Міністерство освіти і науки України

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра інформатики

**Магістерська робота**

освітній ступінь – магістр

на тему: **«Реалізація мови процедурного програмування типа Algol-60 засобами Haskell»**

Виконав: студент 2-го року навчання,

спеціальності

121 «Інженерія Програмного Забезпечення»

Пінкевич В.М.

Керівник

доцент к. н.

Проценко В.С.

_ травня 2025 р.

Київ – 2025

**Календарний план виконання роботи**

**Тема:** Реалізація мови процедурного програмування типа Algol-60 засобами Haskell

**Календарний план виконання роботи:**

| № п/п | Назва етапу кваліфікаційної роботи | Термін виконання етапу | Примітка |
|---|---|---|---|
| 1. | Отримання теми роботи | Вересень 2024 р. | |
| 2. | Отримання завдання на кваліфікаційну роботу | Жовтень 2024 р. | |
| 3. | Ознайомлення із літературою та джерелами за темою роботи | Жовтень - листопад 2024 р. | |
| 4. | Знайомство із засобами розробки системи | Листопад – грудень 2024 р. | |
| 5. | Розробка синтаксичного аналізатора | Січень – лютий 2025 р. | |
| 6. | Розробка семантичного аналізатора | Лютий - березень 2025 р. | |
| 7. | Розробка інтерпретатора | Березень – квітень 2025 р. | |
| 8. | Написання текстової частини роботи | Квітень - травень 2025 р. | |
| 9. | Перегляд кваліфікаційної роботи науковим керівником | Квітень - травень 2025 р. | |
| 10. | Внесення змін до роботи згідно з зауваженнями наукового керівника | Квітень - травень 2025 р. | |
| 11. | Захист кваліфікаційної роботи | Червень 2025 р. | |

Студент Пінкевич В.М.

Керівник Проценко В.С.

**Зміст**

**Анотація**

У роботі описується процес розробки інтерпретатора підмножини мови програмування Pascal. Коротко розглядається історія виникнення та особливості мов програмування сімейства ALGOL. Описується чому саме мову Pascal було обрано для побудови інтерпретатора, а також яку підмножину елементів мови Pascal було імплементовано. Проводиться огляд стандартних етапів компіляції та інтерпретації, принципів їх реалізації. На основі оглянутих етапів формується структура інтерпретатора, який розробляється. Детально описується етап розробки кожної з частин інтерпретатора відповідно до етапу інтерпретації, який вони реалізовують. Наводяться приклади програм, що можуть бути виконані розробленим інтерпретатором, та результати виконання описаних програм. Визначаються переваги та недоліки використання розробленого інтерпретатора, а також можливості його розширення для підтримки більшої кількості елементів мови програмування Pascal.

**Використані скорочення**

ALGOL (Algorithmic Language) – сімейство імперативних мов програмування, першою з яких є ALGOL 58, що була розроблена у 1958 році.

АСД (Абстрактне синтаксичне дерево) – структура даних, яка використовується для проміжного представлення програми при виконанні синтаксичного аналізу.

ЗА (Запис активації) – структура даних, яка використовується для зберігання інформації про стан блоку програми під час її виконання.

**Вступ**

Будь-яка мова програмування потребує компілятора або інтерпретатора для виконання програм, написаних цією мовою. Для багатьох мов програмування існує одразу декілька компіляторів або інтерпретаторів. Якщо значна кількість функціоналу мови програмування є стандартизованою, то компілятори відрізняються лише у незначних деталях, а декілька компіляторів потрібно для можливості виконання програми на різних платформах. Для мов програмування, стандарти яких не передбачають певних важливих функцій, таких як робота із введеннням-виведенням даних, різні компілятори створюються із метою розширити початковий функціонал мови та додати елементи, яких не вистачає. В такому випадку, одна й та ж програма може потребувати значних змін у вихідному коді для можливості використання її з декількома компіляторами. В той же час, якщо для мови існують лише компілятори але немає інтерпретаторів, для запуску навіть досить простих програм необхідно встановлювати та використовувати компілятор, що вимагає додаткових витрат часу на опанування навичок роботи із компілятором. Для таких випадків одним із можливих рішень є розробка інтерпретаторів, які можуть виконувати програму без необхідності її компіляції. Це зменшує кількість часу та зусиль, витрачених на встановлення компіляторів та навчання роботі з ними, і дозволяє зосередитись на розробці програм.

Об’єктом дослідження є вивчення принципів розробки та реалізації компіляторів та інтерпретаторів мов програмування, а також їх особливостей, пов’язаних зі специфічними аспектами конкретних мов програмування.

Предметом дослідження є створення інтерпретатора, який може виконувати програми, написані мовою програмування Pascal, при цьому відслідковуючи синтаксичні та семантичні помилки, які в інтерпретаторах зазвичай виявляються лише під час безпосереднього виконання програми.

Метою роботи є розробка інтерпретатора обраної підмножини мови програмування Pascal, який може виконувати програми, що складаються з елементів, які належать цій підмножині, без необхідності компіляції програм. Використання мови програмування Haskell дозволить дослідити як особливості функціональних мов програмування можуть бути використані для спрощення розробки інтерпретаторів та компіляторів.

Текстова частина роботи складається з п’яти розділів.

У першому розділі розглядаються основні особливості та спільні риси мов програмування сімейства ALGOL та ALGOL-подібних мов. Окрему частину розділу присвячено нововведенням у мові програмування Pascal порівняно з ALGOL 60 та причинам вибору саме цієї мови для розробки її інтерпретатора.

У другому розділі описуються обрані інструменти розробки інтерпретатора.

Третій розділ повністю описує процес розробки інтерпретатора. Спочатку розглядається загальна структура програм на Pascal. Після цього обирається підмножина елементів мови Pascal, що буде реалізована, та описується синтаксис програм, що можуть бути виконані розробленим інтерпретатором. Далі виділяються основні етапи роботи інтерпретатора і детально описується процес розробки кожного модуля, який реалізує певний етап виконання інтерпретатора.

У четвертому розділі наводяться приклади програм, що можуть бути виконані інтерпретатором, а також результати роботи відповідних програм.

У п’ятому розділі підсумовуються результати розробки інтерпретатора, визначаються переваги та недоліки його використання, можливості подальшого розширення його функціоналу для підтримки більшої кількості елементів мови Pascal, а також робляться висновки щодо виконаної роботи.

# Розділ 1. Мова ALGOL 60 та похідні від неї мови програмування

## 1.1 Мова програмування ALGOL 60

Сімейство мов ALGOL – сукупність імперативних мов програмування, яка складається з 3 мов програмування: ALGOL 58, ALGOL 60 та ALGOL 68. Хоча ALGOL 58 і є першою мовою цього сімейства, найбільш відомою є ALGOL 60, оскільки саме вона вплинула на велику кількість мов програмування, що були розроблені після неї. Зокрема, саме в ALGOL 60 вперше було використано блокову структуру програм та можливість визначення вкладених функцій. Проте значна кількість елементів ALGOL 60 була еволюцією ідей, запропонованих у ALGOL 58, тому спочатку необхідно описати особливості мови ALGOL 58.

ALGOL 58 – перша в сімействі ALGOL мова програмування. В своїй основі вона була схожою на інші тогочасні імперативні мови програмування (наприклад, Fortran). В ALGOL 58 було реалізовано базові типи даних, умовні оператори, цикли, процедури, оператор вибору. Проте ця мова мала декілька ключових відмінностей, які виділяли її серед інших тогочасних мов програмування. По-перше, ALGOL 58 мала частково формалізовані синтаксис і семантику та намагалась запровадити загальну логіку побудови операторів, на відміну від Fortran, граматика якого була описана природною мовою, а значна частина синтаксичних конструкцій та семантика залежали від конкретної реалізації компілятора. По-друге, в ALGOL 58 вперше було запроповано концепцію складеного оператора (“compound statement”), який може містити декілька простих або складених операторів, що можуть бути вкладеними на довільну глибину. В ALGOL 58 складені оператори могли використовуватись лише в потоці керування (“control flow”) та не створювали власної області видимості (“scope”). Ця концепція заклала основу для появи парадигми структурованого програмування. ALGOL 58 не була широко розповсюдженою мовою і мала обмежену кількість

компіляторів, які підтримували її, проте її основним досягненням стало створення основи, з якої почався розвиток сімейства мов ALGOL, зокрема ALGOL 60.

Мова ALGOL 60 була прямим нащадком ALGOL 58, вона розширювала та допрацьовувала велику кількість ідей, закладених в ALGOL 58. В ALGOL 60 вперше було використано форму Бекуса – Наура для опису синтаксису мови, а це "стало революційним кроком у проєктуванні мов програмування і створенні компіляторів" [1].

Концепцію складених операторів було розширено та введено систему блоків ("block"). Кожен блок складався з визначень ("declaration") та інструкцій ("statement") і міг містити вкладені блоки. Область видимості визначень обмежувалась блоком, в якому вони декларувались, та вкладеними блоками. Розбиття на блоки мало декілька переваг. По-перше, блоки дозволяли групувати оператори таким чином, що декілька пов'язаних операторів можна було розглядати як один оператор. По-друге, блоки обмежували область видимості змінних, що дозволяло використовувати змінну з одним і тим же іменем в декількох різних блоках. Використання блоків в майбутньому стане одним із основних принципів для парадигми структурованого програмування.

Завдяки реалізації системи блоків було значно розширено можливості функцій. Було реалізовано чіткий поділ на процедури (не повертають значення) та функції (повертають значення). Функції могли бути вкладеними, що дозволяло обмежити їх область видимості та зробити вихідний код програм більш чітким і структурованим. Крім передачі параметрів за значенням ("call by value") було додано можливість передачі параметрів за іменем ("call by name"). Це спосіб передачі параметрів функції, при якому значення параметра не обраховується при виклику функції. Натомість, в тілі функції значення параметра обчислюється кожен раз, коли він використовується. Також, ALGOL 60 мав повну підтримку рекурсивних функції та рекурсивних викликів (прямих і непрямих).

Одним з основних недоліків ALGOL 60 була відсутність стандартизованих можливостей введення-виведення. Їх реалізація повністю залежала від конкретного компілятора, а тому при розробці програм для різних платформ, програму необхідно було модифіковувати під конкретний компілятор.

Значна кількість мов програмування, що створювались після ALGOL 60, використовували ідеї, запропоновані в ній, або навіть використовували дуже схожу структуру побудови програм. Через сильну схожість на мови сімейства ALGOL та наявність великої кількості спільних характеристик такі мови почали називати ALGOL-подібними.

### 1.2 ALGOL-подібні мови програмування

ALGOL-подібними (або мовами, подібними до ALGOL) називають мови програмування, що успадкували основні ідеї, концепції та синтаксис від мови ALGOL 60. Серед мов, що відносяться до ALGOL-подібних, можна виділити наступні:

- Pascal (1970) — мова програмування зі строгою статичною типізацією, яка базувалась на парадигмі структурованого програмування та розроблялась як мова для вивчення програмування.
- Modula / Modula-2 (1975/1978) — мова, яка розвивала ідеї Pascal. Основною відмінністю від Pascal було додавання системи модулів.
- Ada (1980) — надійна мова програмування з підтримкою паралелізму. Однією з основних ідей мови є пошук якомога більшої кількості помилок у програмі на етапі компіляції щоб звести до мінімуму кількість можливих помилок часу виконання.

- Simula (1967) — перша мова програмування з підтримкою об'єктно-орієнтованої парадигми. Побудована на базі ALGOL 60 і вважається її розширенням.
- C (1972) — низькорівнева мова програмування, яка перейняла значну частину синтаксису та структурну модель ALGOL.
- C++, Java, C#, D, Swift, Rust — сучасні мови програмування, які надихались більш пізніми мовами, проте вважаються ALGOL-подібними, оскільки синтаксис та структура програм написаних цими мовами наслідують ALGOL.

Усі ALGOL-подібні мови мають низку спільних характеристик, серед яких:

- Синтаксична схожість — схожа структура запису операторів, виразів та блоків. Прикладом є використання ключових слів “begin” та “end” або символів “{“ та “}” для позначення початку та кінця блоків. Також для умовних операторів та циклів використовується суворий синтаксис (структура “if-then-else” для умовних операторів, ключові слова “for” та “while” для циклів).
- Блокова структура програм — програма складається з ієрархії блоків, кожен з яких має власну область видимості.
- Структурний підхід до побудови програм — використання умовних операторів та циклів замість операторів безумовного переходу (типу “goto”).
- Статична типізація.
- Наявність формально визначеної граматики (часто – з використанням форми Бекуса – Наура).

Оскільки ALGOL-подібні мови часто успадковують ідеї та концепції одна від одної та від попередніх мов, вони також підтримують одні й ті ж парадигми програмування:

- Імперативне програмування — передбачає, що програма описує чітку послідовність дій, які виконує комп’ютер.
- Структуроване програмування (основа якого якраз була закладена в ALGOL 58 та ALGOL 60) — передбачає чіткий поділ програми на підпрограми, блоки, а також використання циклів та умовних операторів замість операторів безумовного переходу.
- Модульне програмування (у Modula, Ada) — передбачає поділ програми на незалежні частини (модулі), які можуть розроблятись, змінюватись та компілюватись окремо від інших модулів.
- Об’єктно-орієнтоване програмування (у Simula, C++, Java) — передбачає побудову програм, основними одиницями яких є об'єкти і класи.

Серед ALGOL-подібних мов особливе місце займає мова Pascal. Вона намагалася зберегти більшість ідей та концепцій, успадкованих від ALGOL 60, в той же час розширюючи (а в деяких аспектах – спрощуючи) їх, що робило цю мову більш легкою та гнучкою у використанні. Саме це дозволило їй стати стандартним вибором для вивчення основ програмування.

### 1.3 Мова програмування Pascal

Pascal – імперативна статично типізована мова програмування, розроблена Ніклаусом Віртом у 1970 році як мова для вивчення програмування. Вона успадкувала основні особливості ALGOL 60, такі як блокова структура програм, статична типізація, акцент на використанні парадигми структурованого програмування, використання формальної граматики та форми Бекуса-Наура для опису синтаксису. Ніклаус Вірт описував причини створення мови Pascal таким чином: “Pascal був спроєктований як прямий нащадок ALGOL 60 і мав на меті заохочувати хороші практики програмування через структурне програмування та організацію даних.”. [2] Зміни, зроблені у Pascal, розширювали можливості

розробки програм, в той же час намагаючись зберегти переваги ALGOL 60. Серед основних та найбільш значних нововведень варто виділити наступні:

- Розширена типова система. Крім базових типів, доступних в ALGOL 60, у Pascal було додано складніші типи, такі як множини, переліки та діапазони.
- Наявність складених типів. У Pascal було додано масиви, записи та указники, яких не було в ALGOL 60, що дозволило розробляти більш складні та гнучкі програми.
- Спрощений синтаксис. Pascal мав чіткі правила іменування змінних, функцій, процедур та типів, що робило його хорошим вибором для вивчення основ програмування.
- Стандартизація введення-виведення. Стандартна бібліотека Pascal містить функції, процедури та типи для введення-виведення даних, на відміну від ALGOL 60, який не мав такого стандарту, а тому усі процеси введення-виведення залежали від реалізації конкретного компілятора.

Більша гнучкість та простота написання програм стали основними причинами, чому в ході роботи було вирішено розробляти саме інтерпретатор мови програмування Pascal. Ще однією причиною такого вибору є мала кількість наявних інтерпретаторів для Pascal, в той час як компіляторів для різних діалектів досить багато. Оскільки Pascal містить досить велику кількість типів, функцій та процедур у стандартній бібліотеці, було вирішено обрати найбільш необхідну для написання програм множину елементів Pascal, яку буде реалізовано в інтерпретаторі.

# Розділ 2. Інструменти розробки інтерпретатора

## 2.1 Мова програмування Haskell

Haskell — це функціональна, статично і строго типізована мова програмування з лінивою семантикою обчислень. Основними принципами розробки програм на Haskell є композиція “чистих” функцій та функцій вищого рівня для того щоб уникнути повторення коду та забезпечити його перевикористання. Ще однією перевагою Haskell є легкість роботи з абстраціями високого рівня, що “зменшує шаблонний код, покращує підтримку та дозволяє авторам компіляторів описувати трансформації декларативно”. [3] Особливостями Haskell, які спрощують розробку компіляторів та інтерпретаторів, є:

- Використання абстракцій високого рівня, завдяки яким можна легко описувати структури даних (наприклад, абстрактні синтаксичні дерева) та операції над ними у декларативній формі.
- Алгебраїчні типи даних, які чудово підходять для представлення синтаксичних конструкцій мови, таких як умовні оператори, вирази та цикли.
- Зіставляння із взірцем (“Pattern matching”), яке значно спрощує роботу із АСД і дозволяє писати лаконічний код для його обробки.
- Вбудовані монадичні структури (наприклад, “Either” та “Maybe”), які дозволяють зберігати та модифікувати стан виконання і помилки виконання.
- Типова безпека та статична типізація, які “полегшують аналіз і верифікацію правильності інтерпретаторів та компіляторів”. [4, с. 365]

## 2.2 Бібліотека Parsec

Parsec — це бібліотека для побудови синтаксичних аналізаторів мовою Haskell. Основною концепцією бібліотеки є використання комбінарів, тобто використання простих парсерів для побудови більш складних парсерів. Такий

підхід “дозволяє програмістам створювати парсери за допомогою комбінаторів у модульний і декларативний спосіб, що забезпечує як читабельність, так і повторне використання коду”. [4, с. 377] Серед переваг цієї бібліотеки найбільш важливими є:

- Декларативний стиль, завдяки якому код синтаксичних аналізаторів дуже нагадує граматику мови, яку вони розпізнають.
- Зручна система обробки помилок. Parsec надає точну інформацію про помилки, яка включає повідомлення про помилку та позицію елементу вхідного потоку, при обробці якого сталась помилка. Це значно полегшує діагностику синтаксичних проблем.
- Підтримка бектрекінгу. Бібліотека дозволяє повертатись до попереднього елемента якщо обробка наступного закінчилась невдачею, завдяки чому можна зручно обробляти неоднозначні граматики.
- Ефективність та простота використання. Parsec не є найшвидшою бібліотекою для синтаксичного аналізу (наприклад, вона програє бібліотеці Attoparsec, основною перевагою якої є швидкість роботи), проте баланс між швидкістю, читабельністю і кількістю можливостей робить її чудовим вибором для написання парсерів мов програмування.

### 2.3 The Haskell Tool Stack

Stack (The Haskell Tool Stack) — це інструмент для управління проєктами у Haskell, який спрощує їх створення, побудову, тестування та розгортання. Він “забезпечує відтворювану збірку, автоматичне керування залежностями та ізоляцію середовища, що робить його надійним інструментом для професійної розробки на Haskell”. [5] Серед інших особливостей Stack варто виділити наступні:

- Простота встановлення та використання. Наприклад, він автоматично встановлює GHC (компілятор Haskell) та всі залежності, необхідні для проєкту.
- Вбудована підтримка тестування і документації. Stack дозволяє зручно запускати юніт-тести, генерувати документацію та перевіряти якість коду.
- Підтримка в інтегрованих середовищах розробки. Stack добре інтегрується з Haskell Language Server та має розширення для Visual Studio Code, яке забезпечує підтримку автодоповнення, перевірку типів і підказки.

Обрані інструменти спрощують розробку інтерпретатора та роблять її більш зручною. Мова Haskell є надійною основою для побудови інтерпретатора, бібліотека Parsec значно пришвидшує процес розробки синтаксичного аналізатора, Stack дозволяє уникнути проблем із залежностями, їх конфліктами та версіями, а також спрощує процес тестування та збірки інтерпретатора в фінальний виконуваний застосунок.

# Розділ 3. Розробка інтерпретатора

## 3.1 Структура програм на Pascal

Програми на Pascal мають чітко визначену структуру. Будь-яка програма складається із блоків, вкладених один в одного. Блоки поділяються на 3 типи:

1. Програмний блок – основний блок програми. Кожна програма повинна обов'язково містити лише один програмний блок. Програмний блок є кореневим блоком програми і всі інші блоки повинні бути вкладеними в нього.
2. Блок функції – блок, який створюється при визначенні функції. Крім визначень із секції визначень, він також містить визначення параметрів функції (вони вважаються окремими змінними) та спеціальної змінної, яка відповідає за повернення значення функції.
3. Блок процедури – блок, який створюється при визначенні процедури. Виконує ту ж роль, що й блок функції, проте не містить змінної для повернення значення, бо процедури у Pascal не повертають значення.

Блоки поділяються на 2 секції: секція визначень і секція виконання.

У секції визначень відбувається оголошення змінних і визначення функцій та процедур що будуть використовуватись у блоці. Послідовність оголошень і визначень є важливою, оскільки змінні, функції та процедури можуть бути використані лише після того, як вони були оголошені або визначені. Секція визначень у вихідному коді повинна знаходитись перед секцією виконання, бо якщо розмістити її після секції виконання, то інструкції з секції виконання не зможуть використати змінні, функції та процедури, розміщені в секції визначень.

У секції виконання описується послідовність інструкцій, що мають бути виконані. Інструкції можуть використовувати змінні, функції та процедури із секції оголошення поточного блоку та усіх батьківських блоків, але не можуть

бути використані елементи із блоків, вкладених у поточний блок. Прикладами інструкцій є оператори присвоєння, умовні оператори, цикли.

Функції та процедури – це підпрограми, які містять список інструкцій, що необхідно виконати. Вони потрібні щоб уникнути повторення коду у різних місцях програми. Натомість, цей код можна помістити у функцію чи процедуру, уникаючи дублювання. Функції відрізняються від процедур тим, що вони можуть повертати значення, а процедури – ні.

Визначення будь-якої функції чи процедури створює новий блок відповідного типу. Це дає можливість створювати всередині функцій та процедур вкладені функції та процедури, які можуть використовуватись лише у межах батьківського блоку. Завдяки цьому немає потреби оголошувати усі використані в програмі елементи всередині програмного блоку, можна оголосити їх в блоці, де вони будуть використовуватись. Це дозволяє зменшити складність програми та уникнути проблем із уже зайнятими ідентифікаторами, коли декільком змінним, функціям чи процедурам потрібно надати однакову назву.

### 3.2 Вибір підмножини мови Pascal, яку буде реалізовано

Оскільки Pascal має велику кількість можливостей (різні типи даних, види інструкцій, операцій, тощо) при розробці інтерпретатора було вирішено обрати певну підмножину елементів Pascal, які будуть реалізовані. Це дозволить зберегти можливість створення великої кількості програм та спростити їх написання завдяки меншій кількості доступних елементів.

Найбільше змін відбулося із типами даних, оскільки було вирішено залишити лише вбудовані типи даних, які найчастіше використовуються. Pascal має як певну кількість вбудованих типів даних, так і можливість створювати власні типи даних. Усі типи даних можна поділити на 3 групи:

1. Прості (“Simple”) – типи даних, які не мають структури, а представляють лише набори значень. Прості типи поділяються на 2 групи:
   1.1. Порядкові (“Ordinal”) – типи даних, кількість елементів яких є скінченною. Більшість вбудованих типів даних є порядковими. Серед порядкових типів даних найбільш вживаними є:
      1.1.1. Тип цілих чисел (“Integer”).
      1.1.2. Символьний тип (“Char”).
      1.1.3. Булевий тип (“Boolean”). Булевий тип також є підтипом переліків. Переліки – це типи даних, кожен елемент яких чітко визначається власною іменованою константою.
   1.2. Дійсний (“Real”) – вбудований тип даних, який представляє дійсні числа.
2. Указники (“Pointers”) – тип даних, значенням змінних якого є адреса пам’яті, яка може вказувати на елемент іншого типу.
3. Структуровані (“Structured”) – типи даних, які можуть містити більше ніж один елемент певного типу. Вони називаються структурованими, оскільки вони “будуються” із інших типів даних. Прикладами структурованих типів даних у Pascal є масиви, множини, записи.

З усіх перелічених типів даних було вирішено реалізувати прості типи даних, які є вбудованими в Pascal: цілі числа, дійсні числа, символи та булевий тип. Для спрощення операцій введення та виведення було реалізовано підтримку стрічкового типу даних (у Pascal він має назву “String”).

Інструкції у Pascal можна поділити на декілька груп:

1. Оператори присвоєння – використовуються для присвоєння значення змінним. Базовий оператор присвоєння позначається послідовністю символів “:=”. Також доступні 4 додаткові оператори присвоєння (наприклад “+=” та “-=”), які поєднують базовий оператор присвоєння із відповідною математичною операцією.

2. Умовні оператори – використовуються для виконання інструкцій залежно від заданої умови (або умов). Перший умовний оператор - “if … then … else”. Він перевіряє умову, яка задається після ключового слова “if”. Якщо умова істинна – виконується інструкція задана після ключового слова “then”. Якщо умова хибна – виконується інструкція задана після ключового слова “else”. Частина “else” є опційною, якщо вона відсутня, то при хибності умови програма просто продовжує виконання. Другий умовний оператор – “case … of”. Він дозволяє виконати інструкції залежно від значення виразу, вказаного після ключового слова “case”. Після ключового слова “of” повинен знаходитись список пар “значення - інструкція”, який визначає, яка інструкція повинна бути виконана в залежності від значення виразу.
3. Цикли – використовуються для виконання інструкцій повторно поки умова є дійсною. Перший цикл – “while … do”. На кожній ітерації він перевіряє умову, задану після ключового слова “while”, причому умова повинна мати булевий тип. Якщо умова істинна - виконується тіло циклу, задане після ключового слова “do”, після цього починається нова ітерація. Якщо умова хибна – виконання циклу закінчується і продовжують виконуватись інструкції, що слідують за циклом. Оскільки умова циклу перевіряється до виконання тіла циклу, то цикл “while” може виконатись 0 або більше разів. Другий цикл - “repeat … until”. Він працює як цикл “while”, проте має ключову відмінність – умова циклу перевіряється після виконання тіла циклу, тому цикл “repeat” виконується 1 раз або більше. Третій цикл – “for … do”. Він відрізняється від інших циклів тим, що його тіло виконується фіксовану кількість разів, тому його умова не є булевою і має окремий тип.
4. Складений оператор “begin … end” – спеціальний оператор, який використовується, якщо потрібно щоб програма розглядала певний набір інструкцій як одну інструкцію. Він використовується щоб мати можливість

вказати декілька інструкцій для виконання в інших операторах: “if”, “case” та “while”.

5. Виклик процедури. Процедури у Pascal не повертають значення, а тому виклик процедури не може бути використаний як вираз і повинен бути окремою інструкцією.
6. Оператор безумовного переходу “goto” – дозволяє із будь-якого місця програми перейти до іншого місця програми, позначеного спеціальною міткою.

З усіх інструкцій було вирішено реалізувати підтримку базового оператора присвоєння “:=”, умовного оператора “if”, циклів “while” та “repeat”, складеного оператора та виклик процедури. Усі можливості оператора “case” можна реалізувати з використанням оператора ”if”. Схожим чином, цикл “for” можна замінити циклом “while” без втрати можливостей.

Вирази у Pascal будуються з використанням базових математичних операцій, логічних операцій, операцій порівняння та унарних операцій. Усі вони є базовими для роботи з типами даних, що будуть реалізовані, тому жодних операцій не було вилучено.

Блокова побудова програм є основою Pascal, а тому усі особливості блоків, такі як вкладеність та обмеження області видимості також будуть реалізовані. Це дозволяє також реалізувати можливість рекурсивних викликів функцій та процедур.

### 3.3 Синтаксис Pascal

Після визначення підмножини елементів Pascal, яка повинна бути реалізована, потрібно описати синтаксис мови, опираючись на структуру програм на Pascal. Синтаксис Pascal описується контекстно-вільною граматикою з

використанням форми Бекуса-Наура у стандарті ISO 7185 [6]. Синтаксис програм, що можуть бути виконані розробленим інтерпретатором, є видозміненою версією синтаксису, описаного у стандарті, з якого було вилучено елементи, що не підтримуються інтерпретатором.

### 3.3.1 Синтаксис програми

*<program> ::=* ***program*** *<identifier> ; <block> .*

*<identifier> ::= <letter > {<letter or digit>}*

*<letter or digit> ::= <letter> | <digit> | _*

Будь-яка програма на Pascal починається із заголовка. Він складається з ключового слова *program* за яким слідує ідентикафітор (він визначає назву програми). Ідентифікатор – це послідовність літер, чисел та символів “_”, яка починається літерою. Після заголовку повинен знаходитись основний блок програми, який завершується крапкою. Заголовок та основний блок розділяються символом “;”.

### 3.3.2 Синтаксис блоку

*<block> ::= <variable declaration part> <procedure and function declaration part> <statement part>*

*<variable declaration part> ::= <empty> |* ***var*** *<variable declaration> {; <variable declaration>} ;*

*<procedure and function declaration part> ::= {<procedure or function declaration > ;}*

*<procedure or function declaration > ::= <procedure declaration > | <function declaration >*

*<statement part> ::= <compound statement>*

Кожен блок складається із секції визначень та секції виконання. Визначення змінних, функцій та процедур не є обов'язковим, в той час як наявність секції виконання є обов'язковою. Визначення можуть розташовуватись у довільному порядку, проте порядок визначень впливає на можливість використання визначених елементів, оскільки елементи не можуть бути використані до того, як були визначені. Таким чином, будь-які визначення, що знаходяться після секції виконання, не будуть доступні для використання у цій секції. Секція виконання повинна бути складеним оператором.

### 3.3.3 Синтаксис визначення змінних

*<variable declaration part> ::= <empty> |* ***var*** *<variable declaration> {; <variable declaration>} ;*

*<variable declaration> ::= <identifier> {,<identifier>} : <type>*

*<type> ::= Boolean | Char | String | Integer | Real*

Визначення змінних починається ключовим словом var, після якого знаходиться одне або більше оголошень змінних, розділених символом ";". Оголошення змінних складається із одного або більше ідентифікаторів, розділених комою. Після останнього ідентифікатора знаходиться символ ":", за яким слідує назва типу змінної. Оскільки в інтерпретаторі список доступних типів є чітко визначеним і можливість створення власних типів відсутня, тип повинен бути одним із 5 доступних типів. Така структура дозволяє уникнути необхідності створювати декілька окремих оголошень для декількох змінних однакового типу,

оскільки можна згрупувати змінні і одним оголошенням створити декілька змінних одного типу.

### 3.3.4 Синтаксис визначення функцій

*<function declaration> ::= <function heading> <block>*

*<function heading> ::=* ***function*** *<identifier> ( <formal parameter section> {;<formal parameter section>} ) : <type> ;*

*<formal parameter section> ::= <identifier> {, <identifier>} : <type>*

Визначення функцій поділяється на 2 частини – заголовок і тіло функції. Заголовок починається ключовим словом *function*, після якого знаходиться назва функцій, яка є ідентифікатором. За назвою слідує список параметрів розділених символом “;”. Він починається і закінчується символами “(” та “)” відповідно. Список параметрів може містити 0 або більше параметрів. Як і при визначенні змінних, якщо декілька параметрів мають однаковий тип, їх можна перелічити через кому і вказати спільний тип після символу “:”. Після списку параметрів знаходиться символ “:”, після нього – тип значення, яке повертає функція. Заголовок функції закінчується символом “;”. Тіло функції представляє собою блок, тому функція може мати вкладені змінні, функції та процедури.

### 3.3.5 Синтаксис визначення процедур

*<procedure declaration> ::= <procedure heading> <block>*

*<procedure heading> ::=* ***procedure*** *<identifier> ( <formal parameter section> {;<formal parameter section>} );*

Визначення процедур майже повністю повторює визначення функцій, але має 2 ключові відмінності. По-перше, заголовок процедури починається ключовим

словом *procedure*. По-друге, оскільки процедури не повертають значення, то у заголовку після списку параметрів не потрібно вказувати тип результату. Тому заголовок процедури завершується символом “;” одразу після списку параметрів. Усі інші особливості визначення функцій повторюються для визначення процедур.

### 3.3.6 Синтаксис інструкцій

*<statement> ::= <simple statement> | <structured statement>*

*<simple statement> ::= <assignment statement> | <procedure statement> | <empty statement>*

*<assignment statement> ::= <variable> := <expression>*

*<variable> ::= <identifier>*

*<procedure statement> ::= <procedure identifier> (<actual parameter> {, <actual parameter> })*

*<actual parameter> ::= <expression> | <variable>*

*<structured statement> ::= <compound statement> | <if statement> | <repetitive statement>*

*<compound statement> ::=* ***begin*** *<statement> {; <statement> }* ***end;***

*<if statement> ::=* ***if*** *<expression>* ***then*** *<statement> |* ***if*** *<expression>* ***then*** *<statement>* ***else*** *<statement>*

*<repetitive statement> ::= <while statement> | <repeat statement>*

*<while statement> ::=* ***while*** *<expression>* ***do*** *<statement>*

*<repeat statement> ::=* ***repeat*** *<statement> {; <statement>}* ***until*** *<expression>*

Більшість інструкцій у Pascal будуються з інших інструкцій та виразів, розміщених між ключовими словами.

Порожня інструкція зазвичай використовується в більш складних інструкціях, якщо при певних умовах не потрібно виконувати ніяких дій.

Оператор присвоєння, складається із назви змінної, якій потрібно присвоїти значення, послідовності символів “:=” та виразом, який представляє значення змінної.

Оператор виклику процедури складається із назви процедури та списку параметрів, розділених комами, які знаходяться між символами “(” та “)”. Параметрами процедури можуть бути як змінні, так і вирази.

Умовний оператор та цикли використовують вираз, що повертає булеве значення, як умову та інші інструкції для виконання дій залежно від умови.

Складений оператор використовується якщо при виконанні умовного оператора чи циклу “while” потрібно замість однієї інструкції вказати декілька.

### 3.3.7 Синтаксис виразів

*<expression> ::= <simple expression>* | *<simple expression> <relational operator> <simple expression>*

*<relational operator> ::=* = | <> | < | <= | >= | >

*<simple expression> ::= <term>* | *<sign> <term>*| *<simple expression> <adding operator> <term>*

*<sign> ::=* + | -

*<adding operator> ::=* + | - | ***or***

*<term> ::= <factor>* | *<term> <multiplying operator> <factor>*

*<multiplying operator> ::=* * | / | ***div*** | ***mod*** | ***and***

*<factor> ::= <variable>* | *<unsigned constant>* | *( <expression> )* | *<function designator>* | ***not*** *<factor>*

*<function designator> ::= <function identifier ( <actual parameter> {, <actual parameter>} )*

*<function identifier> ::= <identifier>*

*<unsigned constant> ::= <unsigned number> | <string>*

*<unsigned number> ::= <unsigned integer> | <unsigned real>*

*<unsigned integer> ::= <digit> {<digit>}*

*<unsigned real> ::= <unsigned integer> . <unsigned integer>*

*<string> ::= '<character> {<character>}'*

Вирази у Pascal будуються із значень базових типів, унарних та бінарних операцій, викликів функцій та операції “()”, яка підвищує пріоритет виконання виразу. Пріоритет операцій у Pascal визначається синтаксисом.

Цілі числа задаються послідовністю літер, дійсні числа – двома цілими числами, розділеними крапкою (ціла та дробова частини дійсного числа), булеві значення – константами “true” та “false”, символи – власними літералами в лапках, стрічки – послідовністю символів в лапках (також стрічки можуть бути порожніми).

Виклик функції виконується так само як і виклик процедури: вказується назва функції та список параметрів, розділених комами, які знаходяться між символами “(” та “)”.

Найвищий пріоритет мають операції виклику функцій, операція “()”, а також унарні оператори “+”, “-“ та “not”. Наступними є операції множення, ділення, цілочисельного ділення, остачі та кон'юнкції. Після них виконуються операції додавання, віднімання та диз’юнкції. Найменш пріоритетними є операції

порівняння. Якщо в одному виразі використовуються декілька операцій з однаковим пріоритетом, вони виконуються зліва направо.

### 3.4 Загальна структура інтерпретатора

Загальні принципи побудови компіляторів та інтерпретаторів для різних мов програмування досить схожі між собою (типові етапи зображено на рисунку 3.4.1) і можуть бути використані для побудови власних компіляторів та інтерпретаторів.


*Рисунок 3.4.1. Типові етапи компіляції*

Зазвичай компіляцію чи інтерпретацію програми поділяють на послідовні етапи, причому результати попереднього етапу використовуються для виконання наступного етапу. Найбільш часто використовуваними етапами є:

1. Лексичний аналіз. Лексичний аналізатор виконує поділ вихідного коду програми на окремі елементи (токени), які мають значення для програми, в той же час відкидаючи непотрібні частини (наприклад, якщо синтаксис мови не вимагає певної кількості відступів чи символів табуляції між елементами). Токени поділяються на групи: числові константи, ідентифікатори, спеціальні символи, оператори, тощо. Визначення токенів та їх типів залежить від синтаксису мови програмування, оскільки для різних мов одна й та ж послідовність символів у вихідному коді може мати зовсім різне значення. Результатом лексичного аналізу є потік токенів.
2. Синтаксичний аналіз. Синтаксичний аналізатор “використовує токени, розпізнані лексичним аналізатором, для створення деревоподібного проміжного представлення, яке описує граматичну структуру потоку токенів ”. [7, с. 8] Якщо під час перевірки потоку токенів помилок не було виявлено, синтаксичний аналізатор будує АСД, яке використовується наступними етапами компілятора чи інтерпретатора. Якщо ж в ході аналізу токенів було знайдено помилку, синтаксичний аналізатор завершує роботу і повертає помилку компіляції. Зазвичай помилка містить інформацію про конкретний токен (або декілька токенів), які викликали помилку, та їх позицію у вихідному файлі програми.
3. Семантичний аналіз. Семантичний аналізатор отримує на вхід АСД і “використовує його для перевірки семантичної констистентності вихідної програми із визначенням мови програмування”. [7, с. 8] По-

перше, перевіряється існування змінних, функцій та інших елементів, які використовуються у виразі чи інструкції. Якщо програма має більше ніж одну область видимості, перевіряється доступність певного елемента в поточній області видимості. Інформація про змінні, функції та їх область видимості може зберігатись в тому ж АСД. Проте часто для цього використовується окрема структура даних, яка називається таблицею імен. Вона дозволяє співставити назву елемента із інформацією про нього. Для змінної це може бути її тип, для функції – інформація про кожен параметр, його назва і тип, та тип результату функції. По-друге, перевіряється відповідність типів змінних та виразів. Виконується перевірка типів даних при використанні операцій у виразах, присвоєнні значень змінним, передачі параметрів функціям. Якщо операції можуть застосовуватись для різних типів, перевіряється що типи є сумісними і обраховується тип результату виконання операції. Процес перевірки типів схожий на процес виконання програми, тільки замість конкретних значень (наприклад, додавання цілого та дійсного чисел) перевіряється, що операція додавання передбачає додавання значень цілого та дійсного типу і обраховується результат додавання значень цього типу. Результатом роботи семантичного аналізатора може бути доповнене АСД, звичайне АСД та символьна таблиця, цілком нова, окрема структура даних, яка будується на етапі семантичного аналізу або помилка, якщо елемент не було знайдено в області видимості або було використано неправильний тип даних у виразі.

4. Генерація проміжного коду. На цьому етапі вихідні дані, отримані від етапу семантичного аналізу, перетворюються на проміжне представлення програми. Воно може бути ніяк не пов’язаним із початковим представленням мови програмування. Часто проміжне

представлення є набагато простішим ніж початкове, тому воно краще підходить для виконання оптимізації чи генерації фінального представлення програми. Проміжне представлення програми є результатом виконання цього етапу. Цей етап є необов’язковим, зазвичай він використовується при розробці компіляторів, які повинні підтримувати велику кількість платформ, оскільки “проміжне представлення дозволяє підтримувати декілька вихідних мов програмування та платформ з меншою кількістю необхідних зусиль”. [8, с. 13]

5. Оптимізація. На цьому етапі “вихідна програма може бути замінена іншою, яка має таку ж семантику, але імплементує її більш ефективно”. [8, с. 14] Таким чином, оновлена програма працює так само, як і вихідна, але виконує меншу кількість операцій і є більш швидкою. Прикладами оптимізації є заміна виразів, які можуть бути обрахованими на етапі компіляції, їхніми значеннями, щоб уникнути необхідності робити одні й ті ж операції щоразу під час виконання програми, або пошук змінних, які ніяк не використовуються в програмі, та вилучення їх з програми. Зазвичай оптимізації досить сильно залежать від мови програмування, для якої вони виконуються, тому цей етап є специфічним для кожного окремого компілятора. Результатом оптимізації є оновлена програма. Цей етап також є необов’язковим.
6. Генерація фінального коду. Цей етап є останнім, він використовує результати або семантичного аналізатора, або проміжне представлення програми (якщо компілятор має етап генерації проміжного коду) і “перетворює його на представлення вихідної програми цільовою мовою”. [7, с. 10] Цей етап також сильно

відрізняється залежно від компілятора, оскільки повністю опирається на обране фінальне представлення програми.

Опираючись на описані вище етапи роботи компілятора та інтерпретатора було вирішено дещо спростити структуру інтерпретатора, який буде розроблено, і залишити 3 етапи (їх зображено на рисунку 3.4.2):

1. Синтаксичний аналіз. Цей етап об’єднано із етапом лексичного аналізу через те, що бібліотека Parsec, яка використовується для побудови синтаксичного аналізатора, надає зручні інструменти для перетворення вихідного коду одразу в АСД, без необхідності спочатку створювати потік токенів. Результатом цього етапу є АСД.
2. Семантичний аналіз. Оскільки Pascal – це статично типізована мова, яка складається з блоків з різними областями видимості, для коректної роботи програми необхідно перевірити правильність типів при побудові виразів та доступність елементів в конкретній області видимості. Для виконання семантичного аналізу буде використано АСД, отримане від синтаксичного аналізатора, а також додатково буде побудовано таблицю імен, яка потрібна для перевірки правильності використання змінних, функцій та процедур.
3. Інтерпретація. На цьому етапі побудоване АСД буде використовуватись для виконання програми. Оскільки всі можливі перевірки виконуються на етапі семантичного аналізу, на етапі інтерпретації можуть виникати лише помилки часу виконання.

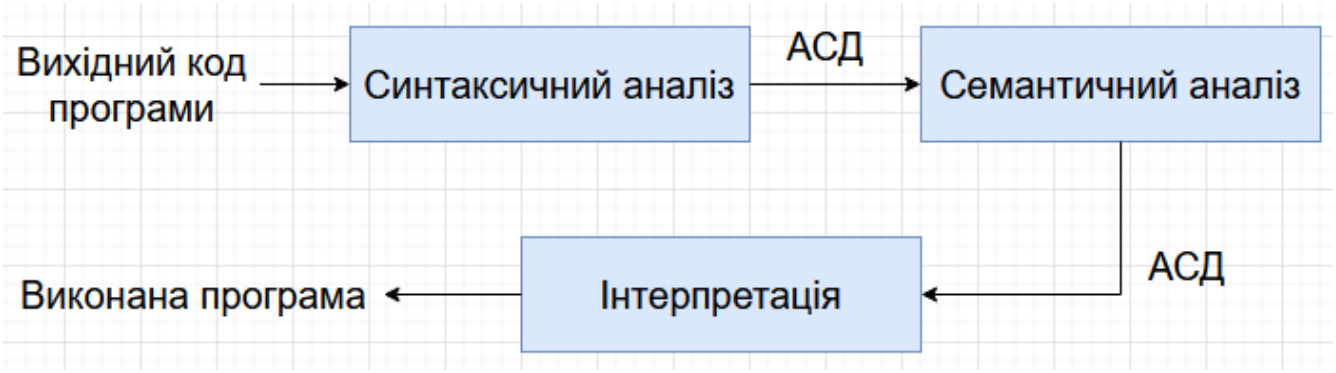



*Рисунок 3.4.2. Етапи виконання розробленого інтерпретатора*

Базуючись на описаних вище етапах, програму можна розбити на наступні модулі:

1. *Parser* - синтаксичний аналізатор (парсер).
2. *Analyzer* - семантичний аналізатор.
3. *Interpreter* - інтерпретатор.
4. *Lexic* – визначення АСД (спільний модуль, який використовується усіма іншими модулями).
5. *Main* - головний модуль, у якому знаходяться функції, необхідні для безпосереднього виконання інтерпретатора.

### 3.5 Розробка модуля Lexic для представлення АСД

Перед розробкою парсера необхідно визначити АСД, яке повинен створити парсер. АСД визначається у модулі *Lexic* (у файлі *Lexic.hs*). Основна мета цього модуля – у ньому визначені усі типи, що формують АСД і використовуються парсером, семантичним аналізатором та інтерпретатором. Усі типи, визначені у цьому модулі, експортуються щоб бути доступними для використання іншими модулями. Цей модуль не містить жодних функцій, оскільки призначений лише для визначення типів АСД.

У Haskell для створення типів даних використовується ключове слово *data*. Тип даних повинен мати один або більше конструкторів, які використовуються для створення сутностей відповідного типу. Різні конструктори можуть містити абсолютно різні поля, проте вони все одно представляють один тип. Ця особливість буде використана пізніше, при створенні типів даних для опису інструкцій та виразів.

Тип *Program* є початком АСД, оскільки він містить інформацію про заголовок програми та основний блок програми, представлений типом *Block*. Тип *Block* представляє собою блок у Pascal і містить інформацію про визначені у ньому змінні, процедури, функції та тіло блоку. Інформації про визначені змінні, функції та процедури зберігається у типах *Variable*, *Function* та *Procedure* відповідно. Тип *FormalParam* використовується для представлення формальних параметрів функцій та процедур. Варто зазначити, що оскільки функції та процедури у Pascal створюють власні блоки, то вони містять інформацію про блоки типу *Block*, тобто типи *Block*, *Function* та *Procedure* використовують один одного у своїй структурі. Це дозволяє представити вкладені блоки в АСД. Визначення усіх перелічених типів зображено на рисунку 3.5.1.

```
data Program = Program {pHeader :: Identifier, pBody :: Block} deriving (Show)

data Block = Block {bDeclarations :: [Declaration], bBody :: Statement} deriving (Show)

data Declaration = VarDecl [Variable] | FuncDecl Function | ProcDecl Procedure deriving (Show)

data Variable = Var {vName :: Identifier, vType :: DataType, vValue :: Maybe Value} deriving (Show)

data Function = Function {fName :: Identifier, fParams :: [FormalParam], fResType :: DataType, fBlock :: Block} deriving (Show)

data Procedure = Procedure {pName :: Identifier, pParams :: [FormalParam], pBlock :: Block} deriving (Show)
```

*Рисунок 3.5.1. Визначення типів АСД, що відповідають програмі, блокам та визначенням*

Усі інструкції, що будуть доступні в інтерпретаторі представлені типом *Statement* (рисунок 3.5.2). Це оператор присвоєння (конструктор *Assignment*), виклик процедури (конструктор *ProcCall*), складений оператор (конструктор *Compound*), умовний оператор “if” (конструктор *If*), цикли “while” (конструктор *While*) та “repeat” (конструктор *Repeat*). Усі інструкції зберігають різну

інформацію, проте все одно представляють один і той же тип *Statement*. Для створення різних сутностей одного й того ж типу використовуються різні конструктори.

```
data Statement
  = Assignment {aName :: Identifier, aValue :: Expression}
  | ProcCall {pcName :: Identifier, pcParams :: [Expression]}
  | Compound [Statement]
  | If {iCondition :: Expression, iIfRoute :: Statement, iElseRoute :: Maybe Statement}
  | While {wCondition :: Expression, wBody :: Statement}
  | Repeat {rCondition :: Expression, rBody :: [Statement]}
  deriving (Show)
```

*Рисунок 3.5.2. Визначення типу Statement*

Вирази, із яких будуються інструкції, представлені типом *Expression* (рисунок 3.5.3). Доступні вирази: константа базового типу (конструктор *Val*), використання змінної (конструктор *VarRef*), виклик функції (конструктор *FuncCall*), унарний оператор (конструктор *UnOp*), бінарний оператор (конструктор *BinOp*), операція “()” (конструктор *Paren*). Доступні унарні та бінарні операції представлені окремими типами *UnaryOp* та *BinaryOp* відповідно, кожен з яких має конструктори, які представляють усі доступні операції.

```haskell
data Expression
  = Val Value
  | VarRef Identifier
  | FuncCall {fcName :: Identifier, fcParams :: [Expression]}
  | UnOp UnaryOp Expression
  | BinOp {boOp :: BinaryOp, boLeft :: Expression, boRight :: Expression}
  | Paren Expression
  deriving (Show)

data UnaryOp
  = Not
  | UnaryPlus
  | UnaryMinus
  deriving (Show, Eq, Ord)

data BinaryOp
  = Plus
  | Minus
  | Mul
  | Div
  | FullDiv
  | Mod
  | Eql
  | Neql
  | Gt
  | Gte
  | Lt
  | Lte
  | And
  | Or
  | Xor
  deriving (Show, Eq, Ord)
```

*Рисунок 3.5.3. Визначення типу Expression та типів, які він використовує*

Тип *Identifier* (рисунок 3.5.4) представляє усі ідентифікатори, які зустрічаються у програмі. Він використовується для представлення інформації про

```haskell
data Identifier = Identifier {idValue :: String}
  deriving (Show)
```

*Рисунок 3.5.4. Визначення типу Identifier*

назву програми (у заголовку), назви змінних, функцій та процедур, назви формальних параметрів функцій та процедур.

Вбудовані типи даних також є ідентифікаторами, проте оскільки вони є фіксованими, було вирішено створити окремий тип *DataType* (рисунок 3.5.5) який представляє усі доступні типи даних.

```
data DataType
  = DTBoolean
  | DTInteger
  | DTReal
  | DTChar
  | DTString
  deriving (Show, Eq)
```

*Рисунок 3.5.5. Визначення типу DataType*

Для представлення значень базових типів було створено тип *Value* (рисунок 3.5.6). Для кожного доступного базового типу використовується окремий конструктор, який співставляє цей тип із типом даних у Haskell.

```
data Value
  = Boolean Bool
  | IntNum Int
  | RealNum Double
  | Character Char
  | Str String
  deriving (Show)
```

*Рисунок 3.5.6. Визначення типу Value*

### 3.6 Розробка модуля Parser - синтаксичного аналізатора (парсера)

Після визначення типів для представлення АСД потрібно розробити синтаксичний аналізатор, який буде перетворювати вихідний код програми в АСД. У модулі *Parser* (вихідний файл *Parser.hs*) визначаються усі необхідні для виконання синтаксичного аналізу функції. Його основна мета – розпізнати вихідний код програми і перетворити його на АСД. Цей модуль експортує лише

одну функцію – *applyParser* (рисунок 3.6.1), яка використовується для виконання синтаксичного аналізу програми. Вона має 2 параметри: шлях до файлу (використовується бібліотекою Parsec для повернення інформації про помилки) та вміст файлу (вихідний код програми, що має бути розпізнана). Ця функція повертає значення типу *Either ParseError Program*. У Haskell тип *Either* – це тип, який дозволяє інкапсулювати результат виконання обчислень і повернути або власне результат, якщо виконання було успішним, або помилку виконання. Якщо побудова АСД закінчилась успішно - результатом є АСД типу *Program*, якщо ні – результатом є помилка типу *ParseError*. Це дозволяє завершити виконання інтерпретатора одразу, якщо помилка відбулась на етапі синтаксичного аналізу, та вивести інформацію про помилку користувачеві для її виправлення.

```
applyParser :: String -> String -> Either ParseError Program
applyParser = parse programP
```

*Рисунок 3.6.1. Імлементація функції applyParser*

Концепція бібліотеки Parsec полягає в створенні невеликих парсерів, які виконують задачу синтаксичного аналізу одного елементу, і подальшій комбінації цих парсерів для виконання аналізу складних конструкцій. Відповідно, розробку варто почати із створення допоміжних парсерів, які потрібні для аналізу багатьох більш складних елементів.

Прикладами допоміжних парсерів (рисунок 3.6.2) є парсери для аналізу спеціальних символів, які зустрічаються у синтаксисі Pascal (наприклад, “:”, “;”, “(” , “)”). У Parsec для створення таких парсерів використовується функція *char*. Вона приймає на вхід символ, який необхідно розпізнати, і повертає парсер, який розпізнає цей символ.

```haskell
hashP :: Parser Char
hashP = char '#'

singleQuoteP :: Parser Char
singleQuoteP = char '\''

colonP :: Parser Char
colonP = char ':'

semicolonP :: Parser Char
semicolonP = char ';'

openParenP :: Parser Char
openParenP = char '('

closeParenP :: Parser Char
closeParenP = char ')'
```

*Рисунок 3.6.2. Імлементація допоміжних парсерів*

Ще однією поширеним способом створення парсерів у Parsec є створення функцій, які приймають на вхід існуючий парсер і повертають новий парсер із модифікованою логікою. У Parsec визначено функції *skipMany* та *skipMany1*, які приймають на вхід будь-який парсер і дозволяють пропустити 0, 1 або більше елементів, які може розпізнати вхідний парсер. Стандартною практикою є додавання цифри 1 в кінці назви функції, якщо вона створює парсер, який повинен розпізнати багато елементів і мінімум один елемент повинен бути розпізнаний, і використання назви функції без жодних цифр, якщо допускається що жоден елемент із вхідного парсера не може бути розпізнаний. Використовуючи ці функції досить легко створити парсери *anySpacesP* та *anySpaces1P*, які пропускають усі символи відступу у вихідному коді програми. Щоб створити ці функції достатньо створити додатковий парсер *anySpace*, який розпізнає будь-який символ відступу. Для його побудови використовується оператор “<|>”, який дозволяє застосувати декілька парсерів послідовно і повертає результат першого

парсера, який успішно розпізнав вхідний елемент. Визначення усіх перелічених функцій можна побачити на рисунку 3.6.3.

```
anySpaces1P :: Parser ()
anySpaces1P = skipMany1 anySpace

anySpacesP :: Parser ()
anySpacesP = skipMany anySpace

anySpace :: Parser Char
anySpace = space <|> newline <|> tab
```

*Рисунок 3.6.3. Імлементація парсерів для пропуску символів відступу*

Однією із стандарних практик при розробці парсерів з використанням Parsec є усунення зайвих символів відступу після кожного синтаксичного елемента (ідентифікатора, оператора, тощо). Це дозволяє комбінувати парсери різних елементів між собою без необхідності використання додаткових парсерів щоразу щоб позбутися відступів перед та після елемента. Щоб уникнути постійного застосування раніше визначених парсерів *anySpacesP* та *anySpaces1P* можна створити допоміжні функції *lexemeP* та *lexeme1P*. Ці функції приймають на вхід парсер будь-якого елемента і повертають модифікований парсер цього елемента,

```
lexeme1P :: Parser a -> Parser a
lexeme1P p = do
  x <- p
  _ <- anySpaces1P
  return x

lexemeP :: Parser a -> Parser a
lexemeP p = do
  x <- p
  _ <- anySpacesP
  return x
```

*Рисунок 3.6.4. Імлементація функцій для побудови парсерів, що пропускають символи відступу після вхідного парсера*

який також ігнорує усі відступи, які знаходяться після елемента. Реалізації цих функцій наведено на рисунку 3.6.4.

Ще одним важливим допоміжним парсером є парсер який дозволяє розпізнавати ідентифікатори у вихідному коді, оскільки ідентифікатори використовуються для назв змінних, функцій та процедур. Ідентифікатором у Pascal є послідовність літер, цифр та символів “_”, яка починається із літери або символа “_”. Цей парсер має назву *identifierP* і для його визначення використовується раніше описаний оператор “<|>” та функція *many*, яка дозволяє створити парсер, який розпізнає 0 або більше послідовних елементів, що розпізнаються вхідним парсером. Додатково також було визначено парсер *allowedIdentifierP*, який також розпізнає ідентифікатори, проте повертає помилку, якщо ідентифікатор є одним із ключових слів у Pascal. Парсер *identifierP* застосовується у випадках, коли потрібно розпізнати ідентифікатор, який може бути ключовим словом (наприклад, інструкції можуть починатись ключовими словами “if” чи “while” які є ідентифікаторами, а можуть ідентифікатором, який відноситься до назви процедури, що викликається, чи назви змінної, якій присвоюється значення). Парсер *allowedIdentifierP* застосовується у місцях, де точно відомо, що ідентифікатор повинен бути не ключовим словом (у заголовку

```
identifierP :: Parser Identifier
identifierP = do
  bg <- letter <|> underscoreP
  rest <- many (letter <|> digit <|> underscoreP)
  return (Identifier {idValue = bg:rest})
  where underscoreP = char '_'

allowedIdentifierP :: Parser Identifier
allowedIdentifierP = do
  iden@(Identifier {idValue}) <- identifierP
  if (elem idValue reservedKeywords)
  then unexpected idValue
  else return (iden)
```

*Рисунок 3.6.5. Імлементація парсерів ідентифікаторів*

програми, назвах змінних, функцій, процедур, назвах параметрів функцій та процедур). Визначення обох парсерів можна побачити на рисунку 3.6.5.

Оскільки кількість типів даних у інтерпретаторі фіксована, було створено парсер *dataTypeP* (рисунок 3.6.6), який розпізнає назви усіх типів даних, доступних у Pascal. Його визначення можна побачити на рисунку. Для його створення було використано функцію *try*, яка дозволяє реалізувати бектрекінг у Parsec. Вона використовується, якщо декілька парсерів розпізнають елементи із однаковим початком і необхідно спробувати декілька різних парсерів, щоб правильно розпізнати елемент. Наприклад, назва цілочисельного типу “Integer” починається на одну й ту ж літеру, що й ключове слово “If”, а тип дійсних чисел “Real” має перші 2 літери у назві такі ж, як ключове слово “Repeat”.

```
dataTypeP :: Parser DataType
dataTypeP = do
  v <- lexemeP (try (string "Integer") <|> try (string "Real") <|> try (string "Boolean") <|> try (string "Char") <|> try (string "String"))
  return
    ( case v of
        "Integer" -> DTInteger
        "Real" -> DTReal
        "Boolean" -> DTBoolean
        "Char" -> DTChar
        "String" -> DTString
    )
```

*Рисунок 3.6.6. Імлементація парсера типів даних*

Після визначення допоміжних парсерів з їх допомогою можна визначити парсери, які розпізнають основні елементи програми. Парсером, який починає

```
programP :: Parser Program
programP = do
  _ <- anySpacesP
  name <- headerP
  block <- blockP
  _ <- lexemeP (char '.')
  return (Program{pHeader = name, pBody = block})
 where
  headerP = do
    _ <- lexemeP (string "program")
    ident <- lexemeP allowedIdentifierP
    _ <- lexemeP semicolonP
```

*Рисунок 3.6.7. Імлементація парсера типу Program*

розпізнавання програми є *programP*, його реалізацію наведено на рисунку 3.6.7. Відповідно до раніше визначеного синтаксису, він розпізнає ключове слово *program*, ідентифікатор, який визначає назву програми, основний програмний блок за допомогою парсера *blockP* і крапку, яка позначає завершення програми.

Парсер *blockP* (реалізацію наведено на рисунку 3.6.8) розпізнає 0 або більше визначень змінних, функцій та процедур, а після визначень – тіло блоку, яке складається із складеного оператора і починається ключовим словом *begin*. Для розпізнавання кожного елемента використовуються окремі парсери.

```
blockP :: Parser Block
blockP = do
  declarations <- many (varDeclP <|> funcDeclP <|> procDeclP)
  _ <- lexemeP (string "begin")
  sttm <- compoundStatementP
  return
    ( Block
        { bDeclarations = declarations,
          bBody = sttm
        }
    )
```

*Рисунок 3.6.8. Імлементація парсера типу Block*

Визначення змінних розпізнаються парсером *varDeclP* (реалізацію наведено на рисунку 3.6.9). Він розпізнає ключове слово *var*, за яким слідує список назв змінних та їх типів.

```
varDeclP :: Parser Declaration
varDeclP = do
  _ <- lexeme1P (string "var")
  varName <- lexemeP allowedIdentifierP
  _ <- lexemeP colonP
  varType <- dataTypeP
  _ <- lexemeP semicolonP
  return (VarDecl [Var{vName = varName, vType = varType, vValue = Nothing}])
```

*Рисунок 3.6.9. Імлементація парсера типу Declaration для визначень змінних*

Парсери *funcDeclP* та *procDeclP* (рисунок 3.6.10), які розпізнають визначення функцій та процедур, досить схожі між собою. Спочатку розпізнається відповідно ключове слово (*function* або *procedure*), за ним слідує ідентифікатор, після нього – список формальних параметрів. Для розпізнавання списку параметрів використовується допоміжний парсер *formalParamListP*, це дозволяє уникнути необхідності описання однієї і тієї ж логіки для двох різних парсерів. Після списку параметрів для функції додатково розпізнається тип результату. Останнім розпізнається тіло функції чи процедури з використання раніше описаного парсера *blockP*. Неважко помітити, що парсери функцій, процедур та блоків рекурсивно викликають один одного. Це досить схоже на те, як було описано визначення цих елементів раніше у формі Бекуса-Наура. Однією із переваг Parsec є те, що ця бібліотека дозволяє створювати парсери, використання яких є дуже близьким до опису синтаксису мови, що розпізнається.

```
funcDeclP :: Parser Declaration
funcDeclP = do
  _ <- lexeme1P (string "function")
  name <- lexemeP allowedIdentifierP
  formalParams <- parenthesisP formalParamListP
  _ <- lexemeP colonP
  returnType <- dataTypeP
  _ <- lexemeP semicolonP
  block <- blockP
  _ <- lexemeP semicolonP
  return (FuncDecl Function{fName = name, fParams = formalParams, fResType = returnType, fBlock = block})

procDeclP :: Parser Declaration
procDeclP = do
  _ <- lexeme1P (string "procedure")
  name <- lexemeP allowedIdentifierP
  formalParams <- parenthesisP formalParamListP
  _ <- lexemeP semicolonP
  block <- blockP
  _ <- lexemeP semicolonP
  return (ProcDecl Procedure{pName = name, pParams = formalParams, pBlock = block})

formalParamListP :: Parser [FormalParam]
formalParamListP = sepBy formalParamP (lexemeP semicolonP)
 where
  formalParamP = do
    paramName <- lexemeP allowedIdentifierP
    _ <- lexemeP colonP
```

*Рисунок 3.6.10. Імлементація парсерів типу Declaration для визначень функцій та процедур*

Для розпізнавання усіх видів інструкцій використовується парсер *statementP* (рисунок 3.6.11). Першим кроком є розпізнавання ідентифікатора. Якщо він відповідає одному із ключових слів – викликається додатковий парсер який розпізнає саме цю інструкцію. Більшість додаткових парсерів визначаються всередині парсера *statementP*, оскільки вони використовуються тільки всередині цього парсера. Парсер *compoundStatementP*, який розпізнає складений оператор, було визначено окремо тому, що він використовується для розпізнавання тіл блоків, функцій та процедур.

Додаткові парсери зазвичай розпізнають лише одну інструкцію, тому їх реалізація є досить простою. Для кожного із них розпізнаються відповідні ключові слова та елементи (вирази чи вкладені інструкції). Проте серед них варто звернути увагу на 2 парсери.

Перший – парсер *ifStatementP*, який розпізнає умовний оператор “if”. Його особливістю є те, що секція “else”, є опційною, а тому залежно від її наявності потрібно або розпізнавати інструкцію, що слідує за ключовим словом *else*, або завершити розпізнавання раніше.

Другий - парсер *assignmentOrProcCallP*. Він використовується для розпізнавання операторів присвоєння та виклику процедури. Лише ідентифікатора недостатньо щоб визначити тип інструкції, потрібно додатково визначити, до чого відноситься ідентифікатор: це назва змінної в операторі присвоєння, чи назва процедури для виклику. Для цього потрібно спробувати розпізнати наступні декілька символів. Якщо наступним символом є “(” – це виклик процедури, якщо наступні 2 символи – це послідовність “:=” – це оператор присвоєння. Визначивши точний тип інструкції, використовується один із двох відповідних допоміжних парсерів щоб завершити розпізнавання інструкції.

```haskell
statementP :: Parser Statement
statementP = do
  iden <- lexemeP identifierP
  case idValue iden of
    "if" -> ifStatementP
    "while" -> whileStatementP
    "repeat" -> repeatStatementP
    "begin" -> compoundStatementP
    _ -> assignmentOrProcCallP iden
 where
  ifStatementP = do
    cond <- expressionP
    _ <- lexeme1P (string "then")
    ifSttm <- statementP
    hasElse <- optionMaybe (lexeme1P (string "else"))
    ( case hasElse of
        Just _ ->
          ( do
              elseSttm <- statementP
              return
                ( If
                    { iCondition = cond,
                      iIfRoute = ifSttm,
                      iElseRoute = Just elseSttm
                    }
                )
          )
        Nothing ->
          return
            ( If
                { iCondition = cond,
                  iIfRoute = ifSttm,
                  iElseRoute = Nothing
                }
            )
    )
  whileStatementP = do
    cond <- expressionP
    _ <- lexeme1P (string "do")
    sttm <- statementP
    return (While{wCondition = cond, wBody = sttm})
  repeatStatementP = do
    sttms <- statementListP
    _ <- lexemeP (string "until")
    cond <- expressionP
    return (Repeat{rCondition = cond, rBody = sttms})
  assignmentOrProcCallP iden = do
    ch <- optionMaybe openParenP
    case ch of
      Just _ -> procCallP
      Nothing -> assignmentP
   where
    procCallP = do
      exprs <- paramListP
      _ <- lexemeP closeParenP
      return (ProcCall{pcName = iden, pcParams = exprs})
    assignmentP = do
      _ <- lexemeP (string ":=")
      expr <- expressionP
      return (Assignment{aName = iden, aValue = expr})

compoundStatementP :: Parser Statement
compoundStatementP = do
  sttms <- statementListP
  _ <- lexemeP (string "end")
```

*Рисунок 3.6.11. Імлементація парсерів типу Statement для розпізнавання інструкцій*

Для розпізнавання виразів було визначено декілька парсерів: *expressionP*, *simpleExpressionP*, *termP* та *factorP*. Реалізацію усіх парсерів (крім *factorP*) наведено на рисунку 3.6.12. Вони мають схожу структуру, оскільки усі вони розпізнають вирази, які є бінарними операціями. Відрізняються лише бінарні операції, що розпізнаються кожним парсером. Така реалізацію досить точно відтворює синтаксис мови Pascal, описаний у формі Бекуса-Наура, і дозволяє реалізувати різний пріоритет виконання бінарних операцій.

```
expressionP :: Parser Expression
expressionP = chainl1 simpleExpressionP binOpC
 where
  binOpC = do
    op <-
      lexemeP
        ( try (string ">=")
            <|> try (string "<=")
            <|> try (string "<>")
            <|> string ">"
            <|> string "<"
            <|> string "="
        )
    return (binOpConstructor $ binOp op)

simpleExpressionP :: Parser Expression
simpleExpressionP = chainl1 termP binOpC
 where
  binOpC = do
    op <-
      lexemeP
        ( string "+"
            <|> string "-"
            <|> try (string "or")
            <|> try (string "xor")
        )
    return (binOpConstructor $ binOp op)

termP :: Parser Expression
termP = chainl1 factorP binOpC
 where
  binOpC = do
    op <-
      lexemeP
        ( string "*"
            <|> string "/"
            <|> try (string "div")
            <|> try (string "mod")
            <|> try (string "and")
        )
    return (binOpConstructor $ binOp op)
```

*Рисунок 3.6.12. Імлементація парсерів типу Expression для розпізнавання виразів бінарних операцій*

Парсер *factorP* (рисунок 3.6.13) розпізнає усі типи виразів, які не є бінарними операціями: унарні операції, операцію “()”, використання змінної, виклик функції та використання констант базових типів. Як і при розпізнаванні інструкцій, для кожного типу виразів використовується окремий допоміжний парсер, визначений всередині парсера *factorP*. Парсер *varRefOrFuncCallP* схожий на парсер *assignmentOrProcCallP*, оскільки у ньому схожим чином потрібно розпізнати наступний символ щоб розрізнити використання змінної та виклик функції.

```
factorP :: Parser Expression
factorP = do
  _ <- anySpacesP
  expr <-
    parenExpressionP
      <|> notFactorP
      <|> signFactorP
      <|> unsignedConstantP
      <|> varRefOrFuncCallP
  return (expr)
 where
  parenExpressionP = do
    expr <- parenthesisP expressionP
    return (Paren expr)
  notFactorP = do
    _ <- try (lexeme1P (string "not"))
    expr <- lexemeP factorP
    return (UnOp Not expr)
  signFactorP = do
    sign <- lexemeP (oneOf "+-")
    expr <- lexemeP factorP
    return (UnOp (if sign == '+' then UnaryPlus else UnaryMinus) expr)
  varRefOrFuncCallP = do
    iden <- lexemeP allowedIdentifierP
    ch <- optionMaybe openParenP
    ( case ch of
        Just _ -> funcCallP iden
        Nothing -> return (VarRef iden)
      )
   where
    funcCallP iden = do
      exprs <- paramListP
      _ <- lexemeP closeParenP
      return (FuncCall{fcName = iden, fcParams = exprs})
  unsignedConstantP = do Val <$> valueP
```

*Рисунок 3.6.13. Імлементація парсера типу Expression для розпізнавання усіх виразів крім бінарних операцій*

Парсер *valueP* (рисунок 3.6.14) розпізнає значення базових типів. Для цілочисельного типу – це послідовність цифр, для типу дійсних чисел – це 2 цілих числа, розділених крапкою, для булевого типу – це ключові слова *true* та *false*. Символьний та стрічковий типи є особливими, оскільки є 2 способи вказати їх значення. Перший спосіб – вказати один символ (для символьного типу) або 0 чи більше символів (для стрічкового типу), які починаються і закінчуються символом “”. Якщо необхідно використати сам символ “” як значення його потрібно продублювати (тобто вказати послідовність “””). Другий спосіб – вказати 1 або більше ASCII-кодів символів, кожен з яких починається символом “#” (тобто, стрічку “ab” також можна створити записом “#97#98”).

```
valueP :: Parser Value
valueP = unsignedNumberP <|> boolP <|> stringP
 where
  unsignedNumberP = do
    integralPart <- numberP
    dot <- optionMaybe (char '.')
    case dot of
      Nothing -> do
        _ <- anySpacesP
        return (IntNum (read integralPart))
      Just _ -> do
        fractionalPart <- lexemeP numberP
        return (RealNum (read (integralPart ++ "." ++ fractionalPart)))
  boolP = do
    v <- lexemeP (try (string "true") <|> try (string "false"))
    return
      ( case v of
          "true" -> Boolean True
          "false" -> Boolean False
      )
  stringP = do
    fst <- singleQuoteP <|> hashP
    str <- case fst of
      '\'' -> do
        chs <- many (try ((noneOf "'") <|> (singleQuoteP >> singleQuoteP)))
        _ <- lexemeP singleQuoteP
        return chs
      '#' -> do
        fst <- numberP
        rest <- many (hashP >> numberP)
        _ <- anySpacesP
        return (map (toEnum . read) (fst : rest))
    return (if length str == 1 then (Character (head str)) else (Str str))
```

*Рисунок 3.6.14. Імлементація парсера типу Value*

Розроблений синтаксичний аналізатор може бути легко розширений щоб розпізнавати нові інструкції чи визначення. Проте додавання нових, більш складних типів даних (наприклад, масивів чи записів, які присутні у Pascal) є більш складною задачею. Оскільки усі базові типи даних визначаються іменованими константами і не мають структури, їх легко зберігати у вигляді констант, проте додавання типів даних, що мають структуру чи інші додаткові характеристики, вимагатиме зміни підходу до розпізнавання типів даних.

### 3.7 Розробка модуля Analyzer - семантичного аналізатора

Після виконання синтаксичного аналізу і створення АСД наступним етапом є семантичний аналіз. У модулі *Analyzer* (вихідний файл *Analyzer.hs*) визначаються усі необхідні для виконання семантичного аналізу функції. Його основна мета – перевірити програму на семантичні помилки. Основні види помилок, які аналізуються:

- Помилки невідповідності типів (у виразах, інструкціях, при виклику функцій та процедур);
- Помилки неіснуючих ідентифікаторів (використання змінних, функцій та процедур, які недоступні в області видимості певного блоку програми або взагалі не визначені у програмі);
- Помилки дублювання ідентифікаторів (спроба назвати змінну, функцію, процедуру чи параметр ідентифікатором, який уже використано в області видимості блоку);
- Помилки невідповідності кількості та типів формальних і фактичних параметрів (для функцій та процедур).

Цей модуль експортує лише одну функцію – *applyAnalyzer* (рисунок 3.7.1), яка використовується для виконання семантичного аналізу програми. Вона має

єдиний параметр типу *Program*. Він представляє АСД, яке треба проаналізувати. Ця функція повертає значення типу *Either AnalysisError Analyzer*. Якщо в ході аналізу було знайдено помилки – результатом роботи аналізатора є перша знайдена у програмі помилка типу *AnalysisError*. Якщо виконання аналізу закінчилось без помилок, то результатом є значення типу *Analyzer* (цей тип зберігає інформацію про стан аналізатора), воно використовується для виведення інформації про виконання етапу семантичного аналізу при запуску інтерпретатора в розширеному режимі. Типи *AnalysisError* та *Analyzer* будуть більш детально описані далі.

```
applyAnalyzer :: Program -> Either AnalysisError Analyzer
applyAnalyzer = analyzeProgram startingAnalyzer
```

*Рисунок 3.7.1. Імплементація функції applyAnalyzer*

Для виконання семантичного аналізу необхідно зберігати інформацію про усі визначені в програмі елементи. Для цього використовується тип *Scope* (рисунок 3.7.2). Кожна сутність типу *Scope* відповідає блоку у Pascal і зберігає інформацію про змінні, функції та процедури, визначені у відповідному блоці. Інформація зберігається у вигляді словників, які у Haskell представлені типом *Map*. Словники представляють собою набір пар типу “ключ-значення”. Для усіх елементів ключем є ідентифікатор (назва функції, змінної чи процедури), а значенням – тип, який зберігає інформацію про відповідний елемент. Для змінних – це їх тип, для процедур – назви і типи параметрів, для функцій – усе те ж, що і для процедур, а також інформація про тип даних, який є результатом функції. Додатково, сутності типу *Scope* також мають посилання на батьківську сутність цього ж типу. Воно представлене типом *Maybe*, який у Haskell використовується, якщо значення може бути відсутнім. Завдяки цьому посиланню завжди можна відстежити ланцюг вкладених блоків і здійснювати пошук елементів не лише в поточному блоці, а й у батьківських. Єдиний блок, який не має батьківського – це

основний програмний блок, оскільки він є початковим блоком будь-якої програми на Pascal.

```
data Scope = Scope
  { variables :: Map.Map String VarInfo,
    functions :: Map.Map String FuncInfo,
    procedures :: Map.Map String ProcInfo,
    scopeLevel :: Int,
    parentScope :: Maybe Scope
  }
  deriving (Show)

data VarInfo = VI {viName :: String, viType :: DataType} deriving (Show)
data FuncInfo = FI {fiName :: String, fiParams :: [ParamInfo], fiResType :: DataType} deriving (Show)
data ProcInfo = PRI {priName :: String, priParams :: [ParamInfo]} deriving (Show)
data ParamInfo = PAI {paiName :: String, paiType :: DataType} deriving (Show)
```

*Рисунок 3.7.2. Визначення типу Scope*

Тип *Analyzer* (рисунок 3.7.3) відповідає за створення, оновлення та відстеження сутностей типу *Scope* та збереження інформації з АСД. Сутності цього типу зберігають сутність типу *Scope* яка відповідає поточному блоку та ще одну сутність, яка відповідає програмному блоку.

```
data Analyzer = A
  { currentScope :: Scope,
    globalScope :: Scope
  }
  deriving (Show)
```

*Рисунок 3.7.3. Визначення типу Analyzer*

Для відстеження семантичних помилок використовується тип *AnalysisError* (рисунок 3.7.4). Він містить інформацію про тип помилки, повідомлення, яка надає більш детальну інформацію про помилку, а також опційну інформацію про іншу помилку, яка стала причиною поточної. Тип помилок визначається окремим типом *AnalysisErrorType*. Для кожного визначення, виразу та інструкції існує власний тип помилки (наприклад, *AssignmentError* для оператора присвоєння або *BinaryOperatorError* для бінарних операцій). Інші типи помилок використовуються для більш загальних помилок, які не прив’язані до конкретного елемента. Прикладами таких типів помилок є *IdentifierAlreadyDefinedError* (якщо

ідентифікатор уже використовується) і *TypeMismatchError* (використовується у будь-якому місці програми, де очікуваний та обрахований тип виразу відрізняються).

```
data AnalysisError = AnalysisError AnalysisErrorType String (Maybe AnalysisError) deriving (Show)
data AnalysisErrorType
  = IdentifierAlreadyDefinedError
  | VariableDoesNotExistError
  | FunctionDoesNotExistError
  | ProcedureDoesNotExistError
  | ActualParameterError
  | TypeMismatchError
  | FormalParameterDeclarationError
  | VariableDeclarationError
  | FunctionDeclarationError
  | ProcedureDeclarationError
  | AssignmentError
  | ProcedureCallError
  | IfStatementError
  | WhileStatementError
  | RepeatStatementError
  | VariableReferenceError
  | FunctionCallError
  | UnaryOperatorError
  | BinaryOperatorError
  deriving (Show)
```

*Рисунок 3.7.4. Визначення типу AnalysisError*

Семантичний аналіз програми починається з створення сутності *Analyzer* із початковим станом за допомогою функції *startingAnalyzer* (рисунок 3.7.5). Початковий стан сутності типу *Analyzer* – це наявність блоку, який не містить жодних визначень, окрім процедур *read*, *readln*, *write*, *writeln*. Ці процедури використовуються у Pascal для читання та виведення інформації у стандартний потік введення-виведення. Вони є частиною стандартної бібліотеки Pascal, тому для можливості їх використання потрібно врахувати їх у програмному блоці. У початковому стані використовується одна і та ж сутність типу *Scope*, яка вказує як на поточний блок, так і на програмний блок, оскільки виконання програми починається із програмного блоку.

```
startingAnalyzer :: Analyzer
startingAnalyzer =
  A
    { currentScope = sc,
      globalScope = sc
    }
 where
  sc =
    Scope
      { variables = Map.empty,
        functions = Map.empty,
        procedures =
          Map.fromList
            [ ("write", PRI{priName = "write", priParams = []}),
              ("writeln", PRI{priName = "writeln", priParams = []}),
              ("read", PRI{priName = "read", priParams = []}),
              ("readln", PRI{priName = "readln", priParams = []})
            ],
        scopeLevel = 1,
        parentScope = Nothing
      }
```

*Рисунок 3.7.5. Імплементація функції strartingAnalyzer*

Після створення сутності типу *Analyzer* із початковим станом, виконується аналіз програми. Для аналізу програми необхідно виконати аналіз програмного блоку. Аналіз блоку виконується функцією *analyzeBlock* (рисунок 3.7.6). Для аналізу блоку спочатку виконується аналіз усіх визначень, наявних у блоці, і якщо не було знайдено жодних помилок, виконується аналіз тіла блоку.

```
analyzeProgram :: Analyzer -> Program -> Either AnalysisError Analyzer
analyzeProgram a Program{pHeader, pBody} = analyzeBlock (Right a) pBody

analyzeBlock :: Either AnalysisError Analyzer -> Block -> Either AnalysisError Analyzer
analyzeBlock a@(Left _) _ = a
analyzeBlock a@(Right _) Block{bDeclarations, bBody} = do
  a' <- foldl analyzeDeclaration a bDeclarations
  analyzeStatement (Right a') bBody
```

*Рисунок 3.7.6. Імплементація функцій analyzeProgram та analyzeBlock*

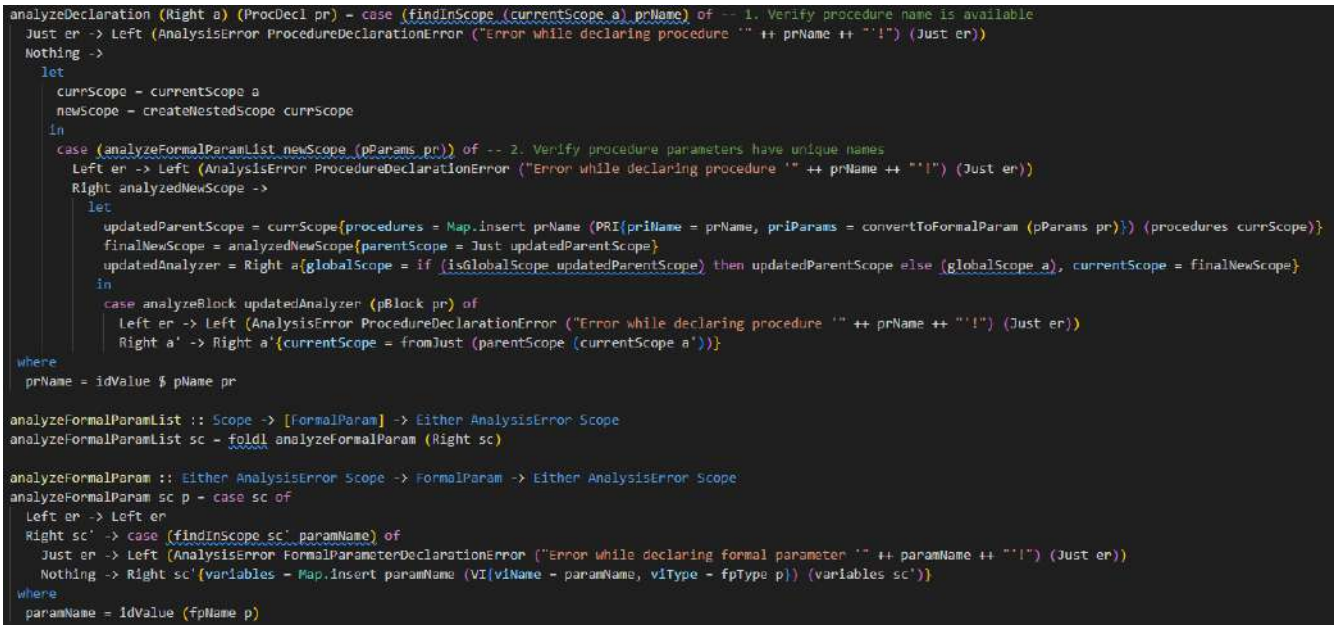
Аналіз визначень виконується функцією *analyzeDeclaration* і залежить від типу визначення.

Щоб визначення змінної було правильним, достатньо перевірити, що назва змінної доступна і в поточному блоці не існує жодних інших змінних, функцій чи процедур із такою ж назвою. Якщо перевірка була успішною, інформація про змінну зберігається в словник змінних поточного блоку (рисунок 3.7.7).

```
analyzeDeclaration a@(Right _) (VarDecl vars) = foldl analyzeVar a vars
 where
  analyzeVar a v =
    let
      varName = idValue (vName v)
      varType = vType v
    in
     case a of
       Left er -> Left er
       Right a' -> case (findInScope (currentScope a') varName) of
         Just er -> Left (AnalysisError VariableDeclarationError ("Error while declaring variable '" ++ varName ++ "'!") (Just er))
         Nothing ->
           let
             updatedScope = (currentScope a'){variables = Map.insert varName (VI{viName = varName, viType = varType}) (variables (currentScope a'))}
           in
            Right a'{globalScope = if (isGlobalScope updatedScope) then updatedScope else (globalScope a'), currentScope = updatedScope}
```

*Рисунок 3.7.7. Імплементація функції analyzeDeclaration для аналізу визначень змінних*

Процес аналізу процедур є дещо складнішим. По-перше, необхідно перевірити, що назва процедури доступна в поточному блоці. По-друге, потрібно перевірити, що у списку параметрів процедури назви усіх параметрів є унікальними. Для перевірки параметрів використовується допоміжна функція *analyzeFormalParam*. Якщо обидві перевірки були успішними, потрібно додати у поточний блок інформацію про процедуру (це дозволяє викликати процедури та функції рекурсивно), створити нову сутність типу *Scope*, яка відповідає блоку процедури, що аналізується, і додати у словник змінних нового блоку параметри процедури, оскільки параметри у процедурах та функціях вважаються повноцінними змінними. Нова сутність типу *Scope* використовується для рекурсивного виконання аналізу блоку процедури за допомогою функції *analyzeBlock*. Якщо у блоку процедури не було знайдено помилок, аналіз процедури завершується успішно (рисунок 3.7.8).



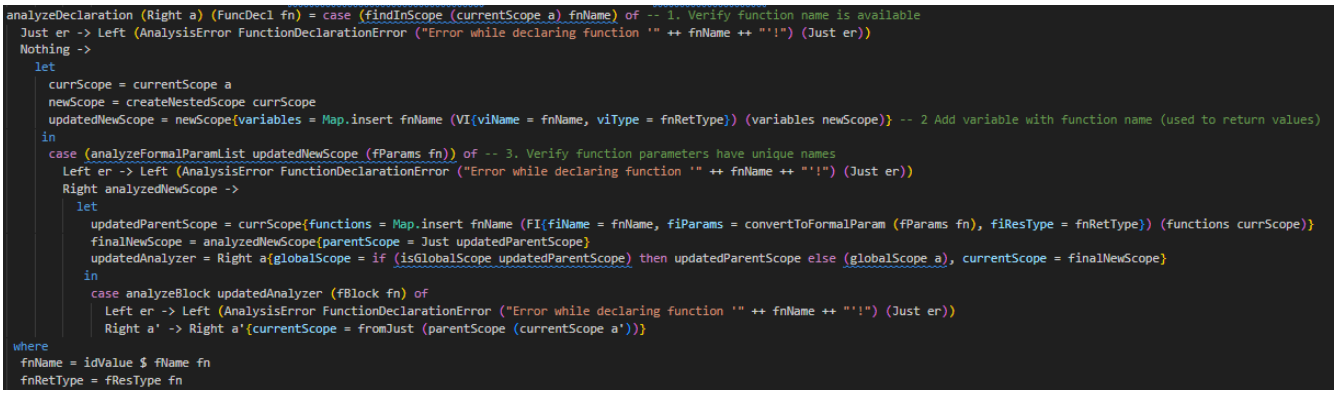
```
analyzeDeclaration (Right a) (ProcDecl pr) = case (findInScope (currentScope a) prName) of -- 1. Verify procedure name is available
  Just er -> Left (AnalysisError ProcedureDeclarationError ("Error while declaring procedure '" ++ prName ++ "'!") (Just er))
  Nothing ->
    let
      currScope = currentScope a
      newScope = createNestedScope currScope
    in
      case (analyzeFormalParamList newScope (pParams pr)) of -- 2. Verify procedure parameters have unique names
        Left er -> Left (AnalysisError ProcedureDeclarationError ("Error while declaring procedure '" ++ prName ++ "'!") (Just er))
        Right analyzedNewScope ->
          let
            updatedParentScope = currScope{procedures = Map.insert prName (PRI{priName = prName, priParams = convertToFormalParam (pParams pr)}) (procedures currScope)}
            finalNewScope = analyzedNewScope{parentScope = Just updatedParentScope}
            updatedAnalyzer = Right a{globalScope = if (isGlobalScope updatedParentScope) then updatedParentScope else (globalScope a), currentScope = finalNewScope}
          in
            case analyzeBlock updatedAnalyzer (pBlock pr) of
              Left er -> Left (AnalysisError ProcedureDeclarationError ("Error while declaring procedure '" ++ prName ++ "'!") (Just er))
              Right a' -> Right a'{currentScope = fromJust (parentScope (currentScope a'))}
  where
    prName = idValue $ pName pr

analyzeFormalParamList :: Scope -> [FormalParam] -> Either AnalysisError Scope
analyzeFormalParamList sc = foldl analyzeFormalParam (Right sc)

analyzeFormalParam :: Either AnalysisError Scope -> FormalParam -> Either AnalysisError Scope
analyzeFormalParam sc p = case sc of
  Left er -> Left er
  Right sc' -> case (findInScope sc' paramName) of
    Just er -> Left (AnalysisError FormalParameterDeclarationError ("Error while declaring formal parameter '" ++ paramName ++ "'!") (Just er))
    Nothing -> Right sc'{variables = Map.insert paramName (VI{viName = paramName, viType = fpType p}) (variables sc')}
  where
    paramName = idValue (fpName p)
```

*Рисунок 3.7.8. Імплементація функції analyzeDeclaration для аналізу визначень процедур*

Процес аналізу функцій майже повністю ідентичний процесу аналізу процедур. Єдиною відмінністю є необхідність додати додаткову змінну у словник змінних під час створення нової сутності типу *Scope*, яка відповідає блоку функції (рисунок 3.7.9). Ця змінна має таку ж назву, як і функція, а її тип відповідає типу значення, яке повертає функція. Це потрібно тому, що для повернення значень із функцій у Pascal немає окремого ключового слова як у багатьох інших мовах програмування. Натомість, для повернення значення із функції у Pascal використовується спеціальна змінна, яка має таку ж назву, як і функція. Для повернення значення необхідно присвоїти значення цій змінній.



```
analyzeDeclaration (Right a) (FuncDecl fn) = case (findInScope (currentScope a) fnName) of -- 1. Verify function name is available
  Just er -> Left (AnalysisError FunctionDeclarationError ("Error while declaring function '" ++ fnName ++ "'!") (Just er))
  Nothing ->
    let
      currScope = currentScope a
      newScope = createNestedScope currScope
      updatedNewScope = newScope{variables = Map.insert fnName (VI{viName = fnName, viType = fnRetType}) (variables newScope)} -- 2 Add variable with function name (used to return values)
    in
      case (analyzeFormalParamList updatedNewScope (fParams fn)) of -- 3. Verify function parameters have unique names
        Left er -> Left (AnalysisError FunctionDeclarationError ("Error while declaring function '" ++ fnName ++ "'!") (Just er))
        Right analyzedNewScope ->
          let
            updatedParentScope = currScope{functions = Map.insert fnName (FI{fiName = fnName, fiParams = convertToFormalParam (fParams fn), fiResType = fnRetType}) (functions currScope)}
            finalNewScope = analyzedNewScope{parentScope = Just updatedParentScope}
            updatedAnalyzer = Right a{globalScope = if (isGlobalScope updatedParentScope) then updatedParentScope else (globalScope a), currentScope = finalNewScope}
          in
            case analyzeBlock updatedAnalyzer (fBlock fn) of
              Left er -> Left (AnalysisError FunctionDeclarationError ("Error while declaring function '" ++ fnName ++ "'!") (Just er))
              Right a' -> Right a'{currentScope = fromJust (parentScope (currentScope a'))}
  where
    fnName = idValue $ fName fn
    fnRetType = fResType fn
```

*Рисунок 3.7.9. Імплементація функції analyzeDeclaration для аналізу визначень функцій*

Аналіз інструкцій виконується функцією *analyzeStatement* і теж залежить від типу інструкції.

Для аналізу оператора присвоєння достатньо перевірити що змінна із заданою назвою існує, а її тип відповідає типу виразу, значення якого потрібно присвоїти (рисунок 3.7.10).

```
analyzeStatement ea@(Right a) Assignment{aName, aValue} = case getVar (currentScope a) (idValue aName) of
  Left er -> Left (AnalysisError AssignmentError "Error during assignment statement!" (Just er))
  Right vi -> case analyzeExpression ea aValue of
    Left er -> Left (AnalysisError AssignmentError "Wrong expression in assignment statement!" (Just er))
    Right dt -> case expectTypes aValue (getExpectedTypes (viType vi)) dt of
      Left er -> Left (AnalysisError AssignmentError "Wrong expression type in assignment statement!" (Just er))
      Right _ -> Right a
```

*Рисунок 3.7.10. Імплементація функції analyzeStatement для аналізу оператора присвоєння*

Аналіз правильності виклику процедури відрізняється для процедур зі стандартної бібліотеки та процедур, визначених у програмі (рисунок 3.7.11). Для процедур *write* та *writeln* достатньо перевірити, що усі параметри є коректними виразами (використовується функція *analyzeWriteParams*). Для процедур *read* та *readln* дозволеними параметрами є лише назви змінних, оскільки ці процедури зберігають результат введення користувача у змінні, які вказані як параметри. Додатково потрібно перевірити, що ці змінні були визначені у програмі (використовується функція *analyzeReadParams*). Для усіх інших процедур треба спочатку перевірити, що процедура з заданою назвою існує. Наступним кроком є перевірка фактичних параметрів. По-перше, треба переконатись, що кількість формальних та фактичних параметрів співпадає. По-друге, для кожного формального параметра потрібно перевірити, що відповідний фактичний параметр є коректним виразом, і його тип відповідає типу формального параметра

(використовується функція *analyzeActualParams*). Реалізації усіх функцій, що використовуються для аналізу параметрів процедур наведено на рисунку 3.7.12.

```
analyzeStatement (Right a) ProcCall{pcName, pcParams} = case getProc (currentScope a) procName of
  Left er -> Left (AnalysisError ProcedureCallError ("Error when calling procedure '" ++ procName ++ "'!") (Just er))
  Right pri ->
    let
      analysisResult
        | elem (idValue pcName) ["write", "writeln"] = analyzeWriteParams a pcParams
        | elem (idValue pcName) ["read", "readln"] = analyzeReadParams a pcParams
        | otherwise = analyzeActualParams a (priParams pri) pcParams
    in
      case analysisResult of
        Left er -> Left (AnalysisError ProcedureCallError ("Error when calling procedure '" ++ procName ++ "'!") (Just er))
        Right a' -> Right a'
 where
  procName = idValue pcName
```

*Рисунок 3.7.11. Імплементація функції analyzeStatement для аналізу оператора виклику процедури*

```
analyzeActualParams :: Analyzer -> [ParamInfo] -> [Expression] -> Either AnalysisError Analyzer
analyzeActualParams a fps aps =
  if fpsl /= apsl
    then Left (AnalysisError ActualParameterError ("Amounts of formal and actual parameters are different! FP amount: " ++ (show fpsl) ++ ", AP amount: " ++ (show apsl)) Nothing)
    else foldl analyzeParamType (Right a) (zip3 fps aps [1 ..])
  where
    fpsl = length fps
    apsl = length aps
    analyzeParamType a' p = case (a', p) of
      (Left er, _) -> Left er
      (a''@(Right _), (fp, ap, index)) -> case analyzeExpression a'' ap of
        Left er -> Left (AnalysisError ActualParameterError ("Wrong expression for actual parameter at position " ++ (show index) ++ "!") (Just er))
        Right dt -> case expectTypes ap (getExpectedTypes (palType fp)) dt of
          Left er -> Left (AnalysisError ActualParameterError ("Error for actual parameter at position " ++ (show index) ++ "!") (Just er))
          Right _ -> a''

analyzeWriteParams :: Analyzer -> [Expression] -> Either AnalysisError Analyzer
analyzeWriteParams a aps = foldl analyzeParamType (Right a) (zip aps [0 ..])
  where
    analyzeParamType a' p = case (a', p) of
      (Left er, _) -> Left er
      (a''@(Right _), (ap, index)) -> case analyzeExpression a'' ap of
        Left er -> Left (AnalysisError ActualParameterError ("Wrong expression for actual parameter at position " ++ (show index) ++ "!") (Just er))
        Right dt -> case expectTypes ap allowedParameterTypes dt of
          Left er -> Left (AnalysisError ActualParameterError ("Error for actual parameter at position " ++ (show index) ++ "!") (Just er))
          Right _ -> a''
    allowedParameterTypes = [DTBoolean, DTChar, DTString, DTInteger, DTReal]

analyzeReadParams :: Analyzer -> [Expression] -> Either AnalysisError Analyzer
analyzeReadParams a aps = foldl analyzeParam (Right a) (zip aps [0 ..])
  where
    analyzeParam a' p = case (a', p) of
      (Left er, _) -> Left er
      (a''@(Right _), (ap@(VarRef _), index)) ->
        case analyzeExpression a'' ap of
          Left er -> Left (AnalysisError ActualParameterError ("Wrong expression for actual parameter at position " ++ (show index) ++ "!") (Just er))
          Right dt -> case expectTypes ap allowedParameterTypes dt of
            Left er -> Left (AnalysisError ActualParameterError ("Error for actual parameter at position " ++ (show index) ++ "!") (Just er))
            Right _ -> a''
      (a''@(Right _), (_, index)) -> Left (AnalysisError ActualParameterError ("Wrong expression for actual parameter at position " ++ (show index) ++ "! Only variable references are allowed as a parameters!") Nothing)
    allowedParameterTypes = [DTChar, DTString, DTInteger, DTReal]
```

*Рисунок 3.7.12. Імплементація функцій для аналізу параметрів процедур*

Для аналізу складеного оператора потрібно перевірити що кожна з інструкцій у складеному операторі є правильною за допомогою рекурсивного виклику функції *analyzeStatement*.

```
analyzeStatement a@(Right _) (Compound sttms) = foldl analyzeStatement a sttms
```

*Рисунок 3.7.13. Імплементація функції analyzeStatement для аналізу складеного оператора*

Аналіз умовного оператора “if” (рисунок 3.7.14) починається з перевірки умови. Вираз що задає умову повинен бути правильним і повертати результат булевого типу. Після цього за допомогою рекурсивного виклику функції *analyzeStatement* виконується аналіз інструкції, яка виконується якщо умова оператора є істинною. Останнім кроком є ще один рекурсивний виклик функції *analyzeStatement* для аналізу інструкції, яка повинна виконатись якщо умова оператора є хибною. Цей крок є необов’язковим, він не виконується якщо в оператора немає частини “else”.

```
analyzeStatement a@(Right _) If{iCondition, iIfRoute, iElseRoute} = case (analyzeExpression a iCondition) of
  Left er -> Left (AnalysisError IfStatementError "Error in conditional expression in 'if' statement!" (Just er))
  Right dt -> case expectType iCondition DTBoolean dt of
    Left er -> Left (AnalysisError IfStatementError "Wrong conditional expression type in 'if' statement!" (Just er))
    Right _ -> case analyzeStatement a iIfRoute of
      Left er -> Left (AnalysisError IfStatementError "Error in 'if' statement!" (Just er))
      ea@(Right _) -> case iElseRoute of
        Nothing -> ea
        Just elseSttm -> case analyzeStatement ea elseSttm of
          Left er -> Left (AnalysisError IfStatementError "Error in 'if' statement!" (Just er))
          ea'@(Right _) -> ea'
```

*Рисунок 3.7.14. Імплементація функції analyzeStatement для аналізу умовного оператора*

Аналіз циклу “while” (рисунок 3.7.15) вимагає аналізу умови циклу та тіла циклу. Щоб умова циклу була коректною спочатку потрібно перевірити, що умова є правильним виразом, а після цього переконатись, що результатом виразу є значення булевого типу. Після перевірки умови виконується перевірка тіла циклу за допомогою рекурсивного виклику функції *analyzeStatement*. Якщо результатами аналізу умови і тіла циклу є виконання без помилок, цикл вважається коректним.

```
analyzeStatement a@(Right _) While{wCondition, wBody} = case (analyzeExpression a wCondition) of
  Left er -> Left (AnalysisError WhileStatementError "Error in conditional expression in 'while' statement! " (Just er))
  Right dt -> case expectType wCondition DTBoolean dt of
    Left er -> Left (AnalysisError WhileStatementError "Wrong conditional expression type in 'while' statement!" (Just er))
    Right _ -> case analyzeStatement a wBody of
      Left er -> Left (AnalysisError WhileStatementError "Error in 'while' statement!" (Just er))
      ea@(Right _) -> ea
```

*Рисунок 3.7.15. Імплементація функції analyzeStatement для аналізу циклу “while”*

Процес аналізу циклу “repeat” (рисунок 3.7.16) є майже повністю ідентичним до аналізу циклу “while”. Єдина відмінність – оскільки у циклі “repeat” тіло циклу може містити декілька інструкцій, а не одну, перевірка тіла циклу

виконується рекурсивним виконання функції *analyzeStatement* для кожної з інструкцій.

```
analyzeStatement a@(Right _) Repeat{rCondition, rBody} = case (foldl analyzeStatement a rBody) of
  Left er -> Left (AnalysisError RepeatStatementError "Error in 'repeat' statement!" (Just er))
  ea@(Right _) -> case analyzeExpression ea rCondition of
    Left er -> Left (AnalysisError RepeatStatementError "Error in conditional expression in 'repeat' statement! " (Just er))
    Right dt -> case expectType rCondition DTBoolean dt of
      Left er -> Left (AnalysisError RepeatStatementError "Wrong conditional expression type in 'repeat' statement!" (Just er))
      Right _ -> ea
```

*Рисунок 3.7.16. Імплементація функції analyzeStatement для аналізу циклу "repeat"*

Аналіз виразів виконується функцією *analyzeExpression*, яка дещо відрізняється від інших функцій аналізу. Якщо всі інші функції мають тип результату *Either AnalysisError Analyzer*, оскільки вони можуть змінювати стан сутності типу *Analyzer*, то тип результату цієї функції - *Either AnalysisError DataType*. Тип *DataType* використовується, оскільки результатом аналізу виразу є тип, значення якого буде створено внаслідок виконання виразу.

Для аналізу константного значення достатньо повернути тип значення (рисунок 3.7.17).

```
analyzeExpression a@(Right _) (Val val) = case val of
  IntNum _ -> Right DTInteger
  RealNum _ -> Right DTReal
  Boolean _ -> Right DTBoolean
  Character _ -> Right DTChar
  Str _ -> Right DTString
```

*Рисунок 3.7.17. Імплементація функції analyzeExpression для аналізу константного значення*

При аналізі використання змінної потрібно спочатку перевірити, що змінна з використаною назвою була визначена, і повернути тип цієї змінної (рисунок 3.7.18).

```
analyzeExpression (Right a) (VarRef iden) = case getVar (currentScope a) (idValue iden) of
  Left er -> Left (AnalysisError VariableReferenceError ("Error when referencing variable '" ++ (idValue iden) ++ "'!") (Just er))
  Right vi -> Right (viType vi)
```

*Рисунок 3.7.18. Імплементація функції analyzeExpression для аналізу використання змінної*

Аналіз виклику функції (рисунок 3.7.19) дуже схожий на аналіз виклику процедури. Проте, оскільки ніяких особливих функцій, які визначені як частина стандартної бібліотеки, не було додано, процес аналізу виклику функцій є однаковим для усіх функцій. По-перше, потрібно перевірити, що функція із вказаною назвою була визначена у програмі. По-друге, необхідно перевірити правильність фактичних параметрів функції (що їх кількість співпадає з кількістю формальних параметрів, що вирази, які задають фактичні параметри, є правильними і їх типи відповідають типам формальних параметрів).

```
analyzeExpression (Right a) (FuncCall{fcName, fcParams}) = case getFunc (currentScope a) funcName of
  Left er -> Left (AnalysisError FunctionCallError ("Error when calling function '" ++ funcName ++ "'!") (Just er))
  Right fi -> case analyzeActualParams a (fiParams fi) fcParams of
    Left er -> Left (AnalysisError FunctionCallError ("Error when calling function '" ++ funcName ++ "'!") (Just er))
    Right _ -> Right (fiResType fi)
 where
  funcName = idValue fcName
```

*Рисунок 3.7.19. Імплементація функції analyzeExpression для аналізу виклику функції*

Процес аналізу виконання унарних операцій (рисунок 3.7.20) складається з 2 кроків. Першим кроком є аналіз виразу, до якого застосовується унарна операція, і отримання типу результати цього виразу. На другому кроці потрібно перевірити чи тип виразу, до якого вона застосовується, є типом, який підтримується даною операцією. Наприклад, для унарної операції “not” єдиним можливим типом, для якого вона застосовується, є булевий тип. Унарні операції “+” та “-” можуть бути застосовані лише для цілих та дійсних чисел. Для реалізації такої перевірки використовується словник *unaryOpDataTypeMap* (рисунок 3.7.21), ключами якого є операції, а значеннями – список доступних типів операндів. Таким чином, для виконання перевірки, достатньо отримати зі словника список доступних типів операндів для заданої операції, і перевірити що тип операнда належить до цього списку.

```
analyzeExpression a@(Right _) (UnOp unOp expr) = case analyzeExpression a expr of
  Left er -> Left (AnalysisError UnaryOperatorError ("Error in expression when using unary operator '" ++ (show unOp) ++ "'!") (Just er))
  Right dt -> case Map.lookup unOp unaryOpDataTypeMap of
    Nothing -> error "unreachable"
    Just dts -> case find (\(exdt, _) -> dt == exdt) dts of
      Nothing -> Left (AnalysisError UnaryOperatorError ("Error when using unary operator '" ++ (show unOp) ++ "'! Expected one of these types: " ++ (show (map fst dts)) ++ ", Got: " ++ (show dt)) Nothing)
      Just (_, resultDT) -> Right resultDT
```

*Рисунок 3.7.20. Імплементація функції analyzeExpression для аналізу використання унарної операції*

```haskell
unaryOpDataTypeMap :: Map.Map UnaryOp [(DataType, DataType)]
unaryOpDataTypeMap =
  Map.fromList
    [ (Not, [(DTBoolean, DTBoolean)]),
      ( UnaryPlus,
        [ (DTInteger, DTInteger),
          (DTReal, DTReal)
        ]
      ),
      ( UnaryMinus,
        [ (DTInteger, DTInteger),
          (DTReal, DTReal)
        ]
      )
    ]
```

*Рисунок 3.7.21. Визначення словника unaryOpDataTypeMap*

Процес аналізу використання бінарних операцій (рисунок 3.7.22) реалізовано тим же способом, що і для унарних операцій. Спочатку перевіряється коректність та обраховуються типи лівого та правого операндів, а після цього перевіряється чи для заданої операції комбінація типів операндів є доступною. Для зберігання комбінацій використовується словник *binaryOpDataTypeMap* (частину визначення якого зображено на рисунку 3.7.23). Ключем, як в словнику для унарних операцій, є власне операція, а значенням є список кортежів із трьох елементів. Перші 2 елементи кортежа є доступними типами лівого і правого операндів, а 3 елемент – типом результату виконання операції над відповідними типами. Наприклад, результатом виконання операції додавання “+” для операндів цілочисельного типу буде значення цілочисельного типу. Проте, якщо один із операндів є числом дійсного типу, то і тип результату буде числом дійсного типу. При використанні тієї ж операції додавання для двох значень символьного типу результатом буде стрічковий тип.

```haskell
analyzeExpression a@(Right _) (BinOp boOp, boLeft, boRight)) = case analyzeExpression a boLeft of
  Left er -> Left (AnalysisError BinaryOperatorError "Left operand error!" (Just er))
  Right ldt -> case analyzeExpression a boRight of
    Left er -> Left (AnalysisError BinaryOperatorError "Right operand error!" (Just er))
    Right rdt -> case Map.lookup boOp binaryOpDataTypeMap of
      Nothing -> error "unreachable"
      Just dts -> case find (\(exldt, exrdt, _) -> (ldt == exldt) && (rdt == exrdt)) dts of
        Nothing -> Left (AnalysisError BinaryOperatorError
          ("Error when using binary operator '" ++ show boOp ++ "'! Expected one of these type combinations: " ++ show (map (\(exldt, exrdt, _) -> (exldt, exrdt)) dts)) ++ ". Got: " ++ show (ldt, rdt)) Nothing)
        Just (_, _, resultDT) -> Right resultDT
```

*Рисунок 3.7.22. Імплементація функції analyzeExpression для аналізу використання бінарної операції*

```
binaryOpDataTypeMap :: Map.Map BinaryOp [(DataType, DataType, DataType)]
binaryOpDataTypeMap =
  Map.fromList
    [ ( Plus,
        [ (DTInteger, DTInteger, DTInteger),
          (DTInteger, DTReal, DTReal),
          (DTReal, DTInteger, DTReal),
          (DTReal, DTReal, DTReal),
          (DTChar, DTChar, DTString),
          (DTChar, DTString, DTString),
          (DTString, DTChar, DTString),
          (DTString, DTString, DTString)
        ]
      ),
      ( Minus,
        [ (DTInteger, DTInteger, DTInteger),
          (DTInteger, DTReal, DTReal),
          (DTReal, DTInteger, DTReal),
          (DTReal, DTReal, DTReal)
        ]
      ),
      ( Mul,
        [ (DTInteger, DTInteger, DTInteger),
          (DTInteger, DTReal, DTReal),
          (DTReal, DTInteger, DTReal),
          (DTReal, DTReal, DTReal)
        ]
      ),
      ( Div,
        [ (DTInteger, DTInteger, DTReal),
          (DTInteger, DTReal, DTReal),
          (DTReal, DTInteger, DTReal),
          (DTReal, DTReal, DTReal)
```

*Рисунок 3.7.23. Частина визначення словника binaryOpDataTypeMap*

Для аналізу операції підвищення пріоритету “()” (рисунок 3.7.24) достатньо виконати аналіз виразу, для якого була застосована ця операція.

```
analyzeExpression a@(Right _) (Paren expr) = analyzeExpression a expr
```

*Рисунок 3.7.24. Імплементація функції analyzeExpression для аналізу операції підвищення пріоритету*

Як і синтаксичний аналізатор, семантичний аналізатор буде нескладно розширити для аналізу додаткових інструкцій чи визначень. Найбільше змін він потребуватиме якщо система типів буде розширена структурованими типами, оскільки аналіз значень структурованих типів є більш складним, аніж поточна реалізація із базовими типами.

### 3.8 Розробка модуля Interpreter – інтерпретатора

Якщо етап семантичного аналізу завершився без помилок, наступним етапом є інтерпретація програми. У модулі *Interpreter* (вихідний файл *Interpreter.hs*) визначаються усі необхідні для інтерпретації програми функції. Його основна мета – інтерпретувати програму використовуючи АСД.

Цей модуль експортує лише одну функцію – *applyInterpreter*, яка використовується для інтерпретації програми. Вона має єдиний параметр типу *Program*. Він представляє АСД, яке використовується для інтерпретації. Ця функція повертає значення типу *IO (Either InterpretationError Interpreter)*. Використання типу *IO* є необхідним, оскільки інтерпретатор може виконувати операції введення-виведення (це буде більш детально описано при описі інтерпретації процедур, що виконують відповідні операції). Якщо в ході інтерпретації програми виникла помилка – результатом інтерпретації є відповідна помилка часу виконання типу *InterpretationError*. Якщо інтерпретація програми закінчилась без помилок, то результатом є значення типу *Interpreter* (цей тип зберігає інформацію про стан інтерпретатора), воно використовується для виведення інформації про стан інтерпретатора після виконання програми в розширеному режимі. Типи *InterpretationError* та *Interpreter* будуть більш детально описані далі.

Імплементація функції *applyInterpreter* (рисунок 3.8.1) є досить простою. Спочатку вона викликає функцію *buildBlockScopeRecord* для побудови сутності типу *ScopeRecord*, яка зберігає інформацію про головний програмний блок, а також усі змінні, функції та процедури, визначені у ньому. Після цього на основі створеної сутності *ScopeRecord* створюється сутність типу *ActivationRecord*, що містить інформацію про стан змінних програмного блоку. Ця сутність першою додається до стеку викликів, оскільки виконання тіла програмного блоку відповідає виконанню усієї програми, а змінні, функції та процедури, визначені у ньому, вважаються глобальними і доступні в усіх блоках програми. Останнім кроком є інтерпретація тіла програмного блоку, яка виконується викликом функції *interpretStatement*. Усі описані типи та функції буде більш детально описано далі.

```
applyInterpreter :: Program -> IO (Either InterpretationError Interpreter)
applyInterpreter p = interpretStatement (pure (Right interpreter)) programBody
 where
  sr = buildProgramScopeRecord p
  programBody = srBody sr
  initialVars = foldl (\varMap varInfo -> insert (viName varInfo) varInfo varMap) empty (variables sr)
  interpreter =
    I
      { callStack =
          [ AR
              { arName = srName sr,
                arLevel = 1,
                arType = RTProgram,
                vars = initialVars
              }
          ],
        currentSR = sr
      }
```

*Рисунок 3.8.1. Імплементація функції applyInterpreter*

Для спрощення інтерпретації програми було створено тип *ScopeRecord* (рисунок 3.8.2), сутності якого зберігають усю необхідну інформацію для інтерпретації блоку програми. Для ідентифікації блоку використовуються поля, що зберігають інформацію про назву блоку (базуючись на назві програми, функції чи процедури, яка створює відповідний блок) і тип блоку (представлений типом *RecordType*, блок може мати тип програми, функції чи процедури). Також

зберігаються поля, які є словниками з інформацією про змінні, функції та процедури, визначені у блоці (інформація про змінні зберігається у сутностях типу *VarInfo*, а про функції та процедури – у сутностях типу *ScopeRecord*, оскільки вони є блоками), і поле, яке представляє тіло блоку. Для блоків функцій і процедур додатково визначені поля, які зберігають інформацію про список параметрів (список сутностей типу *ParamInfo*) та посилання на батьківський блок (теж сутність типу *ScopeRecord*). Блоки функцій також містять інформацію про тип значення, яке є результатом функції. Для представлення опційних полів використовується тип *Maybe*, який був описаний раніше.

```
data ScopeRecord = SR
  { srName :: String,
    srLevel :: Int,
    srType :: RecordType,
    parameters :: Maybe [ParamInfo],
    srReturnType :: Maybe DataType,
    variables :: Map String VarInfo,
    functions :: Map String ScopeRecord,
    procedures :: Map String ScopeRecord,
    srBody :: Statement,
    parentSR :: Maybe ScopeRecord
  }

data RecordType = RTProgram | RTFunction | RTProcedure deriving (Show)

data VarInfo = VI {viName :: String, viType :: DataType, viValue :: Maybe Value} deriving (Show)
data ParamInfo = PAI {paiName :: String, paiType :: DataType} deriving (Show)
```

*Рисунок 3.8.2. Визначення типу ScopeRecord*

Під час виконання програми потрібно зберігати інформацію про поточний стан програми, зокрема про значення усіх змінних, створених в ході виконання. Для цього використовується структура даних під назвою “Запис активації”. Ця структура даних зберігає усю інформацію про стан поточного блоку. В інтерпретаторі ця структура даних представлена типом *ActivationRecord* (рисунок 3.8.3). Як і в сутності типу *ScopeRecord*, для ідентифікації блоку зберігається його назва і тип. Поле *vars* зберігає словник, ключами якого є назви змінних, а значеннями – стан відповідних змінних.

```haskell
data ActivationRecord = AR
  { arName :: String,
    arLevel :: Int,
    arType :: RecordType,
    vars :: Map String VarInfo
  }
  deriving (Show)
```

*Рисунок 3.8.3. Визначення типу ActivationRecord*

Під час виконання програми для виконання кожного блоку потрібно створювати новий ЗА, а після завершення виконання блоку – повертатись до попереднього ЗА, якщо такий існує. Якщо функція чи процедура викликається всередині іншої функції чи процедури, виникає необхідність створити ЗА, що зберігає стан відповідного проміжного виклику. Для цього використовується структура даних, яка має назву “Стек викликів”. Стек викликів дозволяє відстежити інформацію про виклики функцій та процедур, їх послідовність та результати. Для керування стеком викликів та станом програми було створено тип *Interpreter* (рисунок 3.8.4). Поле *callStack* зберігає інформацію про стек викликів (представлене списком ЗА типу *ActivationRecord*, оскільки у Haskell операції роботи зі списками дозволяють використовувати його у якості стеку), а поле *currentSR* зберігає інформацію про усі доступні елементи поточного блоку (представлене типом *ScopeRecord*). Завдяки полю *currentSR* інтерпретатор має можливість коректно використовувати не лише доступні в поточному блоці змінні, функції та процедури, а й визначені у батьківських блоках. Це дозволяє забезпечити коректну роботу із областями видимості. Ще одним наслідком такої реалізації є можливість “затінення” змінної – ситуації, при якій змінна у певній області видимості має таку ж назву, як і змінна в батьківській області видимості. В такій ситуації при звертанні за цією назвою використовується змінна, яка знаходиться у найбільш вкладеній області видимості.

```
data Interpreter = I
  { callStack :: [ActivationRecord],
    currentSR :: ScopeRecord
  }
  deriving (Show)
```

*Рисунок 3.8.4. Визначення типу Interpreter*

Оскільки кожна змінна має область видимості, то інформація про змінні, визначені у блоці, зберігається у ЗА, що відповідає цьому блоку (поле *vars* типу *ActivationRecord*). Значення змінної певного типу у Pascal представляється значенням відповідного типу у Haskell. Типу *Boolean* у Pascal відповідає тип *Bool* у Haskell, типу *Integer* – тип *Int*, типу *Real* – тип *Double*, типу *Character* – тип *Char*, а типу *String* – тип *String* з такою ж назвою. При інтерпретації програми є декілька можливих подій, що впливають на стан змінних. По-перше, значення відповідних змінних можуть змінюватись при застосування оператора присвоєння, це вимагає оновлення значення змінної у словнику. По-друге при створенні нового ЗА (при виклику функції або процедури) у його словник змінних додаються усі параметри (оскільки параметри є змінними) та визначені у блоці змінні. При закінченні виконання функції чи процедури ЗА вилучається зі стеку викликів, а інтерпретатор більше не має доступу до змінних, що знаходились у цьому блоці. На вилучений зі стеку ЗА більше немає жодних посилань, а тому Haskell може очистити пам’ять і видалити ЗА. Відповідно, керування пам’яттю, яку використовують змінні, в інтерпретаторі відбувається автоматично за допомогою вбудованого у Haskell збирача сміття. Таким чином, правильна робота із ЗА та вчасне вилучення їх зі стеку викликів дозволяє перекласти необхідність керування пам’яттю на Haskell, що значно спрощує процес інтерпретації програми, оскільки немає необхідності вручну слідкувати за виділенням та очищенням пам’яті для кожної змінної.

При виконанні програми можуть виникати помилки часу виконання. Вони представлені типом *InterpretationError* (рисунок 3.8.5). Його структура схожа на

структуру типу помилок, що виникають на етапі семантичного аналізу. Він теж містить інформацію про тип помилки, повідомлення про помилку та опційне посилання на помилку, яка є причиною виникнення поточної помилки. Тип помилки визначається окремим типом *InterpretationErrorType*. Основні типи помилок включають *DivisonByZeroError* (виникає при виконання операції ділення якщо дільник дорівнює 0) та *WrongReadProcedureArgumentError* (виникає якщо не вдалося зчитати введену зі стандартного потоку інформацію і перетворити її у змінну відповідного типу). Цей тип помилки є специфічним і використовується тільки при виконанні процедур *read* та *readln*, оскільки їх коректне виконання залежить від інформації, доступної лише під час виконання програми. Останній тип помилки – *WrongTypeError*, помилки цього типу виникають у місцях, де тип виразу не співпадає з очікуваним. Оскільки перед інтерпретацією виконується семантичний аналіз програми, під час виконання програми ця помилка не повинна виникати. Якщо ж вона виникла, це означає що було допущено помилку при розробці семантичного аналізатора.

```
data InterpretationError = InterpretationError InterpretationErrorType String (Maybe InterpretationError) deriving (Show)
data InterpretationErrorType
  = WrongTypeError
  | DivisionByZeroError
  | WrongReadProcedureArgumentError
  deriving (Show)
```

*Рисунок 3.8.5. Визначення типу InterpretationError*

Оскільки всі визначення у Pascal розташовуються у блоці перед тілом блоку, ще перед початком інтерпретації можна побудувати сутності типу *ScopeRecord* для кожного блоку використовуючи АСД. Для побудови *ScopeRecord* з АСД використовується функція *buildProgramScopeRecord* (рисунок 3.8.6). Вона приймає на вхід тип *Program* (який є коренем АСД) і будує сутність типу *ScopeRecord* викликаючи функцію *buildBlockScopeRecord*.

```
buildProgramScopeRecord :: Program -> ScopeRecord
buildProgramScopeRecord Program{pHeader, pBody} = buildBlockScopeRecord (idValue pHeader) 1 RTProgram Nothing Nothing pBody
```

*Рисунок 3.8.6. Імплементація функції buildProgramScopeRecord*

Функція *buildBlockScopeRecord* (рисунок 3.8.7) створює *ScopeRecord* для блоку програми. Параметри функції включають назву, рівень вкладення, тип блоку, список параметрів (для блоків функцій та процедур), тип значення, яке повертає блок (для блоків функцій) а також інформацію про блок, який аналізується (список визначень та тіло блоку). Функція аналізує усі визначення блоку викликаючи функцію *buildDeclarationScopeRecord* для кожного із них, і додає проаналізовані визначення до словників змінних, функцій чи процедур, залежно від типу визначення. Результатом виконання функції є сутність типу *ScopeRecord*, яка зберігає усю необхідну інформацію про блок.

```
buildBlockScopeRecord :: String -> Int -> RecordType -> Maybe [ParamInfo] -> Maybe DataType -> Block -> ScopeRecord
buildBlockScopeRecord nm lvl rt pms rett Block{bDeclarations, bBody} =
  let
    initialSR =
      SR
        { srName = nm,
          srLevel = lvl,
          srType = rt,
          parameters = pms,
          srReturnType = rett,
          variables = empty,
          functions = empty,
          procedures = empty,
          srBody = bBody,
          parentSR = Nothing
        }
  in
    foldl buildDeclarationScopeRecord initialSR bDeclarations
```

*Рисунок 3.8.7. Імплементація функції buildBlockScopeRecord*

Функція *buildDeclarationScopeRecord* отримує на вхід сутність типу *ScopeRecord*, яка зберігає інформацію про поточний блок. Залежно від типу визначення, ця сутність модифікується додаванням нових змінних, функцій та процедур.

Якщо функція отримує на вхід визначення змінної, інформація про змінну додається до словника змінних поточної сутності типу *ScopeRecord* (рисунок 3.8.8). Ця інформація використовується інтерпретатором для створення змінних при створенні ЗА для відповідного блоку в процесі виконання програми.

```
buildDeclarationScopeRecord sr (VarDecl varDecls) = foldl buildVarDeclaration sr varDecls
 where
  buildVarDeclaration sr' vd =
    let
      varName = idValue (vName vd)
      varType = vType vd
      newVI = VI{viName = varName, viType = varType, viValue = vValue vd}
     in
      sr'{variables = insert varName newVI (variables sr')}
```

*Рисунок 3.8.8. Імплементація функції buildDeclarationScopeRecord для визначень змінних*

Якщо функція отримує на вхід визначення процедури, тоді спочатку викликається функція *buildBlockScopeRecord* для побудови сутності типу *ScopeRecord*, яка містить інформацію про цю процедуру. На вхід функції *buildBlockScopeRecord* передаються назва процедури, рівень вкладення на 1 більший, ніж поточний, список параметрів процедури та тіло процедури. Отримана в результаті виконання сутність оновлюється (їй додається посилання на батьківську сутність типу *ScopeRecord*) і зберігається у словнику процедур отриманої на вході сутності типу *ScopeRecord* (рисунок 3.8.9).

```
buildDeclarationScopeRecord sr (ProcDecl pr) =
  let
    prName = idValue (pName pr)
    lvl = (srLevel sr) + 1
    pms = Just (map (\p -> PAI{paiName = (idValue (fpName p)), paiType = fpType p}) (pParams pr))
    retType = Nothing
    body = pBlock pr
    procedureSR = buildBlockScopeRecord prName lvl RTProcedure pms retType body
    updatedSR =
      sr
        { procedures = insert prName (procedureSR{parentSR = Just updatedSR}) (procedures sr)
        }
   in
    updatedSR
```

*Рисунок 3.8.9. Імплементація функції buildDeclarationScopeRecord для визначень процедур*

Якщо функція отримує на вхід визначення функції, виконуються ті ж дії, що й для процедури з невеликими відхиленнями. По-перше, додатково при виклику функції *buildBlockScopeRecord* параметром передається тип результату функції. По-друге, інформація про функцію зберігається у словнику функцій, а не процедур (рисунок 3.8.10).

```
buildDeclarationScopeRecord sr (FuncDecl fn) =
  let
    fnName = idValue (fName fn)
    lvl = (srLevel sr) + 1
    pms = Just (map (\p -> PAI{paiName = (idValue (fpName p)), paiType = fpType p}) (fParams fn))
    retType = Just (fResType fn)
    body = fBlock fn
    functionSR = buildBlockScopeRecord fnName lvl RTFunction pms retType body
    updatedSR =
      sr
        { functions = insert fnName (functionSR{parentSR = Just updatedSR}) (functions sr)
        }
  in
    updatedSR
```

*Рисунок 3.8.10. Імплементація функції buildDeclarationScopeRecord для визначень функцій*

Функція *interpretStatement* інтерпретує інструкцію залежно від її типу.

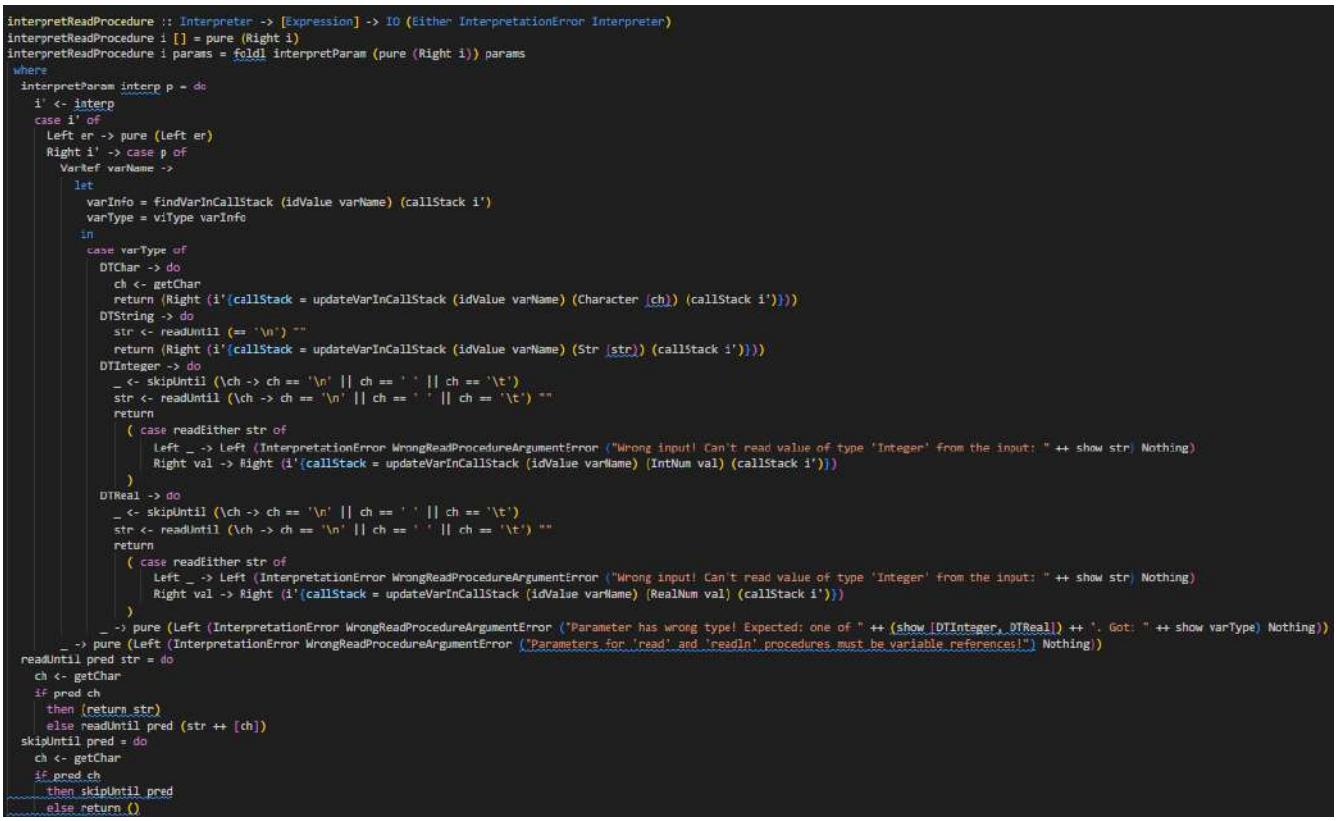
Для виконання оператора присвоєння (рисунок 3.8.11) спочатку обраховується значення виразу, який необхідно присвоїти, за допомогою функції *interpretExpression*. Якщо при обрахуванні виразу помилок не відбулось, то відбувається пошук змінної та ЗА, в якому вона знаходиться (оскільки присвоєння може виконуватись для змінної, що знаходиться не в поточному блоці, а в одному із батьківських блоків), після чого значення змінної зберігається у відповідному ЗА. Якщо ж відбулась помилка, робота інтерпретатора закінчується помилкою часу виконання.

```
interpretStatement :: IO (Either InterpretationError Interpreter) -> Statement -> IO (Either InterpretationError Interpreter)
interpretStatement ioi sttm = do
  intr <- ioi
  case (intr, sttm) of
    (Left _, _) -> return intr
    (Right i, Assignment{aName, aValue}) -> do
      res <- interpretExpression i aValue
      return
        ( case res of
            Left er -> Left er
            Right (i', v) ->
              let
                varName = idValue aName
                vi = findVarInCallStack varName (callStack i')
                finalValue = charAsString (viType vi) v
              in
                Right (i'{callStack = updateVarInCallStack varName finalValue (callStack i')})
        )
```

*Рисунок 3.8.11. Імплементація функції interpretStatement для виконання оператора присвоєння*

Виконання виклику процедури починається із обрахування значень виразів, які є фактичними параметрами процедури, використовуючи функцію *interpretExpression* для кожного з них. Після цього, процес виконання залежить від того, чи є ця процедура однією зі стандартної бібліотеки, чи вона визначена у програмі.

Якщо викликається одна з процедур *read, readln, write, writeln,* то використовуються можливості Haskell для виконання операцій введення-виведення. Для реалізації процедур *write* та *writeln* використовуються функції Haskell *putStr* та *putStrLn,* які виводять значення у стандартний потік введення-виведення. Щоб імплементувати процедури *read* та *readln* так, як вони працюють у Pascal, використовується функція *interpretReadProcedure* (рисунок 3.8.12). Залежно від типу значення, яке необхідно отримати з потоку введення-виведення, ця функція сприймає введену інформацію по-різному. Для цілих чисел очікується послідовність цифр, для дійсних чисел – 2 цілих числа, розділених крапкою. Для символів очікується будь-який символ, причому якщо було введено декілька символів, то у відповідну змінну зберігається лише перший символ, а решта ігнорується. Для стрічок очікується будь-яка послідовність символів, яка закінчується спеціальним символом переходу на новий рядок. Якщо процедури були викликані із декількома параметрами, програма може очікувати декілька рядків введення від користувача, залежно від типу параметрів процедури. Якщо введенні користувачем дані можуть бути перетворені на значення відповідних типів, то ці значення присвоюються змінним, що є аргументами процедури. Якщо хоча б одне зі значень не може бути створене із введених даних – відбувається помилка виконання типу *WrongReadProcedureArgumentError* і робота інтерпретатора завершується.



```haskell
interpretReadProcedure :: Interpreter -> [Expression] -> IO (Either InterpretationError Interpreter)
interpretReadProcedure i [] = pure (Right i)
interpretReadProcedure i params = foldl interpretParam (pure (Right i)) params
 where
  interpretParam interp p = do
    i' <- interp
    case i' of
      Left er -> pure (Left er)
      Right i' -> case p of
        VarRef varName ->
          let
            varInfo = findVarInCallStack (idValue varName) (callStack i')
            varType = viType varInfo
          in
            case varType of
              DTChar -> do
                ch <- getChar
                return (Right (i'{callStack = updateVarInCallStack (idValue varName) (Character ch) (callStack i')}))
              DTString -> do
                str <- readUntil (== '\n') ""
                return (Right (i'{callStack = updateVarInCallStack (idValue varName) (Str str) (callStack i')}))
              DTInteger -> do
                _ <- skipUntil (\ch -> ch == '\n' || ch == ' ' || ch == '\t')
                str <- readUntil (\ch -> ch == '\n' || ch == ' ' || ch == '\t') ""
                return
                  ( case readEither str of
                      Left _ -> Left (InterpretationError WrongReadProcedureArgumentError ("Wrong input! Can't read value of type 'Integer' from the input: " ++ show str) Nothing)
                      Right val -> Right (i'{callStack = updateVarInCallStack (idValue varName) (IntNum val) (callStack i')})
                  )
              DTReal -> do
                _ <- skipUntil (\ch -> ch == '\n' || ch == ' ' || ch == '\t')
                str <- readUntil (\ch -> ch == '\n' || ch == ' ' || ch == '\t') ""
                return
                  ( case readEither str of
                      Left _ -> Left (InterpretationError WrongReadProcedureArgumentError ("Wrong input! Can't read value of type 'Integer' from the input: " ++ show str) Nothing)
                      Right val -> Right (i'{callStack = updateVarInCallStack (idValue varName) (RealNum val) (callStack i')})
                  )
              _ -> pure (Left (InterpretationError WrongReadProcedureArgumentError ("Parameter has wrong type! Expected: one of " ++ (show [DTInteger, DTReal]) ++ ". Got: " ++ show varType) Nothing))
        _ -> pure (Left (InterpretationError WrongReadProcedureArgumentError ("Parameters for 'read' and 'readln' procedures must be variable references!") Nothing))
  readUntil pred str = do
    ch <- getChar
    if pred ch
      then (return str)
      else readUntil pred (str ++ [ch])
  skipUntil pred = do
    ch <- getChar
    if pred ch
      then skipUntil pred
      else return ()
```

*Рисунок 3.8.12. Імплементація функції interpretReadProcedure*

Оскільки Haskell є функціональною мовою програмування, функції повинні бути “чистими”, тобто такими, результат виконання яких завжди буде однаковим при одних і тих же вхідних даних. Функції які використовують операції введення-виведення не можуть бути чистими, оскільки залежать від користувача та потоку введення-виведення, який використовується. У Haskell для розділення “чистих” функцій та безпосередньо взаємодії із потоками введення-виведення використовується монада *IO*. Використання цієї монади дозволяє використовувати одиночні операції читання або запису в потоки введення-виведення, або комбінувати декілька таких операцій у послідовності. Проте усі дані, що використовуються у функціях, які використовують такі операції, повинні “обгортати” тип результату у тип *IO*. Таким чином, функція *interpretStatement* (як і функція *applyInterpreter*) повертає значення типу *IO (Either InterpretationError Interpreter)* замість типу *Either InterpretationError Interpreter*.

Для інтерпретації усіх інших процедур, які не потребують введення або виведення даних, процес інтерпретації спільний (рисунок 3.8.13). Перед викликом процедури у стек викликів додається новий ЗА, який містить усі змінні, визначені у процедурі. Після цього рекурсивно викликається функція *interpretStatement*, яка виконує тіло програми. Після завершення виконання тіла процедури з верхівки стеку викликів вилучається ЗА, який відповідає цьому виклику, і виконання програми продовжується.

```
(Right i, ProcCall{pcName, pcParams}) -> do
  res <- interpretParams i pcParams
  case res of
    Left er -> pure (Left er)
    Right (i', parameterValues) ->
      case (idValue pcName) of
        "write" -> do
          putStr (foldl (++) "" (map printValue (reverse parameterValues)))
          return (Right (i'))
        "writeln" -> do
          putStrLn (foldl (++) "" (map printValue (reverse parameterValues)))
          return (Right (i'))
        "read" -> interpretReadProcedure i' pcParams
        "readln" -> interpretReadProcedure i' pcParams
        pcNameStr ->
          let
            currSR = currentSR i'
            procedureSR = findProcedureSR pcNameStr (currentSR i')
            paramVars = buildParameterMap (fromJust (parameters procedureSR)) parameterValues
            declaredVars = foldl (\varMap varInfo -> insert (viName varInfo) varInfo varMap) paramVars (variables procedureSR)
            procedureAR =
              AR
                { arName = (srName procedureSR),
                  arLevel = arLevel (head (callStack i')),
                  arType = srType procedureSR,
                  vars = declaredVars
                }
          in
            do
              res' <- interpretStatement (pure (Right i'{currentSR = procedureSR, callStack = procedureAR : (callStack i')})) (srBody procedureSR)
              return
                ( case res' of
                    Left er -> Left er
                    Right i'' -> Right (i''{currentSR = currSR, callStack = tail (callStack i'')})
                )
```

*Рисунок 3.8.13. Імплементація функції interpretStatement для виконання виклику процедури*

Для виконання складеного оператора потрібно послідовно виконати усі інструкції, що належать складеному оператору (рисунок 3.8.14).

```
(Right _, Compound sttms) -> foldl interpretStatement ioi sttms
```

*Рисунок 3.8.14. Імплементація функції interpretStatement для виконання складеного оператора*

Виконання умовного оператора “if” (рисунок 3.8.15) починається з обчислення виразу, який є умовою оператора. Якщо результатом виразу є булеве значення “true” рекурсивно викликається функція *interpretStatement* для виконання

інструкції, яка має виконатись при істинності умови оператора. Якщо ж результатом виразу є булеве значення “false”, то при наявності частини “else” виконується інструкція, яка належить до цієї частини. Якщо ж частина “else” відсутня, то виконання умовного оператора завершується.

```
(Right i, If{iCondition, iIfRoute, iElseRoute}) -> do
  res <- interpretExpression i iCondition
  case res of
    Left er -> pure (Left er)
    Right (i', v) -> case expectBooleanType v of
      Left er -> pure (Left er)
      Right (Boolean True) -> interpretStatement (pure (Right i')) iIfRoute
      Right (Boolean False) -> case iElseRoute of
        Just sttm -> interpretStatement (pure (Right i')) sttm
        Nothing -> pure (Right i')
```

*Рисунок 3.8.15. Імплементація функції interpretStatement для виконання умовного оператора*

Щоб інтерпретувати цикл “while” (рисунок 3.8.16) спочатку, як і при виконанні умовного оператора “if”, обраховується значення виразу, який є умовою циклу. Якщо умова є істинною, рекурсивно викликається функція *interpretStatement* для інтерпретації тіла циклу. Після виконання тіла циклу процес повторюється спочатку, функція *interpretStatement* викликається для інструкції циклу “while”. Якщо ж умова циклу є хибною, виконання циклу завершується.

```
(Right i, whileSttm@(While{wCondition, wBody})) -> do
  res <- interpretExpression i wCondition
  case res of
    Left er -> pure (Left er)
    Right (i', v) -> case expectBooleanType v of
      Left er -> pure (Left er)
      Right (Boolean False) -> pure (Right i')
      Right (Boolean True) -> do
        res' <- interpretStatement (pure (Right i')) wBody
        case res' of
          Left er -> pure (Left er)
          Right i'' -> interpretStatement (pure (Right i'')) whileSttm
```

*Рисунок 3.8.16. Імплементація функції interpretStatement для виконання циклу “while”*

Процес виконання циклу “repeat” (рисунок 3.8.17) нагадує процес для циклу “while”. Основною відмінністю є порядок виконання, оскільки для циклу “repeat” спочатку виконуються інструкції з тіла циклу, а після їх виконання перевіряється значення умови циклу. Така особливість гарантує, що тіло циклу “repeat” буде виконано хоча б 1 раз, в той час як тіло циклу “while” може не виконатись взагалі, якщо умова циклу одразу є хибною. Ще однією відмінністю є необхідність виконання декількох інструкцій із тіла циклу, оскільки тіло циклу “repeat” може містити більше ніж одну інструкцію. В той же час, тіло циклу “while” повинне містити лише одну інструкцію, а якщо потрібно застосувати декілька інструкцій, треба використати складений оператор.

```
(Right i, repeatSttm@(Repeat{rCondition, rBody})) -> do
  res <- foldl interpretStatement (pure (Right i)) rBody
  case res of
    Left er -> pure (Left er)
    Right i' -> do
      res' <- interpretExpression i' rCondition
      case res' of
        Left er -> pure (Left er)
        Right (i'', v) -> case expectBooleanType v of
          Left er -> pure (Left er)
          Right (Boolean False) -> pure (Right i'')
          Right (Boolean True) -> interpretStatement (pure (Right i'')) repeatSttm
```

*Рисунок 3.8.17. Імплементація функції interpretStatement для виконання циклу “repeat”*

Для обчислення виразів використовується функція *interpretExpression*. На відміну від функції *interpretStatement*, яка повертає сутність типу *Interpreter* з оновленим станом інтерпретатора, вона повертає кортеж двох елементів типу *(Interpreter, Value)* (рисунок 3.8.18). Першим елементом кортежу є сутність типу *Interpreter*, яка містить інтерпретатор (оскільки виклики функцій можуть змінювати стан інтерпретатора), а другим – сутність типу *Value*, яка представляє значення виразу, що обраховується.

```
interpretExpression :: Interpreter -> Expression -> IO (Either InterpretationError (Interpreter, Value))
```

*Рисунок 3.8.18. Сигнатура функції interpretExpression*

Якщо вираз складається лише з константи примітивного типу, то його результатом є значення цієї константи (рисунок 3.8.19).

```
interpretExpression i (Val v) = pure (Right (i, v))
```

*Рисунок 3.8.19. Імплементація функції interpretExpression для обчислення значення константи*

Якщо вираз є використанням змінної, то його результатом є значення відповідної змінної (рисунок 3.8.20). Якщо змінна була оголошена, проте ще не ініціалізована, результатом є стандартне значення відповідного типу (для його отримання використовується функція *getDefaultValue*, імплементацію якої наведено на рисунку 3.8.21). Для цілих та дійсних чисел – це 0, для булевого типу – константа “false”, для символьного типу – “нульовий” символ “NUL” (який застосовується для позначення кінця текстової стрічки), для стрічкового типу – порожня стрічка.

```
interpretExpression i (VarRef iden) =
  pure
    ( case varValue of
        Just v -> Right (i, v)
        Nothing -> Right (i, getDefaultValue varType)
    )
 where
  vi = findVarInCallStack (idValue iden) (callStack i)
  varType = viType vi
  varValue = viValue vi
```

*Рисунок 3.8.20. Імплементація функції interpretExpression для обчислення значення змінної*

```
getDefaultValue :: DataType -> Value
getDefaultValue DTInteger = IntNum 0
getDefaultValue DTReal = RealNum 0.0
getDefaultValue DTBoolean = Boolean False
getDefaultValue DTChar = Character (toEnum 0)
getDefaultValue DTString = Str ""
```

*Рисунок 3.8.21. Імплементація функції getDefaultValue*

Якщо вираз є унарною операцією (рисунок 3.8.22), спочатку обраховується значення виразу, до якого застосовується ця операція. Після цього, до отриманого

значення застосовується відповідна операція (“not” – перетворює булеве значення на протилежне, унарний “+” просто повертає число, до якого застосовується, а унарний “-” – змінює знак числа, до якого застосовується).

```
interpretExpression i (UnOp unOp expr) = do
  res <- interpretExpression i expr
  return
    ( case res of
        Left er -> Left er
        Right (i', v) -> case unOp of
          Not -> case v of
            Boolean val -> Right (i', Boolean (not val))
            _ -> Left (InterpretationError WrongTypeError ("Wrong value type in '" ++ (show unOp) ++ "' operator!") Nothing)
          UnaryPlus -> case v of
            IntNum _ -> Right (i', v)
            RealNum _ -> Right (i', v)
            _ -> Left (InterpretationError WrongTypeError ("Wrong value type in '" ++ (show unOp) ++ "' operator!") Nothing)
          UnaryMinus -> case v of
            IntNum val -> Right (i', IntNum (-val))
            RealNum val -> Right (i', RealNum (-val))
            _ -> Left (InterpretationError WrongTypeError ("Wrong value type in '" ++ (show unOp) ++ "' operator!") Nothing)
    )
```

*Рисунок 3.8.22. Імплементація функції interpretExpression для обчислення результату застосування унарної операції*

Для обрахування результату застосування бінарної операції (рисунок 3.8.23) спочатку потрібно обрахувати значення виразів, до яких застосовується операція, а потім застосувати відповідну операцію до отриманих значень. У специфікації Pascal вказано, що для бінарних операцій порядок обрахунку операндів “залежить від імплементації” [6, с. 48]. В поточній реалізації обраховується спочатку значення лівого операнда, а потім правого. Для булевих операторів “and” (логічне множення) та “or” (логічне додавання) було реалізовано логіку “короткого замикання”, тобто якщо значення лівого операнда дозволяє визначити результат операції, то значення правого операнда не обраховується. Для операції “and” якщо значення лівого операнда - “false”, то результатом операції буде “false”. Для операції “or” якщо значення лівого операнда – “true”, то результатом операції є “true”. Особливості обрахунку операндів потрібно враховувати, якщо обидва операнди є викликами фунцій, які впливають на змінні, доступні обом функціям, оскільки результат виконання функцій може бути неочікуваним. Результатом виконання операцій “/” (ділення), “div” (цілочисельне ділення) та “mod” (остача

від ділення) може бути помилка типу *DivisionByZeroError*, якщо значення правого операнда (дільника) дорівнює 0.

```
interpretExpression i (BinOp{boOp, boLeft, boRight}) = do
  resl <- interpretExpression i boLeft
  case resl of
    Left er -> pure (Left er)
    Right (i', lv) -> case boOp of
      And -> case lv of
        Boolean False -> pure (Right (i', Boolean False))
        Boolean lvv@True -> do
          resr <- interpretExpression i' boRight
          return
            ( case resr of
                Left er -> Left er
                Right (i'', Boolean rvv) -> Right (i'', Boolean (lvv && rvv))
                Right _ -> Left (InterpretationError WrongTypeError ("Wrong value type in '" ++ (show boOp) ++ "' operator!") Nothing)
            )
        _ -> pure (Left (InterpretationError WrongTypeError ("Wrong value type in '" ++ (show boOp) ++ "' operator!") Nothing))
      Or -> case lv of
        Boolean True -> pure (Right (i', Boolean True))
        Boolean lvv@False -> do
          resr <- interpretExpression i' boRight
          return
            ( case resr of
                Left er -> Left er
                Right (i'', Boolean rvv) -> Right (i'', Boolean (lvv || rvv))
                Right _ -> Left (InterpretationError WrongTypeError ("Wrong value type in '" ++ (show boOp) ++ "' operator!") Nothing)
            )
        _ -> pure (Left (InterpretationError WrongTypeError ("Wrong value type in '" ++ (show boOp) ++ "' operator!") Nothing))
```

*Рисунок 3.8.23. Імплементація функції interpretExpression для обчислення результату застосування бінарної операції*

Результатом виконання операції підвищення пріоритету “()” є результат виконання виразу, до якого вона була застосована (рисунок 3.8.24).

```
interpretExpression i (Paren expr) = interpretExpression i expr
```

*Рисунок 3.8.24. Імплементація функції interpretExpression для обчислення результату застосування операції підвищення пріоритету*

Процес обчислення виклику функції (рисунок 3.8.25) схожий на процес інтерпретації виклику процедури. Спочатку обраховуються значення виразів, які є фактичними параметрами функції. Наступним кроком є створення нового ЗА та додавання його до стеку викликів. Словник змінних створеного ЗА містить параметри функції та їх значення, змінні, оголошені всередині функції, а також змінну із назвою функції, яка є неініціалізованою і використовується для повернення результату виконання функції. Після цього виконується тіло функції. Після виконання тіла функції необхідно отримати значення змінної з ім’ям

функції з ЗА, що відповідає функції. Значення цієї змінної є результатом виконання функції. Останнім кроком є вилучення ЗА зі стеку викликів.

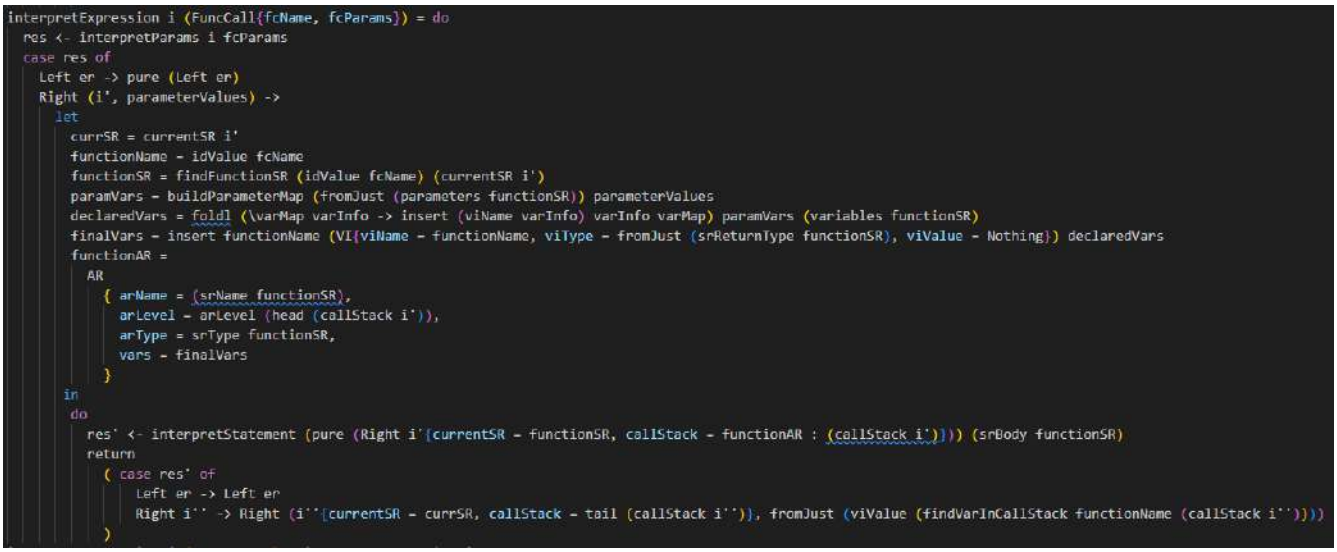



*Рисунок 3.8.25. Імплементація функції interpretExpression для обчислення результату виклику функції*

Розроблений інтерпретатор підтримує усі інструкції та вирази, що створюються синтаксичним аналізатором та аналізуються семантичним аналізатором. Він досить легко може бути розширений для підтримки більшої кількості елементів Pascal. Кількість необхідних змін для додавання нових елементів залежить безпосередньо від того, який елемент додається. Наприклад, додавання циклу “for” буде вимагати незначних змін, оскільки принцип його роботи дуже схожий на уже реалізовані цикли “while” та “repeat”. В той же час, зміна системи типів та додавання структурованих типів даних вимагатиме змін як на етапі інтерпретації інструкцій, так і при інтерпретації виразів.

### 3.9 Розробка модуля Main – головного модуля програми

Для виконання програм у Haskell необхідно визначити функцію *main* – вона є точкою входу в програму, після виконання цієї функції виконання програми завершується. Для імплементації цієї функції (а також декількох допоміжних) було вирішено створити окремий модуль. Цей модуль має назву *Main* (вихідний

файл – *Main.hs*). Його основна мета – отримати ввід користувача та забезпечити виконання бажаної команди.

Функція *main* (рисунок 3.9.1) викликається про запуску виконуваного файла програми. Вона викликає функцію *getArgs* для отримання аргументів, переданих програмі користувачем, перевіряє, що користувач передав хоча б 1 аргумент (команду), та викликає функцію *executeCommand* для виконання введеної команди.

```
main :: IO ()
main = do
  args <- getArgs
  if (length args < 1)
    then putStrLn "ERROR: Command was not provided"
    else executeCommand (head args) (tail args)
```

*Рисунок 3.9.1. Імплементація функції main*

Функція *executeCommand* (рисунок 3.9.2) відповідає за виконання команди, заданої користувачем. Вона має 2 параметри – назву команди та список аргументів команди. У роботі було реалізовано лише 1 команду “run”, яка виконує інтерпретацію програми із заданого вихідного файлу. Відповідно, якщо користувач використав команду “run”, то викликається функція *executeRunCommand*, яка реалізує логіку цієї команди. Якщо користувач передав іншу команду, то виводиться повідомлення про помилку, яке каже, що команда не може бути розпізнана.

```
executeCommand :: String -> [String] -> IO ()
executeCommand "run" args = executeRunCommand args
executeCommand cmd _ = putStrLn ("ERROR: Unknown command '" ++ cmd ++ "'")
```

*Рисунок 3.9.2. Імплементація функції executeCommand*

Функція *executeRunCommand* (рисунок 3.9.3) відповідає за виконання команди “run”. Вона має єдиний параметр – список аргументів для команди “run”. Першим кроком виконання функції є перевірка аргументів команди. Спочатку перевіряється, що список аргументів непорожній, оскільки команда “run” вимагає

рівно 1 аргумент – шлях до файлу із програмою, яку необхідно інтерпретувати. Якщо перевірка першого аргументу є успішною, то виконується перевірка на наявність опційного прапорця “--debug”, який повинен знаходитись після шляху до файлу. Він відповідає за виконання інтерпретації програми у розширеному режимі. Якщо цього прапорця немає – програма інтерпретується у звичайному режимі. Якщо команда має більше ніж 2 аргументи, то наступні аргументи ігноруються. Другим кроком є інтерпретація програми із вихідного файлу, яка виконується функцією *runInterpreter*.

```
executeRunCommand :: [String] -> IO ()
executeRunCommand args =
  if (length args < 1)
    then putStrLn "ERROR: expecting file path to run the program"
    else
      let
        filePath : flags = args
        isDebug = (length flags > 0) && (head flags == "--debug")
       in
        runInterpreter filePath isDebug
```

*Рисунок 3.9.3. Імплементація функції executeRunCommand*

Функція *runInterpreter* (рисунок 3.9.4) виконує інтерпретацію програми, визначеної в заданому вихідному файлі. Вона має 2 параметри: шлях до файлу з програмою, яку потрібно інтерпретувати, та булеве значення, яке вказує, в якому режимі потрібно виконати інтерпретацію: звичайному (значення False) або розширеному (значення True). У звичайному режимі при інтерпретації програми не виводиться ніяка додаткова інформація. В розширеному режимі функція виводить інформацію про фінальний стан кожного етапу: побудоване АСД після синтаксичного аналізу, стан семантичного аналізатора після семантичного аналізу та стан інтерпретатора після виконання програми. Розширений режим виконання використовується в основному під час розробки інтерпретатора для можливості відстеження та перевірки правильності роботи кожного з етапів інтерпретатора. Виконання функції можна поділити на декілька кроків. На першому кроці функція

отримує вихідний код програми із вказаного файлу. Якщо файлу із вказаним шляхом не існує, виконання функції закінчується відповідною помилкою. Наступними кроками є послідовний виклик функцій *applyParser*, *applyAnalyzer* та *applyInterpreter* із модулів *Parser*, *Analyzer* та *Interpreter* відповідно. Якщо на будь-якому із етапів відбувається помилка, інформацію про помилку виводиться у стандартний потік введення-виведення і процес інтерпретації припиняється на цьому етапі.

```haskell
runInterpreter :: String -> Bool -> IO ()
runInterpreter filePath isDebug = do
  handle <- openFile filePath ReadMode
  contents <- hGetContents handle
  putStrLn ("Interpreting source file: '" ++ filePath ++ "'...")
  putStrLn ""
  ifDebug (putStrLn "Parsing program...")
  case applyParser filePath contents of
    Left er -> putStrLn (show er)
    Right pr -> do
      ifDebug (putStrLn (show pr))
      ifDebug printSeparator
      ifDebug (putStrLn "Analyzing parsed program...")
      case applyAnalyzer pr of
        Left er -> putStrLn (show er)
        Right a -> do
          ifDebug (putStrLn (show a))
          ifDebug printSeparator
          ifDebug (putStrLn "Interpreting program...")
          ifDebug (putStrLn "Program output:")
          intrRes <- applyInterpreter pr
          case intrRes of
            Left er -> putStrLn (show er)
            Right i -> do
              ifDebug (putStrLn "")
              ifDebug printSeparator
              ifDebug (putStrLn "Final interpreter state:")
              ifDebug (putStrLn (show i))
  hClose handle
 where
  ifDebug fn = if isDebug then fn else pure ()
  printSeparator = putStrLn (take 40 $ repeat '-')
```

*Рисунок 3.9.4. Імплементація функції runInterpreter*

Розроблений модуль *Main* містить усі необхідні функції для використання інтерпретатора. При збільшенні кількості доступних команд інтерпретатора їх виконання може бути імплементоване в окремому модулі. В такому випадку, у модулі *Main* буде наявна лише функція *main*, основною задачею якої буде отримання аргументів від користувача та виклик функцій з інших модулів для виконання користувацьких команд.

# Розділ 4. Використання інтерпретатора

## 4.1 Робота з інтерпретатором під час розробки

Компіляція вихідного коду інтерпретатора та збірка проєкту щоб протестувати внесені зміни може зайняти значну кількість часу, що не є ефективним в процесі розробки. Для Haskell існує інтерпретатор GHCi, який дозволяє на ходу інтерпретувати код Haskell. Однією із переваг GHCi є можливість завантажувати модулі для інтерпретації та викликати будь-яку функцію, визначену в модулі, щоб перевірити результат виконання. Ця особливість використовувалась для швидкого тестування оновлень після внесення змін до вихідного коду інтерпретатора.

Інтерпретатор запускається командою *ghci* з кореневої директорії проєкту. Оскільки вихідний код програми розбитий на різні модулі та директорії, для виконання його в інтерпретаторі потрібно застосувати декілька додаткових команд. Модулі *Lexic*, *Parser*, *Analyzer* та *Interpreter* знаходяться у директорії *src*, в той час як модуль *Main* знаходиться у директорії *app*. Тому при спробі завантажити модуль *Main* для інтерпретації виникне помилка, оскільки використані у ньому модулі не можуть бути знайдені (рисунок 4.1.1). Схожа помилка може виникнути, якщо потрібно завантажити модуль, що знаходиться в іншому вихідному файлі, навіть якщо обидва модулі знаходяться в одній директорії. Щоб вказати інтерпретатору додаткову директорію для пошуку модулів необхідно виконати команду *:set -isrc*. Після цього можна використати команду *:l .\app\Main.hs* для завантаження модуля *Main* в інтерпретатор (рисунок 4.2.2). Після виконання цих команд, інтерпретатор налаштований на виконання функцій із модуля *Main*.

```
PS D:\Projects\haskell\pascal-interpreter> ghci
GHCi, version 9.6.6: https://www.haskell.org/ghc/  :? for help
ghci> :l .\app\Main.hs
[1 of 2] Compiling Main             ( app\Main.hs, interpreted )

app\Main.hs:3:1: error:
    Could not find module 'Analyzer'
    Use -v (or `:set -v` in ghci) to see a list of the files searched for.
  |
3 | import Analyzer
  | ^^^^^^^^^^^^^^^

app\Main.hs:4:1: error:
    Could not find module 'Interpreter'
    Use -v (or `:set -v` in ghci) to see a list of the files searched for.
  |
4 | import Interpreter
  | ^^^^^^^^^^^^^^^^^^

app\Main.hs:5:1: error:
    Could not find module 'Parser'
    Use -v (or `:set -v` in ghci) to see a list of the files searched for.
  |
5 | import Parser
  | ^^^^^^^^^^^^^
Failed, no modules loaded.
ghci>
```

*Рисунок 4.1.1. Невдала спроба завантаження модуля Main*

```
PS D:\Projects\haskell\pascal-interpreter> ghci
GHCi, version 9.6.6: https://www.haskell.org/ghc/  :? for help
ghci> :set -isrc
ghci> :l .\app\Main.hs
[1 of 6] Compiling Lexic            ( src\Lexic.hs, interpreted )
[2 of 6] Compiling Interpreter      ( src\Interpreter.hs, interpreted )
[3 of 6] Compiling Analyzer         ( src\Analyzer.hs, interpreted )
[4 of 6] Compiling Parser           ( src\Parser.hs, interpreted )
[5 of 6] Compiling Main             ( app\Main.hs, interpreted )
Ok, five modules loaded.
ghci>
```

*Рисунок 4.1.2. Використання додаткових команд для завантаження модуля Main*

Однією з причин виділення функції *runInterpreter* в окрему функцію замість використання її коду напряму в функції *executeRunCommand* є те, що таким чином

значно легше виконувати її в інтерпретаторі, без необхідності використання додаткової команди для встановлення аргументів користувача Це полегшує тестування програми при внесенні змін. Приклад виконання функції напряму в інтерпретаторі наведено на рисунку 4.1.3.

```
PS D:\Projects\haskell\pascal-interpreter> ghci
GHCi, version 9.6.6: https://www.haskell.org/ghc/  :? for help
ghci> :set -isrc
ghci> :l .\app\Main.hs
[1 of 6] Compiling Lexic            ( src\Lexic.hs, interpreted )
[2 of 6] Compiling Interpreter      ( src\Interpreter.hs, interpreted )
[3 of 6] Compiling Analyzer         ( src\Analyzer.hs, interpreted )
[4 of 6] Compiling Parser           ( src\Parser.hs, interpreted )
[5 of 6] Compiling Main             ( app\Main.hs, interpreted )
Ok, five modules loaded.
ghci> runInterpreter "./sources/power.pas" False
Interpreting source file: './sources/power.pas'...

Please, enter the number and pow to calculate: 2 5
2 to the power of 5 = 32
ghci>
```

*Рисунок 4.1.3. Виклик функції runInterpreter в інтерпретаторі GHCi*

### 4.2 Збірка проєкту

Для збірки інтерпретатора у виконуваний файл, який може застосовуватись без інтерпретатора чи компілятора Haskell, використовується Stack. Це інструмент, який полегшує збірку багатомодульних проєктів із додатковими залежностями.

Усі налаштування Stack знаходяться у вихідному файлі *package.yaml* (рисунок 4.2.1) з використанням мови YAML. У ньому вказуються усі директорії із модулями, що використовуються у проєкті, директорія і файл головного модуля, список залежностей, параметри компілятора та метаінформація про проєкт. На основі цього файлу Stack може створювати додаткові файли у директорії проєкту, що необхідні для його роботи.

package.yaml

```yaml
name:                pascal-interpreter
version:             1.0.0
github:              "KrAxmalL/pascal-interpreter"

extra-source-files:
- README.md

# Metadata used when publishing your package
# synopsis:            Short description of your package
# category:            Web

# To avoid duplicated efforts in documentation and dealing with the
# complications of embedding Haddock markup inside cabal files, it is
# common to point users to the README.md file.
description:         Please see the README on GitHub at <https://github.com/KrAxmalL/pascal-interpreter#readme>

dependencies:
- base >= 4.7 && < 5
- parsec
- containers

ghc-options:
- -Wall
- -Wcompat
- -Widentities
- -Wincomplete-record-updates
- -Wincomplete-uni-patterns
- -Wmissing-export-lists
- -Wmissing-home-modules
- -Wpartial-fields
- -Wredundant-constraints
- -XNamedFieldPuns

library:
  source-dirs: src

executables:
  pascal-interpreter-exe:
    main:                Main.hs
    source-dirs:         app
    ghc-options:
    - -threaded
    - -rtsopts
    - -with-rtsopts=-N
    dependencies:
    - pascal-interpreter

tests:
  pascal-interpreter-test:
    main:                Spec.hs
    source-dirs:         test
    ghc-options:
    - -threaded
    - -rtsopts
    - -with-rtsopts=-N
    dependencies:
    - pascal-interpreter
```

*Рисунок 4.2.1. Налаштування Stack у файлі package.yaml*

Після налаштування Stack для створення виконуваного файлу використовується команда *stack build*. Залежно від налаштувань компілятора, у процесі виконання команди може з'являтись велика кількість попереджень щодо вихідного коду, які варто виправити. Якщо на певному етапі збірки відбудеться помилка, збірка завершиться помилкою і виведеться повідомлення про її причину. Якщо ж збірка виконається успішно, виведеться повідомлення про шлях до виконуваного файлу, який було створено і який може бути використано для виконання програми (рисунок 4.2.2). У випадку поточного проєкту, фінальний виконуваний файл має назву *pascal-interpreter-exe.exe*, він створюється всередині директорії *.stack-work*, яку створює Stack і в якій зберігаються усі артефакти його роботи.

```
PS D:\Projects\haskell\pascal-interpreter> stack build

Warning: Stack is not using a Stack-supplied MSYS2.
pascal-interpreter-1.0.0: unregistering (local file changes: app\Main.hs src\Analyzer.hs src\Interpreter.hs
src\Lexic.hs)
pascal-interpreter> build (lib + exe) with ghc-9.6.6
Preprocessing library for pascal-interpreter-1.0.0..
Building library for pascal-interpreter-1.0.0..
[3 of 3] Linking .stack-work\\dist\\effaccc7\\build\\pascal-interpreter-exe\\pascal-interpreter-exe.exe [Objects changed
]
pascal-interpreter> copy/register
Installing library in D:\Projects\haskell\pascal-interpreter\.stack-work\install\fb26c89a\lib\x86_64-windows-ghc-9.6.6\p
ascal-interpreter-1.0.0-APr2YzVQqAW1lHA8p9YM8a
Installing executable pascal-interpreter-exe in D:\Projects\haskell\pascal-interpreter\.stack-work\install\fb26c89a\bin
Registering library for pascal-interpreter-1.0.0..
PS D:\Projects\haskell\pascal-interpreter>
```

*Рисунок 4.2.2. Приклад успішного виконання команди stack build*

Усі кроки, описані у цьому розділі, можуть бути використані для збірки та запуску інтерпретатора на власній системі. Для цього достатньо завантажити вихідний код проєкту з платформи GitHub [15], встановити Stack та необхідні інструменти для Haskell (компілятор GHC, який також може бути встановлений за допомогою Stack) і виконати команди, описані у розділі. Проєкт може бути зібраний іншими інструментами, доступними для Haskell, з використанням лише файлів із вихідним кодом, проте оскільки у проєкті було вирішено використовувати Stack, то інші методи збірки не розглядалися і не описуються у роботі.

### 4.3 Робота з виконуваним файлом інтерпретатора

Після створення виконуваного файлу його можна використовувати як окремий застосунок для інтерпретації програм на Pascal. Щоб уникнути необхідності кінцевим користувачам виконувати збірку проєкту на локальних системах, створений виконуваний файл було завантажено на платформу GitHub [16] у той же репозиторій, що і вихідний код проєкту. Вихідний файл було створено та протестовано на операційній системі Windows 11, тому він не буде працювати на операційних система macOS та на системах сімейства Linux. На інших версіях Windows можливість запуску виконуваного файлу не була перевірена.

Для інтерпретації програм виконуваним файлом необхідно у командній стрічці виконати команду *<Назва_виконуваного_файлу> run <Шлях_до_файлу> --debug*. Назва виконуваного файлу за замовчуванням – *pascal-interpreter-exe.exe,* шлях до файлу може бути відносним або абсолютним. Прапорець *--debug* є опційним і використовується для виконання інтерпретації в розширеному режимі. При неправильному використанні команди можливі декілька помилок:

- Якщо виконуваний файл було виконано без жодних параметрів (.*pascal-interpreter-exe.exe*) – виникне помилка, що жодної команди не було вказано (рисунок 4.3.1).
- Якщо командою для виконуваного файлу є будь-яка команда, окрім “run” (.*pascal-interpreter-exe.exe random-command*) – виникне помилка, що команда не була розпізнана (рисунок 4.3.2).
- Якщо команда “run” була вказана без жодний аргументів (.*pascal-interpreter-exe.exe run*) – виникне помилка, що шлях до файлу не було вказано (рисунок 4.3.3).

- Якщо файлу зі вказаним шляхом не існує (.*pascal-interpreter-exe.exe run .\random-file-path.pas*) – виникне помилка, що файл не було знайдено (рисунок 4.3.4).

```
PS D:\Projects\haskell\pascal-interpreter\.stack-work\install\fb26c89a\bin> .\pascal-interpreter-exe.exe
ERROR: Command was not provided
PS D:\Projects\haskell\pascal-interpreter\.stack-work\install\fb26c89a\bin>
```

*Рисунок 4.3.1. Запуск виконуваного файлу без аргументів*

```
PS D:\Projects\haskell\pascal-interpreter\.stack-work\install\fb26c89a\bin> .\pascal-interpreter-exe.exe random-command
ERROR: Unknown command 'random-command'
PS D:\Projects\haskell\pascal-interpreter\.stack-work\install\fb26c89a\bin>
```

*Рисунок 4.3.2. Запуск виконуваного файлу з неіснуючою командою*

```
PS D:\Projects\haskell\pascal-interpreter\.stack-work\install\fb26c89a\bin> .\pascal-interpreter-exe.exe run
ERROR: expecting file path to run the program
PS D:\Projects\haskell\pascal-interpreter\.stack-work\install\fb26c89a\bin>
```

*Рисунок 4.3.3. Запуск виконуваного файлу з командою run без шляху до файлу*

```
PS D:\Projects\haskell\pascal-interpreter\.stack-work\install\fb26c89a\bin> .\pascal-interpreter-exe.exe run ./random-fi
le-path.pas
pascal-interpreter-exe.exe: ./random-file-path.pas: openFile: does not exist (No such file or directory)
PS D:\Projects\haskell\pascal-interpreter\.stack-work\install\fb26c89a\bin>
```

*Рисунок 4.3.4. Запуск виконуваного файлу з командою run із неіснуючим шляхом*

При запуску програми із коректними аргументами виконується інтерпретація програми, що знаходиться в заданому вихідному файлі. Щоб запустити інтерпретацію у звичайному режимі, потрібно виконати команду без опційного прапорця *--debug*. Прикладом такого запуску є виконання програми для пошуку заданої користувачем кількості чисел Фібоначчі, вихідний код якої знаходиться у директорії *sources* у файлі *fibonacci.pas* у вихідному проєкті (цю програму буде більш детально описано у наступному розділі). На локальній системі ця програма була виконана наступною командою: *.\pascal-interpreter-exe.exe run D:\Projects\haskell\pascal-interpreter\sources\fibonacci.pas*. Результатом виконання команди є виконання інтерпретації програми із вихідного файлу. Результат виконання команди на локальний системі наведено на рисунку 4.3.5.

```
PS D:\Projects\haskell\pascal-interpreter\.stack-work\install\fb26c89a\bin> .\pascal-interpreter-exe.exe run D:\Projects
\haskell\pascal-interpreter\sources\fibonacci.pas
Interpreting source file: 'D:\Projects\haskell\pascal-interpreter\sources\fibonacci.pas'...

Please, enter the required number of Fibonacci numbers to calculate: 10
Fibonacci number at position 0 - 0
Fibonacci number at position 1 - 1
Fibonacci number at position 2 - 1
Fibonacci number at position 3 - 2
Fibonacci number at position 4 - 3
Fibonacci number at position 5 - 5
Fibonacci number at position 6 - 8
Fibonacci number at position 7 - 13
Fibonacci number at position 8 - 21
Fibonacci number at position 9 - 34
Fibonacci number at position 10 - 55
PS D:\Projects\haskell\pascal-interpreter\.stack-work\install\fb26c89a\bin>
```

*Рисунок 4.3.5. Запуск виконуваного файлу для інтерпретації програми у звичайному режимі*

Цю ж програму можна інтерпретувати у розширеному режимі додавши до попередньої команди прапорець *--debug* після шляху до файлу. Розширений режим крім безпосередньої інтерпретації програми також виводить у стандартний потік введення-виведення інформацію про побудоване АСД, а також кінцеві стани семантичного аналізатора та інтерпретатора. Команда для виконання інтерпретації у розширеному режимі має такий вигляд: *.\pascal-interpreter-exe.exe run*


*Рисунок 4.3.6. Запуск виконуваного файлу для інтерпретації програми у розширеному режимі*

*D:\Projects\haskell\pascal-interpreter\sources\fibonacci.pas --debug*, результат її виконання наведено на рисунку 4.3.6.

### 4.4 Програма для обрахування чисел Фібоначчі

Розглянемо програму, яка обраховує задану користувачем кількість чисел із послідовності Фібоначчі (рисунок 4.4.1).

Обрахунок числа Фібоначчі під заданим номером виконує функція *calculateFibonacci*. Вона має єдиний параметр *position* цілочисельного типу, який вказує на порядковий номер числа, яке необхідно обрахувати, і повертає значення цілочисельного типу. Реалізація функції є рекурсивною, оскільки це найбільш простий спосіб імлементації. Якщо позиція числа є від'ємним числом, результатом функції є -1 (спосіб продемонструвати, що порядковий номер повинен бути невід'ємним числом). Для позицій 0, 1 і 2 повертаються відповідні значення чисел, які є добре відомими. Якщо ж позиція числа - це 3 або більше, то функція викликається рекурсивно для пошуку значень чисел на двох попередніх позиціях. Сума отриманих значень і є шуканим числом.

У програмі визначено 2 змінні: *index* та *amount*. Змінна *amount* зберігає введену користувачем кількість чисел Фібоначчі, яку необхідно порахувати. Змінна *index* використовується щоб відстежувати позицію поточного обрахованого числа.

Програма починається виведенням у стандартний потік введення-виведення повідомлення, яке описує що необхідно ввести користувачеві. Після цього, програма зберігає введене користувачем число у змінній *amount*. Останнім кроком є виконання циклу "while", який для усіх позицій, починаючи з 0 і закінчуючи заданою користувачем, виводить повідомлення про позицію числа та саме число.

```
program Fibonacci;

function calculateFibonacci(position: Integer): Integer;
begin
if position < 0 then
  calculateFibonacci := -1
else if position = 0 then
  calculateFibonacci := 0
else if position < 3 then
  calculateFibonacci := 1
else
  calculateFibonacci := calculateFibonacci(position - 1) +
                        calculateFibonacci(position - 2);
end;

var index: Integer;
var amount: Integer;

begin
  write('Please, enter the required number of Fibonacci numbers to calculate: ');
  read(amount);

  index := 0;
  while (index <= amount) do
    begin
      writeln('Fibonacci number at position ', index, ' - ', calculateFibonacci(index));
      index := index + 1;
    end;
end.
```

*Рисунок 4.4.1. Вихідний код програми, яка обраховує задану кількість чисел Фібоначчі*

```
PS D:\Projects\haskell\pascal-interpreter\.stack-work\install\fb26c89a\bin> .\pascal-interpreter-exe.exe run D:\Projects
\haskell\pascal-interpreter\sources\fibonacci.pas
Interpreting source file: 'D:\Projects\haskell\pascal-interpreter\sources\fibonacci.pas'...

Please, enter the required number of Fibonacci numbers to calculate: 13
Fibonacci number at position 0 - 0
Fibonacci number at position 1 - 1
Fibonacci number at position 2 - 1
Fibonacci number at position 3 - 2
Fibonacci number at position 4 - 3
Fibonacci number at position 5 - 5
Fibonacci number at position 6 - 8
Fibonacci number at position 7 - 13
Fibonacci number at position 8 - 21
Fibonacci number at position 9 - 34
Fibonacci number at position 10 - 55
Fibonacci number at position 11 - 89
Fibonacci number at position 12 - 144
Fibonacci number at position 13 - 233
PS D:\Projects\haskell\pascal-interpreter\.stack-work\install\fb26c89a\bin>
```

*Рисунок 4.4.2. Результат успішної роботи програми, яка обраховує задану кількість чисел Фібоначчі*

Якщо програма виконалась успішно, результатом роботи програми буде

наявність у потоці введення-виведення повідомлень про позицію та значення чисел Фібоначчі (рисунок 4.4.2).

Якщо користувач ввів щось окрім числа, виконання програми завершиться помилкою інтерпретації (рисунок 4.4.3).

```
PS D:\Projects\haskell\pascal-interpreter\.stack-work\install\fb26c89a\bin> .\pascal-interpreter-exe.exe run D:\Projects
\haskell\pascal-interpreter\sources\fibonacci.pas
Interpreting source file: 'D:\Projects\haskell\pascal-interpreter\sources\fibonacci.pas'...

Please, enter the required number of Fibonacci numbers to calculate: 13abc
InterpretationError-[type = WrongReadProcedureArgumentError, message = Wrong input! Can't read value of type 'Integer' f
rom the input: "13abc"]
PS D:\Projects\haskell\pascal-interpreter\.stack-work\install\fb26c89a\bin>
```

*Рисунок 4.4.3. Результат помилкового завершення роботи програми, яка обраховує задану кількість чисел Фібоначчі*

### 4.5 Програма із семантичними помилками

Попередню програму можна модифікувати таким чином, щоб на етапі семантичного аналізу було виявлено помилки (рисунок 4.5.1). Якщо змінити тип змінної *index* з цілочисельного на булевий, програма одразу стає семантично неправильною. По-перше, помилка виникає при спробі присвоєння змінній булевого типу значення цілочисельного типу. По-друге, в умові циклу “while” помилкою є спроба застосувати оператор “<=” (“менше або дорівнює”) для порівняння значень різних типів. По-третє, помилка виникає при виклику функції *calculateFibonacci*, оскільки функція очікує параметр цілочисельного типу, а не булевого.

```
program Fibonacci;

function calculateFibonacci(position: Integer): Integer;
begin
if position < 0 then
  calculateFibonacci := -1
else if position = 0 then
  calculateFibonacci := 0
else if position < 3 then
  calculateFibonacci := 1
else
  calculateFibonacci := calculateFibonacci(position - 1) +
                        calculateFibonacci(position - 2);
end;

var index: Boolean;
var amount: Integer;

begin
  write('Please, enter the required number of Fibonacci numbers to calculate: ');
  read(amount);

  index := 0;
  while (index <= amount) do
    begin
      writeln('Fibonacci number at position ', index, ' - ', calculateFibonacci(index));
      index := index + 1;
    end;
end.
```

*Рисунок 4.5.1. Вихідний код програми, що містить семантичну помилку*

Про запуску інтерпретатора для виконання цієї програми у стандартний потік введення-виведення буде виведено інформацію, що було знайдено помилку на етапі семантичного аналізу (рисунок 4.5.2). Оскільки інтерпретатор зупиняє виконання коли знаходить першу помилку, то буде виведено інформацію тільки про одну знайдену помилку.

```
PS D:\Projects\haskell\pascal-interpreter\.stack-work\install\fb26c89a\bin> .\pascal-interpreter-exe.exe run D:\Projects
\haskell\pascal-interpreter\sources\fibonacci.pas
Interpreting source file: 'D:\Projects\haskell\pascal-interpreter\sources\fibonacci.pas'...

AnalysisError-[type = AssignmentError, message = Wrong expression type in assignment statement!, source = AnalysisError-
[type = TypeMismatchError, message = Expression: Val (IntNum 0) has wrong type! Expected one of these types: ["DTBoolean
"]. Actual type: DTInteger]]
PS D:\Projects\haskell\pascal-interpreter\.stack-work\install\fb26c89a\bin>
```

*Рисунок 4.5.2. Приклад семантичної помилки, яка виникає при спробі інтерпретувати програму з неправильно використаними типами даних*

### 4.6 Додаткові приклад програм, що можуть бути інтерпретовані

Приклади програм, що можуть бути виконані інтерпретатором, знаходяться у директорії *sources* серед вихідний файлів проєкту. Крім програми для обрахування чисел Фібоначчі (вихідний файл – fibonacci.pas), описаної у розділі 4.4, у проєкті є ще 2 приклади програм.

Перша програма знаходиться у вихідному файлі *power.pas* і обраховує значення степеня заданого числа. Вона використовується для перевірки коректності виконання інтерпретатором умовного оператора, циклу “while”, процедур введення-виведення та математичних операцій. Вихідний код програми наведено на рисунку 4.6.1.

```
1   program Power;
2
3   function pow(number: Integer; power: Integer): Integer;
4   var index: Integer;
5   begin
6     if (power = 0) then
7       pow := 1
8     else
9       begin
10        index := 1;
11        pow := number;
12        while (index < power) do
13        begin
14          pow := pow * number;
15          index := index + 1;
16        end;
17      end;
18  end;
19
20  var number: Integer;
21  var expectedPow: Integer;
22
23  begin
24    write('Please, enter the number and pow to calculate: ');
25    read(number, expectedPow);
26
27    writeln(number, ' to the power of ', expectedPow, ' = ', pow(number, expectedPow));
28  end.
```

*Рисунок 4.6.1. Вихідний код програми, яка обраховує заданий користувачем степінь числа*

Друга програма знаходиться у вихідному файлі *program.pas* і не виконує жодних корисних дій. Натомість, вона використовується для перевірки правильності виконання інтерпретатором усіх елементів мови Pascal, які ним підтримуються: визначень змінних (усіх доступних типів), функцій та процедур (включно із вкладеними визначеннями), усіх доступних інструкцій, більшості математичних операцій та операцій порівняння, процедур введення-виведення. Також у цій програмі перевіряється, чи коректно обираються елементи відповідно до області визначення, в якій вони використовуються, та чи відбувається “затінення” змінних. Елементи до цієї програми додавались у ході розробки, таким чином відбувалась перевірка та відлагодження правильності інтерпретації доданих елементів. Враховуючи відносно невелику кількість елементів, що підтримуються інтерпретатором, розмір цієї програми не є великим і вона може бути розширена, проте при додаванні більшої кількості елементів кращим рішенням може бути відмова від однієї великої програми і використання більшої кількості малих програм, кожна з яких використовується для перевірки правильності інтерпретації лише одного елемента. Вихідний код цієї програми наведено на рисунку 4.6.2.

```
program HelloWorld;

var a: Integer;
var b: Boolean;
var c: Integer;
var e: Real;
var x: Integer;
var cond: Boolean;
var t: Real;
var k: Integer;
var m: Char;
var s: String;

function add(c: Integer; d: Integer): Integer;
begin
    add := c + d;
end;

procedure print(k: Integer);
    function divide(divisor: Integer): Real;
    begin
        divide := divisor / k;
    end;
begin
    e := divide(14);
end;

begin
    a := -1;
    a := add(1, 4);
    print(5);
    b := a > e;
    if e > 5 then
        begin
            b := (add(a, c) > 5) = (4 / a = c);
            print(add(a, c));
        end
    else
        print(1);
    x := -1;
    cond := true;
    while (x < 10) = (e > 2) do
        begin
            x := x + 3;
        end;

    k := 5;
    repeat
        k := k + 10;
    until k < 3;

    read(m,s);
    writeln('s=', s);
    writeln('m=', m);
end.
```

*Рисунок 4.6.2. Вихідний код програми, яка використовується для перевірки роботи інтерпретатора*

# Розділ 5. Результати роботи

## 5.1 Можливості розширення розробленого інтерпретатора

Розроблений інтерпретатор підтримує наступні елементи мови Pascal:

- Типи даних: цілочисельний тип, тип дійсних чисел, булевий тип, символьний тип та стрічковий тип.
- Унарні операції: усі унарні операції, визначені у Pascal для реалізованих типів даних.
- Бінарні операції: усі бінарні операції, визначені у Pascal для реалізованих типів даних.
- Інструкції: оператор присвоєння, складений оператор, умовний оператор “if”, цикли “while” та “repeat”.
- Блоки: основний програмний блок, функції та процедури.
- Параметри функцій та процедур: можуть передаватись лише за значенням.
- Рекурсивні виклики: функції та процедури можуть бути викликані рекурсивно.
- Вкладені визначення: функції та процедури можуть містити вкладені визначення змінних, функцій та процедур.
- “Затінення” визначень: змінні, функції та процедури у вкладених блоках використовуються замість відповідних елементів із батьківських блоків з такими ж назвами.

Реалізовані елементи дозволяють розробляти велику кількість програм, проте для розробки більш складних програм необхідно розширити розроблений інтерпретатор і додати наступні елементи:

- Складені типи даних: масиви, множини, записи.

- Операції над складеними типами даних: перевірка на належність елемента до множини, читання та оновлення елементів масиву, операції для роботи із полями записів.
- Параметри функцій та процедур: можливість передачі параметрів за посиланням.
- Елементи стандартної бібліотеки: математичні функції, процедури для роботи з потоками введення-виведення.

### 5.2 Висновки

В ході виконання роботи було розроблено інтерпретатор підмножини мови програмування Pascal. Інтерпретатор складається з трьох частин, які відповідають трьом етапам виконання інтерпретації. Кожна з частин є незалежною одна від одної, оскільки усі вони виконують операції з абстрактним синтаксичним деревом, але частини не модифікують його, а використовують додаткові структури даних для виконання своїх задач.

При розробці синтаксичного аналізатора було використано бібліотеку Parsec, яка дозволила розробити синтаксичний аналізатор, що може бути легко розширений при розширені підмножини мови Pascal, яка підтримується інтерпретатором. Підходи бібліотеки Parsec дозволяють описувати парсери таким чином, що їх комбінація та використання виглядає дуже схоже на синтаксис Pascal, описаний формою Бекуса-Наура. Така структура полегшує розуміння вихідного коду розробленого синтаксичного аналізатора.

Семантичний аналізатор було розроблено таким чином, що при його виконанні використовується додаткова структура даних, що існує лише на етапі семантичного аналізу. Таким чином, при необхідності етап семантичного аналізу

може бути вилучений при виконанні інтерпретації, а помилки, які можуть бути виявлені на цьому етапі, стануть помилками часу виконання.

Завдяки особливості структури програм, розроблених мовою Pascal (а саме - поділ кожного блоку на секції визначень та інструкцій), немає необхідності при виконанні секції інструкцій блоку додавати нові визначення змінних, функцій та процедур до інтерпретатора і спиратись на них при безпосередньому виконанні інструкцій. Натомість, перед виконанням інтерпретації тіла блоків інтерпретатор будує окрему структуру даних, що зберігає інформацію про усі блоки та їх вкладеність один в одного. Така структура дозволяє при використанні стеку викликів зберігати у кожному записі активації лише інформацію про змінні та їх значення, що значно спрощує процес інтерпретації програми.

Розроблений інтерпретатор може бути розширений як додаванням пропущених інструкцій із мови Pascal (таких як умовний оператор “case” та цикл “for”), так і додаванням структурованих типів даних та указників. Додавання перелічених інструкцій буде досить простим і не потребуватиме великої кількості змін у структурі інтерпретатора. Натомість, додавання структурованих типів даних вимагатиме внесення значних змін до інтерпретатора для підтримки не лише вбудованих типів, а й типів, визначених користувачем.

Основною перевагою розробленого інтерпретатора є те, що оскільки не потрібно виконувати процес компіляції, інтерпретатор набагато краще підходить для виконання невеликих програм або частин програм. Правильність їх роботи можна перевірити запускаючи їх з використанням інтерпретатора, а після цього частини програм можна використовувати для побудови більших програм або повноцінних застосунків, які потребують компіляції. Таким чином, розроблений інтерпретатор досить добре підходить для використання у навчальних цілях (наприклад, для вивчення основ програмування з використанням мови Pascal),

оскільки його використання є значно легшим, ніж встановлення повноцінного компілятора.

Вихідний код інтерпретатора знаходиться у відкритому доступі на платформі GitHub [15] і може бути використаний у поточному стані, або може бути розширений завдяки додаванню підтримки більшої кількості елементів мови Pascal.

Розроблений інтерпретатор може бути використаний різними способами. Для використання на операційній системі Windows 11 достатньо завантажити створений в ході виконання роботи виконуваний файл із платформи GitHub [16]. Для збірки виконуваного файлу для операційних систем сімейства Linux та macOS можна застосувати інструмент Stack, налаштування якого наявні у вихідному коді проєкту, або використати будь-які інші інструменти для збірки проєктів на Haskell використовуючи лише файли із вихідним кодом інтерпретатора.

**Список використаних джерел**


1. Sebesta R. W. Concepts of Programming Languages / Robert W. Sebesta., 2021. – 816 с.
2. Wirth N. The Programming Language Pascal. Acta Informatica, 1(1) / Niklaus Wirth., 1971. – 35 с.
3. Lipovača M. Learn You a Haskell for Great Good! / Miran Lipovača., 2011. – 400 с.
4. O'Sullivan B., Stewart D., Goerzen J. Real World Haskell / Bryan O'Sullivan, Don Stewart, John Goerzen., 2008. – 710 с.
5. Serrano A. Practical Haskell / Alejandro Serrano., 2019. – 450 с.
6. ISO 7185:1990 - Information technology - Programming languages - Pascal / 1990. – 92 с.
7. Aho. A, Lam M., Sethi R., Ullman J. Compilers Principles, Techniques, and Tools (2 ed.) / Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman., 2006. – 942 с.
8. Nystrom R. Crafting Interpreters / Robert Nystrom., 2021. - 640 с.
9. Документація The Haskell Tool Stack [Електронний ресурс] – Режим доступу до ресурсу: https://docs.haskellstack.org/en/stable/.
10. Документація пакетів Haskell [Електронний ресурс] – Режим доступу до ресурсу: https://hackage.haskell.org/.
11. Документація бібліотеки Parsec [Електронний ресурс] – Режим доступу до ресурсу: https://hackage.haskell.org/package/parsec.
12. Документація типу System.IO [Електронний ресурс] – Режим доступу до ресурсу: https://hackage.haskell.org/package/base-4.21.0.0/docs/System-IO.html.
13. Документація типу Data.Maybe [Електронний ресурс] – Режим доступу до ресурсу: https://hackage.haskell.org/package/base-4.21.0.0/docs/Data-Maybe.html.

14. Документація типу Data.Either [Електронний ресурс] – Режим доступу до ресурсу: https://hackage.haskell.org/package/base-4.21.0.0/docs/Data-Either.html.
15. Вихідний код розробленого інтерпретатора [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/KrAxmalL/pascal-interpreter.
16. Виконуваний файл розробленого інтерпретатора [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/KrAxmalL/pascal-interpreter/releases/tag/v1.0.0.