НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки

Кафедра інформаційних систем та технологій

До захисту допущено:

Завідувач кафедри

_____ Олександр РОЛІК

«___»_____20__ р.

Дипломний проєкт

на здобуття ступеня бакалавра
за освітньо-професійною програмою «Інформаційні управляючі системи та технології»
спеціальності 126 «Інформаційні системи та технології»
на тему: «Інтерпретатор мови ChocoPy»

Виконав (-ла):
студент (-ка) IV курсу, групи ІС-з91
Кравчук Олександр Сергійович _____

Керівник:
кандидат технічних наук, доцент кафедри
Завгородня Ганна Анатоліївна _____

Рецензент:
завідувач кафедри системного проектування,
доктор технічних наук, професор
Мухін Вадим Євгенійович _____

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент (-ка) _____

Київ – 2023 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

Рівень вищої освіти – перший (бакалаврський)

Спеціальність – 126 «Інформаційні системи та технології»

Освітньо-професійна програма «Інформаційні управляючі системи та технології»

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Олександр РОЛІК
«___»_____20__ р.

ЗАВДАННЯ
на дипломний проєкт студенту
Кравчуку Олександру Сергійовичу

1. Тема проєкту «Інтерпретатор мови ChocoPy», керівник проєкту Завгородня Ганна Анатоліївна, к.т.н., доцент, затверджені наказом по університету від «31» травня 2023 р. № 2102-с

2. Термін подання студентом проєкту: 12 червня 2023 року

3. Вихідні дані до проєкту:

- програмне забезпечення повинно мати текстовий інтерфейс і працювати в терміналі ОС Windows та/або Linux;
- обмеження використання системних ресурсів: оперативна пам'ять 256 MB, диск 200 MB;
- програмне забезпечення повинно відповідати формальній специфікації мови ChocoPy та проходити автоматизовані тести;
- тестові дані можуть бути взяті з GitHub репозиторію курсу CS 164 університету Берклі;
- повідомлення про помилки можуть відрізнятися від таких, що зазначені в очікуваних тестових даних, якщо суть і місцезнаходження помилки в коді залишається зрозумілою;

- проходження бенчмарк тестів не є обов'язковим, проте бажане.

4. Зміст пояснювальної записки:
   1) дослідити предметну область;
   2) вивчити документацію мови ChocoPy;
   3) обрати технології розробки;
   4) розробити програмне забезпечення;
   5) провести тестування програмного забезпечення.

5. Перелік графічного матеріалу (із зазначенням обов’язкових креслеників, плакатів, презентацій тощо)
   1) UML–діаграма класів інтерпретатора ChocoPy (кресленик);
   2) Блок–схема алгоритму лексичного аналізу (кресленик);
   3) Блок–схема алгоритму синтаксичного аналізу (кресленик);
   4) UML–діаграма класів семантичного аналізу (кресленик).

6. Дата видачі завдання 1 березня 2023 року

Календарний план

№ з/п	Назва етапів виконання дипломного проєкту	Термін виконання етапів проєкту	Примітка
1.	Вивчення літератури за тематикою проєкту	01.03.2023	
2.	Підготовка матеріалів першого розділу пояснювальної записки	10.04.2023	
3.	Підготовка матеріалів другого розділу пояснювальної записки	17.04.2023	
4.	Підготовка матеріалів третього розділу пояснювальної записки	24.04.2023	
5.	Підготовка матеріалів четвертого розділу пояснювальної записки	01.05.2023	
6.	Програмна реалізація інтерпретатора мови ChocoPy	08.05.2023	
7.	Тестування інтерпретатора мови ChocoPy	15.05.2023	
8.	Підготовка матеріалів п'ятого розділу пояснювальної записки	22.05.2023	
9.	Підготовка матеріалів шостого розділу пояснювальної записки	29.05.2023	

Студент Олександр КРАВЧУК

Керівник Ганна ЗАВГОРОДНЯ

АНОТАЦІЯ

Кравчук О.С. Інтерпретатор мови ChocoPy. КПІ ім. Ігоря Сікорського, Київ, 2023.

Проєкт містить 88 с. тексту, 24 рисунки, 4 таблиці, посилання на 28 літературних джерел, 5 додатків та 4 конструкторські документи.

Ключові слова: AST, BNF, CHOCOPY, EBNF, PYTHON, ІНТЕРПРЕТАТОР, КОМПІЛЯТОР, ЛЕКСИЧНИЧНИЙ АНАЛІЗ, СЕМАНТИЧНИЙ АНАЛІЗ, СИНТАКСИЧНИЙ АНАЛІЗ.

Об’єктом розробки є мова програмування ChocoPy.

Мета розробки – реалізація інтерпретатора мови програмування ChocoPy.

У дипломному проєкті розглянуті теоретичні та практичні аспекти побудови компіляторів та інтерпретаторів. Було розроблено алгоритми лексичного, синтаксичного і семантичного аналізу. Інтерпретація вихідного коду відбувається шляхом обходу AST. Виконано кросплатформний портативний консольний додаток на технології Java у вигляді JAR–файлу.

Програмне забезпечення може бути використано як основа для створення інтерпретаторів і компіляторів інших мов програмування.

SUMMARY

Kravchuk O.S. An Interpreter for ChocoPy. Igor Sikorsky KPI, Kyiv, 2023.

The project contains 88 pages of text, 24 figures, 4 tables, references to 28 literary sources, 5 annexes and 4 design documents.

Keywords: AST, BNF, CHOCOPY, COMPILER, EBNF, INTERPRETER, LEXICAL ANALYSIS, PYTHON, SEMANTIC ANALYSIS, SYNTACTIC ANALYSIS.

The object of development is the ChocoPy programming language.

The purpose of the development – an implementation of the ChocoPy programming language interpreter.

The graduation project considers the theoretical and practical aspects of building compilers and interpreters. Algorithms for lexical, syntactic, and semantic analysis were developed. A source code is interpreted by traversing an AST. A cross–platform portable console application based on Java technology in the form of a JAR file has been developed.

The software can be used as a basis for creating interpreters and compilers for other programming languages.

Номер рядка	Формат	Позначення	Найменування	Кільк. аркушів	Номер екзем.	Примітка
1			Документація загальна			
2						
3			Знову розроблена			
4						
5	А4	ІСз91.020БАК.004 ПЗ	Пояснювальна записка	88		
6	А3	ІСз91.020БАК.004 Д1	Інтерпретатор мови ChocoPy.	1		
7			UML–діаграма класів			
8			інтерпретатора ChocoPy			
9	А3	ІСз91.020БАК.004 Д2	Інтерпретатор мови ChocoPy.	1		
10			Блок–схема алгоритму			
11			лексичного аналізу			
12	А3	ІСз91.020БАК.004 Д3	Інтерпретатор мови ChocoPy.	1		
13			Блок–схема алгоритму			
14			синтаксичного аналізу			
15	А3	ІСз91.020БАК.004 Д4	Інтерпретатор мови ChocoPy.	1		
16			UML–діаграма класів			
17			семантичного аналізу			
18						
19						
20						
21						
22						
23						
24						
25						
26						
27						
28						

Зм.	Аркуш	№ докум.	Підпис	Дата	ІСз91.020БАК.004 ТП		
Розроб.		Кравчук О.С.			Інтерпретатор мови ChocoPy. Відомість проєкту	Літ.	Аркуш	Аркушів
Керівн.		Завгородня Г.А.				Т	1	1
						КПІ ім. Ігоря Сікорського Група ІС-з91		
Затв.								

Пояснювальна записка
до дипломного проєкту
на тему: «Інтерпретатор мови ChocoPy»

Київ – 2023 року

ЗМІСТ

ВСТУП

Компілятори – одна з найважливіших тем комп'ютерних наук. Крім мов загального призначення (C++, Java, Python) є багато предметно–орієнтованих мов (з англ. domain–specific language, DSL), для вирішення конкретних завдань. До них відносяться скриптові мови (CFML, Emacs Lisp, Vim Script, Tcl), шаблонізатори (Jinja, Mustache), конфігураційні файли (INI), генератори парсерів (ANTLR, Lex, Yacc, Bison), мови серіалізації (YAML, JSON) та розмітки даних (HTML, XML, CSS), утиліти для трансформації тексту (Sed, AWK), мови трансформації XML (XSLT, XQuery), командні оболонки (Bash, Scsh), мова для взаємодії з базою даних (SQL), утиліти для побудови програм (Make, cpp), обгортки зв’язування програм, написаних на різних мовах (SWIG, IDL) тощо.

Актуальність проблеми. Майже кожен великий програмний проєкт потребує набір таких мов. За можливості необхідно використовувати готові рішення. Проте існує ймовірність, що доведеться написати власний парсер. Навіть використовуючи існуючу реалізацію, в кінці кінців необхідно буде відлагоджувати та підтримувати її. Для цього потрібно розуміти будову подібних програм.

Сутність і значущість. Дипломний проєкт розглядає практичні і теоретичні аспекти побудови компіляторів та інтерпретаторів. У якості прикладу розглядається інтерпретатор мови програмування ChocoPy [1]. Наразі не знайдено готових розробок інтерпретаторів цієї мови.

Вихідні дані. Реалізація має відповідати формальній специфікації мови ChocoPy. Перевірка коректності роботи має відбуватися через мануальне тестування, а також автоматизовані модульні тести, тестові дані до яких можна взяти з GitHub репозиторію курсу «CS 164. Мови програмування та компілятори» в Каліфорнійському університеті в Берклі [2].

Програмне забезпечення повинно мати текстовий інтерфейс і працювати в терміналі ОС Windows та/або Linux. Обмеження використання системних ресурсів має відповідати мінімальним вимогам: оперативна пам’ять 256 MB, диск 200 MB.

ChocoPy – це мова програмування, розроблена для використання в навчаль–

них цілях на курсах компіляторів для студентів. ChocoPy – це обмежена підмножина Python 3, яку можна легко скомпілювати до цільової архітектури, наприклад RISC–V. Мова повністю специфікована за допомогою формальної граматики, правил типізації та операційної семантики для всіх мовних конструкцій.

Програми ChocoPy можна виконувати безпосередньо в інтерпретаторі Python 3.6+. Програми ChocoPy також можна редагувати за допомогою стандартного підсвічування синтаксису Python. ChocoPy використовує анотації типу Python 3.6 для забезпечення статичної перевірки типів. Система типів підтримує номінальний підтип. Можна писати нетривіальні програми ChocoPy, використовуючи списки, класи та вкладені функції.

Об'єкт розробки – мова програмування ChocoPy.

Предмет розробки – реалізація інтерпретатору ChocoPy.

Даний дипломний проєкт – це академічний проєкт, мета якого є реалізація інтерпретатора мови програмування ChocoPy. Для цього необхідно розв’язати завдання лексичного, синтаксичного, семантичного аналізу та виконання коду.

Практичне значення одержаних результатів. Було отримано навички проєктування компіляторів, проведено порівняльний аналіз ефективності роботи розробленого інтерпретатору з Python та виявлено напрями подальшого вдосконалення.

Дипломний проєкт складається з наступних розділів: вступ, 6 основних розділів, висновки, перелік використаних джерел із 28 найменувань, 5 додатків. Графічна частина включає 4 кресленики формату А3. Загальний обсяг: 88 сторінок.

1 ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Мови програмування

Мова програмування – це система позначень для написання комп'ютерних програм [3]. Мови програмування можуть бути як формальними текстовими мовами, так і графічними. Зазвичай мови програмування описуються за допомогою двох складових: синтаксису (форми) і семантики (значення), які визначає формальна мова. Мови може бути визначена специфікацією (мову програмування C визначає стандарт ISO), в той час як інші мови (такі як Perl) мають переважаючу реалізацію, що вважається еталонною. Реалізація мови програмування – це система для виконання комп'ютерних програм. Існує два загальних підходи до реалізації мови програмування:

– інтерпретація: програма читається як вхідні дані інтерпретатором, який виконує дії, записані в програмі;

– компіляція: програма читається компілятором, який перекладає її на іншу мову, наприклад байт–код або машинний код. Перекладений код може або безпосередньо виконуватися ЦП, або служити вхідними даними для іншого інтерпретатора чи іншого компілятора [4].

Перші комп'ютери розуміли тільки двійковий код, мали обмежену пам'ять, відповідно і перші мови програмування були примітивними. З розвитком технологій були розроблені високорівневі мови програмування, більш продуктивні для програмістів.

Високорівневі мови – це формальні мови, які строго визначені своїм синтаксисом і семантикою, що складають їх архітектуру. Елементи цих формальних мов є:

– абетка (будь–який кінцевий набір символів);

– рядок (кінцева послідовність символів);

– мова (будь–який набір рядків абетки).

Речення в мові можуть бути визначені набором правил, які називаються граматикою. Нотація Бекуса – Наура (з англ. Backus–Naur form, BNF) описує син–

таксис «речень» мови. Ці ідеї випливають із концепцій контекстно–вільної граматики лінгвіста Ноама Чомскі [5]. У теорії формальних мов, контекстно–вільна граматика (з англ. context–free grammar, CFG) – це формальна граматика, правила виведення якої можуть бути застосовані до нетермінального символу незалежно від його контексту.

BNF та його розширення стали стандартними інструментами для опису синтаксису мов програмування, і в багатьох випадках частини компіляторів генеруються автоматично з опису BNF. На рисунку 1.1 можна побачити приклад EBNF (з англ. extended Backus–Naur form) – розширеної форми способу запису правил CFG, яка складається з термінальних символів і правил виводу, які є обмеженням, що регулюють, як термінальні символи можуть бути об'єднані в коректну послідовність. Приклади термінальних символів включають абетко–цифрові символи, розділові знаки та пробіли.

```
digit excluding zero = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
digit                = "0" | digit excluding zero ;
```

Рисунок 1.1 – Приклад нотації EBNF

У наведеному прикладі правило виводу визначає нетермінальну цифру, яка знаходиться зліва від присвоєння. Вертикальна риска представляє альтернативу, а термінальні символи взяті в лапки, за якими слідує крапка з комою як завершальний символ. Таким чином, цифра – це 0 або цифра, за винятком нуля, яка може бути 1 або 2 або 3 і так далі до 9.

Технологія компіляції розвинулась із необхідності строго визначеного перетворення вихідної високорівневої програми в цільову низькорівневу програму для цифрового комп’ютера. Компілятор можна розглядати як фронт–енд для аналізу вихідного коду та бек–енд для генерації цільового коду. Оптимізація між фронт–ендом і бек–ендом може створити більш ефективний цільовий код.

1.2 Будова компілятора

Компілятор – комп’ютерна програма, яка перекладає комп’ютерний код, написаний на вихідній мові програмування (в основному високорівневій), на цільову мову (в основному низькорівневу) для створення виконуваної програми [6].

Незалежно від кількості фаз в архітектурі компілятора, їх можна віднести до одного з трьох етапів, які включають фронт–енд, середній рівень та бек–енд (рисунок 1.2).

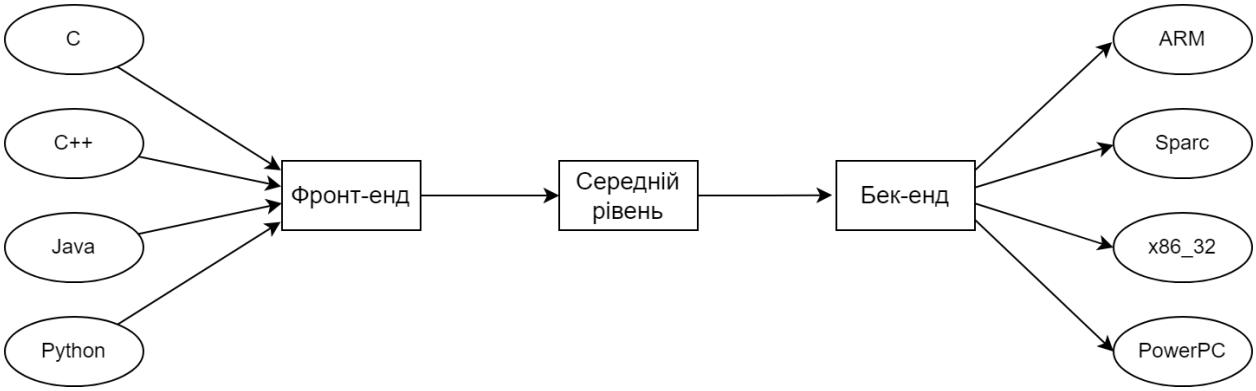

Рисунок 1.2 – Структура компілятора

1.2.1 Фронт–енд

Фронт–енд (з англ. front end) сканує вхідні дані та перевіряє синтаксис і семантику відповідно до певної вихідної мови. Для мов зі статичною типізацією він виконує перевірку типу. Якщо введена програма є синтаксично некоректною або має помилку типу, генерується повідомлення про помилку та/або попередження, зазвичай ідентифікуючи місце у вихідному коді, де було виявлено проблему; у деяких випадках фактична помилка може бути (набагато) раніше в програмі. Фронт–енд складається з основних 4 фаз.

Препроцесор, що підтримує підстановку макросів та умовну компіляцію.

Лексичний аналіз, що розбиває текст вихідного коду на послідовність маленьких фрагментів, які називаються токенами. Цей етап можна розділити на дві

частини: сканування, яке сегментує вхідний текст на синтаксичні одиниці, які називаються лексемами, і призначає їм категорію; і обчислення (з англ. evaluating), яке перетворює лексеми в оброблені значення (наприклад, числа і рядки). Токен – це пара, що складається з назви та значення (необов'язково). Загальні категорії токенів можуть включати ідентифікатори, ключові слова, розділові символи, оператори та літерали (константи). Деякі символи у вхідному файлі взагалі не мають значення (наприклад, пробіли та коментарі) і сканер їх просто відкидає. Синтаксис лексеми зазвичай є звичайною мовою, для її розпізнавання можна використовувати скінченний автомат, створений з регулярного виразу [7].

Синтаксичний аналіз включає аналіз послідовності токенів для визначення синтаксичної структури програми. На цьому етапі зазвичай створюється абстрактне синтаксичне дерево (з англ. abstract syntax tree, AST), яке замінює лінійну послідовність токенів деревом, побудованим відповідно до правил формальної граматики, яка визначає синтаксис мови. Робота аналізатора також включає повідомлення про синтаксичні помилки [8].

На рисунку 1.3 видно, як внутрішні вузли побудованого AST співставляються з відповідними операторами вихідного коду мови програмування, а листя з відповідними операндами.

Семантичний аналіз додає семантичну інформацію до AST та створює таблицю символів (структуру даних, що пов'язує кожен ідентифікатор з інформацією, такою як тип даних і область видимості). Ця фаза виконує семантичні перевірки, такі як перевірка типу, зв'язування об'єктів (зв'язування посилань на змінні та функції з їхніми визначеннями), або обов'язкова ініціалізація всіх локальних змінних перед використанням, відхилення некоректних програм або видача попереджень.

Фронт–енд перетворює вхідну програму в проміжне представлення (з англ. intermediate representation, IR) для подальшої обробки середнім рівнем.

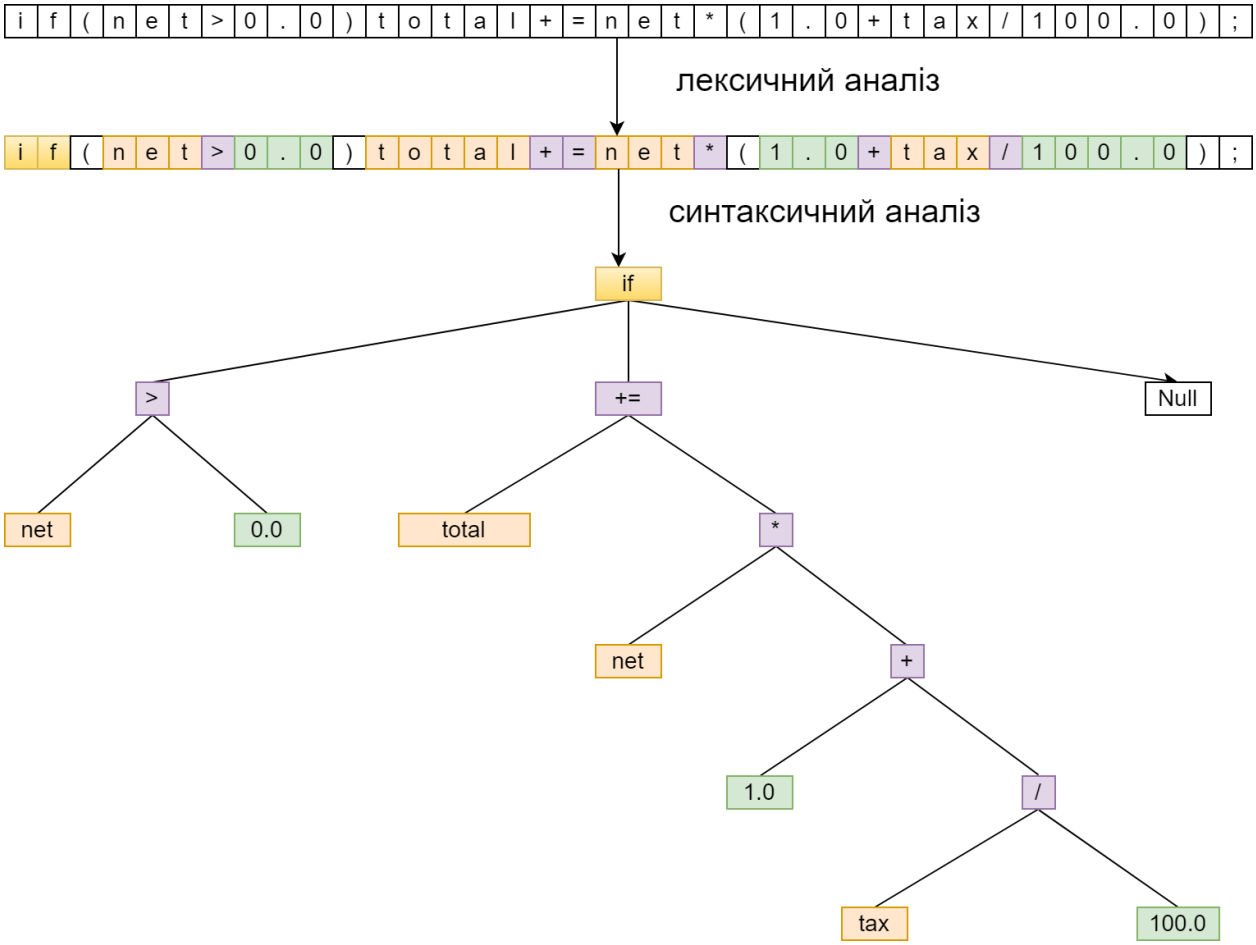

Рисунок 1.3 – Фази лексичного і синтаксичного аналізу

1.2.2 Середній рівень

Середній рівень (з англ. middle end) виконує оптимізацію проміжного представлення, щоб покращити продуктивність і якість створеного машинного коду [9]. Середній рівень містить оптимізації, які не залежать від цільової архітектури центрального процесора (ЦП). Середній рівень складається з 2 основ–них фаз.

Аналіз: це збір програмної інформації з проміжного представлення, отриманого з вхідних даних; Аналіз потоку даних використовується для побудови ланцюжків «використання–визначення» (з англ. use–definition chain) разом із аналізом залежностей, псевдонімів, вказівників тощо. Точний аналіз є основою будь–якої оптимізації компілятора. Граф потоку керування кожної скомпільованої функції та граф викликів програми зазвичай також будуються на етапі аналізу.

Оптимізація: проміжне представлення мови перетворюється у функціональ–но еквівалентні, але швидші (або менші) форми. Популярними методами оптимізації є вбудовування функцій, усунення мертвого коду, згортка констант, оптимізація циклів та навіть автоматичне розпаралелення.

1.2.3 Бек–енд

Бек–енд (з англ. back end) відповідає за оптимізацію, специфічну для архітектури процесора та генерацію коду. Бек–енд складається з 2 основних фаз.

Оптимізація, залежна від деталей цільової архітектури ЦП [10]. Яскравим прикладом є віконна (з англ. peephole) оптимізація, яка переписує короткі послідовності інструкцій асемблера в більш ефективні.

Генерація коду. Є два варіанти: машинний код і байт–код. Проміжне представлення перекладається на, як правило, рідну машинну мову системи. Це включає рішення щодо ресурсів і зберігання, наприклад, які змінні зберігати в регістрах та пам’яті, а також вибір і планування відповідних машинних інструкцій разом із пов’язаними з ними способами адресації пам’яті. Також генеруються дані для полегшення налагодження.

Згенерувавши реальний машинний код, отримується виконуваний файл, який операційна система (ОС) може завантажити безпосередньо на чип. Рідний код працює найшвидше, але його генерація дуже кропітка. Компілятор прив’язаний до певної архітектури: цільовий машинний код для x86 не працюватиме на пристрої ARM.

1.3 Система виконання

1.3.1 Байт–код

Для уникнення несумісності компілятор, замість інструкцій для реального чипа, створює код для гіпотетичної, ідеалізованої машини. Він називається байт–код (також p–code, що означає портативний), оскільки кожна інструкція часто має

довжину в один байт, за якою слідують додаткові параметри. Ці синтетичні інструкції створені для того, щоб трохи точніше відповідати семантиці мови, а не бути тісно прив'язаними до особливостей архітектури будь–якого ЦП.

1.3.2 Віртуальна машина

Якщо компілятор видає байт–код, то його необхідно перекласти, адже немає мікросхеми, що розуміє цей байт–код. Знову ж таки, є два варіанти. Можна написати маленький міні–компілятор для кожної цільової архітектури, який перетворює байт–код у рідний код для цієї машини. В такому разі байт–код використовується як проміжне представлення.

Або можна написати віртуальну машину (з англ. VM), програму, яка емулює гіпотетичний чип, що підтримує віртуальну архітектуру під час виконання [11]. Виконання байт–коду у віртуальній машині повільніше, ніж його завчасне перетворення у рідний код, тому що кожна інструкція повинна бути змодельована кожного разу, коли вона виконується. Натомість отримується простота та портативність. Якщо реалізувати віртуальну машину, скажімо, на C, то можна буде виконувати мову на будь–якій платформі, яка має компілятор C.

1.3.3 Час виконання

Час виконання – це завершальна фаза життєвого циклу комп'ютерної програми, на якій код виконується на ЦП комп'ютера у вигляді машинного коду. Помилки часу виконання виявляються після або під час виконання програми, тоді як помилки часу компіляції виявляються компілятором ще до того, як програма буде виконана. Перевірка типів, розподіл регістрів, генерація та оптимізація коду зазвичай виконуються під час компіляції, але можуть виконуватися і під час виконання, залежно від конкретної мови та компілятора.

Якщо програма скомпільована до машинного коду, тоді ОС достатньо завантажити бінарний файл, і почнеться виконання. Якщо програма скомпільована

в байт–код, потрібно запустити віртуальну машину та завантажити в неї програму.

1.4 Різниця між компілятором та інтерпретатором

Компіляція – це техніка реалізації, яка передбачає переклад вихідної мови в іншу форму, зазвичай нижчого рівня. Бо, наприклад, транскомпілятор (з англ. transpiler) перетворює вихідний код однієї мови програмування на вихідний код іншої мови. Реалізація мови у вигляді компілятора означає, що вона перекладає вихідний код в іншу форму, але не виконує його. Користувач повинен взяти результат і виконати його самостійно. Прикладами компіляторів є: GCC, Clang (для C і C++); CDC 6000, Turbo51 (для Pascal); Flang, Intel oneAPI (для Fortran) і т. ін.

Якщо реалізація є інтерпретатором – це значить, що вона приймає вихідний код і негайно виконує його. Інтерпретатор може використовувати одну з таких стратегій для виконання програми:

– розібрати вихідний код і виконати його поведінку безпосередньо (ранні версії Lisp, діалекти BASIC для міні комп’ютерів);

– перекласти вихідний код у якесь ефективне проміжне представлення або об’єктний код і негайно виконати його (Perl, Matlab, Ruby);

– явно виконати збережений попередньо скомпільований байт–код, створе–ний компілятором і завантажений у віртуальну машину інтерпретатора (CPython, UCSD Pascal).

Багато інтерпретаторів скриптових мов на якійсь з фаз компілюють вихідний код у проміжний байт–код для внутрішнього використання (Java). Про такі мови не можна сказати, що вони є строго компіляторами, або інтерпретаторами, оскільки виконують обидві функції.

Поєднанням двох підходів, AOT–компіляції (з англ. ahead–of–time) перед ви–конанням програми та інтерпретації, є JIT–компіляція (з англ. just–in–time) – виконання комп'ютерного коду, що передбачає компіляцію під час виконання програми. Як тільки код починає виконуватись, JIT–компілятор здатен його оптимізувати по–різному для кожного користувача, залежно від того, як кожен з

них використовує додаток.

Висновки до розділу

Компілятори – одна з найважливіших дисциплін комп'ютерних наук. Вона допомагає глибше зрозуміти, як влаштована мова програмування, а знання принципів їх побудови дозволяє написати власну DSL. Мови програмування пройшли довгий шлях у еволюції від примітивного двійкового коду до високорівневих мов. Складна граматика досить просто описується BNF нотацією.

Мова програмування може бути реалізована у вигляді інтерпретатору або компілятору. Інтерпретатор негайно виконує вхідний код, а компілятор перекладає програму у низькорівневу мову. Обидва підходи мають як переваги, так і недоліки.

Роботу компілятора можна уявити у вигляді конвеєру, що складається з трьох етапів: фронт–енд, середній рівень та бек–енд. Завданням фронт–енду є лексичний, синтаксичний і семантичний аналіз вихідного коду. Середній рівень, отримавши від фронт–енду проміжне представлення програми, аналізує та оптимізує його. Бек–енд оптимізує код відповідно до цільової архітектури ЦП та генерує машинний код для безпосереднього виконання ОС або байт–код для виконання віртуальною машиною.

Машинний код зазвичай швидкий і використовує менше пам’яті та місця на диску. Проте байт–код більш портативний, але потребує встановленого середовища виконання.

2 ВИВЧЕННЯ МОВИ CHOCOPY

2.1 Походження ChocoPy

ChocoPy – це мова програмування, розроблена для використання в навчальних цілях на курсах компіляторів для студентів. Спочатку використовувалась для викладання курсу «CS 164. Мови програмування та компілятори» в Каліфорнійському університеті в Берклі, але з тих пір її використовували кілька інших установ.

Мова ChocoPy є статично типізованою підмножиною Python 3.6. Майже кожна коректна програма ChocoPy також є коректною програмою Python 3.6. Виконання програми ChocoPy, яке не призводить до помилки, зазвичай має таку ж видиму семантику, як і виконання цієї програми в Python 3.6.

Документація містить формальну специфікацію синтаксису мови (правила токенізації + граматика), вичерпні правила типізації для системи типів, а також операційну семантику для всіх мовних конструкцій. Програми ChocoPy можна виконувати безпосередньо в інтерпретаторі Python 3.6+, а також редагувати за допомогою стандартного підсвічування синтаксису Python. ChocoPy використовує анотації типу Python 3.6 для забезпечення статичної перевірки типів [1].

Програма ChocoPy обмежується одним вихідним файлом. На верхньому рівні програма ChocoPy складається з визначень змінних, функцій, класів та операторів. Клас складається з визначень атрибутів і методів. Клас створює тип, визначений користувачем. Визначення функцій можуть бути вкладені в інші методи та функції. Усі назви класів і функції, визначених на верхньому рівні, мають глобальну видимість. Класи, функції та методи не можна перевизначити. Оператори програми можуть містити вирази, присвоєння та оператори потоку керування, такі як умовні оператори та цикли. Результатом обчислення виразу є значення, яке може бути цілим числом, логічним значенням, рядком, об'єктом визначеного користувачем класу, списком або спеціальним значенням None. ChocoPy не підтримує асоціативні масиви, функції першого класу та рефлексивну інтроспекцію. Усі ви–рази статично типізовані. Змінні (глобальні та локальні) і атрибути класу є статично

типізованими та мають лише один тип протягом усього часу існування. Змінні та атрибути явно типізовані за допомогою анотацій. У визначеннях функцій і методів анотації типів використовуються для явного визначення типу повернення значення та типів формальних параметрів.

Рисунок 2.1 містить зразок програми ChocoPy, що ілюструє функції верхнього рівня, оператори, глобальні і локальні змінні та анотації типів. Анотації типу є коректним синтаксисом у Python 3.6, хоча інтерпретатор Python просто ігнорує ці анотації та залишає їх як підказки для інших інструментів. В той час, як ChocoPy забезпечує перевірку статичної типізації під час компіляції. На рисунку 2.1 функція *is_zero* визначена на верхньому рівні. Її формальні параметри *items* та *idx* є явно типізованими як список цілих чисел і ціле число відповідно. Тип повернення значення функції – логічний тип. Функція визначає локальну змінну *val* типу *int*. На верхньому рівні програма визначає глобальну змінну *mylist*, типом якої є список цілих чисел. Функція *is_zero* викликається в операторі верхнього рівня та її результат виводиться за допомогою попередньо визначеної функції *print*.

```
def is_zero(items: [int], idx: int) -> bool:
    val:int = 0             # Type is explicitly declared
    val = items[idx]
    return val == 0

mylist: [int] = None
mylist = [1, 0, 1]
print(is_zero(mylist, 1))   # Prints 'True'
```

Рисунок 2.1 – Приклад програми ChocoPy з функцією

Рисунок 2.2 містить програму ChocoPy, що визначає два класи: *animal* та *cow*. Клас *cow* успадковується від *animal*, який, у свою чергу, успадковується від попередньо визначеного кореневого класу *object*. Логічний атрибут *makes_noize* визначений у класі *animal* і тому успадковується класом *cow*. Клас *cow* перевизначає метод *sound*. Конструктор класу *cow* викликається у передостанньому рядку.

```
class animal(object):
    makes_noise:bool = False

    def make_noise(self: "animal") -> object:
        if (self.makes_noise):
            print(self.sound())

    def sound(self: "animal") -> str:
        return "???"
class cow(animal):
    def __init__(self: "cow"):
        self.makes_noise = True

    def sound(self: "cow") -> str:
        return "moo"
c:animal = None
c = cow()
c.make_noise()      # Prints "moo"
```

Рисунок 2.2 – Приклад програми ChocoPy з класом

2.2 Огляд ChocoPy

Позначення:

– {expr} для позначення виразу в програмі;

– {id} для позначення ідентифікатора, наприклад імені змінної або функції;

– {stmts} для позначення списку операторів програми, розділених символами нового рядка;

– {declarations} для позначення списку оголошень функцій, змінних, атрибутів та/або класів, де це можливо;

– {type} для позначення анотації статичного типу;

– {literal} для позначення константного літералу, такого як ціле число, рядок або ключові слова *True*, *False* або *None*.

2.2.1 Верхній рівень

Програма ChocoPy на верхньому рівні складається з нуля або більше визначень, за якими слідує нуль або більше операторів. Визначення верхнього рівня містять визначення глобальних змінних, функцій та класів. Ці визначення створюють асоціативний масив в глобальній області видимості. Глобальні змінні визначаються за допомогою синтаксису *{id}:{type} = {literal}*, де ідентифікатор визначає ім'я змінної, анотація типу визначає статичний тип змінної, а константний літерал визначає початкове значення змінної. Назви глобальних змінних, функцій та класів мають бути відмінними. Оператори верхнього рівня виконуються в глобальній області видимості; тобто, вирази в операторах верхнього рівня можуть посилатися на сутності, визначені в глобальній області видимості за допомогою ідентифікаторів. Виконання програми ChocoPy починається з першого оператора верхнього рівня і закінчується, коли останній оператор верхнього рівня повністю виконається [12].

2.2.2 Функції

У ChocoPy визначення функції може бути на верхньому рівні програми або бути вкладеним в іншу функцію або метод. Функції не можна перевизначити в тій самій області видимості. Однак визначення функції в поточній області видимості може затіняти функцію, визначену в навколишній області видимості. Визначення функції зображено на рисунку 2.3.

```
def {id}({id}: {type}, ..., {id}: {type}) {return type}:
    {declarations}
    {stmts}
```

Рисунок 2.3 – Визначення функції в ChocoPy

де *{return type}* або порожній, або має форму →*{type}*.

Перший рядок визначає назву функції, розділений комами список з нуля або більше формальних параметрів у дужках та тип повернення значення функції після символу →. Якщо тип повернення порожній, функція може повертати лише значення *None*. Кожен формальний параметр має ім’я та анотацію статичного типу. Тіло функції містить послідовність з нуля або більше оголошень, за якими слідує послідовність з одного або більше операторів. Визначення функції створює нову область видимості.

Оголошення в тілі функції включають визначення локальних змінних, оголошення глобальних і нелокальних змінних і визначення вкладених функцій. Визначення *{id}:{type}={literal}* оголошує локальну змінну з іменем *id*, явно пов’язує її зі статичним типом і вказує початкове значення за допомогою літералу. Оператор *global {id}* використовується для прив’язки імені до глобальної змінної. Аналогічно, оператор *nonlocal {id}* використовується у вкладеній функції для прив’язки імені до змінної, визначеної в навколишній не глобальній області видимості, а саме найближчій навколишній області, що оголошує цю змінну. Оголошення *global* не можна застосовувати на верхньому рівні. Так само, є помилкою оголошувати *nonlocal* змінну за межами вкладеної функції або для посилання на глобальну змінну.

Якщо змінна явно не оголошена у функції, але пов’язана з деякою сутністю – змінною, функцією або класом – у будь–якій навколишній області видимості, тоді її прив’язка неявно успадковується з навколишньої області видимості як змінна лише для читання – такій змінній не можна присвоїти значення в жодному з операторів функції.

2.2.3 Класи

У ChocoPy визначення класу знаходиться на верхньому рівні програми. Класи не можна перевизначити. Також імена класів ніколи не можна перевизначати; тобто програма не може визначати жодну змінну чи функцію з іменем класу. Визначення класу зображено на рисунку 2.4.

```
class {id}({id}):
    {declarations}
```

Рисунок 2.4 – Визначення класу в ChocoPy

У першому рядку вказується ім'я класу, а потім ім'я його суперкласу в дужках. Ім'я класу не повинно бути прив'язане до жодної іншої сутності програми: класу, функції чи змінної. Суперклас має посилатися на клас, який був раніше визначений у програмі, або бути попередньо визначеним класом *object*. Суперклас не може бути примітивним типом *int*, *str* або *bool*.

Тіло класу складається з визначень атрибутів і методів. У ChocoPy атрибути та методи асоціюються з екземплярами об'єктів класу, а не з самими класами; тобто ChocoPy не підтримує поняття статичних членів класу, як в інших мовах. Подібно до визначення змінної, визначення атрибута має вигляд *{id}:{type}={literal}*. Визначення метода має той самий синтаксис, що й визначення функції, з двома важливими обмеженнями:

– визначення методу повинно мати принаймні один формальний параметр;

– перший формальний параметр повинен мати тип класу, в якому визначається цей метод.

Клас визначає атрибути та методи. Клас успадковує атрибути та методи свого суперкласу. Атрибути, що визначені в поточному класі чи успадковані від суперкласу, не можуть бути перевизначені. Методи не можна перевизначити в тому самому класі. Успадковані методи можна перевизначити за умови, що тип повернення значення методу та типи всіх формальних параметрів, крім першого параметра, абсолютно однакові. Перед будь–яким посиланням на атрибут або метод повинен бути вираз та оператор *«крапка»*.

Якщо визначено, що клас *C* має суперклас *P*, тоді клас *C* є підкласом *P*. Якщо *C* є підкласом *P*, тоді *P* має бути або визначеним користувачем класом, який був визначений перед *C* у програмі, або бути попередньо визначеним класом *object*.

Клас *object* не має суперкласу. Оскільки кожен клас ChocoPy (крім *object*) успадковує атрибути та методи від одного суперкласу, ця схема називається прос–

тим успадкуванням (з англ. single inheritance). Відношення підклас/суперклас у класах визначає граф. Оскільки клас може успадковувати лише від іншого визначеного раніше класу, цей граф є деревом з кореневим класом *object*.

Щоб створити об’єкт типу *C*, використовується вираз *C()*. Після виконання з'являється новий об'єкт, атрибути якого ініціалізуються їх визначеними значеннями. Потім викликається метод *__init__* класу *C*. Якщо *__init__* метод не визначено в класі *C*, то викликається успадкований метод *__init__*. Кореневий клас *object* має стандартний метод *__init__* з порожнім тілом.

2.2.4 Ієрархія типів

У ChocoPy кожне ім’я класу також є типом. Основне правило типів в ChocoPy полягає в тому, що якщо метод або змінна очікує значення типу *P*, тоді замість нього можна використовувати будь–яке значення типу *C*, за умови, що *P* є предком *C* в ієрархії класів. Іншими словами, якщо *C* успадковує від *P* прямо чи опосередковано, тоді можна використовувати *C* скрізь, де вистачило б *P*. Коли об’єкт типу *C* може використовуватися замість об’єкта типу *P*, кажуть, що *C* відповідає *P* або що $C \leq P$ (треба розуміти як, *C* знаходиться нижче в дереві успадкування). Відповідність класових типів визначається в термінах графа успадкування. Нехай *A*, *C* і *P* – типи. Тоді відповідність (тобто ≤) визначається таким чином:

– $A \leq A$ для всіх типів *A*;

– якщо *C* є підкласом *P*, тоді $C \leq P$;

– якщо $A \leq C$ і $C \leq P$, тоді $A \leq P$.

Коренем ієрархії класів є попередньо визначений клас *object*. Попередньо визначені типи *int*, *bool* і *str* є підкласами класу *object*. Крім того, для кожного типу *T* у програмі ChocoPy існує тип списку *[T]*, який представляє список, елементи якого мають тип *T*. Наприклад, тип *[int]* представляє список цілих чисел. Типи списків є рекурсивними: тип $[[int]]$ представляє список, кожен з елементів якого є списком цілих чисел. Типи списків не пов’язані один з одним співвідношенням ≤.

Кожен тип списку відповідає класу *object*; тобто *[T]* ≤ *object* для будь–якого типу *T*.

Окрім типів, які можна вказувати в програмах ChocoPy, існує два спеціальні типи: тип *None*, який позначається <*None*>, і тип *[]*, який позначається як < *Empty*>. Оскільки немає можливості явно записувати ці назви типів у ChocoPy, жодна змінна, параметр чи значення, що повертає функція, ніколи не мають жодний з цих типів.

В ієрархії типів <*None*> ≤ *object*, <*Empty*> ≤ *object* і, як зазвичай, <*None*> ≤ <*None*> і <*Empty*> ≤ <*Empty*>, але в інших випадках ці типи не пов'язані з жодним іншим типом.

Щоб описати присвоєння та виклик функції, потрібно дещо інше відношення, яке називається сумісність за присвоєнням (з англ. assignment compatibility) і позначається символом $\leq_a$. Тобто, можна присвоїти або передавати величину типу T_1 до сутності типу T_2, якщо $T_1 \leq_a T_2$ і виконується принаймні одна з наступних умов:

– $T_1 \leq T_2$ (тобто звичайний підтип);

– T_1 є <*None*> та T_2 не є *int*, *bool*, або *str*;

– T_2 є тип списку *[T]* та T_1 є <*Empty*>;

– T_2 є тип списку *[T]* та T_1 є *[*<*None*>*]*, де <*None*> $\leq_a T$.

В останньому випадку два різні типи списків сумісні за присвоєнням. Це зручно для написання таких речей, як *x: [A] = [None, None]*. Щоправда, це можна використати в досить обмеженій кількості випадків. Наприклад, *x: [[A]] = [[None]]* все ще є некоректним кодом, хоча виглядає цілком розумним.

У деяких ситуаціях потрібно використовувати концепцію об'єднання двох або більше типів. З'єднання двох типів *A* і *B* є найменшим типом *C* (з використанням порядку $\leq_a$), таким чином *A* і *B* є сумісними за присвоєнням з *C*.

Оператор об'єднання ⊔ може бути формально визначений таким чином:

$$C = A \sqcup B \tag{2.1}$$

тоді і тільки тоді, коли:

$$(A \leq_a C) \wedge (B \leq_a C) \wedge (\forall D:(A \leq_a D) \wedge (B \leq_a D) \Rightarrow (C \leq_a D)) \quad (2.2)$$

Тобто *C* є об'єднанням *A* і *B* тоді і тільки тоді, коли обидва *A* і *B* сумісні за присвоєнням з *C*, і якщо існує тип *D* такий, що *A* і *B* сумісні за присвоєнням з *D*, тоді *C* також є сумісний за присвоєнням з *D*. Об'єднання будь–яких двох типів завжди існує і є унікальним. Якщо $A \leq_a B$, то:

$$A \sqcup B = B \sqcup A = B; \quad (2.3)$$

В іншому випадку $A \sqcup B$ є просто найменшим спільним предком *A* і *B* в деревоподібній ієрархії типів визначеній відношенням $\leq$.

2.2.5 Значення

У ChocoPy є наступні типи значень.

2.2.5.1 Цілі числа

Цілі числа мають знак і представлені за допомогою 32 бітів. Діапазон цілих чисел – від -2^{31} до $(2^{31}-1)$. Хоча цілі числа успадковують від класу *object*, вони незмінні. Арифметичні операції, які викликають переповнення, призводять до невизначеної поведінки під час виконання програми.

2.2.5.2 Логічні значення

Існує рівно два логічних значення: *True* та *False*.

2.2.5.3 Рядки

Рядки – це незмінні послідовності символів. Рядкові літерали розділяються

подвійними лапками, наприклад *"Hello World"*. Рядки підтримують такі три операції: отримання довжини за допомогою функції *len*, індексування за допомогою синтаксису *s[i]*, та конкатенація – за допомогою синтаксису *s1 + s2*. Як і в Python, ChocoPy не має символьного типу. Індексація рядку повертає новий рядок довжиною 1. Конкатенація повертає новий рядок з довжиною, що дорівнює сумі довжин її операндів.

2.2.5.4 Списки

Списки – це змінювані послідовності фіксованої довжини. Таким чином, списки в ChocoPy поводяться більше як масиви в C. Список створюється за допомогою квадратних дужок, наприклад *[1, 2, 3]* (не є посиланнями на джерела).

Як і рядки, списки типу *[T]* підтримують три операції: *len*, індексування за допомогою синтаксису *lst[i]* та конкатенація за допомогою синтаксису *lst1 + lst2*. Індексування списку типу *[T]* повертає значення типу *T*. Конкатенація двох списків типу $[T_1]$ і $[T_2]$ відповідно повертає новий список типу $[T_3]$, де тип елемента нового списку

$$T_3 = T_1 \sqcup T_2. \tag{2.4}$$

Довжина об’єднаного списку дорівнює сумі довжин обох операндів. Крім того, списки типу *[T]* є змінними та підтримують четверту операцію: присвоєння елемента за допомогою синтаксису *lst[i] = {expr}*, де вираз у правій частині має відповідати типу *T*.

Правила ChocoPy дозволяють створювати досить дивні (але нешкідливі) списки з такими типами, як *[<None>]* та *[<Empty>]*. Єдині змінні та параметри, яким вони можуть бути присвоєні або передані, які мають оголошений тип *object*, оскільки неможливо записати конкретні імена типів *[<None>]* і *[<Empty>]* у ChocoPy.

2.2.5.5 Об'єкти класів, визначених користувачем

Об’єктами маніпулюють за допомогою посилань. Тобто *x = cow()* означає, що змінна *x* посилається на об’єкт типу *cow*. Подальше присвоєння *y = x* означає, що *x* і *y* посилаються на той самий об’єкт типу *cow* у пам’яті. Оператор *is* можна використовувати, щоб визначити, чи два вирази посилаються на один і той самий об’єкт у пам’яті. Об'єкти знищуються, коли вони недоступні з будь–якої локальної, глобальної чи тимчасової змінної.

2.2.5.6 None

None – це спеціальне значення, яке можна присвоїти змінній або атрибуту типу *object*, будь–якому класу, визначеному користувачем або будь–якому типу списку. Оператор *is* можна використовувати, щоб визначити, чи вираз має значення *None*. Для перевірки типу воно має тип <*None*>.

2.2.5.7 Пустий список

Вираз *[]* створює список, який можна присвоїти змінній типу *object* або будь–якому типу списку. З метою перевірки типу, він має тип <*Empty*>.

2.2.6 Вирази

ChocoPy підтримує такі категорії виразів: літерали, ідентифікатори, арифметичні та логічні вирази, вирази відношення, конкатенації, доступу та виклику.

2.2.6.1 Літерали та ідентифікатори

Елементарним виразом є константний літерал або змінна. Змінні обчислю–

ються до значення, що міститься в змінній. Якщо ідентифікатор прив'язаний до глобальної функції чи класу, то він не є коректним виразом сам по собі – він може використовуватися лише в певних виразах, таких як виклики. Це тому що ChocoPy не підтримує функції та класи першого класу (з англ. first–class), тобто не підтримує передачу функцій та класів як аргументів інших функцій.

2.2.6.2 Вирази зі списком

Списки можуть бути створені за допомогою послідовності виразів, розділених комами та розмежованих квадратними дужками: *[{expr}, ...]*. Тип виразу списку, що містить один або більше елементів, – *[T]*, де *T* – найменший спільний предок типів елементів списку в ієрархії типів програми. Іншими словами, *T* є найменший тип, такий щоб тип виразу кожного елемента відповідав *T*.

Використовуючи оператор об'єднання ⊔ можна сказати, що вираз форми *[{expr_1}, {expr_2}, ..., {expr_n}]*, де кожен вираз *{expr_i}* має тип *T_i*, має результатом список типу *[T]*, де:

$$T = T\_1 \sqcup T\_2 \sqcup \ldots \sqcup T\_n. \quad (2.5)$$

Вираз порожнього списку *[]* має спеціальний тип <*Empty*>, який дозволяє присвоїти його (передавати) до змінної (параметра) будь–якого типу списку. Наприклад, якщо змінна x має тип *[int]*, то присвоювання *x = []* дозволяється; *x* буде містити порожній список цілих чисел після цього присвоєння.

2.2.6.3 Арифметичні вирази

ChocoPy підтримує наступні арифметичні вирази для двох операндів типу *int*: *{expr} + {expr}*, *{expr} – {expr}*, *{expr} * {expr}*, *{expr} // {expr}*, і *{expr} % {expr}*. Ці оператори виконують додавання, віднімання, множення, частку та залишок від ділення відповідно. ChocoPy не підтримує ділення *{expr} / {expr}*, яке у Python

повертає дробове значення. Унарний вираз *–{expr}* повертає від'ємне значення цілочисельного операнда. Арифметичні дії повертають ціле значення типу *int*.

2.2.6.4 Логічні вирази

ChocoPy підтримує наступні логічні операції над операндами типу *bool*: *not {expr}, {expr} and {expr}*, і *{expr} or {expr}*, які означають логічне заперечення, кон'юнкцію та диз'юнкцію своїх операндів, відповідно. Логічні вирази повертають логічне значення типу *bool*. Двійкові логічні вирази підтримують обчислення за короткою схемою. Якщо лівий операнд виразу *and* має значення *False*, тоді результат *False* повертається без обчислення правого операнда. Подібним чином, якщо лівий операнд виразу *or* має значення *True*, тоді результат *True* повертається без обчислення правого операнда. Ця семантика важлива, коли вирази в правих операндах містять побічні ефекти.

2.2.6.5 Вирази відношення

ChocoPy підтримує такі реляційні вирази для операндів типу *int*: *{expr} < {expr}, {expr} <= {expr}, {expr} > {expr}, {expr} >= {expr}*. Крім того, операнди виразів *{expr} == {expr}* і *{expr} != {expr}* можуть бути типу *int, bool* або *str*, але обидва операнди мають бути одного типу. На відміну від цього, операнди у виразах *{expr} is {expr}* можуть бути літералом *None* або виразом будь–якого статичного типу, крім *int, bool, str*. Оператори == та *!=* повертають *True* тоді і тільки тоді, коли їхні операнди є відповідно рівними чи нерівними значеннями цілих чисел, логічних значень або рядків. Оператор *is* повертає значення *True* тоді і тільки тоді, коли обидва операнди є один і той самий об'єкт або обидва операнди є *None*.

2.2.6.6 Умовні (тернарні) вирази

Вираз *{expr1} if {expr0} else {expr2}* спочатку обчислює *{expr0}*, який повинен мати тип *bool*. Якщо результат *True*, тоді обчислюється *{expr1}*, і його значення буде результатом умовного виразу. В іншому випадку обчислюється *{expr2}*, і його значення буде результатом умовного виразу.

2.2.6.7 Вирази конкатенації

Вираз *{expr} + {expr}* можна використовувати для об’єднання двох рядків або двох списків; результатом є новий рядок або список відповідно.

2.2.6.8 Вирази доступу

Доступ до атрибута об’єкта можна отримати за допомогою оператора «крапка»: *{expr}.{id}*. Наприклад, *x.y.z* повертає значення, що зберігається в атрибуті *z* об’єкта, отриманого шляхом обчислення виразу *x.y*. Елемент рядка або списку можна отримати за допомогою оператора індексу: *{expr}[{expr}]*. Наприклад, *"Hello"[2+2]* повертає рядок *"o"*. Доступ до рядка або списку *x* з індексом *i* таким, що *i < 0* або *i >= len(x)*, перериває програму з відповідним повідомленням про помилку.

2.2.6.9 Вирази виклику

Вираз виклику має форму *{id}({expr},...)*, де *{expr},...* – розділений комами список з нуля або більше виразів, що є аргументами виклику. Якщо ідентифікатор прив’язаний до функції, оголошеній в глобальній області видимості, вираз обчислюється як результат виклику функції. Якщо ідентифікатор прив'язаний до класу, вираз призводить до побудови нового об’єкта цього класу, чий метод *__init__* викликається з наданими аргументами.

Вираз у формі *{expr}.{id}({expr},...)* викликає метод *{id}* об’єкта, що повертається шляхом обчислення виразу ліворуч від оператора *«крапка»*. Перший аргумент неявний і є об'єктом, метод якого викликається; решта аргументів явно наведено в дужках. Методи викликаються за допомогою динамічної диспетчеризації: якщо динамічний тип об’єкта, тобто тип на момент виконання, є *T*, тоді викликається метод *{id}*, визначений у *T* або успадкований *T*.

2.2.7 Анотації типів

У ChocoPy анотації статичних типів використовуються для явного надання типів змінним, атрибутам, формальним параметрам та типам повернення значень функцій і методів. Анотація типу може посилатися на тип класу *T* або тип списку *[T]*, де *T* є анотацією типу, що відповідає типу елементів списку. Анотації типу класу можуть бути надані в одній із двох форм: як ідентифікатори або як рядкові літерали, що містять назву класу. У ChocoPy можна використовувати будь–яку з двох форм для анотацій. Однак у Python, першу форму не можна використовувати для посилання на тип класу, який ще не визначено, оскільки Python інтерпретується рядок за рядком. Оскільки ChocoPy має поводився так само, як Python, треба використовувати останню форму анотації в описаному вище сценарії. Зокрема, рядкові літерали завжди потрібні в анотації типів для першого формального параметра у визначеннях методу, оскільки тип цього параметра завжди той самий, що й охоплюючий або зовнішній (з англ. enclosing) клас, який ще не повністю визначений.

2.2.8 Оператори

2.2.8.1 Оператори–вирази

Найпростішим оператором є окремий вираз. Вираз обчислюється, а його результат відкидається. Ці типи операторів корисні, коли вони мають побічні ефекти, наприклад *print("Hello")*.

2.2.8.2 Складені оператори: умовні та цикли

ChocoPy підтримує синтаксис *if–elif–else* для умовного потоку керування з необов'язковими *elif* та *else*. Код на рисунку 2.5 є еквівалентним до коду на рисунку 2.6.

```
if {expr1}:
    {body1}
elif {expr2}:
    {body2}
elif {expr3}:
    {body3}
```

Рисунок 2.5 – Оператор *if–elif* в ChocoPy

```
if {expr1}:
    {body1}
else:
    if {expr2}:
        {body2}
    else:
        if {expr3}:
            {body3}
```

Рисунок 2.6 – Оператор *if–else* в ChocoPy

Вирази в умовах *if* та *elif* повинні мати тип *bool*. Тіло оператора відразу після умови *if* або *elif* обчислюється, лише якщо вираз має значення *True*. Якщо вираз має значення *False*, розглядаються наступні блоки *elif* або *else*. Тіло оператора, яке слідує за *else*, обчислюється, лише якщо усі попередні вирази умови мають значення *False*.

ChocoPy підтримує два типи циклів: прості цикли *while* і цикли *for* над списками та рядками. Структура циклів *while* зображена на рисунку 2.7.

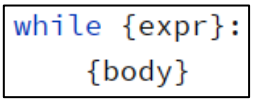

Рисунок 2.7 – Оператор *while* в ChocoPy

Вираз *{expr}* має бути типу *bool*. Тіло циклу багаторазово обчислюється, доки вираз має значення *True* між ітераціями. Цикли *for* можна використовувати для перебору елементів списку або символів рядка. Їх форма зображена на рисунку 2.8.

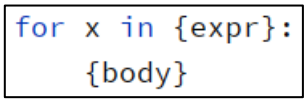

Рисунок 2.8 – Оператор *for* в ChocoPy

Цикли *for* є синтаксичним цукром; наведена вище структура еквівалентна коду, зображеному на рисунку 2.9.

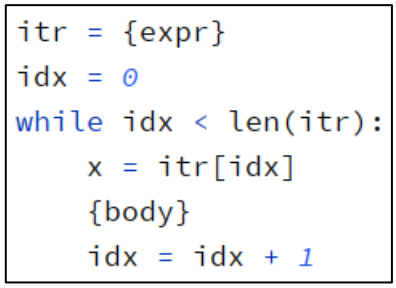

Рисунок 2.9 – Заміна оператора *for* на цикл *while* в ChocoPy

де *len* – попередньо визначена функція довжини, а *itr* та *idx* – тимчасові змінні, які не визначені у початковій області видимості. Цикл *for* не створює нових оголошень для змінної циклу (*x* в наведеному вище прикладі); змінна циклу має бути оголошена перед оператором *for*.

2.2.8.3 Оператор присвоєння

Присвоєння може мати одну з наступних трьох форм:

– *{id} = {expr}* присвоює значення змінній, пов’язаній з ідентифікатором *{id}*;

– *{expr}.{id} = {expr}* присвоює значення атрибуту об’єкта;

– *{expr}[{expr}] = {expr}* присвоює значення елементу списку.

При присвоєнні значення індексу *i* списку *x*, якщо *i < 0* або *i > = len(x)*, то програма переривається після друку відповідного повідомлення про помилку.

Одне присвоєння може призначити те саме значення кільком різним адресатам. Наприклад, код *x = y.f = z[0] = 1* присвоює ціле значення 1 трьом коміркам пам’яті: змінній *x*, атрибуту *f* об'єкта, на який посилається змінна *y*, та першому елементу списку *z* (в такому порядку). Тобто спочатку обчислюється остаточний вираз (права частина). Потім результат присвоюється лівій частині (тобто, зліва від символів =), зліва направо.

2.2.8.4 Оператор pass

Оператор *pass* є *NOP* (з англ. no operation). Стан програми не змінюється, і потік керування просто продовжується до наступного оператора.

2.2.8.5 Оператор return

Оператор *return* припиняє виконання функції та повертає значення (необов’язково) за допомогою синтаксису *return {expr}*. Якщо значення, що повертається, не вказано, повертається значення *None*. Оператор помилково використовувати на верхньому рівні програми поза тілом функції або методу. Під час виконання функції, якщо потік керування досягає кінця тіла функції, не зустрічаючи оператора *return*, тоді значення *None* повертається неявно. Приклад коду з операторами *return* зображено на рисунку 2.10.

У функції *bar* виконання може припинитися через те, що *x > 0* і виконується оператор *return* без значення, що повертається, або *x == 0* і явно повертається значення *None*, або *x < 0* і потік керування досягає кінця функції, неявно повертаючи *None*.

У функціях або методах, які оголошують тип повернення *int, str* або *bool*, усі шляхи виконання повинні містити оператор *return* з виразом, який не є літералом *None*. У визначеннях класу методи *__init__* не повинні мати тип повернення (вказуючи, що вони завжди повертають *None*).

```
def bar(x: int) -> object:
    if x > 0:
        return
    elif x == 0:
        return None
    else:
        pass
```

Рисунок 2.10 – Оператори *return* в ChocoPy

2.2.8.6 Попередньо визначені класи та функції

Функції *print, input* і *len* надаються оточенням виконання.

print приймає аргумент типу *object* і виводить його друковану форму до стандартного потоку виводу, повертає значення *None*. Дозволяються аргументи типів *str*, *int* або *bool*. Інші аргументи призводять до переривання програми з повідомленням про помилку.

input не приймає аргументів і повертає значення типу *str* шляхом читання рядка введення зі стандартного потоку введення, включно з останнім символом перенесення рядка. Функція повертає порожній рядок, коли стандартний потік введення вичерпано.

len приймає аргумент *x* типу *object*, повертаючи його довжину, якщо це рядок або список. Інші аргументи призводять до переривання програми з повідомленням про помилку.

Кожен з попередньо визначених класів *object, int, bool* і *str* визначає метод *__init__*. Виклик цих класів призводить до створення порожнього об'єкту, значень *0*, *False* та *""* (порожній рядок) відповідно.

2.3 Лексична структура

Лексичний аналіз читає вхідний файл і створює послідовності токенів. Токени зіставляються у вхідному рядку за допомогою лексичних правил, які виражаються за допомогою регулярних виразів. При неоднозначності, токен буде містити найдовший можливий рядок, що утворює коректний токен при читанні зліва направо. Існують такі категорії токенів: структура рядків, ідентифікатори, ключові слова, літерали, оператори та роздільники.

2.3.1 Структура рядків

У ChocoPy, як і в Python, пробіли можуть бути важливими як для завершення оператора, так і для відступу в операторі програми. Щоб це зробити, ChocoPy визначає три токена, які походять від пробілів: *NEWLINE*, *INDENT* і *DEDENT*.

2.3.1.1 Фізичні рядки

Фізичний рядок – це послідовність символів, що завершується послідовністю кінця рядка. У вихідних файлах і рядках, можна використовувати наступні послідовності завершення рядка: форму Unix з використанням ASCII LF (*n*), форму Windows використовуючи послідовність ASCII CR LF (*r**n*) або стару форму Macintosh із використанням символу ASCII CR (*r*). Будь–яка з цих форм може використовуватись однаково, незалежно від платформи. Кінець введення програми також слугує неявним обмежувачем для останнього фізичного рядка.

2.3.1.2 Логічні рядки

Логічний рядок – це фізичний рядок, який містить принаймні один токен, який не є пробілом чи коментарем. Закінчення логічного рядка представлено токеном *NEWLINE*. Оператори не можуть перетинати межі логічного рядка за

винятком випадків, коли *NEWLINE* дозволено синтаксисом (наприклад, між операторами в конструкціях потоку керування, таких як цикли *while*).

2.3.1.3 Коментарі

Коментар починається зі знаку решітки (#), який не являється частиною рядкового літералу, та завершується в кінці фізичного рядка. Коментарі ігноруються лексичним аналізатором; для них не генеруються токени.

2.3.1.4 Порожні рядки

Фізичний рядок, який містить лише пробіли, табуляції та, можливо, коментар, ігнорується (тобто токен *NEWLINE* не генерується).

2.3.1.5 Відступ

Початковий пробіл (або табуляція) на початку логічного рядка використовується для розрахунку рівня відступу рядка, який, використовується для визначення групування операторів. Табуляції замінені (зліва направо) на один – вісім пробілів так, щоб загальна кількість символів до та включно з заміною є кратна восьми (правило Unix систем). Загальна кількість пробілів перед першим непробільним символом визначає відступ рядка.

Для створення токенів *INDENT* і *DEDENT* використовуються рівні відступів послідовних рядків за допомогою стеку таким чином: перед тим, як буде прочитано перший рядок вхідної програми, у стек вводиться нуль, який там залишиться до кінця роботи програми. Числа, розміщені в стеку, постійно строго зростатимуть від першого до останнього. На початку логічного рядка ступінь відступу рядка буде порівнюватися з верхнім значенням стеку. Якщо вони рівні, нічого не змінюється.

Якщо відступ більше, то поміщається в стек, і створюється один токен *INDENT*. Якщо відступ менше, це має бути одне з чисел, що зустрічаються в стеку;

всі числа в стеку які більші, видаляються, і для кожного з них генерується токен *DEDENT*. В кінці вхідної програми генерується токен *DEDENT* для кожного числа, що залишилося в стеку та перевищує нуль.

2.3.1.6 Пробіли між токенами

Крім рядкових літералів або початку логічного рядка, табуляції та пробіли можуть використовуватися для розділення токенів як взаємозамінні. Пробіл потрібен між двома токенами лише у разі, якщо їх об'єднання можна інтерпретувати як інший токен (наприклад, *ab* – це один токен, *a b* – це два токена). Пробільні символи не є токенами; вони ігноруються.

2.3.2 Ідентифікатори

Ідентифікатор визначається як неперервна послідовність символів, що містить великі та маленькі літери від *A* до *Z*, підкреслення _ та, окрім першого символу, цифри від 0 до 9.

2.3.3 Ключові слова

Наступні рядки не розпізнаються як ідентифікатори, а натомість розпізнаються як окремі токени ключових слів: *False, None, True, and, as, assert, async, await, break, class, continue, def, del, elif, else, except, finally, for, from, global, if, import, in, is, lambda, nonlocal, not, or, pass, raise, return, try, while, with, yield*.

Не всі ключові слова мають особливе значення в ChocoPy. Наприклад, ChocoPy не підтримує *async* або *await*. Однак ChocoPy використовує той самий список ключових слів, що й Python, щоб уникнути випадків, коли ідентифікатор допустимий у ChocoPy, але не в Python. Отже, деякі ключові слова (такі як *async*) не з'являються будь–де в граматиці та призводять до синтаксичної помилки.

Ідентифікатор може містити ключове слово як підрядок; наприклад, *classic* є

коректним ідентифікатором, хоча містить підрядок *class*. Це випливає з правила найдовшого збігу.

2.3.4 Літерали

Рядкові та цілочисельні літерали зіставляються на етапі лексичного аналізу та представлені рядковими та цілочисельними токенами відповідно. Логічні літерали *True* і *False* просто представлені своїми токенами ключових слів.

2.3.4.1 Рядкові літерали

Рядкові літерали в ChocoPy значно спрощені порівняно з Python. У ChocoPy рядкові літерали – це послідовність символів ASCII, розділених подвійними лапками (включно з ними): *"..."*. Символи ASCII мають бути в межах десяткового діапазону 32–126 включно, тобто бути вище або дорівнювати пробілу та до символу тильди. Сам рядок може містити подвійні лапки, екрановані зворотною косою рискою, наприклад \".

Оскільки рядкові літерали використовуються як для значень, так і для імен типів, зручно розрізняти дві категорії: рядкові літерали, вміст яких має синтаксис ідентифікатора – *IDSTRING*, та інші рядкові літерали – *STRING*.

Значення рядкового токена – це послідовність символів між роздільними подвійними лапками із застосованими керуючими послідовностями. Розпізнаються наступні керуючі послідовності: \", \n, \t, \\, які є подвійними лапками, новим рядком, табуляцією та зворотною косою рискою відповідно. Будь–яка інша керуюча послідовність вважається некоректною. Приклади керуючих послідовностей наведено у таблиці 2.1.

Таблиця 2.1 – Приклади керуючих послідовностей в ChocoPy

Літерал	Значення
"Hello"	Hello
"He\"ll\"o"	He"ll"o
"He\\\"llo"	He\"llo
"Hell\o"	(помилка: "\o" не розпізнано)

2.3.4.2 Цілочисельні літерали

Цілочисельні літерали в ChocoPy складаються з послідовності однієї чи кількох цифр 0–9, де крайня ліва цифра може бути 0 лише тоді, коли це єдиний символ у послідовності. Тобто цілочисельні літерали з ненульовим значенням не можуть мати початкові нулі. Ціле значення таких літералів інтерпретується за основою 10. Максимальне інтерпретоване значення може бути $2^{31} – 1$ для літералу 2147483647. Літерал із значенням, більшим за це обмеження, призводить до лексичної помилки.

2.3.5 Оператори та роздільники

Це розділений пробілами список символів, які відповідають окремим токенам ChocoPy: + – * // % < > <= >= == != = ( ) [ ] , : . →

2.4 Синтаксис

У таблиці 2.2 наведено граматику ChocoPy з використанням розширеної нотації Бекуса–Наура (EBNF). Токени ключових слів представлені **напівжирним шрифтом**. Літерали та пробільні токени представлені ВЕРХНІМ РЕГІСТРОМ. Нетермінальні символи відформатовані *курсивом у нижньому регістрі*. Оператори та роздільники форматуються як є. Позначення ⟦...⟧ використовується для гру–пування одного або більше символів у правилі виведення та не є токеном у вхідній

мові. Символи або групи можуть бути анотованими наступним чином: «?» означає, що попередній символ або група необов'язкові, «*» означає нуль або більше повторюваних випадків, а «+» позначає один або більше повторюваних випадків.

Головоломний поділ виразів на *expr* і *cexpr* фіксує невиразний момент, що ChocoPy (як Python) допускає лише логічні двійкові або унарні вирази як операнди логічних операторів (*and*, *or*, *not*).

В результаті вираз *True == not False* має викликати синтаксичну помилку (коректним виразом є: *True == (not False)*).

Таблиця 2.2. – Граматика синтаксису мови ChocoPy

Нетермінальний символ	Правило виведення
program ::=	⟦*var_def* | *func_def* | *class_def*⟧ * *stmt**
class_def ::=	**class** ID (ID) : NEWLINE INDENT *class_body* DEDENT
class_body ::=	pass NEWLINE | ⟦*var_def* | *func_def*⟧ +
func_def ::=	**def** ID (⟦*typed_var* ⟦, *typed_var*⟧ *⟧ ?) ⟦->type ⟧ ? : NEWLINE INDENT *func_body* DEDENT
func_body ::=	⟦*global_decl* | *nonlocal_decl* | *var_def* | *func_def*⟧ * *stmt*+
typed_var ::=	ID : *type*
type ::=	ID | IDSTRING | [*type*]
global_decl ::=	**global** ID NEWLINE
nonlocal_decl ::=	**nonlocal** ID NEWLINE
var_def ::=	*typed_var* = *literal* NEWLINE
stmt ::=	*simple_stmt* NEWLINE | **if** *expr* : *block* ⟦**elif** *expr* : *block*⟧ * ⟦**else** : *block*⟧ ? | **while** *expr* : *block* | **for** ID **in** *expr* : *block*

Кінець таблиці 2.2

Нетермінальний символ	Правило виведення
simple_stmt ::=	**pass** | *expr* | **return** ⟦*expr*⟧ ? | ⟦*target* =⟧ + *expr*
block ::=	NEWLINE INDENT *stmt*+ DEDENT
literal ::=	**None** | **True** | **False** | INTEGER | IDSTRING | STRING
expr ::=	*cexpr* | **not** *expr* | *expr* ⟦**and** | **or**⟧ *expr* | *expr* **if** *expr* **else** *expr*
cexpr ::=	ID | *literal* | [⟦*expr* ⟦, *expr*⟧ *⟧ ?] | (*expr*) | *member_expr* | *index_expr* | *member_expr* (⟦*expr* ⟦, *expr*⟧ *⟧ ?) | ID (⟦*expr* ⟦, *expr*⟧ *⟧ ?) | *cexpr bin_op cexpr* | – *cexpr*
bin_op ::=	**+** | **–** | ***** | **//** | **%** | **==** | **!=** | **<=** | **>=** | **<** | **>** | **is**
member_expr ::=	*cexpr* . ID
index_expr ::=	*cexpr* [*expr*]
target ::=	ID | *member_expr* | *index_expr*

Оператори в ChocoPy мають такий самий пріоритет, як і в Python. У таблиці 2.3 підсумовано пріоритет операторів у ChocoPy, від найнижчого пріоритету (найменше зв’язування) до найвищого пріоритету (найбільше зв'язування).

Оператори порівняння неасоціативні (тому ChocoPy, на відміну від Python 3, не допускає таких виразів, як $x < y < z$).

Таблиця 2.3 – Пріоритет і асоціативність операторів в ChocoPy

Пріоритет	Оператори	Асоціативність
1	$\cdot\ if\ \cdot\ else\ \cdot$	права
2	or	ліва
3	and	ліва
4	not	н/д
5	$==, !=, <, >, <=, >=, is$	немає
6	$+, -$ (бінарний)	ліва
7	$*, //, \%$	ліва
8	$-$ (унарний)	н/д
9	. , []	ліва

2.5 Правила типів

Правила типів визначають тип кожного виразу ChocoPy у певному контексті. Контекст – це оточення типів, яке описує тип кожного незв'язаного ідентифікатора, що є у виразі.

2.5.1 Оточення типів

У першому наближенні перевірку типів в ChocoPy можна розглядати як висхідний (з англ. bottom–up) алгоритм: тип виразу *e* виводиться з (раніше виведених) типів підвиразу *e*. Наприклад, ціле число 1 має тип *int*; у цьому випадку

немає підвиразів. Як інший приклад, якщо типи e_1 і e_2 є *int*, тоді вираз $e_1 > e_2$ має тип *bool*.

Ускладнення виникає у випадку виразу *v*, де *v* є змінною або функцією. Неможливо сказати, який тип *v* у строго висхідному алгоритмі; потрібно знати тип, оголошений для *v* у більшому виразі. Таке оголошення має існувати для кожної змінної та функції в коректних програмах ChocoPy.

Щоб отримати інформацію про типи ідентифікаторів, використовується оточення типів. Оточення типів складається з чотирьох частин: локальне оточення *O*, оточення методу/атрибута *M*, назва поточного класу *C*, у якому з'являється вираз або оператор, та тип повернення значення функції чи методу *R*, де з'являється вираз або оператор. *C* є ⊥ коли вираз або оператор знаходиться поза класом, тобто як оператор або вираз на верхньому рівні. Подібним чином *R* є ⊥ коли вираз або оператор знаходиться поза функцією або методом, тобто як оператор або вираз на верхньому рівні. Локальне оточення і оточення методу/атрибута є асоціативними масивами. Локальне оточення є функцією форми *O(v) = T*, яка присвоює тип *T* змінній *v*. Це ж оточення також містить інформацію про сигнатуру функції. Наприклад,

$$O(f) = \{T_1 \times \cdots \times T_n \to T_0;\ x_1, \cdots, x_n;\ v_1 : T_1', \cdots, v_m : T_m'\} \tag{2.6}$$

дає тип *f* і означає, що ідентифікатор *f* має формальні параметри $x_1, \ldots, x_n$ типів $T_1, \ldots, T_n$, відповідно, і має тип повернення значення T_0. Ідентифікатори $v_1, \ldots, v_m$ – змінні та вкладені функції, оголошені в тілі *f* і їх типи $T_1', \ldots, T_m'$ відповідно. Оточення методу/атрибута аналогічно зіставляє клас, його атрибути та методи з їхніми типами. Наприклад, *M(C,a) = T* зіставляє атрибут *a* у класі *C* з типом *T*. Аналогічно

$$M(C, m) = \{T_1 \times \cdots \times T_n \to T_0;\ x_1, \cdots, x_n;\ v_1 : T_1', \cdots, v_k : T_k'\} \tag{2.7}$$

зіставляє метод m класу C з його типом. Зокрема, формула позначає, що метод m у класі C має формальні параметри $x_1, \ldots, x_n$ типів $T_1, \ldots, T_n$ відповідно, і має тип повернення значення T_0. Ідентифікатори $v_1, \ldots, v_k$ – це змінні та вкладені функції, оголошені у тілі метода m, та їх типи $T'_1, \ldots, T'_k$ відповідно.

Третім компонентом оточення типів є ім'я класу C, який містить вираз або оператор, що перевіряється на тип. Четвертим компонентом оточення типів є R – тип повернення значення функції або методу, що містить вираз або оператор функції чи методу, що містить вираз або оператор, який перевіряється на тип.

При перевірці типів оголошень функцій та методів потрібно поширювати оточення типів із зовнішньої області видимості у область видимості функції, де успадковується зв'язування будь–якого ідентифікатора, якщо тільки функція не оголошує формальний параметр, змінну або вкладену функцію з тим самим іменем. Нехай O – поточне локальне оточення і f – функція з визначенням типу $\{T_1 \times \ldots \times T_n \to T_0;\ x_1, \ldots, x_n;\ v_1 : T'_1, \ldots, v_m : T'_m\}$.

При перевірці типів у визначення функції f перевіряється її тіло, використовуючи локальне оточення $O[T_1/x_1][T_2/x_2] \ldots [T_n/x_n][T'_1/v_1] \ldots [T'_m/v_m]$, де запис *O[T/c]* використовується для побудови нової пари ключ–значення наступним чином:

$$O[T/c](c) = T, \tag{2.8}$$

$$O[T/c](d) = O(d) \text{ if } d \neq c. \tag{2.9}$$

2.5.2 Правила перевірки типів

Загальна форма правила перевірки типу наступна:

$$\frac{\vdots}{O, M, C, R \vdash e : T} \tag{2.10}$$

Правило слід читати так: у оточенні типів з локальним оточенням *O*, оточен–

ням методу/атрибуту *M*, що міститься у класі *C* та типом повернення значення *R*, вираз *e* має тип *T*. Рядок під лінією є судженням про типізацію: тире «⊢» відокремлює контекст *(O,M,C,R)* від твердження *e:T*. Крапки над лінією позначають інші судження про типи підвиразів *e*. Ці інші судження є гіпотези правила; якщо гіпотези виконуються, то судження під лінією є істинним.

Змінні. Правило для змінних полягає у тому, що якщо оточення присвоює ідентифікатору *id* тип *T*, то вираз *id* має тип *T*:

$$\frac{O(id) = T, \text{where } T \text{ is not a function type.}}{O, M, C, R \vdash id : T} \quad (2.11)$$

Однак, заборонені ідентифікатори з функціональними типами при читанні значень (тобто, коли ідентифікатори використовуються як вирази у синтаксисі). Це просто відображає той факт, що ChocoPy не розглядає функції як значення першого класу (такі, що можна присвоїти чи зберігати).

Решту правил перевірки типів наведено у додатку А.

Висновки до розділу

Мова ChocoPy – це підмножина Python 3.6, яка була розроблена для навчання студентів мовам програмування. Код, написаний на ChocoPy, сумісний з інтерпретатором Python 3.6.

У вільному доступі є детальна технічна документація з описом граматики мови. Головним документом є довідник з мови ChocoPy. Цей документ формально визначає синтаксис мови, правила типізації та операційну семантику.

Верхній рівень програми ChocoPy складається з класів, функцій, глобальних змінних та операторів. Функції можуть мати вкладені функції. Функції завжди повертають значення, за відсутності оператора $return$ буде повернено $None$. Класи мають просте успадкування, коренем ієрархії класів є $object$. Успадковані методи

можна перевизначати, вони викликаються за допомогою динамічної диспетчеризації.

ChocoPy – статично типізована мова. Для кожного типу T існує тип списку [*T*]. Тип може бути вказаний через ідентифікатор або рядок з назвою ідентифікатора. При оголошенні змінної обов'язково має бути вказано початкове примітивне значення змінної. Ключові слова Python, які не підтримуються у ChocoPy, не можна використовувати, щоб забезпечити сумісність з інтерпретатором Python.

Вся граматика ChocoPy вміщається на листі A4, використовуючи розширену нотацію Бекуса–Наура (EBNF). Для порівняння граматика Python 3 займає близько 1300 рядків коду [13]. Оператори в ChocoPy мають такий самий пріоритет, як і в Python.

ChocoPy має чіткі правила перевірки типів. Тип виразу виводиться з типів підвиразів. Для кожної змінної, оператора, виразу наводяться чіткі алгоритми для визначення типу. Знаючи, наприклад, тип аргументу функції, можна на етапі семантичного аналізу перевірити його сумісність з оголошеним параметром функції тощо.

3 МАТЕМАТИЧНЕ ЗАБЕЗПЕЧЕННЯ

3.1 Математична модель формальної мови

У теорії формальних мов граматика описує, як утворювати рядки з абетки мови, які є правильними відповідно до синтаксису мови. Граматика не описує значення рядків або те, що з ними можна робити в будь–якому контексті – лише їхню форму. Формальна граматика визначається як набір правил створення таких рядків у формальній мові.

У класичній формалізації генеративних граматик, вперше запропонованій Ноамом Чомскі у 1950–х роках, граматика G складається з наступних компонентів:

– скінченна множина N нетермінальних символів, яка не перетинається з ряд–ками, утвореними з G;

– скінченна множина Σ термінальних символів, відмінних від N;

– скінченна множина правил виводу P, кожне з яких має вигляд

$$(\Sigma \cup N)^* N (\Sigma \cup N)^* \to (\Sigma \cup N)^*, \qquad (3.1)$$

де * – оператор Кліні, а $\cup$ позначає об'єднання множин. Тобто, кожне правило виведення відображає (з англ. map) один рядок символів в інший, де перший рядок («голова») містить довільну кількість символів за умови, що хоча б один з них не є терміналом. У випадку, коли другий рядок («тіло») складається лише з порожнього рядка, тобто не містить жодного символу, його можна позначити спеціальною нотацією (часто Λ, e або ϵ), щоб уникнути плутанини.

– символ $S \in N$, який є початковим символом, також називається символом речення [14].

Граматика формально визначається як кортеж (N,Σ,P,S).

Коли Ноам Чомскі вперше формалізував генеративні граматики, він класифікував їх за типами, відомими зараз як ієрархія Чомскі. Різниця між цими типами полягає в тому, що вони мають дедалі суворіші правила виводу і тому мо–жуть виражати меншу кількість формальних мов. Двома важливими типами є

контекстно–вільні граматики (тип 2) та регулярні граматики (тип 3). Мови, які можна описати за допомогою такої граматики, називаються контекстно–вільними мовами та регулярними мовами відповідно.

Контекстно–вільна граматика – це формальна граматика, правила виведення якої можуть бути застосовані до нетермінального символу незалежно від його контексту. Зокрема, у контекстно–вільній граматиці кожне правило виводу має вигляд $A \rightarrow \alpha$, де A – один нетермінальний символ, і α – рядок терміналів та/або нетерміналів (α може бути пустим). Незалежно від того, які символи його оточують, єдиний нетермінал A у лівій частині завжди можна замінити на α з правого боку.

Контекстно–вільна граматика G визначається 4–місним кортежем

$$G = (V, \Sigma, R, S), \tag{3.2}$$

де:

– V – скінченна множина, кожен елемент якої $v \in V$ називається нетерміналь–ним символом або змінною. Кожна змінна представляє окремий тип словосполучення або пункту в реченні. Змінні також іноді називають синтаксичними категоріями. Кожна змінна визначає суб–мову мови, визначеної G.

– Σ є скінченною множиною терміналів, відокремлених від V, які складають фактичний зміст речення. Множина терміналів є абеткою мови, визначеною граматикою G.

– R – скінченне відношення у $V \times (V \cup \Sigma)^*$, де зірочка позначає оператор Кліні. Члени R називаються правилами переписування або правилами виведення граматики.

– S – це стартова змінна (або стартовий символ), яка використовується для представлення всього речення (або програми). Вона повинна бути елементом V.

Регулярні мови зазвичай використовують для лексичного аналізу та регулярних виразів, які можуть бути використані як інструмент аналізу. Контекстно–вільні граматики описують мовні конструкції та використовуються для синтаксичного аналізу.

3.2 Математична модель лексичного аналізу

Завданням лексичного аналізу є перетворення послідовності символів, що подається на вхід компілятору, у відповідну послідовність токенів. Зокрема, при кожному виклику сканер повинен знайти найдовшу послідовність символів на вході, починаючи з поточного символу, яка відповідає токену, і повернути його.

Лексичні аналізатори працюють на основі скінченних автоматів (з англ. finite–state machine, FSM). Це абстрактний автомат, який у будь–який момент часу може перебувати точно в одному зі станів кількість яких є скінченною. Скінченний автомат переходить з одного стану в інший, відповідаючи на певні вхідні дані. Списком станів, початковий стан і вхідні дані, які викликають кожен перехід, визначають скінченний автомат [15].

Для сканування вихідного коду використаємо детермінований скінченний автомат (з англ. deterministic finite automaton, DFA). Це скінченний автомат, який приймає або відкидає заданий рядок символів, проходячи через послідовність станів, однозначно визначену цим рядком. Детермінованість означає однозначність прогону обчислень [16].

Детермінований скінченний автомат M – це кортеж з 5 елементів $(Q, \Sigma, \delta, q_0, F)$, який складається з:

– скінченної множини станів Q;

– скінченної множини вхідних символів, яка називається абеткою Σ;

– функції переходу $\delta : Q \times \Sigma \to Q$;

– початкового або стартового стану $q_0 \in Q$;

– множини допустимих станів $F \subseteq Q$.

Нехай $w=a_1 a_2 \ldots a_n$ – рядок абетки Σ. Автомат M приймає рядок w, якщо у Q існує послідовність станів $r_0, r_1, \ldots, r_n$, яка задовольняє наступним умовам:

а) $r_0 = q_0$;

б) $r_{i+1} = \delta(r_i, a_{i+1})$, для $i = 0, \ldots, n–1$;

в) $r_n \in F$.

Перша умова говорить, що автомат починає роботу у початковому стані q_0. Друга умова говорить, що для кожного символу рядка *w* автомат буде переходити зі стану в стан відповідно до функції переходу δ. Остання умова говорить, що автомат приймає *w*, якщо останній вхідний символ *w* призводить до зупинки автомата в одному з допустимих станів. При невиконанні умов, автомат відхиляє рядок. Множина рядків, які приймає *M*, називається мовою, яку розпізнає *M*, і позначається через *L(M)*.

На рисунку 3.1 зображений детермінований скінченний автомат, що приймає лише двійкові числа, які кратні 3. Початковий стан S_0 є одночасно і допустимим. Наприклад, рядок «1001» призводить до послідовності станів S_0, S_1, S_2, S_1, S_0, і, таким чином, приймається.

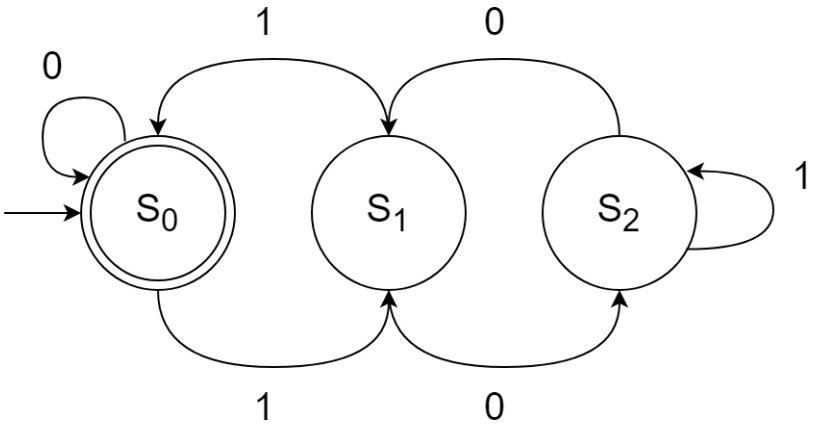

Рисунок 3.1 – Приклад детермінованого скінченного автомата

3.3 Математична модель синтаксичного аналізу

Для синтаксичного аналізу використаємо алгоритм рекурсивного спуску (з англ. recursive–descent parsing). Це різновид низхідного (з англ. top–down) синтаксичного аналізу, побудованого з множини взаємно рекурсивних процедур (або нерекурсивного еквівалента), де кожна така процедура реалізує один з нетермінальних символів граматики. Таким чином, структура результуючої програми точно відображає структуру граматики, яку вона розпізнає [17].

Предиктивний синтаксичний аналізатор – це синтаксичний аналізатор рекурсивного спуску, який не вимагає пошук з поверненням (з англ. backtracking). Предиктивний синтаксичний аналіз можливий лише для класу LL(k)–граматик, які є контекстно–вільними граматиками, для яких існує деяке натуральне число *k*, що дозволяє синтаксичному аналізатору рекурсивного спуску вирішувати, яке правило виведення використовувати, розглядаючи лише наступні *k* токенів вхідних даних [18].

LL(k)–граматика – це контекстно–вільна граматика, яка може бути розібрана LL–синтаксичним аналізатором, за допомогою якого вхідні дані аналізуються зліва направо і будується ліворекурсивне виведення (з англ. leftmost derivation) речення.

LL(k)–синтаксичний аналізатор – це детермінований автомат з магазинною пам'яттю (з англ. deterministic pushdown automaton, DPDA), який має можливість підглядання наступних *k* вхідних символів без їх читання.

Автомат з магазинною пам'яттю (з англ. pushdown automaton, PDA) формально визначається як 7–місний кортеж

$$M = (Q,\Sigma,\Gamma,\delta,q_0,Z,F), \tag{3.3}$$

де Q – скінченна множина станів; Σ – скінченна множина, яка називається вхідною абеткою; Γ – скінченна множина, яка називається абеткою стека; δ – скінченна підмножина $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \times Q \times \Gamma^*$, функція переходів; $q_0 \in Q$ – початковий стан; $Z \in \Gamma$ – початковий символ стеку; $F \subseteq Q$ – множина дозволених кінцевих станів.

Елемент $(p,a,A,q,\alpha) \in \delta$ – це перехід від M. Мається на увазі, що M, у стані $p \in Q$, на вході $a \in \Sigma \cup \{\varepsilon\}$ та з $A \in \Gamma$ як самий верхній символ стека, може прочитати a, змінити стан на q, видалити A, замінивши додаванням $\alpha \in \Gamma^*$. Компонент відношення переходу $(\Sigma \cup \{\varepsilon\})$ використовується для формалізації того, що PDA може або прочитати символ з входу, або продовжити роботу, не читаючи вхідних даних [16].

Нижче наведено приклад формального опису PDA, який розпізнає мову $\{0^n 1^n \mid n \geq 0\}$ за кінцевим станом

$$M = (Q,\Sigma,\Gamma,\delta,q_0,Z,F), \tag{3.4}$$

де стани: $Q = \{p,q,r\}$; вхідна абетка: $\Sigma = \{0,1\}$; абетка стеку: $\Gamma = \{A,Z\}$; початок стеку: $q_0 = p$; початковий символ стеку: Z; дозволені кінцеві стани: $F = \{r\}$. На рисунку 3.2 можна побачити графічне зображення прикладу.

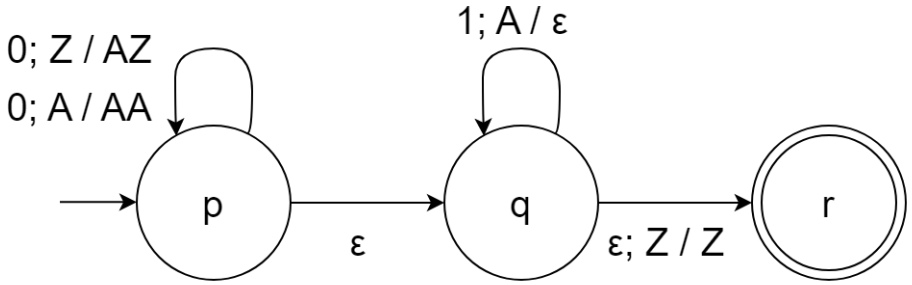

Рисунок 3.2 – Приклад автомата з магазинною пам’яттю

Функція переходів δ складається з шести інструкцій. Перші дві інструкції *(p,0,Z,p,AZ)* та *(p,0,A,p,AA)* говорять, що у стані p кожного разу, коли зчитується символ 0, один символ A виштовхується у стек. Переміщення символу A поверх іншого A формалізується як заміна верхнього A на AA (і аналогічно для переміщення символу A поверх Z).

Третя і четверта інструкції *(p,ϵ,Z,q,Z)* та *(p,ϵ,A,q,A)* говорять про те, що в будь–який момент автомат може перейти зі стану p у стан q.

П'ята інструкція *(q,1,A,q,ϵ)* говорить, що у стані q на кожен прочитаний символ 1 видаляється зі стеку один символ A. Нарешті, шоста інструкція *(q,ϵ,Z,r,Z)* говорить, що автомат може перейти зі стану q в стан r тільки тоді, коли стек складається з одного символу Z.

Висновки до розділу

Регулярні мови лежать в основі лексичного аналізу та можуть бути представлені регулярними виразами. Контекстно–вільні граматики лежать в основі синтаксичного аналізу та представляють мовні конструкції.

Лексичний аналізатор перетворює вхідну послідовність символів у відповідну послідовність токенів. Сканер знаходить найдовшу послідовність символів, яка відповідає токену.

В основі лексичних аналізаторів використовуються скінченні автомати. Для сканування вихідного коду зазвичай використовують детермінований скінченний автомат. Описано математичну модель DFA та приклад його роботи.

Синтаксичний аналізатор будує абстрактне синтаксичне дерево, в якому кожне правило виведення формальної граматики стає його вузлом.

Одним з алгоритмів, які використовуються для синтаксичного аналізу є алгоритм рекурсивного спуску. Він є LL(k)–синтаксичним аналізатором, або детермінованим автоматом з магазинною пам’яттю. Показано формальний опис PDA та наведено приклад його роботи.

LL(k)–граматика дозволяє синтаксичному аналізатору рекурсивного спуску використовувати потрібне правило виведення, беручи до уваги наступні *k* токенів вхідних даних. У власній реалізації парсеру буде розглядатись 1, а в деяких випадках 2 наступні токени.

4 ВИБІР ТЕХНОЛОГІЇ РОЗРОБКИ

4.1 Мова програмування С

С є мовою комп’ютерного програмування загального призначення. Можливості цільових ЦП чітко відображаються функціями С. Вона тривалий час використовується в операційних системах стеках протоколів і драйверах пристроїв.

С – процедурна імперативна мова, підтримує області видимості лексичних змінних, структурне програмування та рекурсію зі статичною системою типів. Була створена, щоб після компіляції забезпечити низькорівневий доступ до пам’яті. Її мовні конструкції ефективно відображаються на машинні інструкції з мінімальною підтримкою часу виконання. Попри свою низькорівневість, мова була розроблена, щоб заохотити кросплатформне програмування. Написана з урахуванням переносимості програма С сумісна зі стандартами та може бути скомпільована для широкого кола комп’ютерних платформ і операційних систем з мінімальними змінами у вихідному коді [19].

4.1.1 Використання в системному програмуванні

С широко використовується для системного програмування при реалізації операційних систем і вбудованих системних програм.

Код, згенерований після компіляції, не вимагає багатьох системних функцій і може бути викликаний з деякого завантажувального коду простим способом – її легко виконати.

Оператори та вирази мови С зазвичай добре відображаються на послідовності інструкцій для ЦП, і, як наслідок, існує низький попит на системні ресурси під час виконання – вона швидка у виконанні.

Завдяки багатому набору операторів мова С може використовувати багато функцій цільових ЦП. Там, де певний ЦП має більш езотеричні інструкції, можна створити варіант мови з, можливо, внутрішніми функціями для використання цих інструкцій – вона може використовувати практично всі особливості цільового ЦП.

Ця мова дозволяє легко накладати структури на блоки двійкових даних, дозволяючи зрозуміти дані, переміщатися по ним та змінювати їх – вона може записувати структури даних, навіть файлові системи.

Мова підтримує багатий набір операторів, включаючи маніпулювання бітами, для цілочисельної арифметики та логіки, і, можливо, різні розміри чисел з рухомою комою – вона може ефективно обробляти належним чином структуровані дані. С – досить маленька мова, лише з кількома операторами та без надто великої кількості особливостей, які створюють великий цільовий код – вона зрозуміла.

С має прямий контроль над виділенням і звільненням пам'яті, що забезпечує розумну ефективність і передбачуваний час для операцій обробки пам'яті, без будь–яких побоювань щодо епізодичних подій збірки сміття – вона має передбачувану продуктивність.

Доступ до апаратного забезпечення платформи можна отримати за допомогою вказівників (з англ. pointer), тому специфічні для системи особливості (наприклад, регістри керування/статусу, регістри вводу/виводу) можна налаштовувати та використовувати за допомогою коду, написаного мовою С – вона добре взаємодіє з платформою, на якій працює.

Залежно від компонувальника та середовища, код С також може викликати бібліотеки, написані мовою асемблера, і може викликатися з мови асемблера – вона добре взаємодіє з іншим кодом нижчого рівня.

С та її погодження викликів та структури компонування зазвичай використовуються разом з іншими мовами високого рівня, при цьому підтримуються виклики як до С, так і з С – вона добре взаємодіє з іншим кодом високого рівня.

С має дуже зрілу та широку екосистему, включаючи бібліотеки, фреймворки, компілятори з відкритим кодом, відлагоджувачі та утиліти, і є стандартом де–

факто. Цілком імовірно, що драйвери вже існують на С або існує подібна архітектура ЦП, як бек–енд частина компілятора С, тому немає підстав обирати іншу мову.

4.1.2 Деякі інші мови самі написані мовою C

Наслідком широкого доступу та ефективності C є те, що компілятори, бібліотеки та інтерпретатори інших мов програмування часто реалізуються на C. Наприклад, еталонні реалізації Python, Perl, Ruby і PHP написані мовою C.

4.1.3 Обмеження

Хоча C була популярною, впливовою і надзвичайно успішною, вона має недоліки. Стандартна обробка динамічної пам’яті за допомогою *malloc* і *free* є схильною до помилок. Серед помилок: витік пам’яті, коли пам’ять виділена, але не звільнена; і доступ до раніше звільненої пам'яті.

Використання вказівників і пряме маніпулювання пам’яттю означає можливе пошкодження пам’яті, можливо, через помилку програміста або недостатню перевірку неправильних даних. Існує деяка перевірка типу, але вона не стосується таких областей, як варіативні (з англ. variadic) функції, і перевірку типу можна банально або ненавмисно обійти. Вона слабко типізована.

Оскільки код, згенерований компілятором, сам по собі містить кілька перевірок, програміст має передбачити врахування всіх можливих результатів: захист від переповнення буфера, перевірки меж масиву, переповнення стека, виснаження пам’яті, а також врахування станів гонитви (з англ. race condition), ізоляції потоку тощо.

Використання вказівників і маніпулювання ними під час виконання означає, що можуть існувати два способи доступу до тих самих даних (накладання псевдонімів), які неможливо визначити під час компіляції. Це означає, що деякі оптимізації, які можуть бути доступні для інших мов, неможливі в C. Деякі зі стандартних функцій бібліотеки, наприклад *scanf* або *strncat*, можуть призвести до переповнення буфера.

Існує обмежена стандартизація щодо підтримки низькорівневих варіантів у згенерованому коді, наприклад: різні погодження викликів функцій та двійковий

програмний інтерфейс; різні конвенції пакування структур; різне впорядкування байтів у великих цілих числах (включаючи порядок байтів). У багатьох реалізаціях мови деякі з цих параметрів можна обробляти за допомогою директиви препроцесора *#pragma*, а деякі – за допомогою додаткових ключових слів, наприклад, використовувати погодження викликів *__cdecl*. Але директива та параметри не підтримуються постійно.

Обробка рядків за допомогою стандартної бібліотеки потребує багато коду та вимагає явного керування пам'яттю. Мова безпосередньо не підтримує об'єктну орієнтацію, інтроспекцію, оцінку виразів під час виконання, узагальнене програмування тощо.

Є кілька засобів захисту від невідповідного використання мовних особливостей, що може призвести до непридатності коду. С не має стандартної підтримки для обробки винятків і пропонує лише коди повернення для перевірки помилок. Функції стандартної бібліотеки *setjmp* і *longjmp* були використані для реалізації механізму *try–catch* за допомогою макросів.

4.2 Мова програмування Java

Java є високорівневою об'єктно–орієнтованою мовою програмування, підтримує класи, створена з метою мати найменше залежностей реалізації. З її допомогою програмісти можуть написати код один раз і виконувати його всюди. Іншими словами скомпільований код на Java не потрібно перекомпільовувати, щоб він працював на підтримуваних Java платформах. Код на Java компілюється в байт–код, який використовує віртуальна машина Java (з англ. JVM), не залежно від архітектури комп'ютера. Синтаксис Java є С–подібним, але менш низькорівневий, ніж мови С і С++. Середовище виконання для Java має динамічні можливості недоступні в традиційних скомпільованих мовах, а саме модифікація коду під час виконання та рефлексія [20].

4.2.1 Система виконання

4.2.1.1 Віртуальна машина Java і байт–код

Java розроблялась з ціллю портативності, тобто програми, написані на Java, мають працювати однаково на будь–якому поєднанні апаратного забезпечення з операційною системою при адекватній підтримці часу виконання. Це стало можливим завдяки компіляції коду, написаного на Java до проміжного представлення, що називається байт–кодом, замість безпосереднього машинного коду, що прив'язаний до архітектури. Байт–код Java аналогічний низькорівневому коду, але його інструкції розроблені для виконання на віртуальній машині, яка написана для апаратного забезпечення хост–комп'ютера.

Кінцеві користувачі як правило користуються середовищем виконання Java (з англ. JRE), що інстальоване на їх пристрої. Стандартні бібліотеки забезпечують типовий спосіб доступу до специфічних для хоста функцій, як–от як графіка, потоки та мережа.

Використання байт–коду робить перенесення програм простим. Проте через додаткові витрати при інтерпретації байт–коду в низькорівневі машинні інструкції інтерпретовані програми майже завжди працювали повільніше, ніж виконувані файли для цільової платформи. Just–in–time (JIT) компілятори, що перекомпільовують байт–коди в машинний код під час виконання, використовувались ще на ранній стадії. Компілятор Java Hotspot – це фактично два компілятори в одному. Сама Java є незалежною від платформи, оскільки JVM перекладає Java байт–код на машинний код платформи, на якій вона встановлена.

4.2.1.2 Продуктивність

Загальноприйняте уявлення, що програми на Java повільніші і потребують більше пам'яті, чим, наприклад, C++. Проте швидкість, з якою виконуються програми на Java, значно покращилася з впровадженням just–in–time компіляції в Java 1.1, додаванням функцій з кращим аналізом коду (таких як внутрішні класи,

класи *Optional* та *StringBuilder* тощо), і оптимізації віртуальної машини HotSpot, яка стала JVM за замовчуванням у 2000 році. У Java 1.5 продуктивність була покращена завдяки додаванню пакета *java.util.concurrent*, включаючи реалізацію без блокування *ConcurrentMap* та інших багатопотокових колекцій, і їх було додатково вдосконалено з Java 1.6.

4.2.1.3 Автоматичне керування пам'яттю

Java має збирач сміття (з англ. garbage collector) для керування пам'яттю протягом життєвого циклу об'єкта. Вихідний код описує, коли створюються об'єкти, а середовище виконання Java очищує пам'ять, коли більше не має потреби в об'єктах. Як тільки на об’єкт не залишається жодного посилання, пам'ять, яку він займає, стає кандидатом для звільнення збирачем сміття. Може статися витік пам’яті, якщо код програми містить посилання на об'єкт, який більше не використовується. Це зазвичай відбувається, коли об'єкти, в яких вже немає потреби, зберігаються у контейнерах, але які поки що використовуються. При виклику метода для видаленого об'єкта, генерується виключення за нульовим вказівником (*NullPointerException*).

Одна з ідей моделі автоматичного керування пам'яттю у Java полягає в тому, що програмісти не зобов’язані керувати пам'яттю вручну. В залежності від мови пам'ять, потрібна для створення об'єктів, може неявно виділятися на стеку або виділятися і вивільнятися з купи (з англ. heap) явно. В цьому випадку програмісті відповідає за керування пам’яттю. Якщо об'єкт не звільняється програмою, відбувається витік пам'яті. Якщо спробувати отримати доступ до пам'яті, яка вже була звільнена, або звільнити її, результат не визначений і важко передбачуваний, і програма, швидше за все, стане нестабільною або припинить роботу з аварійним завершенням. Частково це можуть виправити розумні вказівники, але вони збільшують накладні витрати і складність програми. Збірка сміття не рятує від логічних витоків пам'яті, тобто тих випадків, коли на адреси пам'яті ще є посилання, але вона може не використовуватися.

Збирання сміття на практиці відбувається у будь–який момент. В кращому випадку, це має відбуватися під час простою програми. Воно точно спрацює, коли у купі буде недостатньо вільної пам'яті для створення нового об'єкта, що може призвести до миттєвого зупинення програми. Явне керування пам'яттю у Java неможливе.

У Java не підтримується арифметика С–подібних вказівників, коли для адреси об'єктів можна арифметично додавати або віднімати зсув (з англ. offset). Таким чином збирач сміття може переміщувати об'єкти, на які є посилання, цим забезпечується безпеку типів.

Так само як у об'єктно–орієнтованих мовах (наприклад, C++), примітивні типи даних у Java зберігаються для об’єктів безпосередньо у полях або для методів у стеку, а не у купі, як для не примітивних типів даних. Так було вирішено при розробці Java для забезпечення продуктивності.

У Java є декілька різних збирачів сміття. З 9–ї версії Java, у HotSpot за замовчуванням працює Garbage First збирач сміття (G1GC). Однак, існує також декілька інших збирачів сміття, які можна застосувати для управління купою. Проте багатьом програмам на Java достатньо буде G1GC. Раніше Java 8 використовувала паралельний збирач сміття.

Попри автоматичне керування пам'яттю, програміст повинен правильно обробляти інші види ресурсів, такі як з'єднання з базами даних, мережеві з'єднання, дескриптори файлів і т. ін., особливо при наявності виключень.

4.2.2 Нові функції в Java у версіях з 8 по 17

Якщо раніше нові версії Java виходили раз у 2 роки, то тепер цикл розробки скоротився до півроку. З 8 по 17 версії в Java було реалізовано багато нових функцій, зокрема:

– лямбда–вирази, потоки (8 версія);

– виведення типу локальної змінної – оголошення змінної за допомогою клю–чового слова *var* без явного вказування типу (10 версія);

– збирач сміття ZGC з високою продуктивністю та меншим часом зупинок програми (11 версія);

– вираз *switch* та текстові блоки (13 версія);

– класи *record* для зменшення написання шаблонного коду такого як, гетери, сетери, функції *hashcode, equals*, *toString*, конструктори. Зіставлення зі взірцем (з англ. pattern matching) для оператора *instanceof* (14 версія);

– зіставлення зі взірцем для оператора *switch* (17 версія);

– загальна продуктивність у Java 17 порівняно зі старими версіями значно вища, незалежно від того, який збирач сміття використовується.

4.3 Мова програмування Python

Python – це проста, але потужна і гнучка мова програмування. Вона має ефективний підхід до об'єктно–орієнтованого програмування, та ефективні структури даних. Елегантний синтаксис і динамічна типізація Python, а також інтерпретована природа роблять її ідеальною мовою для написання сценаріїв і швидкої розробки додатків у багатьох сферах на більшості платформ.

4.3.1 Особливості дизайну

Python є мультипарадигмальною мовою програмування. Об'єктно–орієнтоване програмування та структурне програмування підтримуються повністю, а також багато функцій функціонального і аспектно–орієнтованого програмування. Інші парадигми, як–от проєктування за контрактом та логічне програмування, підтримуються завдяки розширенням.

Динамічна типізація, поєднання підрахунку посилань та збирач сміття, що виявляє цикли, допомагають Python керувати пам'яттю. Він застосовує динамічне розпізнавання імен: пізнє зв'язування імен змінних і методів, коли виконується програма.

Python в деякій мірі підтримує функціональне програмування в стилі Lisp.

Вона має функції *reduce*, *map* та *filter*; словники, спискові вирази, множини та вирази генераторів. Два модулі стандартної бібліотеки (*functools* та *itertools*) використовують запозичені зі Standard ML і Haskell функціональні інструменти.

Її основна філософія підсумована в документі «Дзен Пайтона» (PEP 20), який включає в себе такі афоризми, як:

– гарне краще за потворне;

– очевидне краще за неочевидне;

– просте краще ніж складне;

– складне краще ніж заплутане;

– має значення легкість читання [21].

Замість того, щоб вбудовувати всю свою функціональність в ядро, Python був розроблений з можливістю розширення використовуючи модулі. Компактна модульність зробила її популярною при додаванні програмованих інтерфейсів до існуючих додатків.

Python має простий, незахаращений синтаксис та граматику, та надає розробникам вибір методології для написання коду. На противагу Perl, у якого девіз – «існує більше одного способу зробити це», Python сповідує філософію «повинен бути один – і краще лише один – явний спосіб це зробити».

Щоб не ризикувати якістю задля незначного збільшення швидкості, при розробці Python уникають передчасної оптимізації та відкидають патчі до некритичних частин еталонної реалізації CPython. Якщо важлива швидкість, критичні до часу функції з коду Python можуть бути переміщені до модулів розширення, написаних на низькорівневих мовах, як–от C, або використати PyPy, що є компілятором just–in–time. Можна використати реалізацію Cython, яка перекладає код Python на C та на рівні C робить прямі виклики API в Python інтерпретатор.

Python має бути легкою для читання мовою. Її форматування візуально не скупчене, бо часто використовує ключові слова на англійській мові там, де інші мови застосовують розділові знаки. На противагу іншим мовам, вона не використовує фігурні дужки для розділення блоків; після операторів крапка з ко–

мою дозволена, але використовується рідко.

4.3.2 Типізація

Python використовує качину типізацію (з англ. duck typing) і має типізовані об'єкти. Обмеження типу не перевіряються при компіляції; натомість, операції над об'єктом можуть завершитися невдачею, тому що він не має відповідного типу. Незважаючи на динамічні типи, Python є строго типізованою мовою, забороняючи операції, що нечітко визначені (наприклад, коли додається рядок і число), натомість, аби неявно намагатися розібратися в них.

Розробники на Python можуть визначати власні типи використовуючи класи, що використовуються для об'єктно–орієнтованого програмування. Створення нових екземплярів класів відбувається шляхом виклику класу (наприклад, *SomeClass()* чи *OtherClass()*), класи відповідно є екземпляри метакласу *type*, що дозволяє метапрограмування та рефлексію.

Python підтримує послідовну типізацію. Синтаксис Python дозволяє вказувати статичні типи, але у реалізації за замовчуванням CPython вони не перевіряються. Експериментальна необов'язкова перевірка статичних типів, mypy, підтримує перевірку типів під час компіляції.

4.3.3 Модуль ast

ast – модуль стандартної бібліотеки Python. Коли запускається Python–файл (*.py*), код спочатку компілюється в AST, а потім перетворюється на байт–код (файли *.pyc*). Під час виконання байт–код передається віртуальній машині Python для інтерпретації. Генерація AST є найважливішою функцією ast, але є й інші способи використання модуля [22].

Використовуючи модуль *ast* програми на Python можуть обробляти дерева граматики абстрактного синтаксису Python. Використовуючи власний парсер Python можна виконувати наступні завдання:

– розібрати рядок коду Python;

– здійснювати навігацію та маніпулювати вузлами AST;

– аналізувати синтаксичні помилки в коді Python;

– створювати власні трансформери AST;

– генерувати код на Python з об'єктів AST;

– проводити аналіз коду та лінкування;

– візуалізувати абстрактні синтаксичні дерева за допомогою *ast*;

– такі IDE, як IntelliJ, використовують AST для розбору файлів, щоб зрозумі–ти структуру коду;

– пакет flake8 працює на основі AST для знаходження стилістичних помилок в коді.

Отже, Python має інструменти, які можна використати при написанні компілятора як мінімум для самого себе. Тобто Python може спростити етап фронт–енда компіляції ChocoPy, надаючи готові рішення.

Висновки до розділу

Вибір технології для розробки програмного продукту залежить від цілей, строків, та наявних ресурсів. Оскільки ціллю є розробка інтерпретатора ChocoPy, для цього підходять всі розглянуті мови: C, Java, Python.

Java – строго типізована мова, що дозволяє уникнути тривіальних помилок типів під час виконання програми. Python має вбудований модуль *ast*, який можна було б використати у фронт–енді компілятора, оскільки ChocoPy є підмножиною Python. Обравши C, можна було б досягти швидкодії інтерпретатору Python.

Але IT–команда в основному обирає для розробки замовлення технологію, якою володіє найкраще. Враховуючи досвід роботи з Java та наявність великої кількості допоміжних інструментів для збірки і тестування проєкту, складність C (інакше кажучи обмеженість часу для написання проєкту) та бажання розробити всі етапи інтерпретатору самостійно (не використовувати готовий парсер Python), було обрано Java.

5 РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

5.1 Засоби розробки

На початку роботи над проєктом необхідно визначитись з інструментами. Їх вибір впливає на якість та швидкість роботи. Для розробки інтерпретатора використаємо Java версії 17, яка є останньою LTS–версією, тобто компанія Oracle буде підтримувати її до вересня 2026 року [23]. Вся реалізація буде виконана на чистій Java без використання фреймворків або спеціалізованих бібліотек, окрім тестування, що буде описано в наступному розділі.

Як інструмент збірки використаємо Apache Maven 3. Він потрібен для автоматизації завантаження бібліотек залежностей, побудови пакунків з виконуваним кодом, та виконання тестів.

Для керування версіями файлів використаємо систему Git. Вона дозволяє безпечно працювати над новими версіями програмного забезпечення з можливістю відкату до попередніх версій коду. Також це незамінний інструмент для організації спільної роботи, коли над проєктом працює більше одного розробника.

Інтегрованим середовищем розробки буде слугувати Intellij IDEA – де–факто стандарт для Java–програмістів. Це потужний редактор коду, який дозволяє зручно робити пошук у проєкті, використовуючи регулярні вирази, редагувати код у декількох місцях одночасно, має гнучку інтеграцію з Git, що дозволяє порівнювати різні версії файлів, має потужний статичний аналізатор, який рекомендує варіанти рефакторингу коду.

Оскільки ChocoPy є підмножиною Python 3.6, не зайвим буде інсталювати цю або вищу версію Python для порівняння роботи власної реалізації з реальним інтерпретатором, адже код, написаний на ChocoPy має так само працювати з Python. Для перевірки коректності коду знадобиться онлайн компілятор з графічним інтерфейсом на вебсайті ChocoPy [24].

Розробка буде відбуватися на ОС Linux Mint 21.1 – дистрибутиві Linux на основі Ubuntu та Debian. На Linux зручно інсталювати програмне забезпечення та налаштовувати оточення користувача.

5.2 Структура програмного забезпечення

Інтерпретатор буде виконуваним файлом, що не потребує інсталяції. Це консольний додаток, який приймає аргументом один текстовий файл з вихідним кодом, і виводить в стандартний потік результат виконання коду або повідомлення про помилки. При виклику програми з кількістю аргументів, що відрізняється від одного, буде показано повідомлення про помилку.

Буде використано лише 2 етапи компіляції: фронт–енд і бек–енд. Фронт–енд складатиметься з фаз лексичного, синтаксичного та семантичного аналізу. Блок–схему роботи інтерпретатора можна побачити на рисунку 5.1. На кожному кроці будуть перевірятись помилки, і при їх наявності зупинятиметься подальша робота та показуватимуться повідомлення про знайдені помилки у стандартному потоці для виведення помилок. Повідомлення про помилку містить номер рядка, на якому вона була знайдена.

Інтерпретатор буде однопрохідним тобто читатиме вихідний код програми користувача рівно один раз. Перша фаза лексичного аналізу сканує текст і передає набір коректних токенів до синтаксичного аналізатора, який перевіряє правильність послідовності токенів, та будує AST.

Семантичний аналіз перевіряє існування використаних змінних, чи дійсно значення змінних відповідають їх оголошеним типам, чи сумісні операнди виразу, рівність кількості параметрів і переданих аргументів функції, виводить тип виразу, правильність місця використання оператору *return, global, nonlocal* тощо.

Бек–енд не буде генерувати машинний код чи байт–код, а одразу виконувати інструкції в AST і показувати результат їхньої роботи. Після виконання коду інтерпретатор завершує роботу і повертає керування у командну оболонку, в якій був викликаний. Для повторного виконання програми необхідно ще раз виконати консольну команду з ім'ям додатку та передати аргументом текстовий файл з вихідним кодом ChocoPy.

Програма не записує події своєї роботи в лог, а також не модифікує вхідний файл та будь–які інші файли чи системні ресурси.

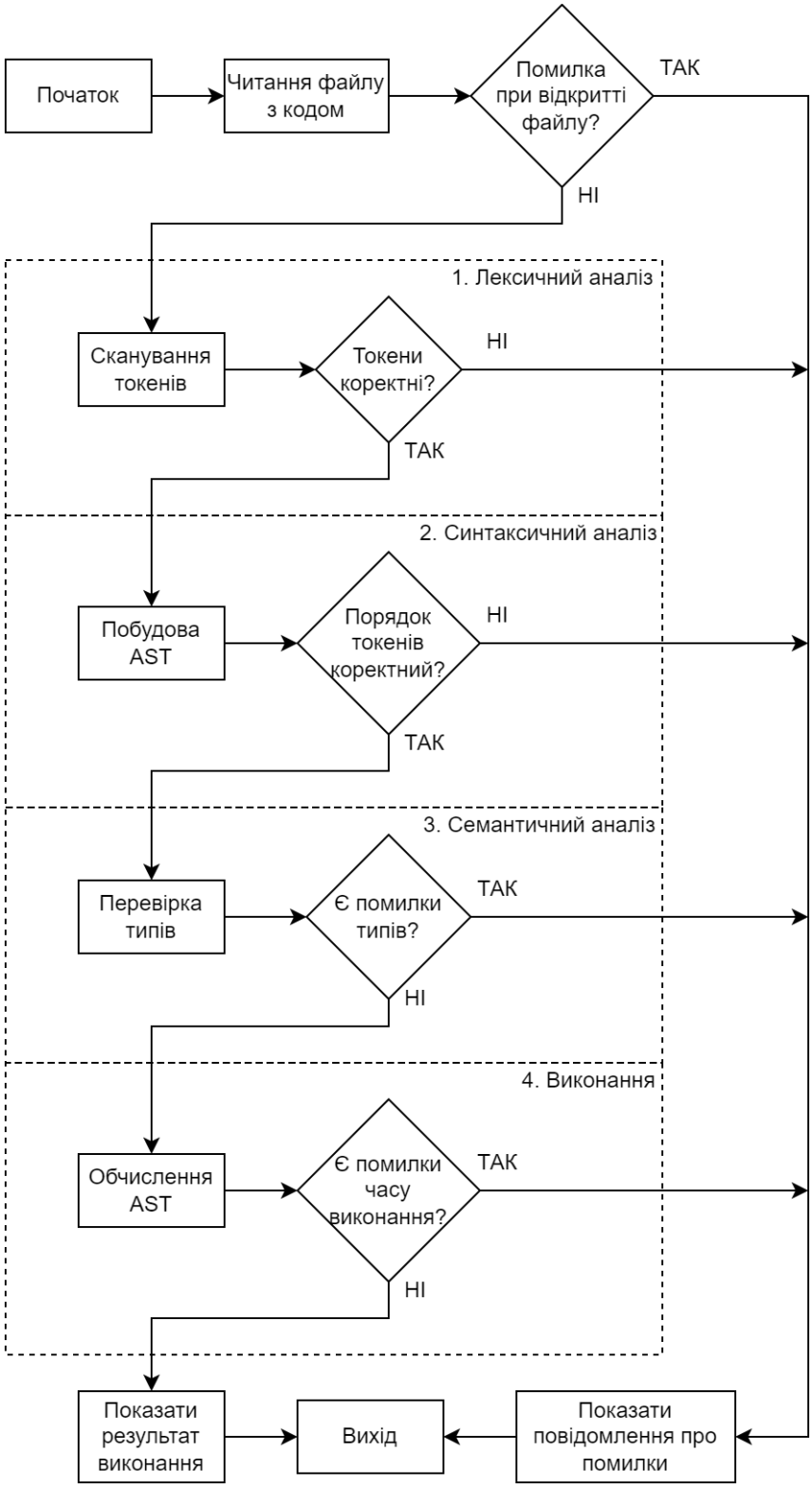

Рисунок 5.1 – Блок–схема роботи інтерпретатору ChocoPy

UML–діаграму класів інтерпретатора можна побачити на кресленику ІСз91.020БАК.004 Д1. Головним класом з усією бізнес логікою буде *ChocoPy.java*. Він використовує 4 класи для кожної з фаз компіляції.

Scanner.java (лексичний аналіз) використовує класи *Token.java* для створення об’єктів токенів з лексем та *TokenType.java* для типів токенів. *Scanner.java* сканує вихідний код, перетворюючи його у список токенів і передає його до *Parser.java*.

Parser.java (синтаксичний аналіз), аналізуючи токени, створює вузли абстрактного синтаксичного дерева за допомогою класів *Stmt.java* (для операторів) та *Expr.java* (для виразів). Це абстрактні класи, які наслідуються дочірніми класами для відповідних інструкцій. Наприклад, *Stmt.If.java* для оператору *if–elif–else*, або *Expr.Logical.java* для логічного виразу. Оскільки ChocoPy – строго типізована мова, для відображення типів необхідний клас *ValueType.java*, а також його нащадки для підтримуваних типів.

Resolver.java (семантичний аналіз) реалізує інтерфейси *Stmt.Visitor.java* і *Expr.Visitor.java* для обходу усіх вузлів дерева AST для перевірки типів. *Resolver.java* використовує *ClassInfo.java* для відображення усіх класів коду ChocoPy.

Interpreter.java (виконання коду) так само реалізує інтерфейси *Stmt.Visitor.java* і *Expr.Visitor.java* з метою виконання коду, обходячи кожен з вузлів AST. *Interpretator.java* зберігає змінні кожної області видимості у об’єктах класу *Environment.java*. Створені об’єкти функцій та класів зберігаються у об’єктах класів *ChocoPyFunction.java* та *ChocoPyClass.java* відповідно. Вони реалізують інтерфейс *ChocoPyCallable.java*. Об’єкти ChocoPy класів зберігаються, використовуючи клас *ChocoPyInstance.java*, атрибути – за допомогою класу *ChocoPyAttribute.java*. Для обробки помилок часу виконання використовується клас *RuntimeError.java*.

5.3 Лексичний аналіз

Код лексичного аналізу складається з циклу *while*, у якому в операторі *switch* перевіряється кожен символ вихідного коду і для нього створюється токен. Вкінці завжди додається *EOF* токен, означаючи кінець файлу – це знадобиться у наступній фазі фронт–енда компіляції.

В залежності від токену аналізується один, два, або більше символів. Блок–схему роботи циклу лексичного аналізу можна побачити на кресленику ІСз91.020БАК.004 Д2. Наприклад, ліва кругла дужка (створює токен типу *LEFT_PAREN*. Для токену *ARROW* потрібно два символи – та >. Коментар починається з символу #, і всі символи до кінця рядку, або кінця файлу, в залежності, що зустрінеться раніше, ігноруються. Тобто для коментаря токен не створюється.

Рядок починається з символу “ та продовжується до наступного такого ж символу, за умови що він не екранований (список керуючих послідовностей ChocoPy наведено у підпункті 2.3.4.1). При зчитуванні лексеми рядку перевіряються допустимі символи та відбувається заміна керуючих послідовностей. Згенерований токен рядку, в залежності від змісту, матиме один з двох рядкових типів *IDSTRING* або *STRING*.

Числа починаються з першої десяткової цифри і скануються до останнього такого символу у послідовності. Числовий рядок перетворюється на ціле число. Воно проходить перевірку на входження у допустимі межі, а також відсутність нулів на початку лексеми.

Ідентифікатор зчитується за допомогою жадібного алгоритму, використовуючи максимальну кількість абетко–цифрових символів. Перевіряється, чи лексема не є зарезервованим ключовим словом Python, яке не підтримується в ChocoPy, але використання якого в якості імені змінної може призвести до помилки в інтерпретаторі Python. Адже код ChocoPy є валідним кодом Python. Тип токена ідентифікатора є або одним з ключових слів (наприклад, *AND*) або *ID*.

Після кожного токену виконується перевірка на необхідність додати або видалити відступ – токени *INDENT* і *DEDENT* відповідно.

5.4 Синтаксичний аналіз

Синтаксичний аналізатор отримує на вхід від лексичного аналізатора список

токенів і будує з них абстрактне синтаксичне дерево. На кресленику ІСз91.020БАК.004 Д3 зображена блок–схема порядку роботи алгоритму синтаксичного аналізу. На верхньому рівні в залежності від поточного токену може бути оголошення змінної, функції, класу або оператор.

Змінна складається з імені та типу, розділених двокрапкою, та обов'язкового початкового примітивного значення. Об'єкт класу або списку можна ініціалізувати тільки значенням *None*.

Функція має ім'я, необов'язкові параметри, що є оголошеннями змінних без початкового значення, необов'язкове значення повернення функції (якщо відсутнє, то буде *None*) та тіло. Тіло функції складається з необов'язкових оголошень *global*, *nonlocal*, локальних змінних, внутрішніх функцій, а також хоча б одного оператора.

Клас складається з власного імені, імені суперкласу та тіла. Тіло повинне мати у своєму складі хоча б один атрибут чи метод, або оператор *pass* – інструкція, яка наказує нічого не робити.

Оператор може бути складеним (*if–elif–else*, цикли *while* та *for*) або простим (*return*, *pass*, присвоєння та вираз). Для того, щоб врахувати пріоритет операторів, вираз аналізує їх у порядку відповідно до таблиці 2.3. Останнє, що аналізується, це виклик функції або методу, атрибуту об'єкта, чи індексу списку.

Оскільки у ChocoPy є множинне присвоєння (з англ. multiple assignment), необхідно в циклі перебрати всі вирази, розділені символом =, та зберегти їх у одному операторі, щоб потім опрацювати у семантичному аналізаторі.

Тип змінної «список» може приймати будь–яку глибину (наприклад, [[*int*]] – список списків цілочисельних значень). При синтаксичному аналізі можуть виникнути помилки, про які необхідно повідомити користувача. Проте перед зупинкою роботи програми потрібно знайти і показати якомога більше помилок.

Після кожного оператора виконуватиметься перевірка у методі *synchronize*. Якщо буде знайдена хоча б одна помилка, то пошук наступного оператора зупиниться. Натомість керування перейде до головного класу інтерпретатора, який зупинить хід програми і виведе повідомлення про помилки у стандартний потік для виведення помилок.

5.5 Семантичний аналіз

Після того як AST побудовано на попередній фазі, потрібно обійти кожен його вузол для семантичного аналізу. Для цієї мети зручно використати шаблон проєктування «Відвідувач». Він був створений, щоб можна було додавати нові операції, не змінюючи існуючих операцій та елементів, над якими виконуються ці операції [25].

На кресленику ІСз91.020БАК.004 Д4 можна побачити велику кількість класів нащадків *Expr.java* і *Stmt.java* у взаємозв'язку з класом *Resolver.java*. Проте, при необхідності додати новий підклас (наприклад *Stmt.Assert.java*) або нову дію (*Intepreter.java* для виконання коду), клас *Resolver.java* та підкласи абстрактних класів *Expr.java* і *Stmt.java* мають залишитись без змін.

Через статичну типізацію семантичний аналіз – єдина фаза компіляції, в якій доводиться обходити AST двічі – для аналізу оголошень та операторів. Спершу аналізуються тільки оголошення класів, функцій, змінних, оператори *global* і *nonlocal*. Сигнатури функцій і методів та змінних і атрибутів зберігаються у поточній області видимості. Інформація про класи зберігається у глобальній області видимості.

Під час другого проходу ця інформація використовується для пошуку імен та типів змінних, пошуку атрибутів і методів класу, що використовуються у виразах, на предмет їх існування або визначення дублікатів. Також це потрібно для виведення підтипів виразів для розуміння, чи результат виконання інструкції може бути присвоєно змінній.

В деяких виразах, коли існує два і більше непов'язаних типи, потрібно виконати операцію злиття типів, для подальшого аналізу можливості присвоєння або використанню в іншій операції. До них відносяться вирази: умовний (тернарний), списку і додавання списків.

Ще проводиться перевірка відповідності операндів операціям над ними, спів–ставлення кількості параметрів функцій і аргументів, можливості перевизначення методу, правильність контексту використання операторів (наприклад, *return* може

бути тільки в тілі функції). Для кожної з цілей виразу присвоєння створюється окремий вузол виразу для подальшої перевірки.

5.6 Інтерпретатор

На останньому етапі роботи виконується код, обходячи AST за допомогою шаблону проєктування «Інтерпретатор» [25]. Це найпростіший, але, в той же час, найповільніший спосіб виконання програмного коду. Акцент зроблено на коректності роботи інтерпретатору, а не його швидкості. Так само, як і при семантичному аналізі, використовується шаблон проєктування «Відвідувач», що можна побачити на рисунку Б.1 у додатку Б.

Для кожного оточення створюється асоціативний масив змінних цього оточення, також є посилання на зовнішнє оточення, щоб можна було дістатись до оточень верхніх рівнів аж до глобального оточення.

У глобальному оточенні попередньо визначено 3 бібліотечні функції (*print*, *input*, *len*) та 4 типи (*object*, *int*, *bool*, *str*).

У виразах, що потребують двох операндів, спочатку виконується лівий опе–ранд, потім правий. Логічний вираз виконує рівно стільки операцій, скільки потрібно: *OR* не виконує правий операнд, якщо лівий повертає *True*, так само *AND* не виконує правий операнд, якщо лівий повертає *False*.

Для виконання коду функцій використовується клас *ChocoPyFunction.java*, а для методів класи *ChocoPyClass.java* і *ChocoPyInstance.java* для конструкторів і об’єктів відповідно. Зі спільного функціоналу у них є метод *bind*, який зв’язує виклик функції з оточенням, яке було на момент визначення функції.

Множинне присвоєння виконує вираз зі значенням рівно один раз, після чого по черзі присвоює значення цільовим змінним зліва направо.

Для виведення помилки часу виконання використовується клас *RuntimeError.java*, який повертає різні коди виходу в залежності від типу помилки: 1 – неправильний аргумент; 2 – ділення на нуль; 3 – індекс виходить за межі значень рядка чи списку; 4 – операція, використовуючи *None* (додавання до *None*, виклик

атрибуту у об’єкта *None* тощо) [26].

Висновки до розділу

Отже, була спроєктована структура та ключові алгоритми інтерпретатора. Відповідно до цього розроблено програмне забезпечення. Використано наступні інструменти: Java 17, Apache Maven 3, Git, Intellij IDEA, ОС Linux.

Власна імплементація ChocoPy має 2 етапи компіляції: фронт–енд і бек–енд. Фронт–енд складається з фаз лексичного, синтаксичного та семантичного аналізу. Бек–енд виконує роль інтерпретатора.

Консольний додаток отримує на вхід текстовий файл з вихідним кодом ChocoPy, який читається один раз. При неправильній кількості аргументів або некоректному розширенні файлу показується повідомлення з помилкою. Для виконання інтерпретатора необхідно виконати консольну команду з передачею текстового файлу з вихідним кодом у якості аргументу.

Основний потік роботи програми можна описати наступним чином. Клас *Scanner.java*, який виконує лексичний аналіз, сканує символи, створює токени з лексем та передає список токенів до синтаксичного аналізатору *Parser.java*. Він будує з них абстрактне синтаксичне дерево відповідно до правил формальної граматики. Після отримання AST клас *Resolver.java* виконує семантичний аналіз, читаючи визначення змінних, функцій та класів перед перевіркою на типи решти виразів та операторів. Він додає до вузлів абстрактного синтаксичного дерева інформацію про виведені типи виразів, яка використовується вузлами вищих рівнів. Інтерпретатор проходиться по усьому AST і виконує код, використовуючи однойменний шаблон проєктування.

Результат роботи програми виводиться в стандартний потік виводу, якщо код викликає функцію *print*. За наявності помилок компіляції або часу виконання, виводяться відповідні повідомлення в стандартний потік для виведення помилок.

6 ПРОВЕДЕННЯ ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

6.1 Інструкція з використання

Програмне забезпечення не потребує інсталяції, не пише у лог–файли і не створює конфігураційні файли. ОС неважлива, необхідно лише, щоб у системі було встановлено середовище JRE версії 17+. Мінімальні системні вимоги додатку такі самі, як і для Java 17:

– диск: 124 MB для JRE (виконання), 245 MB для JDK (пакування);

– оперативна пам’ять: 128 MB;

– процесор: Pentium 2 266 MHz [27].

Java–архів *chocopy.jar* займає 108 KB, використовує до 5 MB оперативної пам’яті. Інтерпретатор має текстовий інтерфейс. Для його використання необхідно у терміналі виконати команду *java -jar chocopy.jar FILE.py*, де *FILE.py* – ім’я файлу з вихідним кодом програми, написаній на мові ChocoPy. Приклад неправильного виконання зображено на рисунку 6.1 (відсутність файлу з вихідним кодом).

Рисунок 6.1 – Неправильне виконання інтерпретатора ChocoPy

Приклад правильного виконання показано на рисунку 6.2.

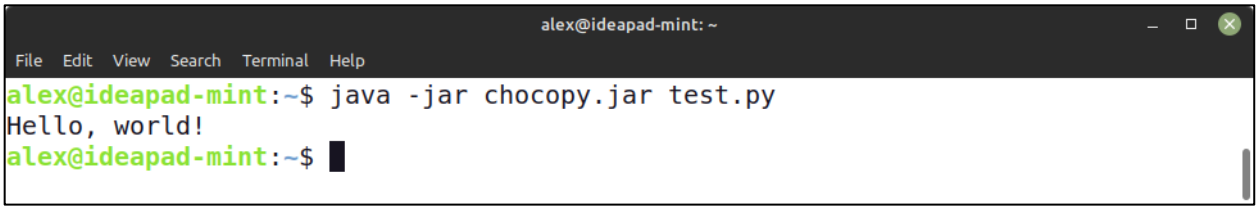

Рисунок 6.2 – Правильне виконання інтерпретатора ChocoPy

Код програмного забезпечення знаходиться в GitHub репозиторії за посиланням у додатку В. Для того, щоб зібрати Java архів з вихідного коду, необхідно виконати команду $mvn\ clean\ package$. Для цього має бути встановлені JDK 17+ та Apache Maven 3.

6.2 Автоматизоване тестування

Для перевірки коректності реалізації усіх етапів компіляції необхідні автоматизовані тести з великою кількістю тестових сценаріїв для покриття якомога більшої кількості випадків роботи. Використаємо фреймворк JUnit 5. Дана бібліотека створена для модульного тестування, та дозволяє тестувати окремі методи (модулі) написаного програмного забезпечення.

Оскільки ChocoPy використовувалась для викладання курсу «CS 164. Мови програмування та компілятори» в Каліфорнійському університеті в Берклі, для його проведення була створена платформа, що містить тестові дані для перевірки коректності імплементації компілятора.

Три GitHub репозиторії курсу мають близько 250 скриптів з тестовим вихідним кодом, як коректним, так і не коректним для перевірки правильності повідомлень про помилки [28]. Мною були додані додаткові тести для токенів, рядків, відступів та крайніх випадків (наприклад пустий вхідний файл тощо). Таким чином проєкт наразі містить 272 тести.

Всього є 4 тестові класи *ScannerTest.java*, *ParserTest.java*, *ResolverTest.java* та *InterpreterTest.java* для кожної з реалізованих фаз інтерпретатора відповідно. Всі тестові класи мають параметричні тести зі спільною логікою. Параметрами виступають тестові файли з кодом ChocoPy. Логіка тесту полягає в читанні рядку коду, передачу його в один з методів відповідного класу інтерпретатора, читанні файлу з очікуваним результатом, та порівнянням з фактичним результатом роботи метода, який тестується. На рисунку 6.3 зображено приклад тестових даних для перевірки умовного (тернарного) виразу.

Тести *ScannerTest.java* перевіряють лексичний аналізатор і для коректного

коду порівнюють список згенерованих токенів, приклад яких можна побачити на рисунку 6.4.

```
3 if 1 > 2 else 4
```

Рисунок 6.3 – Тестовий файл *resolver/expr_if.py*

```
NUMBER 3
IF
NUMBER 1
GREATER >
NUMBER 2
ELSE
NUMBER 4
EOF
```

Рисунок 6.4 – Тестовий файл *scanner/expr_if.py.tokens*

Тести *ParserTest.java* перевіряють синтаксичний аналізатор, а саме правильність згенерованого абстрактного синтаксичного дерева. Тестові дані курсу CS 164 мають формат JSON, який строго прив'язаний до класів фреймворку та бібліотеки для генерування JSON об'єктів. Рисунок Г.1 у додатку Г містить приклад файлу з очікуваним результатом у форматі JSON.

Поміняємо формат на YAML – зручніший для читання формат розмітки даних. На лівій частині (а) рисунку 6.5 зображені ті самі дані, але без інформації про місцезнаходження кожного виразу або оператору, та структурою, що відображає класи власної реалізації.

Тести *ResolverTest.java* перевіряють правильність виведення типів даних виразів у вузлах AST. Цей самий тестовий сценарій, тільки з інформацією про виведений тип даних зображено на правій частині (б) рисунку 6.5. На ньому можна побачити рядки *inferredType:int* та *inferredType:bool*, які вказують на те, що вирази мають цілочисельний та логічний типи даних відповідно.

Для виведення інформації про AST у форматі YAML використаємо шаблон

проєктування «Відвідувач» так само, як у фазах семантичного аналізу та інтерпретації. Відповідну UML–діаграму класів можна побачити на рисунку Д.1 у додатку Д.

```yaml
statements:
  - statement:
      class: Stmt.Expression
      expr:
        class: Expr.Ternary
        onTrue:
          class: Expr.Literal
          value: 3
        condition:
          class: Expr.Binary
          left:
            class: Expr.Literal
            value: 1
          operator: ">"
          right:
            class: Expr.Literal
            value: 2
        onFalse:
          class: Expr.Literal
          value: 4
```

а

```yaml
statements:
  - statement:
      class: Stmt.Expression
      expr:
        class: Expr.Ternary
        inferredType: int
        onTrue:
          class: Expr.Literal
          inferredType: int
          value: 3
        condition:
          class: Expr.Binary
          inferredType: bool
          left:
            class: Expr.Literal
            inferredType: int
            value: 1
          operator: ">"
          right:
            class: Expr.Literal
            inferredType: int
            value: 2
        onFalse:
          class: Expr.Literal
          inferredType: int
          value: 4
```

б

Рисунок 6.5 – Тестові файли *resolver/expr_if.py.ast* (а) та *resolver/expr_if.py.ast.typed* (б) у форматі YAML

Тести *InterprerTest.java* виконують вихідний код тестових файлів, які містять виклики вбудованої функції *print* і порівнюють правильність виведених даних.

Якщо вхідні тестові файли мають коректний код ChocoPy, то очікувані ре–

зультати залежно від тестового класу можуть бути такими:

– для *ScannerTest.java* – список токенів;

– для *ParserTest.java* – вузли AST у форматі YAML;

– для *ResolverTest.java* – типізовані вузли AST у форматі YAML (такі, що містять виведені типи для виразів);

– для *InterpreterTest.java* – виведення результатів роботи викликів методу *print*.

Якщо вихідний код тестового файлу містить помилку, то в результаті тесту мають бути показані повідомлення про помилки. Для тестового файлу, який зображено на рисунку 6.6, наведено приклад файлу з очікуваними помилками на рисунку 6.7.

```
1   x:int = 0
2   y:int = 0
3   z:bool = False
4
5   x = z = 1
6   x = y = None
7   x = y = []
8   x = a = None
9   x = a = []
10  x = y = True
```

Рисунок 6.6 – Тестовий файл *resolver/bad_assign_expr.py*

```
[line 5] TypeError: expected type 'bool', got type 'int'
[line 6] TypeError: expected type 'int', got type '<None>'
[line 7] TypeError: expected type 'int', got type '<Empty>'
[line 8] TypeError: expected type 'int', got type '<None>'
[line 8] NameError: name 'a' is not defined in current scope
[line 9] TypeError: expected type 'int', got type '<Empty>'
[line 9] NameError: name 'a' is not defined in current scope
[line 10] TypeError: expected type 'int', got type 'bool'
```

Рисунок 6.7 – Тестовий файл *resolver/bad_assign_expr.py.errors*

Для виконання тестів у проєкті необхідно запустити команду *mvn test*, результат роботи якої можна побачити на рисунку 6.8. Для наочності, тести були виконані у середовищі Intellij IDEA, яке дозволяє гнучко проводити тестування для окремих методів, класів і пакетів, використовуючи графічний інтерфейс.

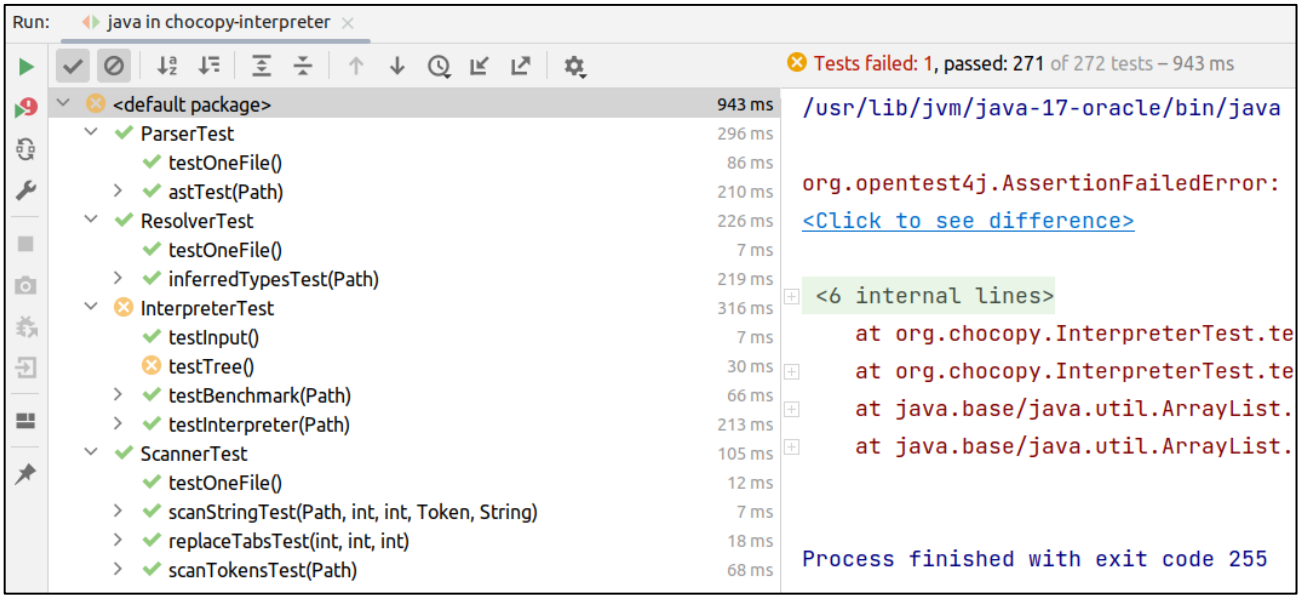

Рисунок 6.8 – Результат виконання автоматичних модульних тестів

Хибним виявився тільки один тест. При значенні $n = 1$ у вихідному коді скрипту *interpreter/benchmark/tree.py* тест проходить, отже його код є коректним. Проте при *n = 2* ChocoPy зупиняє роботу з помилкою *java.lang.StackOverflowError*. 12 операторів вихідного коду створюють AST з 224 вузлами. Середній розмір об’єкту одного виразу чи оператора займає 36 байт, що було виміряно за допомогою Java–агенту, скориставшись Instrumentation API.

Проте через велику кількість вузлів AST при виконанні коду відбувається багато викликів функцій, в кожну з яких передаються об’єкти виразів і операторів, що призводить до переповнення стеку викликів. За замовчуванням у Java розмір стеку дорівнює 1 MB. Але збільшення стеку до максимального значення в 1 GB не вирішує проблему. При збільшенні ітерацій циклу *while* до тестового значення *n = 100* так само збільшується кількість викликів функцій при виконанні коду в AST, що знову ж таки призводить до переповнення стеку викликів.

6.3 Оцінка продуктивності інтерпретатора

Тести продуктивності або бенчмарк (з англ. benchmark) тести використовуються з метою оцінки відносної продуктивності програми, а саме її швидкості, об'єму використаної пам'яті, кількості виконаних операцій за одиницю часу тощо. Застосовані тести перевіряють арифметичні операції, алгоритми пошуку, операції над списками та стандартні бібліотечні функції. Оскільки код ChocoPy можна виконувати в інтерпретаторі Python версії 3.6 і вище, було проведено порівняння швидкості роботи власної реалізації з інтерпретатором версії Python 3.10.6 за допомогою консольної утиліти *time*. У таблиці 6.1 наведено результати вимірювань часу виконання бенчмарк скриптів.

Таблиця 6.1 – Порівняння власного інтерпретатору з Python

	Час роботи, сек		Відношення часу роботи chocopy.jar до Python 3.10.6
	chocopy.jar	Python 3.10.6	
tree.py (n=1)	0.135	0.018	7.5
exp.py	0.188	0.013	14.5
prime.py	0.138	0.016	8.6
sieve.py	0.149	0.013	11.5
stdlib.py	0.140	0.018	7.8

З таблиці можна побачити, що в середньому час виконання скриптів в інтерпретаторі Python у 10 разів швидше за власну реалізацію на Java.

6.4 Напрямки подальшого вдосконалення

Можливі напрямки покращення проєкту можна розділити на 2 частини:

– інтерфейс користувача;

– швидкість роботи.

Повідомлення про помилки були максимально запозичені з відповідних по–

милок інтерпретатору Python. Проте їх можна зробити більше деталізованими. Наприклад, замість *syntax error* написати, якої саме синтаксичної помилки припустився користувач. Також можна додати більше інформації про місцезнаходження помилки. Наразі інтерпретатор показує тільки рядок, у якому сталася помилка. Проте можна показати номер символу в рядку, де сталася помилка. Вищий рівень – показати трасу стеку викликів (з англ. stack trace).

Щодо підвищення швидкості роботи існує кілька шляхів:

– концептуальний підхід до реалізації інтерпретатора;

– використання низькорівневої мови програмування.

Тест, який не пройшов, підводить нас до питання про ефективність стратегії виконання коду. Сучасні інтерпретовані мови програмування перетворюють вихідний код на проміжне представлення, об'єктний код, або байт–код, який одразу виконують. Тобто послідовність інструкцій значно швидше виконуватиметься ніж обхід AST.

Одночасно з використанням байт–коду можна використати мову програмування нижчого рівня, наприклад C. Справа в тому, що Java полегшує розробку, беручи на себе керування пам'яттю. Проте є зворотній бік – час виконання збирача сміття непередбачуваний, що може вплинути на продуктивність. До того ж для роботи JVM необхідна власна оперативна пам'ять (під час тестування використано 249 MB).

Висновки до розділу

Було розглянуто як використовувати та тестувати розроблений інтерпретатор. Програмне забезпечення не потребує інсталяції, лише наявності середовища JRE 17+. Для отримання виконуваного Java архіву потрібно спакувати проєкт за допомогою інструменту Maven.

Для тестування застосовано бібліотеку Junit5 і параметричні тести. Тестові файли взято з репозиторіїв курсу CS 164. JSON формат файлів очікуваних резуль–татів було замінено на YAML формат для полегшення використання і сприйняття.

З 272 тестів не пройшов тільки один бенчмарк тест. Оскільки розроблений інтерпретатор ChocoPy обходить вузли AST при виконанні коду, це спричиняє велику кількість викликів функцій, що призводить до помилки переповнення стеку викликів. Отже, можна зробити висновок, що виконана реалізація цілком відповідає формальній специфікації та проходить тести, а значить є коректною.

Також було порівняно час виконання усіх бенчмарк скриптів з інтерпретатором Python. Власна реалізація виявилася в середньому у 10 разів повільніше. Це знову ж таки вказує на те, що обраний підхід до виконання коду є найменш ефективним як за пам’яттю, так і за часом.

Подальші покращення можуть стосуватись як поліпшення точності повідомлень про помилки, так і зміни підходу щодо імплементації інтерпретатора. Для пришвидшення потрібно використати байт–код та власну реалізацію віртуальної машини на мові програмування нижчого рівня C.

Отже, розробляючи власний інтерпретатор, як і будь–яке інше програмне забезпечення, можна зрозуміти, чому для успішних продуктів було обрано саме ті чи інші інструменти роботи та підходи в реалізації. Тобто, щоб опанувати якусь програму, одним з варіантів вивчення може бути написання власної імплементації.

ВИСНОВКИ

В результаті цього дипломного проєкту було написано інтерпретатор мови програмування ChocoPy. Задля досягнення мети даного проєкту були розв'язані такі завдання, як: лексичний, синтаксичний, семантичний аналіз і виконання коду AST. Задля розв'язання поставлених завдань були розроблені алгоритми ключових етапів роботи: сканування токенів, розрахунок відступу рядка, парсингу виразу множинного присвоєння, подвійного проходу AST для оцінки типів тощо.

В розділі опису предметної області було наведено теоретичні засади мов програмування. Було досліджено роботу компіляторів, їх складові частини та підходи до їхньої побудови.

У розділі опису мови ChocoPy надано вичерпну інформацію щодо лексичної структури, синтаксису, граматики та правил перевірки типів.

Розділ математичного забезпечення розкриває формальні описи застосованих абстрактних моделей та алгоритмів.

Четвертий розділ порівнює основні технології розробки та визначає, яка саме буде використовуватись. Було обрано оптимальну на даний момент мову програмування, враховуючи обмеження наявних ресурсів.

Опис проєктування і розробки програмного забезпечення наведено в четвертому розділі, що включає обґрунтування використаних засобів розробки, архітектуру інтерпретатора, огляд основних складових системи.

У розділі тестування програмного забезпечення наведено інструкцію з користування, результати автоматизованого тестування і оцінка продуктивності у порівнянні з інтерпретатором Python. Також зазначені напрямки подальшого вдосконалення програмного продукту.

Додатки містять розширену інформацію щодо правил перевірки типів ChocoPy, приклад оригінального тестового файлу у форматі JSON, а також ілюстрування використання шаблону проєктування «Відвідувач».

Графічні матеріали зображають блок–схеми роботи алгоритмів лексичного і синтаксичного аналізу, а також UML–діаграми класів семантичного аналізу та ін–

терпретатора ChocoPy.

Розроблене програмне забезпечення повністю відповідає висунутим вимогам у завданні на дипломний проєкт. Надано посилання на GitHub репозиторій з вихідним кодом програми.

Виконано кросплатформний портативний консольний додаток на технології Java у вигляді JAR–файлу без використання фреймворку курсу CS 164. Використання системних ресурсів не перевищує висунутих обмежень. Віртуальна машина Java використала 249 MB, а додаток 5 MB оперативної пам’яті під час тестування. Встановлене середовище JRE 17 займає 178 MB на диску, а розмір файлу *chocopy.jar* – 108 KB.

Відповідність специфікації перевірена проходженням автоматизованих модульних тестів, тестові дані для яких взяті з GitHub репозиторію курсу CS 164. Не пройшов тільки один бенчмарк тест через помилку переповнення стеку викликів. В порівнянні з інтерпретатором Python власна реалізація працює в середньому в 10 разів повільніше.

Дана розробка може слугувати основою для створення інтерпретаторів та компіляторів для інших мов програмування.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Rohan Padhye, Koushik Sen, and Paul N. Hilfinger. 2019. ChocoPy: a programming language for compilers courses. In Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH–E (SPLASH–E 2019). Association for Computing Machinery, New York, NY, USA, pp. 41–45. doi:10.1145/3358711.3361627.

2. Course: CS 164. Programming Languages and Compilers at UC Berkeley, Fall 2022. URL: https://www2.eecs.berkeley.edu/Courses/CS164 (Accessed 11.04.2023).

3. Aaby, Anthony (2004). Introduction to Programming Languages. URL: https://uilis.usk.ac.id/oer/files/original/be28b69d281a44f8b2b7aa8af26e3768.pdf (Accessed 12.04.2023), p. 234.

4. Aarne Ranta (2012). Implementing Programming Languages. An Introduction to Compilers and Interpreters. College Publications. ISBN 978–1–84890–064–6, pp. 16-18.

5. Chomsky, Noam; Lightfoot, David W. (2002). Syntactic Structures. Walter de Gruyter. ISBN 978–3–11–017279–9, p. 117.

6. Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman (2007). Compilers: Principles, Techniques, and Tools (2nd Edition). Addison–Wesley. ISBN 0-321–48681–1, p. 1038.

7. Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman (2007). Compilers: Principles, Techniques, and Tools (2nd Edition). Addison–Wesley. ISBN 0-321–48681–1, pp. 5–6, 109–189.

8. Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman (2007). Compilers: Principles, Techniques, and Tools (2nd Edition). Addison–Wesley. ISBN 0-321–48681–1, pp. 8, 191–300.

9. Blindell, Gabriel Hjort (3 June 2016). Instruction selection: principles, methods, and applications. Springer. ISBN 978–3319340197, p. 177.

10. Cooper, Keith Daniel; Torczon, Linda (2012). Engineering a compiler (2nd ed.). Elsevier. ISBN 978–0120884780, p. 8.

11. J. E. Smith and Ravi Nair, «The architecture of virtual machines» in Computer, vol. 38, no. 5, pp. 32–38, May 2005, doi:10.1109/MC.2005.173.

12. ChocoPy v2.2: Language Manual and Reference (November 23, 2019). URL: https://chocopy.org/chocopy_language_reference.pdf (Accessed 18.04.2023).

13. Python 3 Full Grammar specification. Documentation. URL: https://docs.python.org/3/reference/grammar.html (Accessed 19.04.2023).

14. N. Chomsky, «Three models for the description of language» in IRE Transactions on Information Theory, vol. 2, no. 3, pp. 113–124, September 1956, doi:10.1109/TIT.1956.1056813.

15. Wang, J. (2019). Formal Methods in Computer Science (1st ed.). Chapman and Hall/CRC. doi:10.1201/9780429184185, p. 34.

16. Hopcroft, John E.; Motwani, Rajeev; Ullman, Jeffrey D. (2013). Introduction to Automata Theory, Languages, and Computation (3rd ed.). Pearson. ISBN 978-1292039053, p. 496.

17. Burge, W.H. (1975). Recursive Programming Techniques. Addison–Wesley. ISBN 0–201–14450–6, p. 277.

18. Watson, D., (2017). A practical approach to compiler construction. Springer. ISBN 978–3–319–52789–5, p. 254.

19. The current C programming language ISO standard (C17). URL: https://www.open-std.org/jtc1/sc22/wg14/ (Accessed 02.05.2023)

20. Trail: The Reflection API. Java Tutorials. URL: https://docs.oracle.com/javase/tutorial/reflect/index.html (Accessed 03.05.2023)

21. PEP 20 – The Zen of Python. URL: https://peps.python.org/pep-0020 (Accessed 04.05.2023).

22. Python ast module – Abstract Syntax Trees. Documentation. URL: https://docs.python.org/3/library/ast.html (Accessed 05.05.2023)

23. Oracle Java SE Support Roadmap (Updated March 22, 2022). URL: https://www.oracle.com/java/technologies/java-se-support-roadmap.html (Accessed 23.05.2023).

24. ChocoPy: A Programming Language for Compilers Courses. URL: https://chocopy.org (Accessed 24.05.2023).

25. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1994). Design Patterns: Elements of Reusable Object–Oriented Software. Addison–Wesley. ISBN 0-201–63361–2, p. 416.

26. ChocoPy v2.2: RISC–V Implementation Guide (October 31, 2019). URL: https://chocopy.org/chocopy_implementation_guide.pdf (Accessed 25.05.2023).

27. Java Platform, Standard Edition Installation Guide. URL: https://docs.oracle.com/javase/8/docs/technotes/guides/install/toc.html (Accessed 30.05.2023).

28. CS 164: Programming Assignment 1 test data by cs164berkeley. URL: https://github.com/cs164berkeley/pa1-chocopy-parser/tree/master/src/test/data (Accessed 31.05.2023).

ДОДАТОК А

Правила перевірки типів ChocoPy

(обов'язкове)

Формули в додатку А взято з документації мови ChocoPy [А.1]. Ці правила є продовженням пункту 2.5.2 пояснювальної записки.

Визначення та присвоювання змінних. Правило присвоювання, як і інші, використовує відношення $\leq_a$. (пункт 2.2.4 пояснювальної записки). Правило говорить, що присвоюваний вираз e_1 повинен мати тип T_1, який є сумісний за присвоюванням з типом T ідентифікатора id у середовищі типів:

$$\frac{\begin{array}{l} O(id) = T \\ O, M, C, R \vdash e_1 : T_1 \\ T_1 \leq_a T \end{array}}{O, M, C, R \vdash id = e_1} \quad \text{(А.1)}$$

Визначення змінних підпорядковуються аналогічному правилу:

$$\frac{\begin{array}{l} O(id) = T \\ O, M, C, R \vdash e_1 : T_1 \\ T_1 \leq_a T \end{array}}{O, M, C, R \vdash id\colon T = e_1} \quad \text{(А.2)}$$

Двокрапка, що використовується під лінією у попередньому правилі, є двокрапкою у синтаксисі для анотацій типів.

Списки операторів та визначень. Ці типи перевіряють, чи всі визначення компонентів та оператори проходять перевірку типів.

Правило перевірки типів для оператору визначення списку:

$$\frac{\begin{array}{l} O, M, C, R \vdash s_1 \\ O, M, C, R \vdash s_2 \\ \vdots \\ O, M, C, R \vdash s_n \\ n \geq 1 \end{array}}{O, M, C, R \vdash s_1 \text{ NEWLINE } s_2 \text{ NEWLINE } \ldots \ s_n \text{ NEWLINE}} \quad \text{(A.3)}$$

Правило перевірки типів оператора $pass$:

$$\frac{}{O, M, C, R \vdash \texttt{pass}} \quad \text{(A.4)}$$

Правило перевірки типів операторів–виразів:

$$\frac{O, M, C, R \vdash e : T}{O, M, C, R \vdash e} \quad \text{(A.5)}$$

Правило перевірки типу для літералів логічного типу $False$:

$$\frac{}{O, M, C, R \vdash \texttt{False} : bool} \quad \text{(A.6)}$$

Правило перевірки типу для літералів логічного типу $True$:

$$\frac{}{O, M, C, R \vdash \texttt{True} : bool} \quad \text{(A.7)}$$

Правило перевірки типу для літералів цілочисельного типу:

$$\frac{i \text{ is an integer literal}}{O, M, C, R \vdash i : int} \quad \text{(A.8)}$$

Правило перевірки типу для літералів рядкового типу:

$$\frac{s \text{ is a string literal}}{O, M, C, R \vdash s : str} \tag{A.9}$$

Літералу $None$ присвоюється тип $< None >$:

$$\frac{}{O, M, C, R \vdash \texttt{None} : \texttt{<None>}} \tag{A.10}$$

Арифметичні та числові оператори відношення. Правило перевірки типів оператору унарного віднімання:

$$\frac{O, M, C, R \vdash e : int}{O, M, C, R \vdash \texttt{-}\, e : int} \tag{A.11}$$

Правило перевірки типів арифметичних операторів:

$$\frac{\begin{array}{c} O, M, C, R \vdash e_1 : int \\ O, M, C, R \vdash e_2 : int \\ op \in \{+, -, *, //, \%\} \end{array}}{O, M, C, R \vdash e_1 \, op \; e_2 : int} \tag{A.12}$$

Правило перевірки типів операторів відношення:

$$\frac{\begin{array}{c} O, M, C, R \vdash e_1 : int \\ O, M, C, R \vdash e_2 : int \\ \bowtie \in \{\texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}, \texttt{==}, \texttt{!=}\} \end{array}}{O, M, C, R \vdash e_1 \bowtie e_2 : bool} \tag{A.13}$$

Логічні оператори. Правило перевірки типів логічних операторів порівняння:

$$\frac{O, M, C, R \vdash e_1 : bool \quad O, M, C, R \vdash e_2 : bool \quad \bowtie \in \{\texttt{==}, \texttt{!=}\}}{O, M, C, R \vdash e_1 \bowtie e_2 : bool} \tag{A.14}$$

Правило перевірки типів логічного оператора AND:

$$\frac{O, M, C, R \vdash e_1 : bool \quad O, M, C, R \vdash e_2 : bool}{O, M, C, R \vdash e_1 \texttt{ and } e_2 : bool} \tag{A.15}$$

Правило перевірки типів логічного оператора OR:

$$\frac{O, M, C, R \vdash e_1 : bool \quad O, M, C, R \vdash e_2 : bool}{O, M, C, R \vdash e_1 \texttt{ or } e_2 : bool} \tag{A.16}$$

Правило перевірки типів логічного оператора NOT:

$$\frac{O, M, C, R \vdash e : bool}{O, M, C, R \vdash \texttt{not } e : bool} \tag{A.17}$$

Правило перевірки типів умовних (тернарних) виразів:

$$\frac{O, M, C, R \vdash e_0 : bool \quad O, M, C, R \vdash e_1 : T_1 \quad O, M, C, R \vdash e_2 : T_2}{O, M, C, R \vdash e_1 \texttt{ if } e_0 \texttt{ else } e_2 : T_1 \sqcup T_2)} \tag{A.18}$$

Рядкові операції. Правило перевірки типів рядкової операції порівняння:

$$\frac{O, M, C, R \vdash e_1 : str \quad O, M, C, R \vdash e_2 : str \quad \bowtie \in \{\texttt{==}, \texttt{!=}\}}{O, M, C, R \vdash e_1 \bowtie e_2 : bool} \tag{A.19}$$

Правило перевірки типів рядкової операції конкатенації:

$$\frac{O, M, C, R \vdash e_1 : str \quad O, M, C, R \vdash e_2 : str}{O, M, C, R \vdash e_1 \texttt{ + } e_2 : str} \tag{A.20}$$

Правило перевірки типів рядкової операції вибору символу за індексом:

$$\frac{O, M, C, R \vdash e_1 : str \quad O, M, C, R \vdash e_2 : int}{O, M, C, R \vdash e_1[e_2] : str} \tag{A.21}$$

Правило перевірки типів оператору ‘is’:

$$\frac{\begin{array}{l} O, M, C, R \vdash e_1 : T_1 \\ O, M, C, R \vdash e_2 : T_2 \\ T_1, T_2 \text{ are not one of } int, str, bool \end{array}}{O, M, C, R \vdash e_1 \texttt{ is } e_2 : bool} \tag{A.22}$$

Конструювання об'єктів. Якщо T – ім'я класу, то вирази конструювання об'єктів цього класу можуть бути записані наступним чином:

$$\frac{T \text{ is a class}}{O, M, C, R \vdash T() : T} \tag{A.23}$$

Зображення списків. Правило перевірки типів зображення списків:

$$\frac{\begin{array}{l} n \geq 1 \\ O, M, C, R \vdash e_1 : T_1 \\ O, M, C, R \vdash e_2 : T_2 \\ \vdots \\ O, M, C, R \vdash e_n : T_n \\ T = T_1 \sqcup T_2 \sqcup \ldots \sqcup T_n \end{array}}{O, M, C, R \vdash [e_1, e_2, \ldots, e_n] : [T]} \tag{A.24}$$

Пустий список є особливим випадком:

$$\frac{}{O, M, C, R \vdash \texttt{[]} : \texttt{<Empty>}} \tag{A.25}$$

Операції зі списками. Правило перевірки типів операції конкатенації списків:

$$\frac{\begin{array}{l} O, M, C, R \vdash e_1 : [T_1] \\ O, M, C, R \vdash e_2 : [T_2] \\ T = T_1 \sqcup T_2 \end{array}}{O, M, C, R \vdash e_1 \texttt{ + } e_2 : [T]} \tag{A.26}$$

Правило перевірки типів операції вибору елемента списку за індексом:

$$\frac{\begin{array}{l} O, M, C, R \vdash e_1 : [T] \\ O, M, C, R \vdash e_2 : int \end{array}}{O, M, C, R \vdash e_1[e_2] : T} \tag{A.27}$$

Правило перевірки типів операції присвоєння елемента списку:

$$\frac{\begin{array}{l} O, M, C, R \vdash e_1 : [T_1] \\ O, M, C, R \vdash e_2 : int \\ O, M, C, R \vdash e_3 : T_3 \\ T_3 \leq_a T_1 \end{array}}{O, M, C, R \vdash e_1[e_2] = e_3} \tag{A.28}$$

Доступ до атрибутів, присвоєння та ініціалізація. Для доступу до атрибутів використовується оточення класу–члену M. Правило перевірки типів при читанні атрибуту класу:

$$\frac{\begin{array}{l} O, M, C, R \vdash e_0 : T_0 \\ M(T_0, id) = T \end{array}}{O, M, C, R \vdash e_0.id : T} \tag{A.29}$$

Правило перевірки типів при присвоєнні значення атрибуту класу:

$$\frac{\begin{array}{l} O, M, C, R \vdash e_0 : T_0 \\ O, M, C, R \vdash e_1 : T_1 \\ M(T_0, id) = T \\ T_1 \leq_a T \end{array}}{O, M, C, R \vdash e_0.id = e_1} \quad \text{(A.30)}$$

Правило перевірки типів при ініціалізації атрибуту класу:

$$\frac{\begin{array}{l} M(C, id) = T \\ O, M, C, R \vdash e_1 : T_1 \\ T_1 \leq_a T \end{array}}{O, M, C, R \vdash id\colon T = e_1} \quad \text{(A.31)}$$

Множинні присвоєння. Множинне присвоєння перевіряється на тип шляхом декомпозиції на окремі одиночні присвоєння, як показано нижче:

$$\frac{\begin{array}{l} n > 1 \\ O, M, C, R \vdash e_0 : T_0 \\ O, M, C, R \vdash e_1 = e_0 \\ \vdots \\ O, M, C, R \vdash e_n = e_0 \\ T_0 \neq \texttt{[<None>]} \end{array}}{O, M, C, R \vdash e_1 = e_2 = \cdots = e_n = e_0} \quad \text{(A.32)}$$

Обмеження $T_0 \neq [< None >]$ дозволяє уникнути делікатної проблеми безпеки типів. Небезпечно допускати існування двох різних представлень списку з різними типами елементів. Тип $[< None >]$ може виникати тільки з відображень списків. Оскільки значення такого представлення одразу ж споживається при присвоюванні до однієї змінної, параметру або операнду ($+$ для списків). Але множинне присвоювання відкриває можливості для подібних програм:

$x: [A] = None$

$y: [[int]] = None$

$x = y = [None]$ # попереду проблема

$x[0] = A()$

$print(y[0][0])$ # ???

Java, наприклад, сама потрапляє у цю прив'язку і тому потребує виключення $ArrayStoreException$ під час виконання. Щоб уникнути цього у ChocoPy, консервативно забороняється використовувати значення типу $[< None >]$ у множинному присвоєнні.

Існує ще один дуже делікатний момент, який ховається у випадку, коли e_0 має тип $< Empty >$ (тип порожнього списку). У цьому випадку, однак, не потрібне спеціальне правило, оскільки у ChocoPy немає методу $.append$, який би дозволяв додавати елементи до порожнього списку. Якби це було не так, можна було б отримати таку ситуацію:

$A: [int] = None$

$B: [str] = None$

$A = B = []$

$A.append(3)$

і згодом $B[0]$ повертало б значення 3, яке, безумовно, не є рядком.

Застосування функцій. Правило перевірки типів виклику функції:

$$\frac{\begin{array}{l} O, M, C, R \vdash e_1 : T_1'' \\ O, M, C, R \vdash e_2 : T_2'' \\ \vdots \\ O, M, C, R \vdash e_n : T_n'' \\ n \geq 0 \\ O(f) = \{T_1 \times \cdots \times T_n \to T_0;\ x_1, \ldots, x_n;\ v_1 : T_1', \ldots, v_m : T_m'\} \\ \forall 1 \leq i \leq n : T_i'' \leq_a T_i \end{array}}{O, M, C, R \vdash f(e_1, e_2, \ldots, e_n) : T_0} \quad \text{(A.33)}$$

Щоб перевірити на тип виклик функції, кожен з аргументів функції має спочатку бути перевірений на тип. Тип кожного аргументу має відповідати типу пов’язаного формального параметра функції. Виразу виклику присвоюється оголошений тип повернення значення функції.

Вирази диспетчеризації методу перевіряються на тип аналогічно. Ключова відмінність від виразу виклику функції полягає в тому, що цільовий метод визначається шляхом звернення до середовища методів/атрибутів з використанням типу виразу об'єкта–одержувача:

$$\frac{\begin{array}{l} O, M, C, R \vdash e_1 : T_1'' \\ O, M, C, R \vdash e_2 : T_2'' \\ \vdots \\ O, M, C, R \vdash e_n : T_n'' \\ n \geq 1 \\ M(T_1'', f) = \{T_1 \times \cdots \times T_n \to T_0;\ x_1, \ldots, x_n;\ v_1 : T_1', \ldots, v_m : T_m'\} \\ T_1'' \leq_a T_1 \\ \forall i.\ 2 \leq i \leq n : T_i'' \leq_a T_i \end{array}}{O, M, C, R \vdash e_1.f(e_2, \ldots, e_n) : T_0} \quad \text{(A.34)}$$

Оператори $return$. Саме тут вступає в дію середовище з типом повернення значення функції. Правило перевірки типів оператора $return$ зі значенням:

$$\frac{\begin{array}{l} O, M, C, R \vdash e : T \\ T \leq_a R \end{array}}{O, M, C, R \vdash \texttt{return}\ e} \quad \text{(A.35)}$$

Правило перевірки типів оператора $return$ без значення:

$$\frac{\texttt{<None>} \leq_a R}{O, M, C, R \vdash \texttt{return}} \quad \text{(A.36)}$$

Правило перевірки типів умовних операторів:

$$\frac{\begin{array}{l} O, M, C, R \vdash e_0 : bool \\ O, M, C, R \vdash b_0 \\ O, M, C, R \vdash e_1 : bool \\ O, M, C, R \vdash b_1 \\ \vdots \\ O, M, C, R \vdash e_n : bool \\ O, M, C, R \vdash b_n \\ n \geq 0 \\ O, M, C, R \vdash b_{n+1} \end{array}}{O, M, C, R \vdash \texttt{if}\ e_0 \texttt{:} b_0\ \texttt{elif}\ e_1 \texttt{:} b_1 \ldots\ \texttt{elif}\ e_n \texttt{:} b_n\ \texttt{else:} b_{n+1}} \quad \text{(A.37)}$$

Правило перевірки типів оператора *while*:

$$\frac{\begin{array}{l} O, M, C, R \vdash e : bool \\ O, M, C, R \vdash b \end{array}}{O, M, C, R \vdash \texttt{while}\ e \texttt{:} b} \quad \text{(A.38)}$$

Правило перевірки типів оператора for над рядками:

$$\frac{\begin{array}{l} O, M, C, R \vdash e : str \\ O(id) = T \\ str \leq_a T \\ O, M, C, R \vdash b \end{array}}{O, M, C, R \vdash \texttt{for}\ id\ \texttt{in}\ e \texttt{:} b} \quad \text{(A.39)}$$

Правило перевірки типів оператора for над списками:

$$\frac{\begin{array}{l} O, M, C, R \vdash e : [T_1] \\ O(id) = T \\ T_1 \leq_a T \\ O, M, C, R \vdash b \end{array}}{O, M, C, R \vdash \texttt{for}\ id\ \texttt{in}\ e \texttt{:} b} \quad \text{(A.40)}$$

Визначення функцій. Щоб описати визначення функції f, треба перевірити тіло функції f у середовищі де O розширено зв'язками для імен, явно оголошених у f:

$$\frac{\begin{array}{l} T = \begin{cases} T_0, & \text{if -> is present,} \\ \texttt{<None>}, & \textit{otherwise.} \end{cases} \\ O(f) = \{T_1 \times \cdots \times T_n \to T;\ x_1, \ldots, x_n;\ v_1 : T'_1, \ldots, v_m : T'_m\} \\ n \geq 0 \qquad m \geq 0 \\ O[T_1/x_1] \ldots [T_n/x_n][T'_1/v_1] \ldots [T'_m/v_m], M, C, T \vdash b \end{array}}{O, M, C, R \vdash \texttt{def}\ f(x_1 : T_1, \ldots, x_n : T_n)\ [\![\texttt{->}\ T_0]\!]^? : b} \tag{A.41}$$

Таке саме правило перевірки типів і для визначень методів:

$$\frac{\begin{array}{l} T = \begin{cases} T_0, & \text{if -> is present,} \\ \texttt{<None>}, & \textit{otherwise.} \end{cases} \\ M(C, f) = \{T_1 \times \cdots \times T_n \to T;\ x_1, \ldots, x_n;\ v_1 : T'_1, \ldots, v_m : T'_m\} \\ n \geq 1 \qquad m \geq 0 \\ C = T_1 \\ O[T_1/x_1] \ldots [T_n/x_n][T'_1/v_1] \ldots [T'_m/v_m], M, C, T \vdash b \end{array}}{O, M, C, R \vdash \texttt{def}\ f(x_1 : T_1, \ldots, x_n : T_n)\ [\![\texttt{->}\ T_0]\!]^? : b} \tag{A.42}$$

Визначення класів. Визначення класів перевіряються на типи шляхом розповсюдження відповідного середовища типів:

$$\frac{O, M, C, R \vdash b}{O, M, \bot, R \vdash \texttt{class}\ C(S) : b} \tag{A.43}$$

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

A.1. ChocoPy v2.2: Language Manual and Reference (November 23, 2019). URL: https://chocopy.org/chocopy_language_reference.pdf (Accessed 18.04.2023)

ДОДАТОК Б

UML–діаграма класів фази інтерпретації

(довідковий)

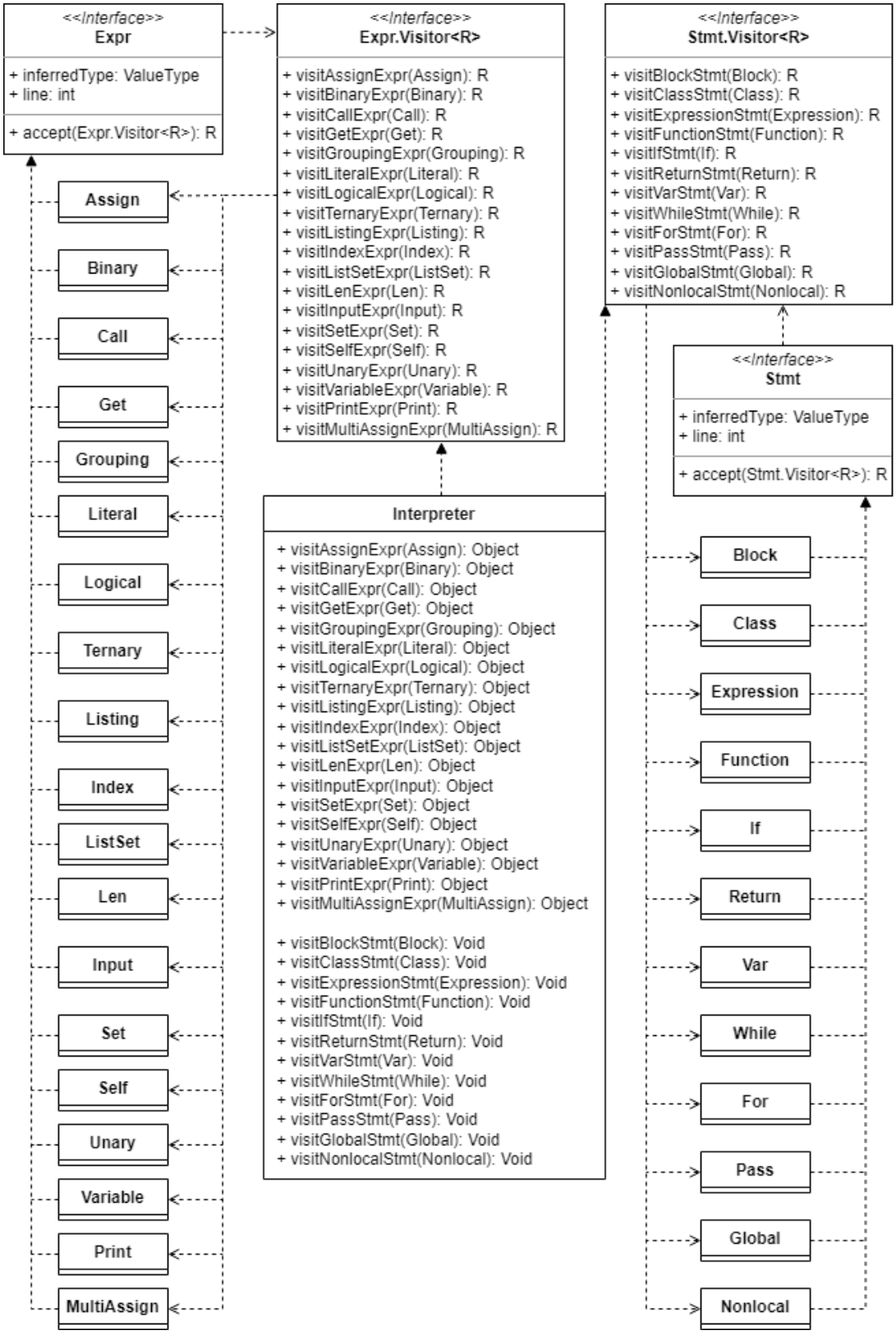

Рисунок Б.1 – UML–діаграма класів фази інтерпретації

ДОДАТОК В

Посилання на код проєкту

(обов’язкове)

https://github.com/alex-krav/chocopy-interpreter

Рисунок В.1 – QR–код з посиланням на код проєкту

ДОДАТОК Г

Тестовий файл у форматі JSON

(довідковий)


```
{
  "kind" : "Program",
  "location" : [ 1, 1, 1, 18 ],
  "declarations" : [ ],
  "statements" : [ {
    "kind" : "ExprStmt",
    "location" : [ 1, 1, 1, 17 ],
    "expr" : {
      "kind" : "IfExpr",
      "location" : [ 1, 1, 1, 17 ],
      "condition" : {
        "kind" : "BinaryExpr",
        "location" : [ 1, 6, 1, 10 ],
        "left" : {
          "kind" : "IntegerLiteral",
          "location" : [ 1, 6, 1, 6 ],
          "value" : 1
        },
        "operator" : ">",
        "right" : {
          "kind" : "IntegerLiteral",
          "location" : [ 1, 10, 1, 10 ],
          "value" : 2
        }
      },
      "thenExpr" : {
        "kind" : "IntegerLiteral",
        "location" : [ 1, 1, 1, 1 ],
        "value" : 3
      },
      "elseExpr" : {
        "kind" : "IntegerLiteral",
        "location" : [ 1, 17, 1, 17 ],
        "value" : 4
      }
    }
  } ],
  "errors" : {
    "errors" : [ ],
    "kind" : "Errors",
    "location" : [ 0, 0, 0, 0 ]
  }
}
```


Рисунок Г.1 – Тестовий файл *pa1/sample/expr_if.py.ast* у форматі JSON

ДОДАТОК Д

UML–діаграма класів для AstPrinter.java

(довідковий)

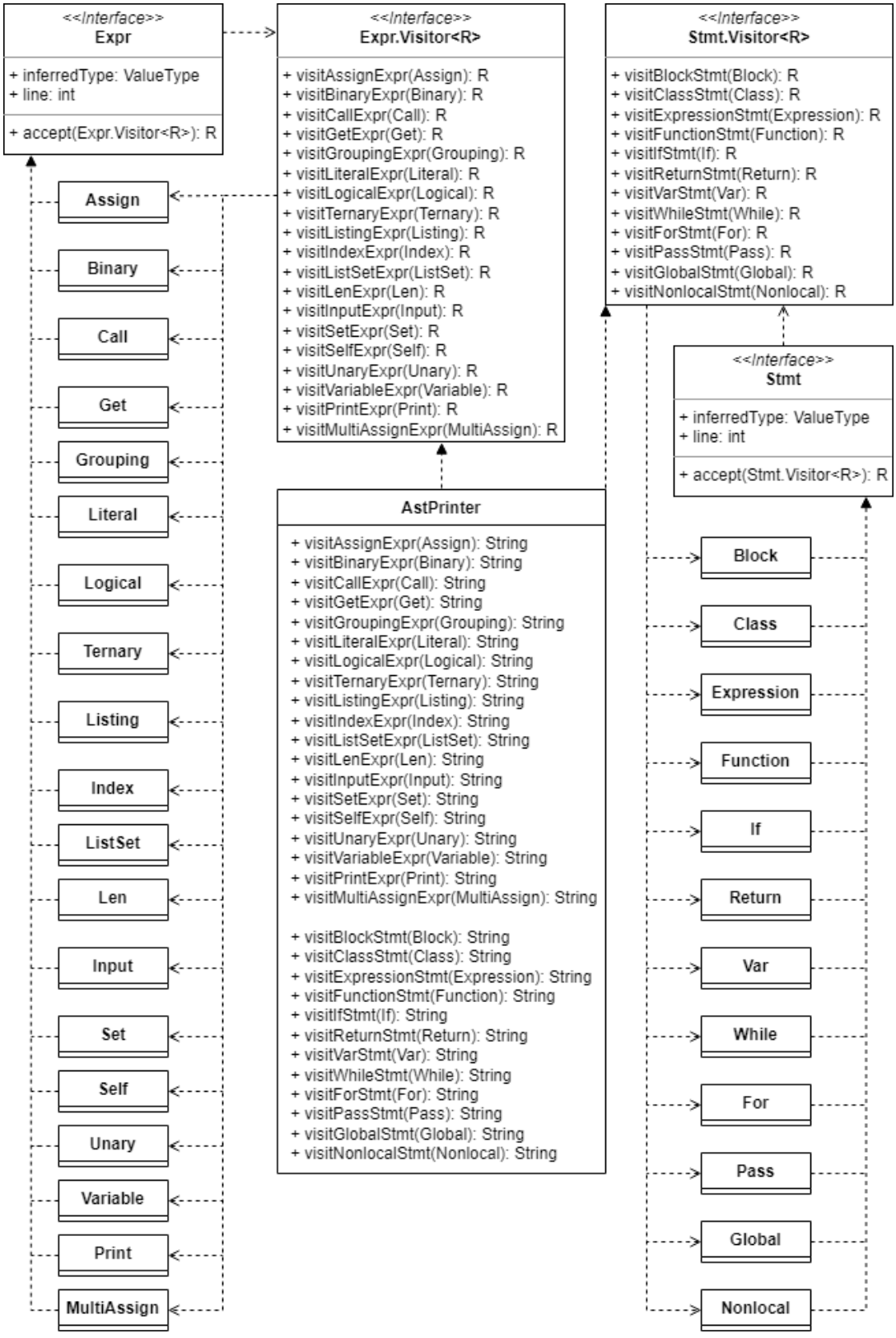

Рисунок Д.1 – UML–діаграма класів для AstPrinter.java