Міністерство освіти і науки України
Національний університет «Києво-Могилянська академія»
Факультет інформатики
Кафедра мультимедійних систем

Кваліфікаційна робота

освітній ступінь – бакалавр

на тему: **«ЗАСТОСУВАННЯ КОНТЕКСТНИХ УМОВ ДЛЯ ПОБУДОВИ ОПЕРАЦІЙНОЇ СЕМАНТИКИ В РЕАЛІЗАЦІЇ МОВИ ПРОГРАМУВАННЯ»**

Виконав: студент 4-го року навчання,
Освітньої програми «Комп'ютерні науки», 122
Білогрудов Даніїл Вячеславович
Керівник _____Бублик В.В._____, кандидат фіз.-мат. наук, доцент

Рецензент _____

Кваліфікаційна робота захищена з оцінкою _____

Секретар ЕК _____
«____» _____ 2024 р.

Київ – 2024

ГРАФІК ПІДГОТОВКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

Графік узгоджено «____» _____ 2024 р.

№ з/п	**ПЕРЕЛІК РОБІТ**	Термін виконання	Дата ознайомлення наукового керівника	Підпис наукового керівника	Примітки
1.	Вибір теми, затвердження її на засіданні кафедри та закріплення наукового керівника. Узгодження календарного графіка підготовки кваліфікаційної роботи. Ознайомлення студента з критеріями оцінювання кваліфікаційної роботи.	жовтень			
2.	Вивчення джерел літератури, матеріалів архівів, періодичних видань, збір та узагальнення фактів, даних	жовтень – листопад			
3.	Складання плану кваліфікаційної роботи та узгодження з науковим керівником	листопад			
4.	Написання розділів роботи	листопад – березень			
5.	Проміжний контроль виконання роботи	лютий			
6.	Написання кваліфікаційної роботи в цілому, ознайомлення з її першим варіантом наукового керівника	січень – березень			
	Розділ 1 Формальна семантика мов програмування	січень			
	Розділ 2 Структура інтерпретатора мови програмування	лютий			
	Розділ 3 Реалізація інтерпретатора	березень			
7.	Повне завершення написання кваліфікаційної роботи, оформлення її згідно з вимогами й подання на відгук науковому керівнику	квітень – початок травня			
8.	Подання кваліфікаційної роботи для перевірки письмових робіт студентів НаУКМА на відповідність вимогам академічної доброчесності	середина травня			
9.	Подання на зовнішню рецензію	середина травня			
10.	Підготовка до захисту кваліфікаційної роботи на засіданні кафедри: написання доповіді та виготовлення ілюстративного матеріалу	до 14 травня			
11.	Попередній захист кваліфікаційної роботи на засіданні кафедри	14 травня			
12.	Подання кваліфікаційної роботи на кафедру з усіма супроводжувальними документами	до 23 травня			
13.	Публічний захист кваліфікаційної роботи перед екзаменаційною комісією	30 травня			

Науковий керівник Бублик Володимир Васильович

Виконавець кваліфікаційної роботи Білогрудов Данiїл Вячеславович

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра мультимедійних систем

Освітній ступінь бакалавр

Спеціальність «Комп’ютерні науки», 122

ЗАТВЕРДЖУЮ

Завідувач кафедри Жежерун О.П.

_____

«____» _____ 2024 року

З А В Д А Н Н Я

ДЛЯ КВАЛІФІКАЦІЙНОЇ РОБОТИ СТУДЕНТУ

Білогрудову Даніїлу Вячеславовичу

1. Тема роботи: «Застосування контекстних умов для побудови операційної семантики в реалізації мови програмування»,

керівник роботи Бублик Володимир Васильович, кандидат фіз.-мат. наук, доцент,

затверджені наказом вищого навчального закладу від «__» _____ 2024 року №__

2. Строк подання студентом роботи – 23 травня 2024 року

3. План роботи:

Вступ

1. Формальна семантика мов програмування
2. Структура інтерпретатора мови програмування
3. Реалізація інтерпретатора

Висновки

Список використаної літератури

Додаток А. Специфікація мови програмування MPL

Додаток В. Контекстно-вільна граматика мови MPL

Додаток С. Правила операційної семантики мови MPL

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

DSL – Domain-specific language (предметно-орієнтована мова)

COC – Структурна операційна семантика

JIT – Just-in-Time (тип компіляції)

АСД – Абстрактне синтаксичне дерево

CFG – Context-free grammar (контекстно-вільна граматика)

BNF – Backus-Naur form (нотація Бекуса-Наура)

EBNF – Extended Backus-Naur form (розширена нотація Бекуса-Наура)

ДСА – Детермінований скінченний автомат

АНОТАЦІЯ

Ця робота присвячена розробці інтерпретатора мови програмування та дослідження ролі контекстних умов у загальній архітектурі обробника. У роботі розглядаються підходи до формальної семантики мов програмування (з окремим акцентом на операційну семантику), складові частини інтерпретатора, реалізація обробки контекстних умов і реалізація інтерпретатора загалом.

Ключові слова: операційна семантика, контекстні умови, мови програмування, інтерпретатори.

ВСТУП

Актуальність теми та практичне значення

Мови програмування не стоять на місці і безперервно розвиваються. Досі триває розробка нових стандартів таких поширених мов, як C++, Rust, Python, Java тощо, і постійно створюються нові мови, покликані вирішити новоз'явлені проблеми. Як приклад можна навести Carbon – експериментальна мова, яка розробляється Google для покращеної підтримки C++ проєктів, або Zig, яка прагне стати нащадком C з акцентом на більшу надійність коду. Зважаючи на масштаб сучасних систем, перед розробниками компіляторів і інтерпретаторів постають нові задачі, особливо щодо забезпечення передбачуваності і коректності виконання програмного коду.

Також варто звернути увагу на зрослу роль предметно-орієнтовних мов програмування (Domain-specific languages, DSL), які створюються для вузькоспрямованих задач в тій чи іншій галузі. Такими є, наприклад, мови для систем складання коду (Maven, Gradle), чи GDScript – вбудована мова програмування для ігрового рушія Godot. У таких мовах значення правильності виконання програм так само важливо, як і у мовах загального призначення.

Існує два основних методи виконання програм – компіляція і інтерпретація. Кожен з двох методів має свої переваги і недоліки та сфери застосування, де він проявляє себе найкраще. Якщо казати про інтерпретатори, то це прототипування і інтерактивне програмування, автоматизація і платформо-незалежне виконання коду. У будь-якому випадку, надважливим є якісне проєктування мови, якому може посприяти розробка операційної семантики для неї. Добре продумане застосування операційної семантики при розробці мови програмування здатне послугувати проміжною реалізацією транслятора, забезпечити його формальну коректність і виявити конструктивні недоліки на ранньому етапі розробки.

Навіть для звичайних розробників розуміння принципів роботи мов програмування може стати при нагоді. Велика кількість мов або повністю інтерпретовані, або мають реалізацію, засновану на інтерпретаторі. До них входять Python, JavaScript, PHP та інші. Тож, знання деталей реалізації мови може бути корисним для кращого розуміння мови й оптимізації коду.

Мета і завдання роботи

Мета роботи – дослідження методів розробки інтерпретатора мови програмування із застосуванням контекстних умов у процесі трансляції програм. Для досягнення мети були поставлені наступні *завдання*:

1. Дослідити теоретичні основи формалізації мов програмування, а також різні підходи до визначення операційної семантики.
2. Розглянути основні складові компоненти інтерпретатора та їхню роль у трансляції вихідного коду та роль контекстних умов на етапі інтерпретації програми.
3. Розробити і протестувати інтерпретатор згідно з попередньо визначеною граматикою та застосуванням правил структурної операційної семантики.

Об’єкт і предмет дослідження

Об’єкт дослідження – формальна семантика мов програмування, абстрактна граматика, архітектура транслятора мови.

Предмет дослідження – підходи до обробки контекстних умов у реалізації транслятора мови та розробка інтерпретатора мови програмування за попередньо визначеною абстрактною граматикою.

Структура роботи

Кваліфікаційна робота складається з анотації, переліку прийнятих скорочень, вступу, трьох розділів, висновку, списку використаної літератури і трьох додатків.

Перший розділ містить відомості про історію досліджень операційної семантики мов програмування, підходи до формалізації, а також детальний опис структурної операційної семантики.

Другий розділ присвячений опису роботи інтерпретатора та його основних складових компонентів – лексичного, синтаксичного і семантичного аналізаторів.

Третій розділ зосереджений на реалізації інтерпретатора мови програмування, описі використаних технологій і перевірці коректності роботи.

РОЗДІЛ 1. ФОРМАЛЬНА СЕМАНТИКА МОВ ПРОГРАМУВАННЯ

1.1 Поняття семантики мов програмування

1.1.1 Порівняння природних мов і мов програмування

Будь-яку природну мову можна описати двома загальними концептами – синтаксисом і семантикою. Синтаксис відноситься до граматичних правил мови, тобто структури, за якою слова впорядковані у реченні. Але просто граматично правильного речення може бути недостатньо.

Можна взяти як приклад наступну фразу: «Безбарвні зелені ідеї сплять несамовито». Незважаючи на те, що речення побудоване згідно з усіма граматичними правилами української мови, йому бракує найголовнішого – сенсу. Тут у роль вступає семантика – змістовне наповнення, яке надає мові сенс. Саме це дозволяє мові виконувати її головне призначення – комунікацію.

Хоча мови програмування функціонально суттєво відрізняються від природних мов, до них застосовуються ті самі поняття синтаксису і семантики. Синтаксис мови програмування – це сукупність правил, що визначають які комбінації символів у рядку складають коректну програму, і якщо програма коректна – яка її структура [1]. Таке саме визначення можна віднести і до природних мов – яка комбінація букв складатиме граматично коректне речення.

У свою чергу, *семантика* прагне визначити точний математичний зміст програми [2]. У той час, коли семантика в природних мовах досліджує значення кожного окремого слова у реченні і як ці слова взаємодіють між собою задля донесення загального сенсу, семантика мов програмування визначає математичне значення кожної складової програми (змінної, виразу, твердження) і які операції обчислювач має виконати, щоб перетворити вхідні дані на коректні вихідні.

Головна відмінність ролі семантики у природних мовах і мовах програмування полягає у точності. Коли людське мовлення може бути

інтерпретоване по-різному в залежності від різних умов, програми мусять бути визначені формально і точно. Це випливає з призначення – в мовах програмування головною метою є забезпечення правильності та передбачуваності виконання, у той час як природні мови забезпечують ефективну комунікацію, яка може допускати неоднозначність.

1.1.2 Історія досліджень

Потреба у математичному підґрунті для формального визначення мов програмування з'явилася у 1950-х роках, у період поступового переходу від низькорівневого програмування до перших високорівневих мов, таких як FORTRAN і ALGOL [3]. Якщо раніше спосіб написання програм напряму залежав від технічної специфікації обчислювача, на якому вона виконувалася, то перші високорівневі мови дали змогу розробникам писати машинно-незалежні програми.

Одночасно з високорівневими мовами з'являється потреба у програмі, яка власне перекладатиме код, написаний людиною, у послідовність команд, яка зрозуміла машині, які згодом стануть відомі як компілятори та інтерпретатори. Але на той момент вони створили більше проблем, ніж рішень – компілятори містили у собі велику кількість програмних помилок, які ускладнювали написання надійного програмного забезпечення. Це призвело до потреби у формалізації мов програмування шляхом введення математичних технік доказу коректності стану програми. Незважаючи на суто математичну природу формалізації, було вирішено скористатися вже вище описаними термінами з галузі лінгвістики – синтаксис і семантика.

У своїй праці «Assigning meanings to programs» («Призначення значенн програмам») [4], Роберт Флойд дав наступне визначення семантиці:

«Семантичне визначення мови програмування в нашому підході – засноване на синтаксичному визначені. Воно мусить вказувати які фрази в

синтаксично коректній програмі представляють команди, і які умови мають бути накладені на інтерпретацію в сусідстві з іншими командами» [4, с. 20].

Надалі напрацювання Флойда були використанні Тоні Гоаром у його статті «An axiomatic basis for computer programming» («Аксіоматична основа комп'ютерного програмування») [5] для визначення логіки, яка згодом отримала назву на його честь – Гоара – формальної системи, яка дозволяє доводити правильність програм за допомогою множини логічних правил. У подальшому ця логіка ляже в основу аксіоматичної семантики.

Інші науковці того часу теж намагалися надати визначення семантики, на той час нового концепту у інформатиці. Едсгер Дейкстра у статті «On the design of machine independent programming languages» [14] («Про проектування машинно-незалежних мов програмування») висловився наступним чином:

«Оскільки метою мови програмування є опис процесів, я розглядаю визначення її семантики як розробку, опис машини, яка реагує на довільний опис процесу цією мовою фактичним виконанням цього процесу. […] При проектуванні мови ця концепція «визначальної машини» має допомогти нам забезпечити однозначність семантичної інтерпретації текстів» [14].

Ця ідея певної машини, обчислювача, за допомогою якої має визначатись мова програмування лягла в основу операційного підходу до семантики [3]. Хоча надалі Дейкстра і відмовився від такого підходу до визначення семантики, ці ідеї були надалі розвинуті Даною Скоттом [7] і Ґордоном Плоткіном [11].

1.2 Підходи до семантики

Виділяють три головних класи підходів до формальної семантики мов програмування – денотаційна семантика, аксіоматична семантика й операційна семантика. Хоча й існують інші підходи, такі, як категорична семантика (яка користується теорією категорій) чи ігрова семантика (заснована на теорії ігор), які використовують певні математичні логічні концепти за основу, при розробці мов програмування найчастіше використовуються саме три основних підходи.

1.2.1 Денотаційна семантика

Денотаційна семантика концентрується на представленні мов програмування як математичних моделей – значення фраз програми визначаються як елементи певної абстрактної математичної структури [6]. Основи денотаційної семантики були закладені у 1970 році американським математиком Даною Скоттом у його статті «Outline of a mathematical theory of computation» («Нарис математичної теорії обчислень») [7]. У цій роботі Скотт за допомогою денотаційної семантики (яку він початково назвав «математичною») описує програму як математичну функцію, яка перетворює вхідні дані на вихідні. Ця функція складається з денотатів кожної окремої фрази програми, які об'єднані за допомогою композиції – головного принципу денотаційної семантики.

За денотаційною семантикою, кожний вираз P програми характеризується денотацією $[\![P]\!]$ – математичним об'єктом, який представляє вклад виразу P до значення будь-якої завершеної програми в якій він зустрічається [6].

Як приклад нотації можна показати представлення певної змінної x в умовній мові програмування:

$[[x]](\sigma) = \sigma(x)$, де σ – стан програми.

Нині денотаційна семантика використовується в більшості для визначення принципів роботи паралельних і розподілених систем, а також для функціональних мов програмування, таких як Haskell.

1.2.2 Аксіоматична семантика

Аксіоматична семантика – це підхід, у якому значення програмних фраз подається через аксіоми і набір теорем для окремо взятої програми [6]. Аксіоматична семантика напряму походить від логіки Гоара, описаної у [5]. Ця

семантика базується на певних твердженнях (предикати зі змінними), які визначають стан програми у окремо взятий час виконання. Якщо твердження справджуються до і після виконання програмної фрази, то програма коректна.

За принципами аксіоматичної семантики, програма складається з послідовності команд, кожне з яких анотоване припущеннями про стан програми до і після виконання. Справдження припущень означає коректність програми. Припущення до виконання називається *передумовою*, а після виконання – *післяумовою*. Загальний вигляд нотації виглядає наступним чином [8]:

$$\{ PRE \}\ \mathrm{C}\ \{ POST \},$$

де PRE – передумова, C – команда і $POST$ – післяумова. Пара $< PRE, POST >$ є *специфікацією* команди C. Також виділяються поняття *часткової коректності* програми – коли програма, запущена з правдивою передумовою, призводить до правдивості післяумови, і *повної коректності* – часткова коректність із забезпеченням закінчення виконання програми.

Як приклад аксіоматичної семантики для певного твердження можна навести наступний вираз:

$$\{ k = 5 \}\ \mathrm{k} := \mathrm{k} + 1\ \{ k = 6 \},$$

або звести до загального вигляду:

$$\{ P[V \to E] \}\ \mathrm{V} := \mathrm{E}\ \{ P \}$$ [8, с. 399].

Можна помітити, що аксіоматична семантика дуже схожа на вираз assert у багатьох мовах програмування. Твердження цієї семантики можна зобразити псевдокодом наступним чином:

```
assert(k == 5);
k = k + 1;
assert(k == 6);
```

Отже, цілком логічно, що аксіоматична семантика використовується для тестування коректності програм та алгоритмів (зокрема і для засобів

автоматичного тестування) і статичного аналізу (перевірка ствердження інваріант програм).

1.2.3 Операційна семантика

Операційна семантика – підхід, який фокусується на обчисленні кожного окремого виразу мови програмування. У той час, коли аксіоматична семантика описує зміну стану при обчислені, а денотаційна семантика співвідносить вирази до значень напряму, операційна семантика описує *як* обчислити той чи інший вираз.

Видання [9] надає наступне визначення: операційна семантика мови програмування – це математичне визначення її обчислювального співвідношення $e \Rightarrow v$, де e – це програма мови, а v – це вихідні програми. Операційну семантику можна порівняти з математично визначеним інтерпретатором, перевага якого – це точність, бо специфіка мови, якою написаний інтерпретатор, і особливості обчислювача, на якому програма виконується, нівелюються, так як використовуються суто математичні концепти.

У статті «The origins of structural operational semantics» («Витоки структурної операційної семантики») [11], Ґордон Плоткін зазначає, що вперше згадка про операційну складову семантики зустрічається у Скотта в [7]:

«Дуже добре прагнути до більш «абстрактного» і «чистого» підходу до семантики, але якщо ми хочемо досягти успіху, не можна повністю ігнорувати операційні аспекти» [7, с. 169].

Операційна семантика традиційно поділяється на два підходи:

1) *Структурна операційна семантика* (або семантика малого кроку)

Започаткована Плоткіном у 1981 році у «Структурному підході до операційної семантики» [10]. Головна ідея структурної операційної семантики – визначення поведінки програми шляхом визначення поведінки її складових

частин, що надає структурний і індуктивний погляд на семантику. СОС користується моделлю переходів станів для опису поведінки.

Також існує такий підвид семантики малого кроку, як редукційна семантика, що користується множиною правил редукції для визначення поведінки програми. Такий підхід може бути корисним для опису об'єктно-орієнтовної моделі чи концепцій паралельних обчислень.

2) *Природна операційна семантика* (або семантика великого кроку)

Природна операційна семантика розглядає вирази програми як один великий «крок», і напряму співвідносить вирази з їхніми значеннями. Цей підхід фокусується на кінцевому результаті, радше ніж на проміжних етапах.

Хоча природна операційна семантика є більш інтуїтивно зрозумілою та легшою у визначенні, ніж структурна операційна семантика, вона не надає часом важливої інформації про структуру виразу, що є важливим при забезпечені коректності програми.

1.3 Структурна операційна семантика

Структурна операційна семантика ледь не найчастіше використовується для визначення семантичних правил мов програмування. Відстеження стану програми, включно з інформацією про складові програми, дозволяє легше доводити такі аспекти, як типобезпечність програми.

Правила структурної операційної семантики базуються на попередньо визначеному синтаксисі мови програмування. Ключовим аспектом СОС є відношення переходу, яке показує власне перехід однієї конфігурації у іншу. Використовуючи нотацію, надану Плоткіном у [10], перехід можна зобразити наступним чином:

$$\langle C, \sigma \rangle \rightarrow \langle C', \sigma' \rangle,$$

де C – це певний синтаксичний конструкт (вираз, твердження, ідентифікатор тощо), а σ – поточний стан середовища.

Відношення переходу визначаються за допомогою правил виведення, які визначають умови, за яких ці відношення можливі. Правило виведення визначається наступним чином:

$$\frac{P_1 \quad P_2 \quad \dots \quad P_n}{C},$$

де P_i це умови виведення, а C – власне сам вивід. У СОС умовами можуть бути попередні переходи чи певний поточний стан.

Як приклад, можна розглянути правила виведення для певного граматичного правила мови. Візьмемо наступне правило контекстно-вільної граматики:

$$E \Rightarrow n \mid E_1 + E_2,$$

де E – вираз мови, а n – числове значення. Для цієї граматики можливо вивести наступні семантичні правила:

1. Числове значення переходить само у себе:

$$\langle n, \sigma \rangle \to \langle n, \sigma \rangle$$

2. Додавання двох виразів є можливим при забезпеченні переходу обидвох операндів до числового значення:

$$\frac{\langle E_1, \sigma \rangle \to \langle E_1', \sigma' \rangle}{\langle E_1 + E_2, \sigma \rangle \to \langle E_1' + E_2, \sigma' \rangle}$$

$$\frac{\langle E_2, \sigma \rangle \to \langle E_2', \sigma' \rangle}{\langle n + E_2, \sigma \rangle \to \langle n + E_2', \sigma' \rangle}$$

$$\langle n_1 + n_2, \sigma \rangle \to \langle n_3, \sigma \rangle$$

1.4 Генералізація контекстно-вільної граматики

У статті українського кібернетика Олександра Летичевського «Генералізація концепту контекстно-вільної граматики» [17] був запропонований підхід до поєднання синтаксичних і семантичних правил шляхом впровадження додаткової змістової інформації до символів контекстно-вільної граматики, зокрема типів. За словами автора, це дозволяє одночасно зберегти переваги рекурсивних граматик і зручність алгоритмів синтаксичного аналізу.

У роботі за основу була взята граматика мови ALGOL 60 – вона була розширена новими символами:

```
<variable of type …>
<array identifier of type …>
<function designator of type …>
<left part list of type …>
<formal parameter of type …>
```

де типами можуть бути real, integer, Boolean тощо. На прикладі операції присвоєння модифікована граматика приймає вигляд:

```
<variable of type real> :: = <formal parameter of type real> |
<formal parameter of type real array> [ <subscript list> ]
```

Такий підхід був названий *генералізованою* граматикою. Він дозволяє впровадити певний ряд контекстних умов у граматику, що робить можливим їхню перевірку ще на етапі синтаксичного аналізу

1.5 Висновок

У цьому розділі було досліджено теоретичні основи формальної семантики мов програмування. Було продемонстровано взаємозв’язок між поняттями синтаксису і семантики у природніх мовах та мовах програмування,

де в обох випадках синтаксис відповідає за граматичну коректність написаного, а семантика – за надання сенсу і значень.

Було розглянуто історію досліджень галузі, її започаткування на ранніх етапах розвитку інформатики і поступове впровадження семантичної теорії при проєктуванні мов програмування. З цією метою були опрацьовані роботи таких науковців, як Дана Скотт, Тоні Гоар, Ґордон Плоткін та інших, які заклали основу для різних підходів до визначення семантики, що забезпечило можливість формального доведення коректності програм.

Описані основні підходи до операційної семантики, які включають в себе аксіоматичну, денотаційну і операційну семантики мов програмування. Була розглянута їхня нотація, випадки застосування і відмінності між ними.

Також окремо була розглянута структурна операційна семантики за визначенням Плоткіна. На прикладі було продемонстровано визначення правил контекстних умов і переваги даного підходу при проєктуванні та розробці мов програмування.

Загалом, було доведено потребу у формальній семантиці як інструменту визначення чіткої і однозначної поведінки програм за допомогою математичних і логічних понять, що є критично важливим при реалізації мов програмування, які використовуються у розробці складних і критичних систем.

РОЗДІЛ 2. СТРУКТУРА ІНТЕРПРЕТАТОРА МОВИ ПРОГРАМУВАННЯ

2.1 Визначення інтерпретатора мови програмування

Програма, яка написана високорівневою мовою програмування, не здатна виконуватись тільки за наявності вихідного коду. Через те, що високорівневі мови – це компроміс між машинозчитуваним кодом і мовою, яку розуміє людина, має існувати окрема програма, яка перекладатиме дану програму у формат, «зрозумілий» обчислювачу. Такі програми сукупно називаються трансляторами – утиліти, які трансформують вихідний код у цільову програму машинною мовою.

Як вже було зазначено у вступі, існує два основних типи трансляторів – компілятори і інтерпретатори. Головна їхня відмінність полягає у тому, що компілятор перетворює вихідний код високорівневої мови у об'єктний код, який потім може виконуватись на машині без участі компілятора. У свою чергу інтерпретатор повністю відповідає за виконання програми, проходячись нею рядок за рядком, виступаючи при цьому віртуальною машиною, яка як раз використовує високорівневу мову замість машинної [12].

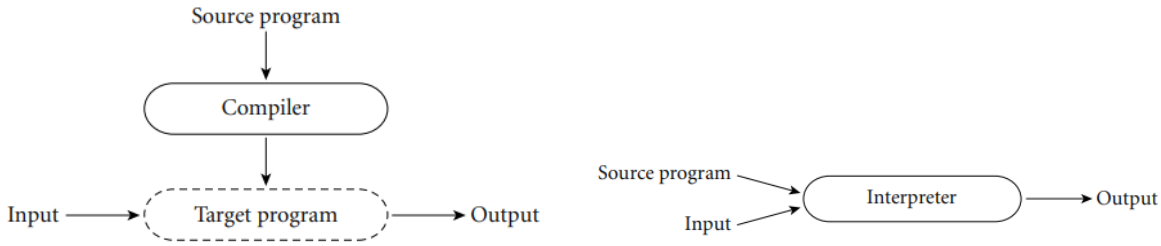

Рисунок 1. Порівняння виконання програми компілятором і інтерпретатором [12]

У цій роботі було обрано віддати перевагу реалізації саме інтерпретатора, переважно через порівнянну простоту розробки у порівнянні з компілятором.

Також у порівнянні з компіляторами, інтерпретаторам властива більша гнучкість у процесі виконання коду. Це дозволяє, наприклад, простіше реалізувати зневаджувач вихідного коду, або ширше контролювати послідовність виконання. У таких мовах, як Lisp і Prolog, це виявляється у особливості автоматичного дописування нового коду просто під час виконання програми. Але інтерпретація має і свої недоліки – головним чином це більш повільна швидкодія програми [12].

Однак, транслятори багатьох мов програмування одночасно поєднують у собі риси компіляції і інтерпретації. Це досягається чином формування програми проміжною мовою, з подальшою її інтерпретацією у окремому середовищі. У випадку мови Java, використовується Java bytecode, який потім виконується у віртуальній машині JVM. Такий підхід відомий, як Just-in-Time (JIT) компіляція [13].

Власне, сама інтерпретація складається з чотирьох ключових частин:

1) *Лексичний аналіз* (сканер) – початковий етап, який приймає на вхід вихідний код програми у вигляді потоку символів і розбиває його на токени – найменші значущі елементи програми.
2) *Синтаксичний аналіз* (парсер) – приймає на вхід потік токенів і формує з них проміжну репрезентацію, яка надалі використовуватиметься для виконання, одночасно проводячи перевірку програми згідно з заданою граматикою для цієї мови програмування.
3) *Семантичний аналіз* – проводить перевірку на семантичні обмеження мови (типізація, області видимості змінних тощо).
4) *Виконавець* – приймає на вхід синтаксично і семантично коректну програму (або частину програми) та вхідні дані програми, і покроково виконує її засобами мови, якою написаний інтерпретатор, повертаючи вихідні дані.

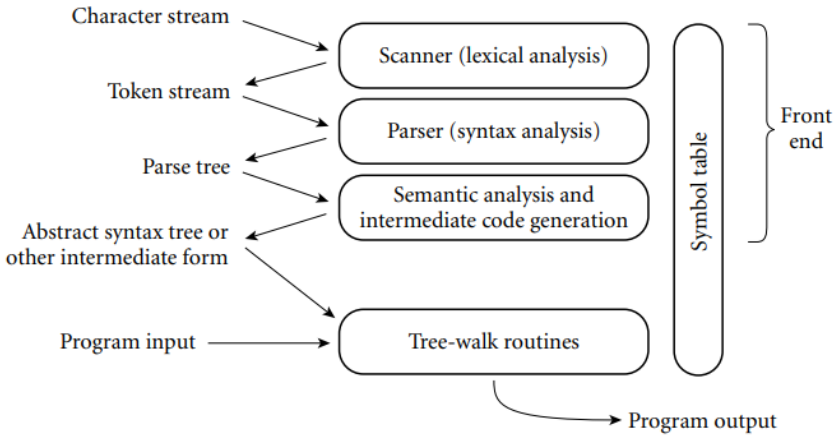

Рисунок 2. Фази інтерпретації [12]

2.2 Підходи до реалізації інтерпретаторів

Виділяється два основних типи інтерпретаторів – інтерпретатори абстрактного синтаксичного дерева і інтерпретатори байт-коду. Головна їхня відмінність полягає у представленні проміжної структури програми під час процесу трансляції – у першому випадку програма представлена деревоподібною структурою, де вузли відповідають за складові програми, а у другому програма набуває вигляду машиноподібного коду, де кожен рядок відповідає певній базовій операції.

Хоча й існують інші типи інтерпретаторів на кшталт інтерпретаторів шитого коду чи інтерпретаторів шаблонів, але вони не є настільки поширеними, як перші дві варіації.

2.2.1 Абстрактне синтаксичне дерево

Ключовий компонент інтерпретаторів абстрактного синтаксичного дерева є власне саме абстрактне синтаксичне дерево. Воно являє собою структуру даних, яка є репрезентацією вихідного коду формальною мовою.

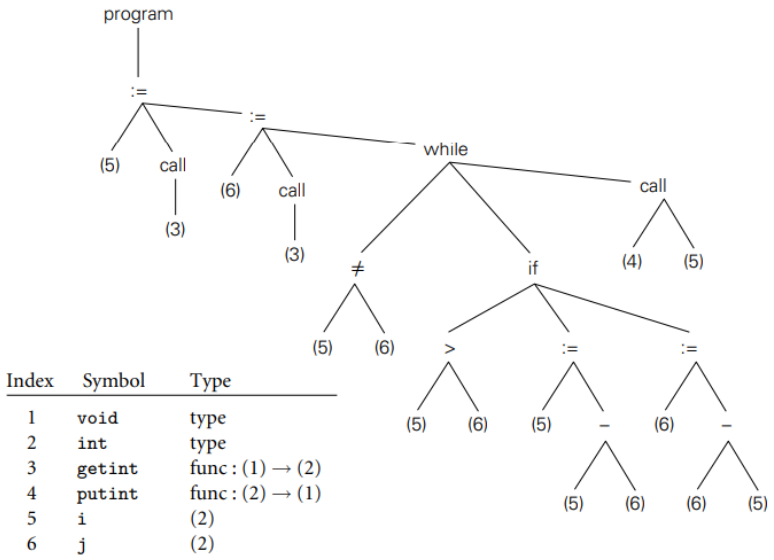

Рисунок 3. Приклад АСД програми для обчислення НСД [12]

У цьому підході інтерпретація відбувається наступним чином – кожне твердження програми обчислюється шляхом обходу дерева. В залежності від специфіки граматики мови, кожен вузол абстрактного синтаксичного дерева може представляти певний вираз, такий, як твердження, ідентифікатор, вираз тощо, який містить необхідну інформацію для обчислень.

Деякі мови програмування використовують абстрактне синтаксичне дерево як ще один проміжний етап перед трансформацією вихідного коду в іншу форму, яка буде використовуватись для подальшої інтерпретації.

Інтерпретатори, засновані на абстрактних деревах, відносно прості у реалізації, але мають суттєвий недолік – при великому розмірі програми дерево призводить до значних додаткових витрат пам’яті, яка витрачається на додаткову інформацію про певний компонент програми і на операції алокації-деалокації.

2.2.2 Байт-код

Байт-код – набір інструкцій, розроблений для виконання вихідного коду інтерпретатором. Байт-код за формою наближений до машинного коду, але

виконується не напряму обчислювачем, а певною віртуальною машиною, яка є складовою інтерпретатора. Це надає можливість легшого крос-платформеного виконання одного і того самого вихідного коду на різних пристроях. Часто для виконання байт-коду використовуються стекові машини.

```
4           0 LOAD_FAST                0 (n)
            2 LOAD_CONST               1 (0)
            4 COMPARE_OP               2 (==)
            6 POP_JUMP_IF_FALSE        6 (to 12)

5           8 LOAD_CONST               1 (0)
           10 RETURN_VALUE

6     >>   12 LOAD_FAST                0 (n)
           14 LOAD_CONST               2 (1)
           16 COMPARE_OP               2 (==)
           18 POP_JUMP_IF_FALSE       12 (to 24)
```

Рисунок 4. Приклад байт-коду, згенерованого інтерпретатором мови Python

Назва байт-код походить від однобайтових кодів операції, з яких код власне і складається. Інструкції складаються власне з самих операцій, констант і посилань на адреси. Під час інтерпретації байт-коду також відбуваються семантичні перевірки.

Реалізаціям інтерпретаторів, які використовують байт-код, не властива проблема надмірного використання пам’яті, як у випадку з абстрактними синтаксичними деревами, але в свою чергу вони є складнішими в розробці.

Поняттям, дотичним до байт-коду, є JIT-компіляція. JIT-компіляція полягає у компіляції програми не до, а під час виконання шляхом перетворення команд байт-коду у машинний код. Як вже було зазначено вище, цей підхід використовує мова програмування Java завдяки Java Virtual Machine (JVM), але також схожий принцип реалізований у мові C# – .NET Common Language Runtime (CLR).

2.3 Специфікація мови програмування MPL

Специфікація мови програмування – це документ, який надає опис структури мови програмування, і визначає її синтаксис, семантику і прагматику. Специфікація може використовуватися так і розробниками трансляторів мови, так і користувачами, які пишуть програми цією мовою.

Для подальшого визначення граматики і правил операційної семантики було розроблено специфікацію. За основу була взята мова Pascal, а саме її опис, наведений у [15]. Назвою мови було обрано MPL – від «mini Pascal» («маленький Pascal»). Специфікація наведена у Додатку А.

Було вирішено сфокусуватись на основних структурних елементах мови, без занурення у більш складні деталі реалізації заради мети дослідження ролі контекстних умов і використання правил операційної семантики. Специфікація мови включає у себе:

- Типи даних (bool, int, string);
- Допустимі операції між типами та операції;
- Вирази мови;
- Список ключових слів.

Ця специфікація надалі використовується для її розширення синтаксичними і семантичними правилами, а також для подальшої розробки інтерпретатора мови MPL.

2.4 Реалізація етапів інтерпретації

2.4.1 Лексичний аналіз

Лексичний аналіз – найперший етап інтерпретації програми, за якого вихідний код програми розбивається на токени Токен. – найменша ключова одиниця програми (відповідно до слова у природній мові), яка визначена

граматикою даної мови програмування. Це може бути ключове слово числове значення, назва змінної, символьний рядок та інші складові мови.

Під час цього етапу зазвичай відкидаються незначущі елементи програми, такі, як коментарі, пробіли чи надлишкова пунктуація. Також токени маркуються їхнім положенням у вихідному коді (номер рядка та стовпчика), аби в подальшому надати більш точну інформацію при обробці помилок. До того ж, під час лексичного аналізу перевіряється орфографічна коректність програми – при обробці послідовності символів, яка не задана граматикою мови, процес інтерпретації переходить у аварійний стан.

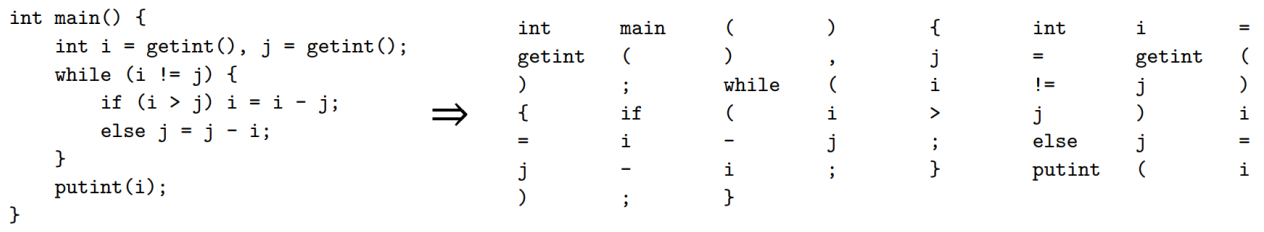

Рисунок 5. Розбиття програми мовою C на токени [12]

Результат лексичного аналізу – послідовність токенів, які надалі передаються синтаксичному аналізатору для формування абстрактного синтаксичного дерева і перевірки програми на відповідність граматиці.

Існує декілька підходів до реалізації лексичного аналізу. У роботі використаний ad-hoc (ситуативний) підхід – послідовне сканування кожного з символів вихідного коду програми і виконання певної дії для формування токену в залежності від значення символу. Для коректної обробки багатосимвольних токенів використовується «підглядання» наступного символу після поточного. Наявні окремі процедури для формування односивольних токенів, літералів, ідентифікаторів, ключових слів і пропуску коментарів і пробілів.

Іншим підходом до лексичного аналізу є застосування регулярних виразів – послідовності символів, який визначає взірець зіставляння для певного

тексту. Регулярні вирази ідеально підходять для визначення правил лексичного аналізу, але їхня коректна програмна реалізація є дещо важчою за «ситуативний» підхід. Наприклад, для мови MPL регулярний вираз для пошуку рядкового літерала виглядає наступним чином:

```
"[^"\\\n]*({0,1}:\\.[^"\\\n]*)*"
```

Регулярні вирази тісно пов’язані з ще одним підходом – скінченними автоматами. Будь-який регулярний вираз можна подати у вигляді недетермінованого скінченного автомата, а згодом – і детермінованого. Саме ДСА використовуються у генераторах лексичних аналізаторів – утиліт, які за допомогою регулярних виразів автоматично генерують ДСА для лексичного аналізу вихідного коду [16]. Прикладами таких генераторів є lex, flex і Jflex.

2.4.2 Синтаксичний аналіз

Синтаксичний аналіз – етап інтерпретації, за якого відбувається перевірка вихідного коду на відповідність граматиці мови програмування, і паралельне формування абстрактного синтаксичного дерева. Формальною граматикою мови найчастіше виступає контекстно-вільна граматика (CFG, context-free grammar) – специфікація мови програмування, яка визначає її правила і допустимі конструкції. Однак, контекстно-вільна граматика не здатна охопити усі аспекти коректності мови. Це вирішується семантичним аналізом, який буде розглянутий у наступному підрозділі.

За визначенням, контекстно-вільна граматика – це формальна граматика, правила відтворення якої мають вигляд

$$A \rightarrow \alpha,$$

де A – це єдиний нетермінальний символ, а α – рядок термінальних та/або нетермінальних символів.

Наприклад, до наступної операції присвоєння

```
a := 2 + 3;
```

можна застосувати наступне правило:

<Stmt> → <Id> := <Expr>;

згідно з яким твердження розбивається на ідентифікатор і вираз. У даному випадку <Id> і <Expr> є нетерміналами. Накладання правил повторюється до досягнення термінальних символів.

Стандартною репрезентацією граматики мов програмування є нотація Бекуса-Наура (BNF, Backus-Naur form), або її варіаціями, наприклад – розширена нотація Бекуса-Наура (EBNF, Extended Backus-Naur form), розроблена Ніклаусом Віртом.

Специфікацією BNF є набір правил, представлених у формі:

`<symbol> ::= __expression__`,

де `<symbol>` це нетермінальна змінна, `::=` – позначення для заміни лівого виразу на правий, і `__expression__` – одна чи більше послідовностей термінальних чи нетермінальних символів, які можуть бути розділені `|` – позначка для вибору однієї з послідовностей.

Згідно зі специфікацією мови MPL була визначена розширена нотація Бекуса-Наура контекстно-вільної граматики для мови програмування MPL, наведена у Додатку В.

Для реалізації синтаксичного аналізатора для мови MPL було обрано підхід нерекурсивного спуску (LL(1) top-down parsing). Він полягає у побудові АСД «згори донизу», послідовно проходячись потоком токенів (сформованих на попередньому етапі). LL (left-most – «найбільш лівий») означає перетворення першого нетермінала згідно із правилом репродукції, формуючи тим самим АСД, «нахилене» ліворуч. Це дозволяє позбутись рекурсивного застосування одного і того самого правила для певної послідовності нетерміналів. Одиниця означає використання «підглядання» на один токен наперед.

Для представлення дерева ідеально пасує шаблон проектування компонувальник (Composite). Компонувальник використовується, коли основна

модель програми може бути представлена у вигляді деревоподібної структури даних. У реалізації шаблону використовується загальний інтерфейс вузла, який визначає метод GetAllChildren() для доступу до всіх дочірніх вузлів.

Вузол абстрактного синтаксичного дерева міститиме інформацію про тип виразу, відповідний токен, значення, якщо це літерал або змінна, і дочірні вузли, якщо він не є листям. Так як вузол містить інформацію про токен, то ми можемо отримати його місцезнаходження (рядок і стовпчик) у разі аварійної ситуації.

Шаблон компонувальник, використаний для реалізації структури абстрактного синтаксичного дерева можливо поєднати з іншим шаблоном – відвідувач, який у подальшому знадобиться при семантичному аналізі та проходженні по дереву для виконання програми.

2.4.3 Семантичний аналіз

Роль семантичного аналізу полягає у знаходженні логічних погрішностей програми, які не були виявлені на етапі синтаксичного аналізу. Це включає в себе використання неоголошених змінних, повторне використання змінних, і в багатьох мовах програмування також типізацію [12].

Потребу у семантичному аналізі можна продемонструвати на прикладі наступного фрагменту програми мовою MPL:

```
var n : str;
for n in 0..2 do
    k := k + 1;
end for;
```

Синтаксично, ця програма є цілком коректною. У випадку з оголошенням твердження циклу, ми маємо «контрольну» зміну n, яка зводиться до ідентифікатора. Але, правила контекстної граматики не дозволяють перевірити, чи ця зміна має цілочисельний тип, як того вимагає специфікація. Очевидно, що таке твердження не матиме сенсу при використанні рядка для ітерації по

чисельному ряду. Або ж, у середині циклу ми маємо операцію присвоєння до певної змінної k, яке теж є синтаксично вірним. Знову ж таки, керуючись виключно граматикою, ми не можемо гарантувати правильний тип змінної для операції додавання до числа, а також чи ця змінна була попередньо оголошена.

Єдиний спосіб забезпечити перевірку на логічні обмеження програми – семантичний аналіз із урахуванням контекстних умов програми. У випадку мови MPL ці умови включають у себе:

- Узгодження типів у бінарних операціях.
- Використання ідентифікатора цілочисельного типу у якості контрольної змінної твердження циклу.
- Використання виразів, які зводяться до цілочисельних значень для визначення меж ітерації.
- Використання виразу, який зводиться до булевого значення у твердженні розгалуження.
- Операції над тільки попередньо оголошеними змінними.

Враховуючи специфікацію мови програмування MPL, наведену у Додатку А і попередньо розроблено контекстно-вільну граматику у Додатку В, було визначено ряд контекстних умов, представлених у вигляді правил структурної операційної семантики. Вони наведені у Додатку С.

Для проведення семантичного аналізу найчастіше використовується символьна таблиця – структура даних, яка зіставляє кожен ідентифікатор з відомою про нього інформацією.

Для перевірки кожного вузла абстрактного синтаксичного дерева використовується шаблон проектування відвідувач. Перевірка проводиться наступним чином – кожен вузол містить оголошений метод Accept() для його «відвідування» і отримання інформації про нього, у той час, як у окремому класі оголошені методи Visit() для кожного типу вузла. Семантичний аналіз починається з кореневого вузла і проводиться рекурсивно до кожного з листя дерева.

2.4.4 Виконання

Виконання чи інтерпретація – фінальний етап роботи інтерпретатора. Маючи синтаксично і семантично перевірену програму, можна оперувати твердженням, що вона коректна і може бути виконана інтерпретатором.

Інтерпретація відбувається за схожим принципом до семантичної перевірки – також за допомогою шаблона виконавець. При рекурсивному обході абстрактного синтаксичного дерева обчислюється значення кожного виразу, представленого вузлом для отримання кінцевого результату програми.

На жаль, етапи синтаксичної та семантичної перевірки не можуть виявити усі вразливості програми. Існує певний клас помилок, які можуть бути виявлені тільки на етапі виконання коду, а отже – не можуть бути перевірені на етапі синтаксичного чи семантичного аналізу. Для мови MPL такі випадки включають у себе:

- Ділення числа на нуль.
- Некоректні дані під час вводу з консолі.

У такому випадку викликається аварійна ситуація часу виконання програми.

Для виконання вихідного коду інтерпретуємої програми, інтерпретатор послуговується вже реалізованими можливостями мови, якою розроблена його реалізація. Це включає в себе операції над базовими типами і обробка вводу/виводу.

2.5 Висновок

У цьому розділі було розглянуто ключові етапи інтерпретації (лексичний, синтаксичний, семантичний аналізи і виконання) та підходи до їхньої реалізації. Було надане визначення інтерпретатора, описані відмінності інтерпретатора від компілятора і типи інтерпретаторів у залежності від представлення проміжної фази програми (абстрактне синтаксичне дерево і байт-код).

Також було визначено специфікацію мови програмування MPL, для якої власне і розробляється інтерпретатор. На основі специфікації були виведені синтаксичні правила у вигляді розширеної нотації Бекуса-Наура, і семантичні правила за допомогою структурної операційної семантики.

Був наданий опис деталей реалізації кожного з етапів інтерпретації, а також використаних взірців проєктування – компонувальника для структури АСД і відвідувача для семантичного аналізу і виконавця.

При розгляді реалізації етапу семантичної перевірки було показано роль коректної обробки правил операційної семантики, які не в змозі охопити контекстно-вільна граматика. На прикладі типізації і областей видимості змінних продемонстровано обробку контекстних умов, а також застосування таблиці символів.

РОЗДІЛ 3. РЕАЛІЗАЦІЯ ІНТЕРПРЕТАТОРА

3.1 Аналіз технічного завдання

Технічне завдання полягає у розробці інтерпретатора мови програмування за наданою специфікацією та сформованими контекстно-вільною граматикою та правилами структурної операційної семантики. Для реалізації задачі з розробки інтерпретатору були поставлені наступні підзадачі:

- Реалізація чотирьох головних складових інтерпретатора – лексичного, синтаксичного, семантичного аналізаторів і виконавця.
- Приймання на вхід програми у вигляді текстового файлу і вивід результатів виконання у консоль.
- Обробка помилок на кожному з етапів інтерпретації.
- Впровадження засобів автоматичного тестування і перевірки коректності роботи кожної із складових.
- Написання тестових програм розробленою мовою програмування.

Специфікація мови програмування наведена у Додатку А, разом з описом контекстно-вільної граматики у Додатку В і контекстних умов у Додатку С.

3.2 Обґрунтування вибору використаних технологій

Для написання реалізації інтерпретатора була обрана мова програмування C# на базі .NET 6.0. Цей вибір обумовлений наступними факторами:

- Об’єктно-орієнтовна парадигма мови C# спрощує розробку загальної архітектури інтерпретатора і реалізації взірців проєктування, які використовуються на етапах інтерпретації.

- Стандартна бібліотека мови C# надає зручні інструменти для роботи з вводом/виводом, а також для обробки тексту, що особливо корисно на етапі лексичного аналізу.
- Крос-платформеність фреймворку .NET 6.0 забезпечує запуск розробленого інтерпретатору на різних операційних системах, таких як Windows, Linux, MacOS тощо.

У якості інтегрованого середовища розробки було обрано Visual Studio 2022. Це зумовлено підтримкою мови C# і фреймворку .NET за замовчуванням, а також потужними можливостями зі зневадження і аналізу коду.

Для реалізації засобів автоматичного тестування був обраний фреймворк MSTest, розроблений спеціально для мови C#. Вибір MSTest обумовлений наступними причинами:

- Інтеграція з інструментами, що використовуються – MSTest був спеціально розроблений під мову C# з урахуванням використання у межах Visual Studio. Це значно спрощує процес інсталяції та подальше використання.
- Простота використання – MSTest має простий і зрозумілий синтаксис, який полегшує написання тестів. Це прискорює розробку і додатково забезпечує повного покриття тестами всіх компонентів інтерпретатора.
- Широка підтримка і документація – MSTest має широку підтримку з боку спільноти розробників та детальну документацію, що допомагає швидко вирішувати виникаючі питання під час розробки.

3.3 Структура проєкту

Внутрішню структуру реалізації інтерпретатора можна представити у вигляді наступної UML-діаграми:

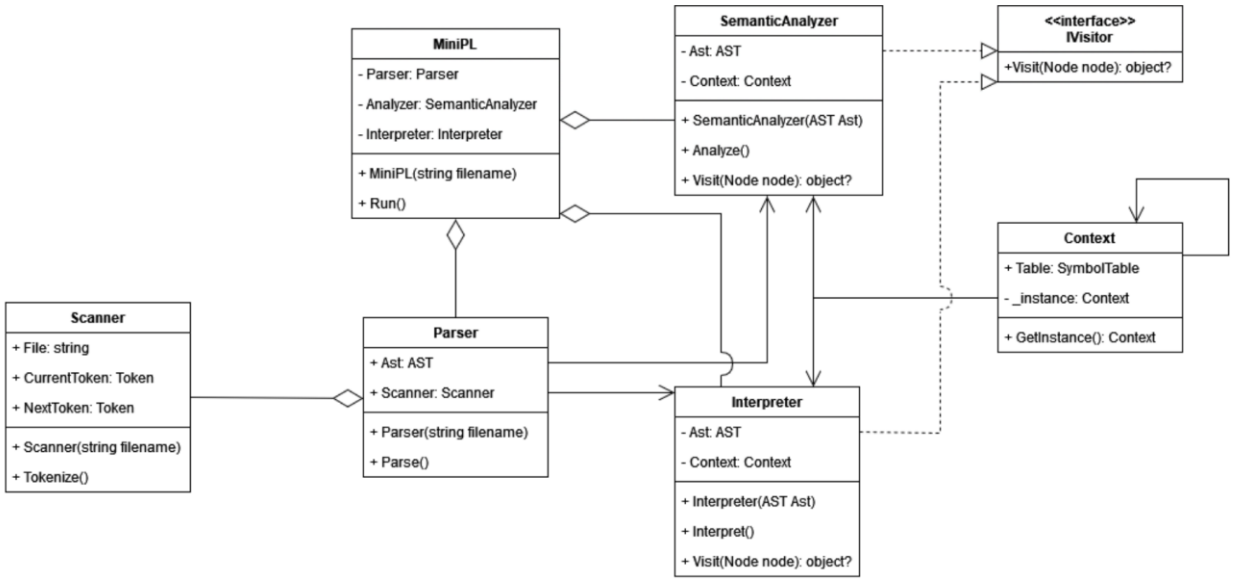

Рисунок 6. UML-діаграма інтерпретатора мови програмування

Враховуючи об'єктно-орієнтовану природу C#, було описано компоненти інтерпретатора згідно з відповідними класами в коді проекту:

- *Scanner*

Спеціалізована реалізація сканера, яка отримує на вхід шлях до файлу з вихідним кодом та генерує відповідні токени. При виклику методу Tokenize() сканер зчитує вихідний код програми і генерує токени, зіставляючи поточний символ з реалізованими правилами лексичного аналізу. Токени зберігаються в атрибутах CurrentToken і NextToken (для «підглядання» під час синтаксичного аналізу). Лексичний і синтаксичний аналізи проводяться одночасно, тобто лексер генерує токени за запитом парсера «на льоту».

- *Parser*

Реалізація синтаксичного аналізатора згідно з правилами EBNF граматики, наведеної у додатку В. При виклику методу Parse() генерується абстрактне синтаксичне дерево з використанням рекурсивного синтаксичного аналізу. Потім абстрактне синтаксичне дерево зберігається в окремому атрибуті для подальшого використання у семантичному аналізаторі та інтерпретаторі.

- *SemanticAnalyzer*

Клас, який реалізує інтерфейс IVisitor (взірець відвідувач) і виконує семантичний аналіз програми. Клас отримує абстрактне синтаксичне дерево, згенероване синтаксичним аналізатором, і при виклику методу Analyze() проходить по дереву для перевірки семантичної коректності програми згідно з правилами операційної семантики, наведеними у додатку С.

- *Interpreter*

Клас, який реалізує інтерфейс IVisitor і виконує програму, представлену у вигляді АСД вираз за виразом. Коли викликається метод Interpret(), виконавець обходить АСД, наданий синтаксичним аналізатором, і виконує кожен з вузлів-тверджень. Цей клас також обробляє ввід/вивід.

Для реалізації взірців компонувальник і відвідувач було оголошено два наступні інтерфейси:

- *IVisitor*

Інтерфейс для реалізації відвідувача у семантичному аналізаторі та виконавці. Він дещо відрізняється від канонічного шаблону відвідувача, оскільки зазвичай методи Visit() нічого не повертають, але було реалізовано повернення значень при відвідуванні вузлів виразів, операндів і токенів для отримання типу вузла (у семантичному аналізі) або значення (в інтерпретації).

- *INode*

Інтерфейс для реалізації компонувальника у синтаксичному аналізаторі. Визначає методи GetAllChildren() для доступу до дочірніх вузлів, Print() для виводу у консоль при зневадженні і Accept() для імплементацій відвідувача.

Для об’єднання усіх ключових компонентів інтерпретатора, а також для задання точки входу виконання було виділено наступні службові класи:

- *MPL*

Основний клас інтерпретатора, який об'єднує всі чотири структурні частини інтерпретатора мови MPL. Коли викликається метод Run(), він

послідовно виконує всі операції для виконання програми, включаючи синтаксичний розбір, семантичний аналіз, інтерпретацію та очищення таблиці символів після виконання. Цей клас також відповідає за обробку помилок.

- *Program*

Точка входу до програми, яка обробляє аргументи командного рядка і запускає інтерпретатор.

До того ж, наявні службові структури даних, які зберігають інформацію про стан програми, що виконується, і її компонентів при інтерпретації:

- *Context*

Клас, який реалізує шаблон одинак. Він зберігає таблицю символів і підтримує різні операції над збереженими змінними, включно з очищенням таблиці. Використовується у семантичному аналізаторі та інтерпретаторі.

- *SymbolTable*

Представлення таблиці символів для семантичного аналізу.

- *TableEntry*

Структура, яка зберігає тип та значення змінної.

- *Token*

Структура для представлення токена, яка зберігає тип, позицію та значення токена.

- *Position*

Структура для представлення позиції токена у вихідному коді, яка зберігає рядок і стовпець.

- *Node*

Представлення вузла абстрактного синтаксичного дерева.

На Рисунку 7 представлення діаграма похідних класів вузла абстрактного синтаксичного дерева, згенерована за допомогою Visual Studio Class Designer. Кожен тип вузла представляє певний тип граматичної структури мови MPL.

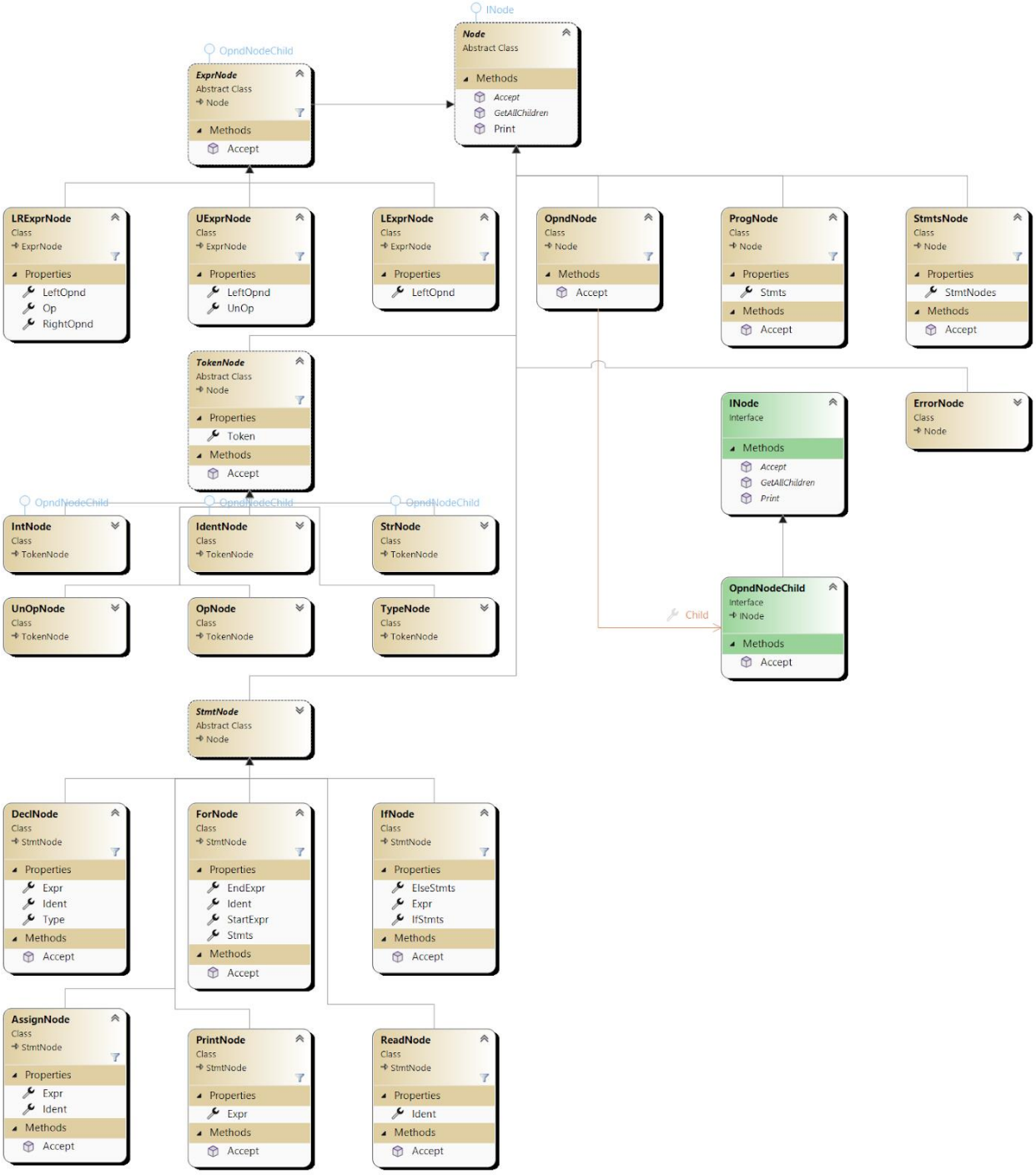

Рисунок 7. Діаграма класів вузлів абстрактного синтаксичного дерева

3.4 Перевірка правильності роботи інтерпретатора

Тестування передбачає перевірку коректності поведінки чотирьох основних структурних частин інтерпретатора. Тести можна розділити на дві основні категорії:

- Забезпечення коректності згенерованих структур та об'єктів при поданні на вхід коректних програм.
- Перевірка того, чи генеруються відповідні помилки при подачі невірних вхідних даних.

Проект MiniPLTests складається з 4 наборів тестів (по одному для кожного етапу), які містять наступні перевірки:

- *ScannerTest*

Перевірка зосереджена на правильному розпізнаванні токенів, наприклад, розпізнавання всіх допустимих токенів, обробка екрануючих символів всередині рядкових літералів, ігнорування пробілів і коментарів у вихідному коді, а також відсутність генерації помилок при обробці допустимих програм.

Інші тести включають обробку неправильного шляху до вихідного коду (генерується помилка FileNotFound), а також обробку інших пограничних випадків, таких як порожній файл, неприпустимі символи, нетермінований рядок або багаторядковий коментар, неприпустимі екрановані послідовності, нелегальні ідентифікатори або числові літерали, а також неприпустимий токен діапазону (генерується помилка LexicalError).

- *ParserTest*

Здійснюється перевірка, чи здатен синтаксичний аналізатор будувати синтаксичні дерева для заданих тверджень, чи може він обробляти складні вкладені вирази, чи здатен він будувати коректні синтаксичні дерева для заданої програми і чи може він обробляти помилки, пов'язані з побудовою синтаксичного дерева, які включають в себе недопустимий токен на початку твердження, пропущену крапку з комою, неочікувані токени тощо (Викликається помилка синтаксичного аналізу – ParseError).

Клас також містить допоміжний метод CompareTrees(), який порівнює AST, згенероване синтаксичним аналізатором, з оголошеним вручну деревом.

- *SemanticTest*

Здійснюється перевірка, чи правильно семантичний аналізатор додає значення до таблиці символів, чи може обробляти коректні програми і таблиці символів для них, а також виявляють різні помилки, такі як перевизначення змінних, використання неоголошених змінних і невідповідність типів в операціях присвоювання або виразах (генерується SemanticError).

- *InterpreterTest*

Здійснюється перевірка на правильність значеннь, які присвоюються змінним у таблиці символів, та чи здатен інтерпретатор коректно обчислювати складні вирази і відловлювати такі помилки, як ділення на нуль, використання неініціалізованих змінних, а також, якщо змінній типу int намагаються присвоїти рядок, який не можна привести до цілого типу (виникає RuntimeError).

Тести також емулюють консольний ввід і перенаправляють консольний вивід для подальших тестів (наприклад, для виконання тестів із заданими вхідними значеннями і перевірки правильності виведення результатів на консоль).

Усі тестові програми зберігаються у папці TestPrograms. Більшість вихідних файлів не є повноцінними програмами, а були створені для тестування конкретного варіанту використання і перевірки того, чи обробляється вся непередбачувана поведінка. Проте, підпапка ValidPrograms містить 5 коректних програм на мові MiniPL.

Всі тести успішно пройдені процесором мови, що можна побачити на Рисунку 8.

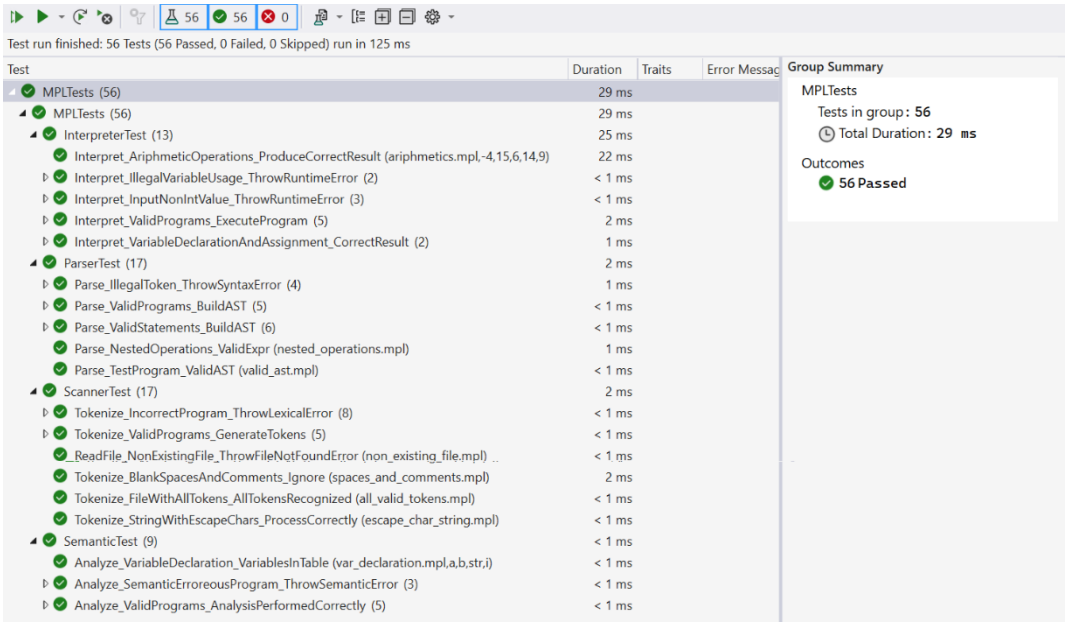

Рисунок 8. Демонстрація успішного виконання тестів інтерпретатора

Наявні два типи обробки аварійних ситуацій, які використовуються при інтерпретації:

- Відновлення у панічному режимі (panic-mode recovery).
- Відновлення у режимі тверджень (statement-mode recovery).

Відновлення у панічному режимі вживається у лексичному аналізі і виконавці. Воно полягає у негайному виклику аварійної ситуації, яка згодом оброблюється у класі MPL. Під час лексичного аналізу, помилка викликається при попаданні на некоректний символ чи при формуванні некоректного токену. У виконавці, помилка викликається під час неочікуваної поведінки часу виконання (наприклад, при діленні на нуль).

Відновлення у режимі тверджень використовується під час синтаксичного і семантичного аналізів. Воно полягає у формуванні списку помилок, і при виникненні аварійної ситуації помилка додається до списку, а поточний етап інтерпретації продовжує свою роботу. Після закінчення етапу, список помилок передається класу MPL і інтерпретація зупиняється.

```
SemanticError: Variable is not declared on line 1 column 5
for str in str..str do end for;
    ^
SemanticError: Variable is not declared on line 1 column 12
for str in str..str do end for;
           ^
SemanticError: Variable is not declared on line 1 column 17
for str in str..str do end for;
                ^
SemanticError: Variable is not declared on line 2 column 5
for i in (2 = 2)..10 do end for;
    ^
SemanticError: Variable type dismatch (expected int, got bool) on line 2 column 16
for i in (2 = 2)..10 do end for;
               ^
SemanticError: Variable type dismatch (expected int, got string) on line 4 column 6
x := "2";
     ^
```

Рисунок 9. Відновлення у режимі тверджень під час семантичного аналізу

3.5 Демонстрація роботи

Запуск програми мовою MPL виконується шляхом виклику файлу виконання інтерпретатора зі шляхом до файлу з вихідним програми MPL у якості аргументу запуску. Отримати файл виконання інтерпретатора можна отримати наступним чином:

```
dotnet build PATH_TO_SOURCE_CODE\MPL -o PATH_TO_OUTPUT_DIRECTRORY
```

де `PATH_TO_SOURCE_CODE` – шлях до директорії, де розміщений проєкт з вихідним кодом інтерпретатора MPL, а `PATH_TO_OUTPUT_DIRECTRORY` – шлях до директорії, де має опинитись сам файл виконання. Для побудови проєкту необхідно мати встановлений фреймворк .NET на комп’ютері.

Надалі сама програма запускається наступною командою:

```
PATH_TO_OUTPUT_DIRECTORY\MPL PATH_TO_SOURCE_FILE
```

де `PATH_TO_SOURCE_FILE` – шлях до файлу з вихідним кодом програми, написаною мовою MPL.

Також є можливість додати аргумент `-debug`, який перед виконанням програми виведе наявні токени і структуру абстрактного синтаксичного дерева у консоль.

```
VAR             var                         Ln: 1    Cl: 1
IDENTIFIER      X                           Ln: 1    Cl: 5
COLON           :                           Ln: 1    Cl: 7
INT             int                         Ln: 1    Cl: 9
ASSIGN          :=                          Ln: 1    Cl: 13
INT_LITERAL     4                           Ln: 1    Cl: 16
PLUS            +                           Ln: 1    Cl: 18
LPAREN          (                           Ln: 1    Cl: 20
INT_LITERAL     6                           Ln: 1    Cl: 21
MUL             *                           Ln: 1    Cl: 23
INT_LITERAL     2                           Ln: 1    Cl: 25
RPAREN          )                           Ln: 1    Cl: 26
SEMICOLON       ;                           Ln: 1    Cl: 27
PRINT           print                       Ln: 2    Cl: 1
IDENTIFIER      X                           Ln: 2    Cl: 7
SEMICOLON       ;                           Ln: 2    Cl: 8
EOF                                         Ln: 2    Cl: 9
ProgNode
  StmtsNode
    DeclNode
      IdentNode [X]
      TypeNode [int]
      LRExprNode
        OpndNode
          IntNode [4]
        OpNode [+]
        OpndNode
          LRExprNode
            OpndNode
              IntNode [6]
            OpNode [*]
            OpndNode
              IntNode [2]
    PrintNode
      LExprNode
        OpndNode
          IdentNode [X]
16
```

Рисунок 10. Демонстрація роботи інтерпретатора

3.6 Висновок

У даному розділі було розглянуто розробку інтерпретатора мови програмування MPL. Описані підзадачі, які були виконанні для досягнення поставленої мети, а також надане обґрунтування вибору використаних технологій, зокрема мови C# і фреймворку .NET для написання інтерпретатора, інтегрованого середовища розробки Visual Studio і фреймворку для тестування MSTest.

Було описано внутрішню структуру проєкту – класів, які відповідають за головні етапі інтерпретації, а також допоміжні класи і структури даних, які використовуються у процесі інтерпретації. Для наглядної демонстрації взаємодії між головними компонентами інтерпретатора були надані візуальні діаграми класів програми.

Також було детально описано підходи до обробки аварійних ситуацій і тестування кожного з компонентів – окремо і в поєднанні між собою, на прикладах як коректних, так і некоректних програм. Було доведено стабільність процесу інтерпретації та відсутність помилок при запуску синтаксично і семантично коректних програм.

Як результат було розроблено інтерпретатор мови MPL, здатний на надійний запуск програм з забезпеченням лексичного, синтаксичного і семантичного аналізу вихідного коду. На прикладі було показано процес запуску програми і результат виконання.

ВИСНОВКИ

Під час написання кваліфікаційної роботи було розглянуто основні підходи до формальної семантики мов програмування, зокрема структурну операційну семантику та її роль під час трансляції вихідного коду програм. Було продемонстровано переваги структурної операційної семантики перед іншими підходами, в наслідок чого вирішено користуватись саме нею. Задля демонстрації застосування контекстних умов у реалізації мови програмування було поставлено за мету розробити інтерпретатор мови програмування.

Для досягнення поставленої мети було розглянуто різні підходи до реалізації інтерпретаторів мов програмування, головні складові етапи інтерпретації і їхні способи втілення. Була визначена специфікація мови програмування MPL, згідно з якою була почата розробка інтерпретатора цієї мови.

За специфікацією спершу була визначена контекстно-вільна граматика у вигляді розширеної нотації Бекуса-Наура, і було продемонстровано, що синтаксичних правил не вистачає для покриття всіх аспектів специфікації. Як наслідок, були розроблені правила структурної операційної семантики для контекстних умов мови MPL у вигляді правил структурної операційної семантики, а також було показано їхню роль при розробці реалізації інтерпретатора – у тому числі при забезпеченні типізації та коректного використання змінних.

Як наслідок, був розроблений інтерпретатор мови програмування MPL із урахуванням попередньо визначених синтаксичних і семантичних правил. На прикладі розробленого інтерпретатора був запропонований ефективний підхід до обробки контекстних умов, зокрема застосування взірця відвідувач і символьної таблиці для зберігання поточного стану інтерпретації програми.

Таким чином, із застосуванням структурної операційної семантики у тандемі з лексичним і синтаксичним аналізом була досягнена мета роботи – коректне застосування контекстних умов при розробці стабільного

інтерпретатора. Надійність роботи виконавця була доведена тестуванням коду та демонстрацією його роботи на прикладі повноцінних програм, написаних мовою MPL.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Ray T. Syntax. Loyola Marymount University. 2024. URL: https://cs.lmu.edu/~ray/notes/syntax/
2. Hutton G. Programming language semantics: It's easy as 1,2,3. Journal of Functional Programming. 2023. Vol. 33. URL: https://doi.org/10.1017/s0956796823000072
3. Astarte T. The history of programming language semantics: an overview. Newcastle University. 2020. URL: https://eprints.ncl.ac.uk/268184
4. Floyd R. W. Assigning Meanings to Programs. Studies in Cognitive Systems (COGS, volume 14). 1967. URL: https://link.springer.com/chapter/10.1007/978-94-011-1793-7_4
5. Hoare C. A. R. An Axiomatic Basis for Computer Programming. The Queen's University of Belfast. 1969. URL: https://dl.acm.org/doi/pdf/10.1145/363235.363259
6. Pitts A. M. Lecture Notes on Denotational Semantics for the Computer Science Tripos, Part II. University of Cambridge Computer Laboratory. 2012. URL: https://www.cl.cam.ac.uk/teaching/1112/DenotSem/dens-notes-bw.pdf
7. Scott D. S. Outline of a mathematical theory of computation. Oxford, Oxford University Computing Laboratory, 1970. 24 p. URL: https://www.cs.ox.ac.uk/files/3222/PRG02.pdf
8. Slonneger K., Kurtz B. Programming Language Foundations. College of Liberal Arts & Sciences, The University of Iowa. 2004. URL: https://homepage.divms.uiowa.edu/~slonnegr/plf/Book/Chapter11.pdf
9. Grant M., Palmer Z., Smith S. Principles of Programming Languages. The Programming Languages Laboratory. 2016. URL: https://pl.cs.jhu.edu/pl/book/
10. Plotkin G. A Structural Approach to Operational Semantics. Computer Science Department, Aarhus University, Aarhus, Denmark. 1981. URL: https://emwww.github.io/home/teaching/immigration_course/plotkin_a_struct

ural_approach_to_operational_semantics.pdf

11. Plotkin G. The Origins of Structural Operational Semantics. J. Log. Algebr. Program. 2004. URL: https://homepages.inf.ed.ac.uk/gdp/publications/Origins_SOS.pdf
12. Scott M. L. Programming Language Pragmatics. Fouth Edition. Elsevier, 2016.
13. The Java® Virtual Machine Specification / T. Lindholm et al. Oracle Docs. 2013. URL: https://docs.oracle.com/javase/specs/jvms/se7/html/index.html
14. Dijkstra E. W. On the Design of Machine Independent Programming Languages. Stichting Mathematisch Centrum, Amsterdam. 1961. URL: https://www.cs.utexas.edu/users/EWD/transcriptions/MCreps/MR34.html
15. Welsh J., McKeag M. Structured System Programming. Queen's University of Belfast, Northern Ireland, Prentice-Hall International. 1980. URL: http://pascal.hansotten.com/uploads/welsh/Structured%20system%20programming.pdf
16. Myers A. Introduction to Compilers. Lexical Analysis and Regular Expressions. Cornell University. 2022. URL: https://www.cs.cornell.edu/courses/cs4120/2022sp/notes.html?id=lexing
17. Letichevskii A. A. A generalization of the concept of context-free grammar. Kibernetika. 1972.

ДОДАТОК А. СПЕЦИФІКАЦІЯ МОВИ ПРОГРАМУВАННЯ MPL

Мова MPL – спрощений варіант реалізації мови програмування Pascal, який містить базові вирази і твердження, арифметичні операції і ввід/вивід даних за допомогою консолі. MPL – інтерпретована мова, програми визначаються у окремих файлах типу «.mpl», для виконання програм інтерпретатору надається шлях до файлу з визначеною програмою.

Визначенні наступні типи даних:

1) int – цілочисельне значення.
2) bool – булевий тип, який може приймати значення true або false.
3) string – рядок символів.

Використовується загальна область видимості для усіх оголошених змінних, кожна змінна має бути оголошена перед її використанням, і тільки один раз.

Визначені дві операції вводу/виводу – зчитування (використовується для присвоєння цілочисельного чи рядкового значення змінними на етапі виконання програми) і вивід значень. Для операцій вводу/виводу використовується стандартна консоль мови, якою розроблена реалізація інтерпретатора.

Для змінних типу int визначенні наступні операції:

- int + int → int – додавання.
- int – int → int – віднімання.
- int * int → int – множення.
- int / int → int – ділення без остачі.
- int < int → bool – перевірка на те, чи перше значення *менше* за друге.
- int > int → bool – перевірка на те, чи перше значення *більше* за друге.

Для змінних типу string визначений наступна операція:

- string + string → string – конкатенація двох рядків.

Для змінних типу bool визначені наступні операції:

- bool & bool → bool – логічне «І».
- !bool → bool – логічне «НІ».

Для всіх типів даних визначена операція:

- type = type → bool – перевірка на однаковість даних.

Твердження for використовується для ітерації по визначеному ряду чисел. Початкове і кінцеве значення обчислюються тільки один раз, на початку циклу. У середині виразу наявний доступ до «змінної контролю», яка містить поточне значення ітерації. Це значення є сталим в межах одної ітерації і не може бути змінене. Воно оголошується за межами циклу і перед ним. Цикл приймає вигляд:

for *int* in *int*..*int* do *expressions* end for.

Твердження if використовується для розгалуження в залежності від значення виразу, що обчислюється на початку блоку. Цей вираз має зводитись до булевого значення, і у випадку зведення до true, вирази у середині блоку виконуються. У випадку зведення до false виконуються вирази у опціональному блоці else. Твердження приймає вигляд:

if *expression → bool* do *expressions* (else *expressions*) end if.

Допустимі ключові слова: var, for, in, end, if, else, read, print, int, bool, string.

ДОДАТОК В. КОНТЕКСТНО-ВІЛЬНА ГРАМАТИКА МОВИ MPL

```
<prog>         ::= <stmts>
<stmts>        ::= <stmt> <stmts_tail>
<stmts_tail>   ::= ";" <stmts_tail'>
<stmts_tail'>  ::= <stmt> <stmts_tail> | ε
<stmt>         ::= <decl> | <assign> | <for> | <read> | <print> | <if>
<decl>         ::= "var" <var_ident> ":" <type> <decl_tail>
<decl_tail>    ::= ":=" <expr> | ε
<assign>       ::= <var_ident> ":=" <expr>
<for>          ::= "for" <var_ident> "in" <expr> ".." <expr> "do"
                   <stmts> "end" for"
<read>         ::= "read" <var_ident>
<print>        ::= "print" <expr>
<if>           ::= "if" <expr> "do" <stmts> <if_tail>
<if_tail>      ::= "else" <stmts> "end" "if" | "end" "if"
<expr>         ::= <opnd> <expr'> | <unary_op> <opnd>
<expr'>        ::= <op> <opnd> | ε
<unary_op>     ::= "!"
<op>           ::= "+" | "-" | "*" | "/" | "&"
<opnd>         ::= <int> | <string> | <var_ident> | "(" <expr> ")"
<type>         ::= "int" | "string" | "bool"
<var_ident>    ::= <ident>
<reserved_keyword> ::= "var" | "for" | "end" | "in" | "do" | "read" |
                       "print" | “int" | "string" | "bool" | "assert" |
                       "if" | "else"
```

Пояснення нотації:

- `<...>` – нетермінальний символ (потребує подальшої заміни).
- `"..."` – термінальний символ (кінцевий).
- `::=` – напрямок підстановки.
- `|` – «АБО», вибір між одною з варіацій.
- `[ ... ]` – індикація опціональності.
- `( ... )*` – «Зірка Кліні», повторення символів 0 або більше разів.

ДОДАТОК С. ПРАВИЛА ОПЕРАЦІЙНОЇ СЕМАНТИКИ МОВИ MPL

1. Присвоєння значення змінній – змінна має бути оголошена.

$$\frac{\Gamma, \sigma \vdash \text{var } x : (\tau, \_) \quad \Gamma \vdash e : \tau}{\Gamma, \sigma \to \Gamma\big[x \longmapsto \big(\tau, \text{val}(e, \sigma)\big)\big], \sigma}$$

Γ – середовище типів, мапування змінних до їхніх типів.

σ – стан програми, мапування змінних до їхніх значень.

$\text{var } x : (\tau, \_)$ – оголошення змінної x з типом τ.

$\Gamma \vdash e : \tau$ – вираз e набуває типу τ.

$\to$ – операція оновлення середовища/стану.

$x \longmapsto \big(\tau, \text{val}(e, \sigma)\big)$ – присвоєння змінній x типу τ значення e.

2. Типізація – операнди виразу мають бути одного типу.

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \to e_1 \text{ op } e_2 : \tau}$$

$e_1 \text{ op } e_2 : \tau$ – певна операція між двома виразами типу τ зводиться теж до типу τ.

3. Твердження розгалуження – вираз умови має зводитись до булевого значення.

$$\frac{\Gamma \vdash e : \text{ bool}}{\Gamma, \sigma \to \text{if } e \text{ do } s_1 [\text{else } s_2] \text{ end if}}$$

s_1, s_2 – твердження.

Твердження розгалуження коректне при зведенні умови e до булевого типу. Квадратні дужки $[\text{else } s_2]$ показують на опціональність блоку else.

4. Змінна може бути викликана тільки в її області видимості.

$$\frac{\Gamma, \sigma \vdash \text{var } x : \tau \quad x \notin \text{dom}(\Gamma)}{\Gamma[x \longmapsto (\tau, \text{undefined})], \sigma}$$

$x \notin \mathrm{dom}(\Gamma)$ – змінна x відсутня у домені Г.

$x \mapsto (\tau, \mathrm{undefined})$ – виклик відсутньої у домені змінної повертає невизначене значення, що є некоректним при виконанні програми.

5. Твердження циклу – контрольна зміна і межі ітерації мають набувати цілочисельного значення.

$$\frac{\Gamma \vdash x : \mathrm{int} \quad \Gamma \vdash e_1 : \mathrm{int} \quad \Gamma \vdash e_2 : \mathrm{int}}{\Gamma, \sigma \to \mathrm{for}\ x\ \mathrm{in}\ e_1 .. e_2\ \mathrm{do}\ s\ \mathrm{end\ for}}$$

Контрольна змінна x і межі ітерації, представлені виразами e_1 і e_2 набувають цілочисельного значення, що є умовою для виконання твердження циклу.