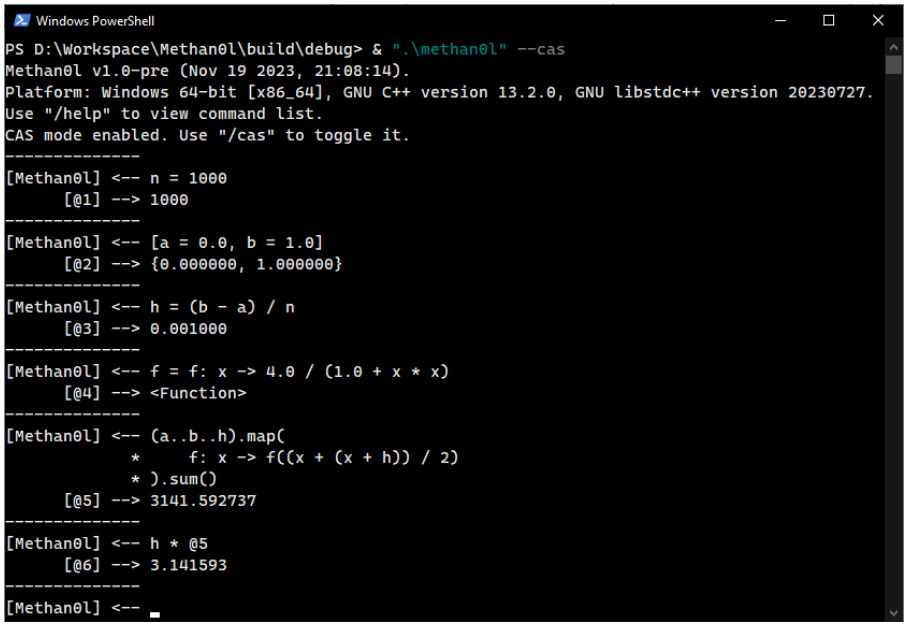

Рисунок 3.13

В результаті виконаних дій отримано апроксимацію числа π з точністю 5 десяткових знаків. В ході виконання обчислення було продемонстровано використання виразів проміжків (див. 1.4.7), маніпуляцію над ітерабельними об’єктами та використання доповнення граматики, визначеного у бібліотеці *LibCLI*, розглянутій раніше.

3.2 Реалізація та використання зовнішніх модулів

Частим призначенням зовнішніх модулів для інтерпретованих мов програмування є створення runtime-прив’язок до вже існуючих нативних бібліотек. Тому для демонстрації функціональності модулів інтерпретатора Methan0l реалізуємо прив’язку бібліотеки для консольної графіки під назвою ncurses [28]. Дана бібліотека написана мовою C і тому створення прив’язок для наданого нею API є тривіальною задачею, яка в основному зводиться до використання макросів *REG_FUNC(name)* та *REG_VAR(name)* у файлі вихідного коду модуля для кожної з публічних функцій ncurses.

Для реалізації нового динамічного модуля встановимо бібліотеку ncurses та створимо головний файл модуля під назвою «ncurses.cpp». В ньому використаємо заголовковий файл API модулів «methan0l.h» та бібліотеки ncurses «ncurses/ncurses.h», до якої буде здійснюватися реєстрація прив’язок. Після цього визначимо блок ініціалізації *LOAD_MODULE*, всередині якого буде визначено прив’язки.

Розглянемо основні з них.

Перш за все створимо runtime-відображення ідентифікатору *NULL* мови C (див. рис. 3.14).

```
VAR(NULL) = (void*)0;
```

Рисунок 3.14

Після чого створимо функції, які надаватимуть доступ до глобальних змінних *stdscr* та *curscr*, які вказують на стандартний та поточний екрани консолі (див. рис. 3.15).

```
function("stdscr", [&]() {return stdscr;});
function("curscr", [&]() {return curscr;});
```

Рисунок 3.15

Далі за допомогою макросу *REG_VAR(name)* зареєструємо всі атрибути та кольори, які можна застосувати до тексту (див. рис. 3.16).

```
REG_VAR(A_NORMAL)
REG_VAR(A_STANDOUT)
REG_VAR(A_UNDERLINE)
REG_VAR(A_REVERSE)
...
```

Рисунок 3.16

Після реєстрації всіх модульних змінних можемо перейти до прив’язування функцій ncurses за допомогою макросу *REG_FUNC(name)* (див. рис. 3.17).

```
REG_FUNC(initscr)
REG_FUNC(endwin)
REG_FUNC(clear)
REG_FUNC(refresh)
REG_FUNC(cbreak)
...
```

Рисунок 3.17

На цьому реєстрацію прив’язок бібліотеки ncurses закінчено.

Як можна побачити, створення runtime-прив’язок інтерпретатора Methan0l для бібліотек, реалізованих мовою програмування C, потребує мінімальної кількості коду та зводиться до перелічення всіх публічних змінних та функцій, наявних у конкретній бібліотеці.

Для перевірки коректності роботи створених прив’язок реалізуємо тестову програму з їх використанням.

Спочатку підключимо створений модуль та реалізуємо функцію виводу тексту по центру консольного вікна. Для цього скористуємося прив’язками функцій ncurses для отримання параметрів вказаного вікна *getmaxx*, *getmaxy* та функцією виводу тексту *mvwprintw* (див. рис. 3.18).

```
import: "$:/modules/ncurses"
print_center = fun: msg, y_offset {
    mx = getmaxx(#screen) - 1
    my = getmaxy(#screen) - 1
    mvwprintw(#screen,
        my / 2 + y_offset,
        (mx - msg.size()) / 2,
        msg
    )
}
...
```

Рисунок 3.18

Реалізована функція приймає два аргументи – текст для виводу *msg* та вертикальний відступ відносно середини екрану *y_offset*.

Після чого також створимо функцію для малювання прямокутника навколо границі видимої області вікна за допомогою того ж набору функцій з прив’язаної бібліотеки ncurses (див. рис. 3.19).

```
draw_border = fun: () {
    wattron(#screen, #A_REVERSE)
    width = getmaxx(#screen) - 1
    height = getmaxy(#screen) - 1
    (0..width).for_each(
        f: x -> mvwprintw(#screen, 0, x, " ")
    )
    (0..width).for_each(
        f: x -> mvwprintw(#screen, height, x, " ")
    )
    (0..height).for_each(
        f: y -> mvwprintw(#screen, y, 0, " ")
    )
    (0..height + 1).for_each(
        f: y -> mvwprintw(#screen, y, width, " ")
    )
}
...
```

Рисунок 3.19

Далі перейдемо до реалізації головної логіки програми. Ініціалізуємо стан ncurses, стилі тексту та зробимо декілька викликів розглянутих вище функцій (див. рис. 3.20).

```
screen = initscr()
noecho()
start_color()
use_default_colors()
wattron(screen, A_REVERSE)
print_center("[Terminal Size: "
    :: getmaxx(screen) :: "x" :: getmaxy(screen) :: "]", 0)
draw_border()
wattron(screen, A_STANDOUT)
print_center(" Methan0l & ncurses Test! ", -getmaxy(screen) / 2)
wrefresh(screen)
getch()
endwin()
```

Рисунок 3.20

Після запуску даної Methan0l-програми побачимо наступний вміст вікна консолі (див. рис. 3.21).

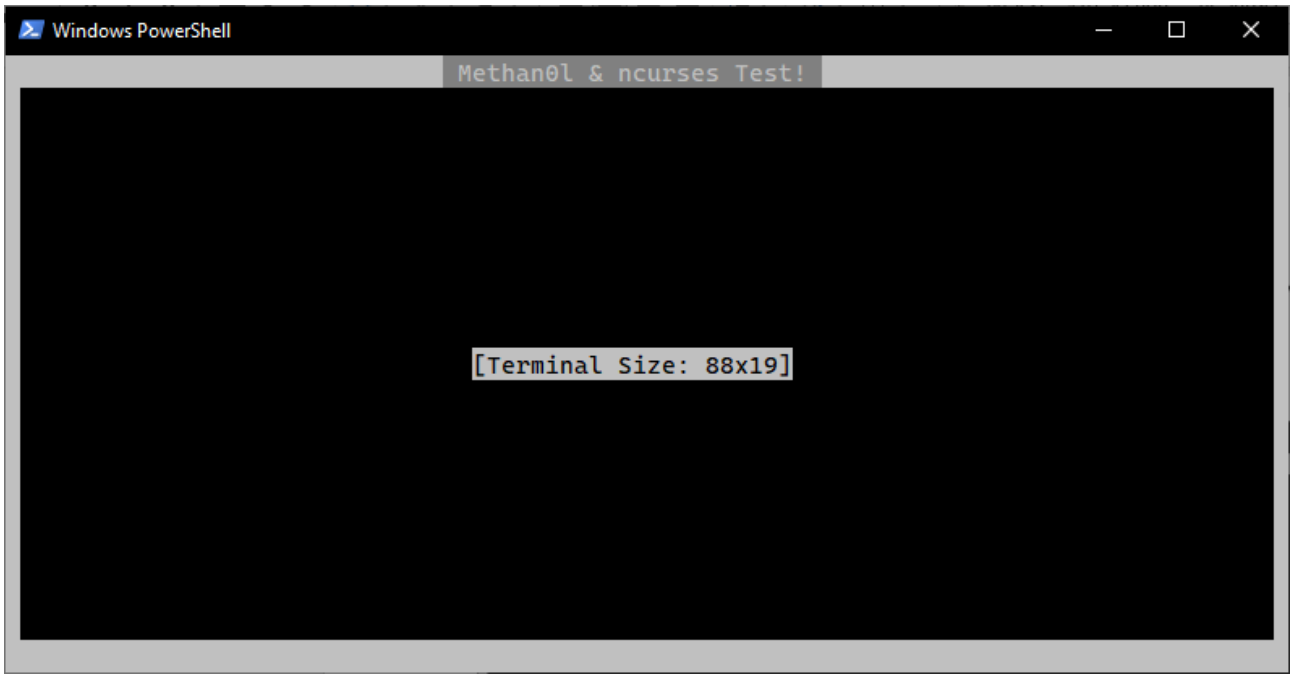

Рисунок 3.21

В результаті було створено повнофункціональну прив’язку нативної бібліотеки консольної графіки ncurses, написаної мовою програмування C, до runtime-середовища інтерпретатора Methan0l у вигляді динамічно-завантажуваного модуля.

3.3 Вбудовування інтерпретатора у існуючий проєкт

Одним з призначень Methan0l також є вбудовування в існуючі програмні продукти, написані мовою програмування C++.

Для демонстрації цього аспекту інтерпретатора було обрано приклад реалізації додатка для малювання під назвою Scribble [29], написаного з використанням фреймворку Qt. Реалізуємо інтеграцію цієї програми з інтерпретатором Methan0l у вигляді можливості автоматизації малювання за допомогою скриптингу.

Оригінальний код Scribble, як і будь-якого іншого додатка, у який є потреба вбудувати інтерпретатор Methan0l, модифікувати не треба. Єдина модифікація, яку було зроблено – додано поле для введення коду та кнопки для його виконання. Безпосередня логіка інтеграції буде повністю самостійною і буде залежати від вже реалізованого API, при цьому основний код програми залежностей від Methan0l не матиме.

Логіка малювання в Scribble реалізована у класі *ScribbleArea*, який описує віджет з підтримкою малювання та надає можливість змінювати параметри інструменту малювання, такі як колір, розмір та позиція. Таким чином, процес вбудовування Methan0l у дану програму зведеться до взаємодії з об’єктом цього класу.

Створимо окремий клас *ScribbleMethan0l*, в конструкторі якого реалізуємо ініціалізацію інтерпретатора та прив’язування всіх необхідних для малювання методів заданого об’єкта *ScribbleArea* у вигляді вільних функцій.

Розглянемо реалізацію конструктора *ScribbleMethan0l*. Перш за все

створимо об’єкт інтерпретатора та вкажемо шлях до його робочої папки для коректності ініціалізації (див. рис. 3.22).

```
ScribbleMethan0l::ScribbleMethan0l(ScribbleArea *scribble)
: scribble(scribble) {
    auto mtlPath = QCoreApplication::applicationDirPath().toStdString();
    mtlPath += "/methan0l";
    interpreter = std::make_unique<mtl::Methan0l>(mtlPath);
    ...
```

Рисунок 3.22

Далі можемо почати реалізацію прив’язок. Будемо використовувати функцію *member*, яка зв’язує об’єкт деякого класу з його методом та перетворює цю комбінацію на єдиний викликуваний об’єкт.

Спочатку прив’яжемо метод *ScribbleArea::beginDraw*, який симулює натискання лівої кнопки миші (див. рис. 3.23).

```
interpreter->register_func(
    "begin_draw",
    interpreter->bind_func(
        mtl::member(scribble, &ScribbleArea::beginDraw)
    )
);
```

Рисунок 3.23

Аналогічним чином створимо прив’язку до всіх основних методів *ScribbleArea*, а саме:

– *ScribbleArea::movePen*, який симулює рух курсору до заданої точки;
– *ScribbleArea::setPenWidth*, який дозволяє налаштовувати розмір пензля;
– *ScribbleArea::setPenColor*, який надає можливість змінювати поточний колір пензля;
– *ScribbleArea::endDraw*, який симулює відпускання лівої кнопки миші.

На цьому налаштування прив’язок завершено. Однак, для приведення їх в дію, для класу *ScribbleMethan0l* також потрібна можливість виконання Methan0l коду, в якому будуть використовуватися реалізовані прив'язки. Для цього було

створено публічний метод *execute*, реалізація якого досить тривіальна та схожа на розглянуту при створенні командної оболонки (див. 3.1).

В якості останнього кроку було додано логіку кнопки виконання Methan0l коду з області для його введення у клас головного вікна додатка *MainWindow*. На цьому процес інтеграції реалізованого інтерпретатора в існуючу програму для малювання закінчено.

Для демонстрації роботи всіх прив'язок реалізуємо невелику тестову програму, яка малюватиме кола та квадрати довільних розмірів та кольорів.

Для цього спочатку створимо функцію малювання квадрату заданого розміру з лівим нижнім кутом у заданих координатах (див. рис. 3.24).

```
draw_square = fun: start_x, start_y, size {
    begin_draw()
    move(start_x, start_y)
    move(start_x + size, start_y)
    move(start_x + size, start_y + size)
    move(start_x, start_y + size)
    move(start_x, start_y)
    end_draw()
}
```

Рисунок 3.24

Як можна помітити, ця функція цілком складається з викликів до прив'язок *ScribbleArea*. Далі реалізуємо функцію малювання кола з заданою центральною точкою та радіусом (див. рис. 3.25).

```
draw_circle = fun: start_x, start_y, radius {
    move(start_x, start_y)
    begin_draw()
    x = fun: phi, r -> r * cos(phi)
    y = fun: phi, r -> r * sin(phi)
    (0..360..0.5).for_each(
        f: phi {
            move(
                start_x + x(phi, radius),
                start_y + y(phi, radius)
            )
        }
    )
    end_draw()
}
```

Рисунок 3.25

Тепер опишемо головну логіку програми. Для демонстрації будемо здійснювати малювання 30 випадкових фігур з випадковими параметрами (рис. 3.26). Для генерації псевдовипадкових чисел використаємо стандартний клас *Random*, реалізація якого використовує клас *std::mt19937* в якості джерела випадковості.

```
rnd = new: Random()
(0..30).for_each(
    f: i {
        set_size(rnd.next_int(1, 20))
        set_color(rnd.next_double(), rnd.next_double(), rnd.next_double())
        x = rnd.next_int(0, 500)
        y = rnd.next_int(0, 500)
        size = rnd.next_int(10, 100)
        if: rnd.next_boolean() {
            draw_circle(x, y, size)
        } else: {
            draw_square(x, y, size)
        }
    }
)
```

Рисунок 3.26

Після натискання на кнопку виконання коду побачимо результат, зображений на рисунку 3.27.

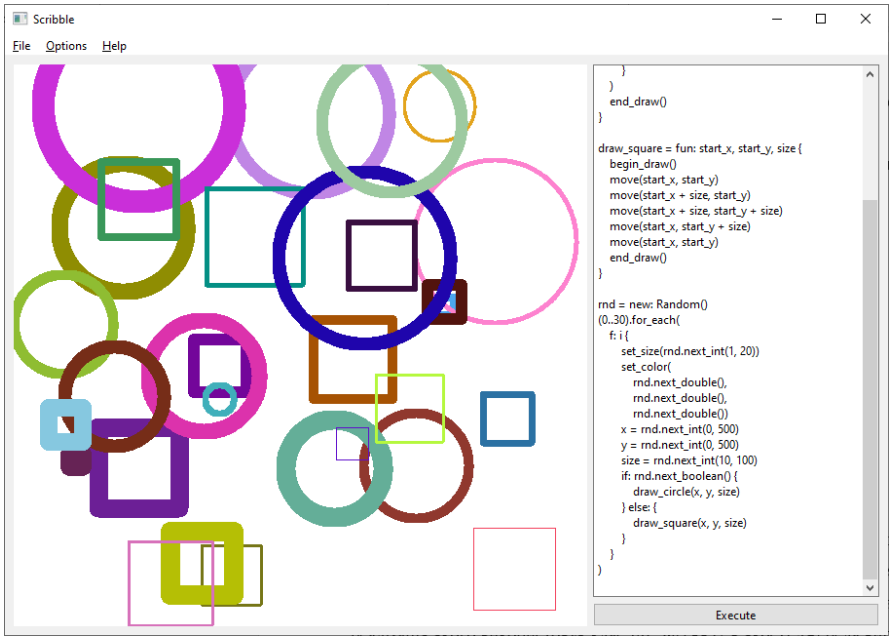

Рисунок 3.27 – Демонстрація вбудовування Methan0l

В результаті в існуючий додаток для малювання та редагування зображень було вбудовано можливість автоматизації дій графічних маніпуляцій. Наведений приклад інтеграції інтерпретатора Methan0l у додаток Scribble демонструє потужність можливостей вбудовування розробленого інтерпретатора та низьку технічну складність реалізації прив’язок існуючих нативних компонентів до його runtime-середовища.

ВИСНОВКИ

В результаті даної роботи було спроєктовано інтерпретовану динамічно-типізовану скриптову мову програмування під назвою Methan0l. Процес створення різних синтаксичних конструкцій був детально розглянутий, обґрунтований та порівняний з іншими існуючими мовами програмування.

Було також реалізовано інтерпретатор створеної мови програмування разом із різноманітними API для його вбудовування в існуючі програмні продукти та створення зовнішніх модулів.

Окрему увагу під час проєктування та реалізації інтерпретатора було приділено максимальній інтероперабельності з мовою програмування C++, на взаємодію з якою він в основному розрахований. Однією з найважливіших складових цього аспекту є реалізація runtime-декораторів стандартних типів C++ в якості найчастіше використовуваних Methan0l-класів.

В повному обсязі було розглянуто і реалізовано механізми визначення довільної лексики та граматики для інтерпретатора Methan0l. Модульна система реєстрації лексичних одиниць та синтаксичних аналізаторів окремих виразів роблять його корисним для використання в якості платформи для реалізації конкретних інтерпретаторів будь-яких предметно-орієнтованих скриптових мов програмування.

На прикладах застосування реалізованого набору інструментів для роботи з runtime-середовищем Methan0l явно продемонстровано виконання головної мети даної кваліфікаційної роботи – розробки мінімалістичної та гнучкої скриптової мови програмування разом з її інтерпретатором, який надає можливість вбудовування у сторонні програмні продукти для розширення їх функціональності.

ПЕРЕЛІК ПОСИЛАНЬ

1. Abelson H., Sussman J. Structure and Interpretation of Computer Programs. MIT Press, 1984. 856 p.
2. Placeholder type specifiers. C++ reference. URL: https://en.cppreference.com/w/cpp/language/auto (дата звернення: 03.07.2023).
3. Thain D. Introduction to Compilers and Language Design. URL: https://www3.nd.edu/~dthain/compilerbook (дата звернення: 03.07.2023).
4. Robert N. Crafting Interpreters. Genever Benning, 2021. 640 p.
5. C++ keywords. C++ reference. URL: https://en.cppreference.com/w/cpp/keyword (дата звернення: 05.07.2023).
6. While loop. C++ reference. URL: https://en.cppreference.com/w/cpp/language/while (дата звернення: 11.07.2023).
7. For loop. C++ reference. URL: https://en.cppreference.com/w/cpp/language/for (дата звернення: 11.07.2023).
8. Range-based for loop. C++ reference. URL: https://en.cppreference.com/w/cpp/language/range-for (дата звернення: 11.07.2023).
9. C++ named requirements: Container. C++ reference. URL: https://en.cppreference.com/w/cpp/named_req/Container (дата звернення: 17.09.2023).
10. Aggregate initialization. C++ Reference. URL: https://en.cppreference.com/w/cpp/language/aggregate_initialization (дата звернення: 03.07.2023).
11. Scope. C++ Reference. URL: https://en.cppreference.com/w/cpp/language/scope (дата звернення: 03.07.2023).
12. Local Variables and Blocks. Programming in Lua. URL: https://www.lua.org/pil/4.2.html (дата звернення: 17.09.2023).
13. Functions. Kotlin Documentation. URL: https://kotlinlang.org/docs/functions.html (дата звернення: 20.09.2023).

14. Non-static member functions. C++ reference. URL: https://en.cppreference.com/w/cpp/language/member_functions (дата звернення: 20.09.2023).

15. The this pointer. C++ reference. URL: https://en.cppreference.com/w/cpp/language/this (дата звернення: 20.09.2023).

16. Abstract Methods and Classes. Oracle Help Center. URL: https://docs.oracle.com/javase/tutorial/java/IandI/abstract.html (дата звернення: 25.09.2023).

17. Default Methods. Oracle Help Center. URL: https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html (дата звернення: 25.09.2023).

18. Summary of Inheritance. Oracle Help Center. URL: https://docs.oracle.com/javase/tutorial/java/IandI/summaryinherit.html (дата звернення: 25.09.2023).

19. Antero T. Classes Versus Prototypes: Some Philosophical and Historical Observations. *Journal of object-oriented programming*. 1997. Vol. 10. P. 44–50.

20. Hitonoriol/Methan0l. GitHub. URL: https://github.com/Hitonoriol/Methan0l (дата звернення: 02.10.2023).

21. Escape sequences. C++ reference. URL: https://en.cppreference.com/w/cpp/language/escape (дата звернення: 02.10.2023).

22. Vaughan R. Top Down Operator Precedence. Massachusetts Institute of Technology, 1973. 11 p.

23. Chunks. Programming in Lua. URL: https://www.lua.org/pil/1.1.html (дата звернення: 02.10.2023).

24. Constructors and member initializer lists. C++ reference. URL: https://en.cppreference.com/w/cpp/language/constructor (дата звернення: 11.10.2023).

25. Using the Python Interpreter. Python 3.12.0 documentation. URL: https://docs.python.org/3/tutorial/interpreter.html#interactive-mode (дата звернення: 11.10.2023).

26. Roy R. The Discovery of the Series Formula for π by Leibniz, Gregory and Nilakantha. *Mathematics Magazine*, 1990. Vol. 63. P. 291–306.

27. Kenneth K. Calculus: Theory And Applications, Volume 1. World Scientific Publishing Company, 2010. 500 p.

28. Thomas D. NCURSES – New Curses. URL: https://invisible-island.net/ncurses/ (дата звернення: 11.10.2023).

29. Scribble Example. Qt Documentation. URL: https://doc.qt.io/qt-6/qtwidgets-widgets-scribble-example.html (дата звернення: 11.10.2023).

30. C++ Operator Precedence. C++ reference. URL: https://en.cppreference.com/w/cpp/language/operator_precedence (дата звернення: 11.07.2023).

31. std::variant. C++ reference. URL: https://en.cppreference.com/w/cpp/utility/variant (дата звернення: 02.10.2023).

ДОДАТОК А

Не обчислені вирази

Так як будь яку граматичну сутність у розроблюваній мові програмування представлено виразом, була реалізована можливість обмеженої маніпуляції цією концепцією, а саме – можливість отримання посилання на вираз без його обчислення за допомогою оператора *noeval*.

Даний оператор повертає значення типу *Expression*, яке посилається на не обчислений вираз правого операнду у AST поточної програми. При обчисленні або виконанні значення типу *Expression* застосовується аналогічна операція до виразу, на який воно посилається.

Цей механізм можна використовувати для лінивих обчислень або для зменшення об’єму коду, виносячи повторювану логіку у не обчислені вирази замість окремих функцій або юнітів. При цьому не обчислені вирази не зберігають контекст, тобто обчислення результату виразу, повернутого *noeval: x* повертатиме різне значення в залежності від значення змінної *x* у конкретній зоні видимості, де буде обчислено вираз.

Приклад використання оператора *noeval* наведено на рисунку А.1.

```
x = noeval: y * 2
y = 2
<% $"Result: {}", x + y
```

Рисунок А.1

Вивід програми зображено на рисунку А.2.

```
Result: 6
```

Рисунок А.2

Даний код еквівалентний обчисленню виразу $2 + 2 * 2$. На першому рядку змінній *x* присвоєно посилання на вираз *y * 2*. Далі відбувається присвоєння числа 2 змінній *y*. Після цього на останньому рядку під час виводу обчислюється значення виразу додавання *x + y*, в ході якого спочатку обчислюється вираз *x*, значення якого – *y * 2*, тобто 4. Далі обчислюється правий операнд *y*, значення якого 2. В результаті всього виразу після накладання оператору додавання отримується число 6.

ДОДАТОК Б

Пріоритет операторів

Кожний оператор в Methan0l, окрім свого значення і синтаксичної категорії, також має деяку властивість, яка виражає «силу», з якою він прив’язується до своїх операндів. Називатимемо цю властивість пріоритетом операторів. Вона представлена цілим числом починаючи з 1. Чим більше це число, тим сильніше оператор прив’язаний до операнду. Рівні пріоритетів базових арифметичних та інших традиційних операторів збігаються з пріоритетністю операторів в C++ [30].

Розглянемо для наочності приклад операторів з різними рівнями пріоритету.

Нехай задано два інфіксних оператори, один префіксний та один постфіксний. Інфіксний оператор (A) має пріоритет 1, інфіксний оператор (B) – пріоритет 2, префіксний оператор (C) – пріоритет 3, а постфіксний оператор (D) – пріоритет 4. Розглянемо наступний вираз:

$$(C)var1(A)var2(A)var3(B)var1(D),$$

де $var1$, $var2$, $var3$ – деякі ідентифікатори.

Для того, щоб перетворити цю послідовність токенів у деревовидну структуру з виразів, інтерпретатор в процесі парсингу користуватиметься значеннями пріоритетів, які вказано вище, групуючи оператори з потрібною кількістю операндів. В результаті внутрішнє цей вираз виглядатиме наступним чином:

$$[[(C)var1](A)var2](A)[var3(B)[var1(D)]].$$

Тобто спочатку буде обчислено $(C)var1$, потім на результат буде накладено (A) з правим операндом $var2$, після чого обчислиться $var1(D)$, а результат буде використано як правий операнд оператору (B) з лівим операндом $var3$. Після чого до двох отриманих результатів у виразі верхнього рівня буде застосовано оператор (A).

В таблиці Б.1 наведено всі рівні пріоритетів операторів, реалізованих в розроблюваній мові програмування.

Таблиця Б.1 – Пріоритети операторів

Пріоритет	Оператори	Опис
1	noeval	Оператор не обчисленого виразу.
2	<% %% %>	Оператори вводу-виводу.
3	+= -= *= /= &= |= ^= >>= <<=	Складене присвоювання.
4	= <-	Присвоювання.
5	:: expr1..expr2..expr3	Рядкова конкатенація, діапазони.
6	expr1 ? expr2 : expr3	Тернарний оператор.
7	||	Булеве OR.
8	&&	Булеве AND.
9	|	Бітове OR.
10	^	Бітове XOR.
11	&	Бітове AND.
12	== !=	Оператори рівності.
13	> < >= <=	Порівняльні оператори.
14	>> <<	Бітовий зсув.
15	+ -	Арифметичні адитивні оператори.
16	* / %	Арифметичні мультиплікативні оператори.
17	-expr !expr ~expr	Префіксні оператори.
18	++expr –expr	Префіксний інкремент, декремент.

Продовження табл. Б.1

Пріоритет	Оператори	Опис
19	expr!	Постфіксний оператор повертання.
20	. @	Оператори доступу.
21	expr1[expr2]	Оператор індексування.
22	expr1(expr2, expr3, …)	Виклик функції.
23	expr++ expr--	Постфіксний інкремент, декремент.
24	const var	Найвищий пріоритет.

Оператор *noeval* має найнижчий пріоритет через його призначення – захоплювати довільні вирази, що знаходяться праворуч від нього та повертати їх у не обчисленій формі. Якщо пріоритет цього оператора був би вище, то для того, щоб використовувати вирази з пріоритетом менше або рівному *noeval* як його операнди, довелося б групувати їх у дужки.

Оператори виводу також мають низький пріоритет зі схожої причини, що й *noeval* – через потребу мати можливість використовувати довільні вирази для виведення їх результату у рядковій формі.

Найвищий же пріоритет мають оператори *const* та *var*. Для *var* таке значення пріоритетності обумовлено тим, що він повинен застосовуватися тільки до ідентифікаторів. Так як ідентифікатори можуть бути операндами інших операторних виразів, існує потреба забезпечити таку умову, щоб якщо ідентифікатор після *var* був у складі деякого виразу, то оператор *var* прив’язувався саме до ідентифікатора, а не до всього виразу.

ДОДАТОК В

Зарезервовані ідентифікатори

Зарезервованими ідентифікаторами називають деякі абетково-цифрові імена, що не можуть бути перевизначені користувачем у вигляді назви функції, змінної тощо. Замість цього вони мають деяке заздалегідь визначене стандартом мови значення або синтаксичний сенс (якщо зарезервований ідентифікатор використовується у операторному виразі).

Незважаючи на те, що всі ключові слова-оператори у мові програмування Methan0l не є зарезервованими, необхідність мати таку категорію виразів все одно присутня. Methan0l має дуже обмежену кількість зарезервованих ідентифікаторів, серед яких:

– *true, false* – булеві константи, що мають відповідні їх назвам значення;
– *newl* – символ перенесення рядка («\n»);
– *nil* – константа, що позначає відсутність значення;
– *break* – ідентифікатор, який може бути використаний у правій частині оператора *return* для виходу з циклу.

ДОДАТОК Г

Менеджер класів

При визначенні класів в Methan0l, вони реєструються у менеджері класів, який описано класом *TypeManager*. Ієрархічні зв'язки між класами та інтерфейсами не підтримуються, тобто концепції внутрішніх класів не існує, через що ім'я кожного класу повинно бути унікальним.

Менеджер класів містить три різних словника типів *IdIndex*, *NameIndex* та *NativeIndex*, в яких індексуються зареєстровані типи, зв'язані з їх унікальними ідентифікаторами, іменами та нативними ідентифікаторами типів.

Існує два перевантаження методу *TypeManager::register_type* для реєстрації нового класу. Один з них використовується напряму в основному для чистих runtime-класів, що визначені повністю Methan0l-кодом. Розглянемо його реалізацію (див. рис. Г.1).

```
void TypeManager::register_type(Shared<Class> type)
{
    auto type_id = type->get_id();
    auto &native_id = type->get_native_id();
    classes.emplace(type_id, type);
    class_index.emplace(type->get_name(), type.get());
    if (type->is_native()) {
        native_classes.emplace(native_id, type.get());
    }
}
```

Рисунок Г.1

Цей метод додає клас у головний індекс (за унікальними ідентифікаторами класів) та в індекс за іменами. Після цього, якщо клас містить нативну частину, то він також додається в індекс нативних класів.

Для реєстрації класів за іменем C++-типу реалізовано спеціальне перевантаження *TypeManager::register_type* (див. рис. Г.2).

```
template<typename T>
inline void register_type(Allocator<T> allocator = {})
{
    if constexpr (is_callable<T>::value) {
        T registrator(context);
        registrator();
    }
    else {
        register_type(
            std::allocate_shared<T>(allocator, context)
        );
    }
}
```

Рисунок Г.2

Тут спочатку відбувається compile-time перевірка на те, чи є заданий тип *T* викликуваним. Якщо ця умова виконується, то створюється об’єкт типу *T* з конструктором, який приймає єдиний аргумент у вигляді посилання на контекст інтерпретатора. Після цього створений об’єкт викликається без аргументів. Даний механізм є частиною API прив’язки нативних C++-класів до runtime-класів інтерпретатора Methan0l, який більше детально розглянемо у наступному пункті. Якщо ж умова не виконується, то вважається, що клас не використовує API прив’язування і виконує всю необхідну ініціалізацію в своєму конструкторі, тому реєстрація відбувається звичайним чином.

ДОДАТОК Д

Форматування рядків

Для форматування даних, представлених у рядковій формі, було також реалізовано стандартний метод *format(template, arg1, arg2, ...)* класу *String* разом з окремим виразом форматованих рядків: *$"template", arg1, arg2, ...*. При обчисленні даний вираз повертає результат форматування.

Форматувальник заміняє спеціальні токени «{}» всередині рядка-шаблону на аргументи. Ці токени можуть бути порожніми або мати номер аргументу, на який їх треба замінити, та один або декілька модифікаторів. Розглянемо всі можливі способи використання токенів форматувальника:

- {} – заміняється наступним аргументом у списку;
- {n}, де *n* – ціле число – заміняється аргументом під номером *n (*номери аргументів починаються з 1. Вказується перед модифікаторами.
- Можливі модифікатори форматування:
- %n – бажана ширина вставленого тексту;
- -l, -r, -c – вирівнювання по лівій, правій сторонах та центру;
- .n – кількість знаків після крапки у числах з плаваючою точкою.

ДОДАТОК Е

Перевантаження операторів індексування

На відміну від більшості інших мов програмування, в Methan0l було реалізовано декілька підвидів оператора індексування, що традиційно позначається як *expr1[expr2]*. Цей оператор не має поведінки за замовчуванням і повинен бути перевантаженим для конкретного класу, щоб мати можливість його використання.

У таблиці Е.1 наведено всі види операторів індексування та синтаксис їх перевантаження.

Таблиця Е.1 – Види операторів індексування

Оператор	Опис	Метод для перевантаження	Кількість аргументів
expr1[expr2]	Отримання елементу за індексом.	`[]`	1
expr1[]	Додавання елементу за посиланням.	`append[]`	0
expr1[~expr2]	Видалення елементу за індексом.	`remove[]`	1
expr1[~]	Очищення контейнеру.	`clear[]`	0
expr1[do: expr2]	Виконання операції для кожного елементу.	`foreach[]`	1
expr1[->expr2]	Вставлення елементу на позицію за індексом.	`insert[]`	1
expr1[a..b..c]	Створення нового контейнеру з елементами з заданого проміжку індексів.	`slice[]`	1

ДОДАТОК Ж

Приклад реалізації виразу

Розглянемо один з найчастіше використовуваних виразів, а саме – вираз визначення списків та множин, який реалізовано у класі *ListExpr*.

Члени класу *ListExpr* наведено на рисунку Ж.1.

```
class ListExpr : public Expression
{
   private:
      ExprList exprs;
      bool as_set;
   public:
      ListExpr(ExprList exprs, bool as_set = false);
   ...
};
```

Рисунок Ж.1

Елементи майбутнього списку або множини зберігаються у списку *exprs*, що має тип *ExprList*. Цей тип представляє з себе список вказівників на вирази. Прапорець *as_set* в даному класі керує тим, який саме об'єкт буде створено при обчисленні виразу: якщо його значення *true*, буде створено об'єкт класу *Set*, інакше – об'єкт класу *List*. Результуючий контейнер буде містити значення, обчислені зі списку виразів *exprs*.

Реалізацію методу *evaluate* для цього виразу наведено на рисунку Ж.2.

```
Value ListExpr::evaluate(Interpreter &context)
{
   return as_set ?
         create_and_populate<ValSet>(context, exprs) :
         create_and_populate<List>(context, exprs);
}
```

Рисунок Ж.2

Цей метод виконує вищеописану логіку, а безпосереднє створення стандартних контейнерних класів відбувається у методі *create_and_populate* (див. рис. Ж.3).

```
template<typename T>
Value create_and_populate(Interpreter &context, ExprList& exprs)
{
    auto container = context.make<T>();
    return container.template as<T>([&](auto &ctr) {
        for (auto &expr : exprs)
            insert(ctr, expr->evaluate(context));
    });
}
```

Рисунок Ж.3

Тут в залежності від типу *T* буде створено відповідний об’єкт runtime-типу. Після цього до нативної частини створеного об’єкта отримується доступ через метод *Value::as*. Отриманий нативний контейнер заповнюється обчисленими зі списку виразів значеннями.

Варто зазначити, що створення будь-якого об’єкта для його використання зсередини runtime-середовища Methan0l потребує його створення саме через контекст інтерпретатора. Інакше встановити зв’язок між нативним та runtime класами об’єкта неможливо.

ДОДАТОК И

Константне обчислення

Через особливості реалізації механізму пошуку ідентифікаторів при обчисленні посилань на змінні, було вирішено створити механізм констант та константного обчислення. Для цього були додані два префіксних оператора: *$* та *const*. Вони мають еквівалентну поведінку, але *$* розрахований на те, що його операндом буде ідентифікатор, а *const* підтримує будь-які вирази.

Константне обчислення відбувається одноразово в процесі парсингу черги токенів і полягає в тому, що операнди операторів *$* та *const* обчислюються, після чого результат обчислення заміняє константний вираз. Таким чином, самі константі оператори не присутні у AST інтерпретатора під час роботи програми, а на їх місці знаходяться буквальні вирази, що містять результат константних обчислень.

Такий підхід значно економить час, що витрачався б на пошук глобальної змінної, визначеної у runtime-середовищі інтерпретатора. Крім того, константні оператори дозволяють виконувати цілі юніти під час парсингу, що дає можливість виконувати обчислювально складні операції тільки один раз перед початком роботи програми.

Приклад використання обох розглянутих операторів константного обчислення наведено на рисунку И.1.

```
const: {
        N = (1..10).sum()
        STR = "String Constant"
}
for: (i, 1..10) <% $"[{}] N + {} = {}", $STR, i, $N + i
foo = const: {
        a = 5..10
        b = 25..30
        return: a.sum() + b.sum()
}
<% $"foo = {}", foo
```

Рисунок И.1

ДОДАТОК К

Структури даних ядра інтерпретатора

Ядро інтерпретатора Methan0l (реалізація всієї загальної логіки обробки програм від лексичного аналізу до виконання) використовує багато різних вузькоспеціалізованих структур даних для забезпечення коректної роботи інтерпретатора та його взаємодії з нативним кодом. Розглянемо найбільш часто застосовувані з них.

Значення. Будь-яке значення у runtime-середовищі інтерпретатора Methan0l описується об’єктом класу *Value*, єдиним полем якого є *value*, що має тип *std::variant<T...>*. Цей тип є безпечним відносно типів аналогом *union* з мови програмування C [31]. Для значень він використовується для того, щоб мати можливість ділити одну й ту ж область пам’яті як для примітивів, так і для об’єктів.

До списку альтернатив *value* входять арифметичні типи, булевий та символьний типи, вказівники на викликувані типи (такі як *Unit* або *Function*), клас *Object* (яким виражається будь-який об’єкт в runtime-середовищі), та клас *std::any*, який використовується для представлення будь-якого нативного об’єкта, для якого не потрібна наявність прямого інтерфейсу взаємодії з runtime-середовищем (наприклад, нативні структури з зовнішніх модулів, що мають передаватися тільки до нативного коду).

Розмір значень класу *Value* дорівнює трьом вказівникам. Тобто, на 32-бітних платформах він дорівнює 16 байтів (з урахуванням вирівнювання даних), а на 64-бітних – 24 байти.

Всі змінні, функції, об’єкти тощо, до яких можна отримати доступ з Methan0l-коду виражаються типом *Value*. Таким чином, цей клас надає можливість інтероперабельності між runtime-середовищем та нативним кодом.

В класі *Value* реалізовано великий набір методів для отримання безпечного відносно типів доступу до поточного значення його об’єктів. Серед таких

методів, наприклад, *Value::get<T>*, який надає доступ до значення за посиланням. Для цього методу також існує окрема шаблонна спеціалізація для отримання доступу до нативного представлення прив'язаних до інтерпретатора класів. У випадку, коли об'єкт *Value* не містить значення запитуваного типу, буде кинуто виняток.

Клас. Класи, визначені у runtime-середовищі Methan0l, або прив'язані до нього описано класом *Class* (див. рис. К.1).

```
class Class
{
   private:
      std::string name;
      TypeID id;
      bool native = false;
      Class *superclass = this;
      std::vector<Class*> interfaces;
      DataTable class_data;
      DataTable proto_object_data;
   ...
```

Рисунок К.1

Поля *name* та *id* повинні однозначно та унікально ідентифікувати визначений клас, тому що в Methan0l відсутня концепція внутрішніх класів або просторів імен. Булевий член *native* встановлюється як *true* тільки тоді, коли даний клас є відображенням деякого нативного C++-класу. Поля *superclass* та *interfaces* відповідно містять вказівники на батьківський клас та всі реалізовані інтерфейси. Найголовнішими ж членами є таблиця даних *class_data*, що описує інформацію про клас (в основному всі методи класу) та *proto_object_data*, що містить початковий стан нового об'єкта, який задано у виразі визначення класу.

Встановлення батьківського класу та додавання реалізованих інтерфейсів реалізовані в методі *Class::inherit* та *Class::implement*. Для обох цих методів існують перевантаження для імен батьківських класів та для вказівників на самі об'єкти *Class*, які будуть наслідувані.

Реєстрацію методів класів реалізовано у методі *Class::register_method*. Цей механізм використовує метод прив'язки функцій інтерпретатора, який буде розглянуто пізніше.

Виклик методів Methan0l-класів також відбувається через клас *Class*. Це необхідно з причини того, що всі методи класів зберігаються у таблиці *Class::class_data* та не містяться у стані окремих об'єктів. Таким чином, при викликанні методу деякого об'єкта він звертається до свого класу та викликає метод *Class::invoke_method(Object&, const std::string&, Args&)*, який, у свою чергу, отримує метод з таблиці *class_data*, після чого викликає *Interpreter::invoke_method*, передаючи йому замість імені методу його викликуваний об'єкт. Реалізація *Interpreter::invoke_method* зображена на рисунку К.2.

```
Value Interpreter::invoke_method(Object &obj, Value &method, Args &args)
{
    auto argcopy = args;
    argcopy.push_front(LiteralExpr::create(obj));
    return method.is<Function>()
            ? invoke(method.get<Function>(), argcopy)
            : method.get<NativeFunc>()(argcopy);
}
```

Рисунок К.2

Як можна побачити, імпліцитний аргумент *this*, що вказує на об'єкт, для якого викликано метод, додається до списку аргументів виклику безпосередньо під час його здійснення. Після цього, в залежності від того, чи є викликуваний об'єкт методу нативним або runtime-об'єктом здійснюється різна логіка виклику.

Створення об'єктів класів реалізовано у методі *Class::create(Args&)*, який спочатку створює не ініціалізований об'єкт, після чого викликає його конструктор.

Об'єкт. Об'єкти класів в Methan0l описано класом *Object*. Він має всього одне поле – таблицю даних об'єкта. Весь його стан зберігається в ній для можливості доступу як з runtime-середовища, так і з нативної частини.

Даний клас містить набір допоміжних методів для роботи з об’єктами, які забезпечують зручний доступ до полів, методів та їх нативного вмісту. Для об’єктів також реалізовано механізм копіювання у методі *Object::copy*, який є тривіальним для runtime-об’єктів, так як зводиться до простого копіювання таблиць даних, але для нативних об’єктів цей механізм реалізовано дещо складніше. Розглянемо його (рис. К.3).

```
template<typename T>
static inline Object copy_native(Object &obj, Allocator<T> alloc = {})
{
    auto new_obj = copy(obj);
    auto &original_native = *mtl::any_cast<std::shared_ptr<T>&>(
        obj.get_native_any()
    );
    auto new_native = std::allocate_shared<T>(alloc, original_native);
    new_obj.set_native(new_native);
    return new_obj;
}
```

Рисунок К.3

Спочатку здійснюється повне копіювання runtime-стану об’єкта, після чого отримується доступ до нативного вмісту переданого об’єкта та створюється новий об’єкт ідентичного йому C++-класу з використанням копіюючого конструктора. Результуючий вказівник з підрахуванням посилань встановлюється як нативний вміст скопійованого об’єкта.

ДОДАТОК Л

Приклад прив’язування оператора

Розглянемо використання механізму прив’язування на прикладі інфіксного оператору *is*, що повертає булеве значення, яке відображає, чи є ліва частина виразу об'єктом типу правої частини. Цей оператор є частиною бібліотеки *LibData* з набору стандартних бібліотек Methan0l. Розглянемо його прив’язку (див. рис. Л.1).

```
infix_operator(Tokens::INSTANCE_OF, LazyBinaryOpr([&](auto lhs, auto rhs) {
    auto l = val(lhs);
    return instanceof(l, rhs);
}));
```

Рисунок Л.1

Як видно, цей оператор є лінивим, тобто якщо нам потрібно обчислити значення операнду, це потрібно робити експліцитно. В даному випадку завжди обчислюється значення тільки лівого операнду, а правий обчислюється тільки за певних умов.

Розглянемо реалізацію методу *LibData::instanceof* (див. рис. Л.2).

```
bool LibData::instanceof(Value &lhs, ExprPtr rhs_expr)
{
    auto type = core::resolve_type(*context, *rhs_expr);
    if (lhs.object()) {
        auto clazz = lhs.get<Object>().get_class();
        return clazz->equals_or_inherits(
            &context->get_type_mgr().get_class(type.type_id())
        );
    }
    else {
        return lhs.type_id() == type.type_id();
    }
}
```

Рисунок Л.2

Перш за все в цьому методі отримується тип даних або класовий тип, який описує права частина. Функція *core::resolve_type* проводить ряд перевірок на те, про який саме тип несе інформацію вираз. Цей вираз, наприклад, може бути ідентифікатором. В такому випадку його ім’я може вказувати на назву класу або на об’єкт деякого класу. Продовжуючи логіку цього випадку проводиться повний аналіз інформації про тип. Далі у методі *LibData::instanceof* виконується перевірка на те, чи є ліва частина об’єктом або примітивом. Для об’єктів виконується перевірка приналежності значення лівої частини типу, отриманого з правої за допомогою методу *Class::equals_or_inherits*. Ця перевірка враховує наслідування та реалізацію інтерфейсів. В протилежному випадку відбувається порівняння ідентифікатора типу даних лівої та правої частин.

Прив’язування інших двох типів операторів відбувається за аналогічним принципом.

ДОДАТОК М

Аргументи командного рядка CLI

Таблиця М.1 – Аргументи командного рядка Methan0l CLI

Формат	Опис
--no-exit	Якщо аргумент вказано, CLI не завершує роботу після виконання програми.
--max-mem=123	Максимальний розмір купи пам’яті інтерпретатора.
--initial-mem=123	Початковий розмір купи пам’яті інтерпретатора.
--no-heap-limit	Якщо аргумент вказано, то при вичерпанні доступної пам’яті інтерпретатор не кине виняток, а продовжить розширювати купу.
--version	Якщо аргумент вказано, CLI виводить версію інтерпретатора та завершує роботу.