Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики факультету інформатики

СИНТАКСИЧНИЙ АНАЛІЗ В HASKELL

Текстова частина до курсової роботи

за спеціальністю «Інженерія програмного забезпечення» 121

Керівник курсової роботи

кандидат фізико-математичних наук,

доцент Проценко В. С.

_____

(підпис)

“___” _____ 2020 р.

Виконала студентка Пивовар О. О.

“___” _____ 2020 р.

Київ 2020

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ

Зав. кафедри інформатики,

к. ф.-м. н. С. С. Гороховський

_____

(*підпис*)

“___” _____ 2020 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на курсову роботу

студентки Пивовар Олена Олександрівни факультету інформатики 3 курсу

Тема: Системний аналіз в Haskell

Зміст ТЧ до курсової роботи:

Анотація

Вступ

Розділ 1. Розуміння процесу синтаксичного аналізу

Розділ 2. Огляд існуючих засобів розробки

Розділ 3. Реалізація синтаксичного аналізу на функціональній мові програмування Haskell

Розділ 4. Порівняння використаних бібліотек

Висновки

Список літератури

Додатки

Дата видачі “___” _____ 2020 р. Керівник _____ (*підпис*)

Завдання отримав _____ (*підпис*)

Тема: Синтаксичний аналіз в Haskell

Календарний план виконання роботи:

№ п/п	Назва етапу курсової роботи	Термін виконання етапу	Примітка
1.	Отримання завдання на курсову роботу	04.11.2019	
2.	Пошук тематичної наукової літератури	08.11.2019	
3.	Ознайомлення із науковою літературою	12.11.2019	
4.	Ознайомлення із принципом синтаксичного аналізу	14.12.2019	
5.	Повторення синтаксису Haskell	27.02.2020	
6.	Створення програми для синтаксичного аналізу із використанням Parsec	18.03.2019	
7.	Аналіз отриманих результатів з керівником	30.03.2020	
8.	Створення програми для синтаксичного аналізу із використанням Attoparsec	01.04.2020	
9.	Створення програми для синтаксичного аналізу із використанням Applicative Parsing	07.04.2020	
10.	Аналіз отриманих результатів з керівником	15.04.2020	
11.	Написання текстової частини	01.05.2020	
12.	Перегляд змісту роботи з керівником	07.05.2020	
13.	Внесення змін до роботи	08.05.2020	
14.	Створення презентації	09.05.2020	

Зміст

Анотація

В процесі виконання роботи відбулося більш детальне ознайомлення із процесом синтаксичного аналізу, його функціями та видами. Також було досліджено різновиди граматик, особливу увагу звернено на КВ-граматику. Було проаналізовано та вивчено особливості використання різних бібліотек Haskell, які надають можливість здійснювати синтаксичний аналіз. Як результат було створено три програми, які використовували функціонал наступних бібліотек: Applicative Parsing, Attoparsec та Parsec.

Ключові слова: синтаксичний аналіз, аналізатор, рекурсивний спуск, лексичний аналіз, граматика, КВ-граматика, дерево розбору, нетермінал, термінал, Haskell, Applicative Parsing, Attoparsec, Parsec.

ВСТУП

Синтаксичний аналіз – це процес, на вході якого є певна послідовність символів, яку необхідно проаналізувати відповідно до заданої граматики, а на виході – певна структура даних, найчастіше дерево розбору. Основна мета аналізу – відповісти на запитання, чи належить до множини ланцюгів, породжених даною мовою, ланцюг, який прийшов на вхід.

Haskell – чудова мова для синтаксичного аналізу, оскільки, використовуючи її, можна легко поєднувати різні простіші блоки у один більш складний. Тобто ми можемо розбити синтаксичний аналіз на більш простіші завдання. При цьому можна не хвилюватися щодо непередбачуваних наслідків. [1] Ці властивості забезпечує функціональний характер даної мови програмування. Саме тому і створено так багато бібліотек, які дозволяють здійснювати аналіз різними способами. Деякі із них здатні працювати лише із регулярними виразами та будувати аналізатори лише для регулярних мов (Applicative Parsing, наприклад). Є й такі, які працюють і з нерегулярними мовами (Parsec, Megaparsec, Attoparsec).

Мета роботи – проаналізувати та визначити особливості використання різних бібліотек Haskell (а саме Applicative Parsing, Attoparsec та Parsec), які надають можливість здійснювати синтаксичний аналіз.

Завданням курсової є створення програми на Haskell для синтаксичного аналізу з використанням різних бібліотек.

Об’єкт дослідження – особливості використання різних бібліотек для синтаксичного аналізу в Haskell.

Робота складається із чотирьох розділів.

У першому розділі описані основні теоретичні відомості про синтаксичний аналіз, його функції, види. Ще згадано про граматики, більше

детально розглянуто контекстно-вільну граматику та її перетворення, неоднозначність. Також більше детально розказано про один із видів – метод рекурсивного спуску, який використовується у багатьох бібліотеках в Haskell, а саме: Parsec, Megaparsec, Attoparsec. Крім того ці бібліотеки надають засоби роботи з помилками, тому у цьому розділі також згадано, про важливість виявлення помилок.

Другий розділ містить короткий огляд найбільш відомих існуючих на даний момент бібліотек для написання синтаксичного аналізу на такій функціональній мові програмування як Haskell.

У третьому розділі описано вибір мови для реалізації синтаксичного аналізу. Також розповідається про особливості написання аналізу за допомогою Applicative Parsing, Attoparsec та Parsec.

У четвертому розділі описано основні відмінності бібліотек, які я помітила під час реалізації синтаксичного аналізу.

Постановка задачі

Розробити три програми, які надають можливість здійснювати синтаксичний аналіз певної мови програмування. Необхідно використати три різні бібліотеки (для кожної програми інша) для здійснення синтаксичного аналізу в Haskell.

Розділ 1. Розуміння процесу синтаксичного аналізу

1.1 Основні визначення

Синтаксичний аналіз – процес під час якого встановлюється, чи певна послідовність символів відповідає правилам заданої граматики.

Породжуюча граматика мови – це $G(V_N, V_T, S, P)$, де V_N – алфавіт нетерміналів, V_T – алфавіт терміналів (V_N та V_T не перетинаються), S – початковий нетермінал, P – скінченна множина правил.

Синтаксичний аналізатор – це функція, яка приймає на вхід текстові дані (в Haskell це може бути String або інша структура даних така як ByteString або Text)[2]. Парсери аналізують вхідні дані відповідно до правил, зазначених у граматиці. Як результат функція повертає певну структуру (деяке проміжне представлення). Це може бути, наприклад, дерево розбору.

Дерево розбору (дерево виводу, синтаксичне дерево) – помічене дерево, що задовольняє умовам:

- кожен вузол має мітку – символ з алфавіту нетерміналів та терміналів або порожній символ (ε);
- мітка кореня – початковий нетермінал;
- мітка вузла – нетермінал, якщо він має хоча б одного сина;
- ε – може бути тільки міткою листа;
- якщо вузол n з міткою A має синів $n_1, n_2 \dots n_k$, перерахованих зліва направо, з мітками $A_1, A_2 \dots A_k$, то $A \to A_1A_2\dots A_k$. [3]

Корона дерева – результат виводу.

1.2 Граматика мови. Контекстно-вільна граматика

Відповідно до ієрархії Хомського формальні граматики діляться на чотири типи:

- граматика типу 0 або граматика з фразовою структурою. Всі її правила мають вигляд α -> β, де $\alpha \in (V_N \cup V_T)+$, $\beta \in (V_N \cup V_T)*$. Породжує рекурсивно-перелічимі множини.
- граматика типу 1 або контекстно-залежна. Правила мають вигляд αAβ -> αγβ, де $A \in V_N$, $\alpha\beta \in (V_N \cup V_T)*$, $\gamma \in (V_N \cup V_T)+$. Породжує контекстно-залежні мови.
- граматика типу 2 або контекстно-вільна. Правила мають вигляд A -> α, де $A \in V_N$, $\alpha \in (V_N \cup V_T)*$. Породжує контекстно-вільні мови.
- граматика типу 3 або автоматна. Всі правила мають вигляд A -> a, B -> Cb, де A, B, C $\in V_N$, a, b $\in V_T$. Породжує регулярну мову. [4]

Контекстно-вільна (КВ) граматика часто використовується для визначення синтаксису мови програмування. Також варто додати, що її часто інтерпретують як граматику типів. У такому випадку нетермінали – складні типи, а термінали – атомарні, у яких не може бути підтипів. Така інтерпретація часто використовується при створенні трансляторів мов.

1.2.1 Ліворекурсивні правила. Перетворення КВ-граматик

Правило виводу виду A -> Aa називається ліворекурсивним. Якщо КВ-граматика має ліворекурсивні правила, то існує еквівалентна їй граматика без ліворекурсивних правил. Нагадаю, що дві граматики еквівалентні, якщо вони генерують одну і ту ж мову.

Якщо у нас в граматиці є наступні правила: A -> Aa, A -> b, то для того, щоб прибрати ліворекурсивні правила, необхідно ввести новий нетермінал A’ і замінити вище згадані правила на наступні: A -> bA’, A’ -> aA’, A’ -> ε. Використовуючи уже нові правила виводу, можна побудувати ті ж слова.

1.2.2 Неоднозначності граматики

Неоднозначна граматика – граматика, у якій для одного і того ж вхідного рядка існує більше одного правостороннього (лівостороннього) виводу.

Давайте розглянемо приклад неоднозначної граматики. Маємо $G(V_N, V_T, S, P)$, де $V_N = \{S\}$, $V_T = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 0, +, -, (,)\}$, $P = \{S \to S+S \mid S - S \mid (S) \mid 1 \mid 2 \mid \ldots \mid 9 \mid 0 \}$. Так як ми маємо два дерева розбору для вхідного слова “1 + 2 – 3” (рис. 1.1). Неоднозначність граматики відображають два можливі варіанти обчислення виразу: (1 + 2) – 3 (ліво-асоціативна) та 1 + (2 – 3) (право-асоціативна).

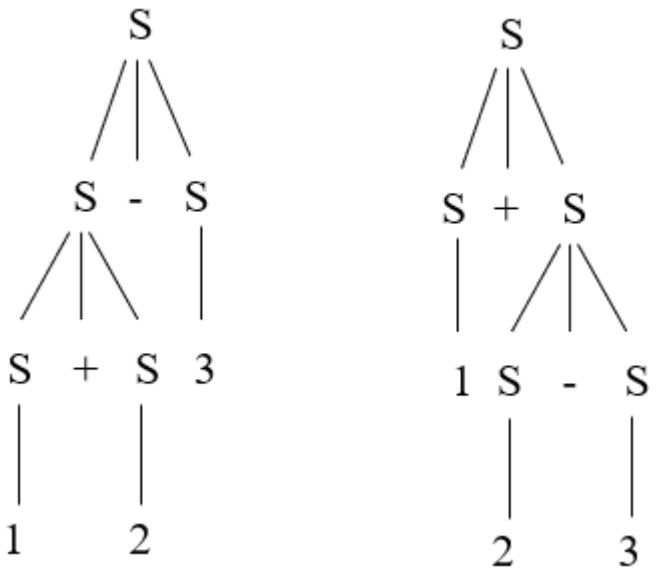

Рис. 1.1 Дерева розбору для "1 + 2 - 3"

Для того, щоб перетворити дану граматику на однозначну, потрібно ввести додатковий нетермінал T. В результаті, отримаємо дві еквівалентні граматики з наступними правилами:

- $P_1 = \{S \to T+S \mid T - S \mid T, T \to (S) \mid 1 \mid 2 \mid \ldots \mid 9 \mid 0 \}$
- $P_2 = \{S \to S+T \mid S - T \mid T, T \to (S) \mid 1 \mid 2 \mid \ldots \mid 9 \mid 0 \}$

У випадку P_1 операції + та – право-асоціативні, у P_2 – ліво-асоціативні. Дерева розбору для того ж виразу “1 + 2 – 3” тепер виглядатимуть інакше (рис. 1.2).

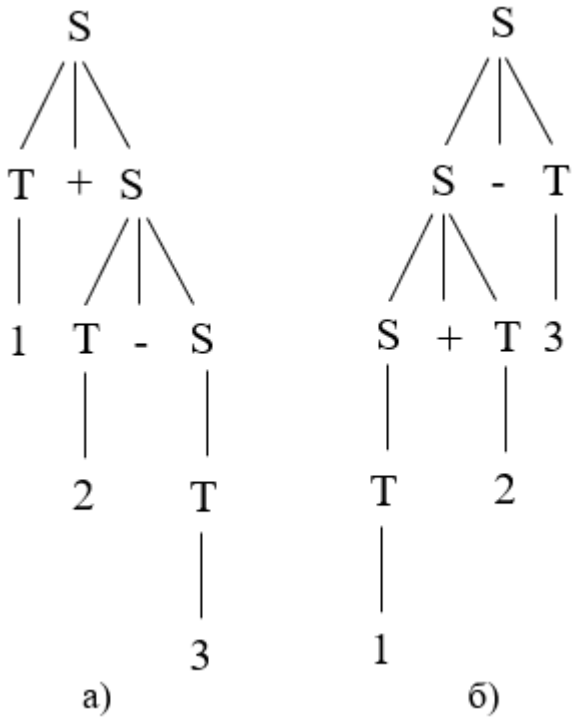

Рис. 1.2 Дерева розбору з використанням право-асоціативних (а) та ліво-асоціативних (б) правил

1.3 Роль синтаксичного аналізатора

Основне завдання синтаксичного аналізу – це перевірка, чи може дана стрічка бути породженою заданою граматикою. Також від нього очікується отримання повідомлень про помилки, які виникли в процесі аналізу. Якщо таких немає та вхідний текст є коректним, то в результаті виконання програми синтаксичний аналізатор будує дерево розбору та передає його наступній частині компілятора для подальшої обробки. [5]

1.3.1 Аналіз вхідних даних

Синтаксичний аналізатор – частина компілятора, яка приймає на вхід результат роботи лексичного аналізатора і аналізує вхідні дані відповідно до

граматики вхідної мови. Іноді ці два етапи об’єднують у один, але такий підхід знижує ефективність компілятора. Є декілька причин, чому фаза аналізу компіляції ділиться на синтаксичний та лексичний аналізи. Одна із основних – спрощення розробки. Розподіл на дві фази часто дозволяє полегшити одну із них. Наприклад, якщо до синтаксичного аналізу ми додамо обробку коментарів та зайвих проміжків (пробілів), то ми цим самим ускладнимо цей процес. Для того, щоб спростити роботу, краще обробляти їх окремо – під час лексичного аналізу.

1.3.2 Обробка помилок

Помилки доволі часто зустрічаються у роботах навіть досвідчених програмістів. Тому хороший компілятор повинен вміти виявляти ці помилки та сповіщати про їх наявність.

Існують різні типи помилок:

- лексичні – некоректно записані ідентифікатори, ключові слова або ж оператори;
- синтаксичні – неправильно розставлені розділові знаки (наприклад зайва дужка або ж відсутня крапка з комою);
- семантичні – невідповідність між типами;
- логічні – зазвичай такі помилки ніяк не видають себе, але при цьому результат виконання програми не відповідає очікуваному. В якості прикладу можна навести такі два оператори у мові програмування C як присвоєння (=) та порівняння (==).

Бажано, щоб обробник подій синтаксичного аналізатора вмів точно повідомляти про існування помилок. Наприклад, він повинен вказувати на місце в програмі, де виявлена помилка: можна виводити рядок із помилкою або ж його номер. Але при цьому, якщо програма коректна, то такий синтаксичний аналізатор не повинен значно сповільнювати її обробку.

1.4 Низхідний синтаксичний аналіз. Рекурсивний спуск

Є три основних типи синтаксичних аналізаторів: універсальні, низхідні (зверху-вниз) та висхідні (знизу-вверх). [5]

Універсальні методи розбору можуть працювати із будь-якою граматикою, але вони дуже неефективні у використанні в компіляторах. До таких методів належать алгоритм Кока-Янгера-Касами (Cocke-Younger-Kasami) та Ерлі (Earley).

Зазвичай в компіляторах використовують або низхідний, або висхідний методи. В обох методах вхідні дані синтаксичний аналізатор опрацьовує зліва направо. Основний принцип роботи даних методів можна зрозуміти з їх назв: низхідний синтаксичний аналізатор працює зверху-вниз, висхідний – навпаки.

Низхідний синтаксичний аналіз – побудова дерева розбору починаючи із кореня та подальше створення вузлів дерева. До алгоритмів даного типу аналізу належить метод рекурсивного спуску (recursive-descent parsing). Цей метод більш потужний та частіше використовується в реальних компіляторах.

Програма синтаксичного аналізу методом рекурсивного спуску складається із набору процедур, по одній для кожного нетерміналу. Робота програми починається із виклику процедури для початкового символу і успішно завершується у випадку сканування всієї вхідної стрічки. [5]

В загальному випадку для рекурсивного спуску може виникнути необхідність у поверненні – повторне сканування вхідних даних. Розглянемо приклад для кращого розуміння.

Нехай у нас дана наступна граматика $G(V_N, V_T, S, P)$, де $V_N = \{S, A\}$, $V_T = \{a, b, c, d\}$, $P = \{ S \to aSA, S \to b, A \to c, A \to dA\}$.

У якості вхідних даних стрічка: w = abc. Щоб побудувати дерево розбору почнемо із вузла зі значенням S. Указник вхідного потоку вказує на “a” –

перший символ w. Беремо першу продукцію S, отримаємо дерево, зображене на рисунку 1.3, а.

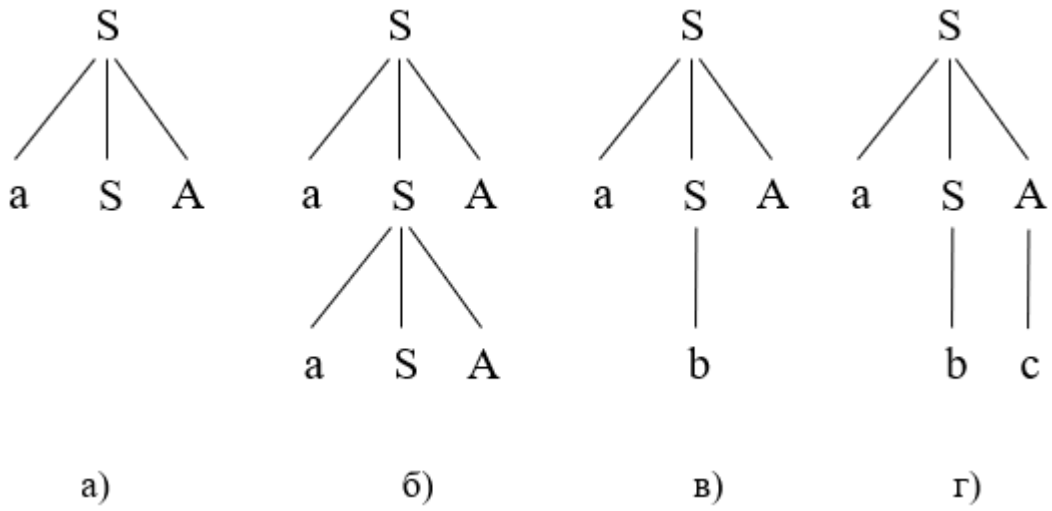

Рис. 1.3 Приклад дерев розбору під час синтаксичного аналізу методом рекурсивного спуску

Так як значення крайнього листка (“a”) дорівнює початковому символу нашої стрічки (“a”), то ми можемо перемістити наш указник на наступний символ вхідного потоку – “b”. Так як наш наступний листок – це “S”, то ми маємо розгорнути його використовуючи перше правило. В результаті отримаємо дерево із рис. 1.3, б. Тепер наш наступний листок – символ “a”, а він не дорівнює наступному символу із вхідного потоку “b”. Отже, ми маємо повернутися до S та перевірити, чи існує альтернативне правило, яке ще не було використано. Таке правило існує: S -> b. Також, нам потрібно і змінити значення указника вхідного потоку: він має вказувати на символ, на який він вказував в той момент, коли ми розгортали S. В цьому прикладі він має вказувати на “b”. Розгорнувши правило S -> b, отримаємо дерево, зображене на рис. 1.3, в. Переміщуємо указник вхідного потоку. Тепер він вказує на символ “c”, а значення наступного листка – “A”. Отже, нам потрібно його розгорнути, використавши перше правило, а саме A -> c. Отримаємо дерева на рис. 1.3, г. Тепер значення листка дорівнює “c”. Воно співпадає із значенням, на яке вказує указник вхідного потоку. Отже, рухаємся далі. А далі у нас закінчилося слово вхідного потоку і листків у дереві, які ми ще не

переглянули, уже немає. Отже, дерево розбору побудоване. Тепер ми можемо сповістити про завершення аналізу та повернути побудоване дерево.

Варто зауважити, що ліворекурсивна граматика може призвести до безкінечного циклу, якщо синтаксичний аналізатор працює методом рекурсивного спуску. Це відбувається тому, що ми щоразу намагатимемося розгорнути нетермінал, який в свою чергу знову ж таки буде намагатися розгорнути цей ж нетермінал. Тоді виникне зациклювання.

Розділ 2. Огляд існуючих засобів розробки

Haskell чудово підходить для реалізації синтаксичного аналізу. Саме тому існує дуже багато бібліотек, які допомагають у цій справі. Є також і ті, які дозволяються здійснювати лексичний аналіз. Завдяки цим усім бібліотекам можна зробити аналіз не лише будь-якої мови програмування, але й будь-чого, що можна описати формальною граматикою.

Серед найбільш відомих бібліотек варто виділити наступні:

- Parsec – одна із основних бібліотек для здійснення синтаксичного та лексичного аналізу. Вона доволі потужна та легка у використанні. До речі, ця бібліотека настільки добре реалізована, що є аналоги і для інших мов програмування. Найвідоміші із них – це JParsec для Java та FParsec для F#. Крім того, вона встановлюється разом із стандартними бібліотеками Haskell.
- Trifecta – ще одна бібліотека для здійснення синтаксичного аналізу. Однією із її переваг є робота з помилками (їх легко читати та інтерпретувати на відмінну від деяких інших бібліотек) [2].
- Megaparsec – бібліотека монадичних аналізаторів, відгалуження Parsec. Це багатофункціональний пакет, який намагається знайти баланс між швидкістю, гнучкістю та якістю обробки помилок [6]. Ця бібліотека чудово працює із токенами, які є результатом роботи лексичного аналізатора Alex. На даний момент бібліотека може працювати із такими типами вхідного потоку як String, ByteString, Text.
- Attoparsec – швидка бібліотека, головна мета якої – ефективна робота із мережевими протоколами та складними текстовими, двійковими форматами файлів [7]. Дана бібліотека використовує монадичний підхід та має багато спільного із Megaparsec.

- Happy – генератор синтаксичних аналізаторів. Він приймає файл, що містить специфікацію граматики БНФ (Форма Бэкуса-Наура), і створює модуль Haskell, що містить аналізатор для цієї граматики. Happy може працювати спільно із лексичним аналізатором, який надається користувачем. Крім того, він здатний аналізувати потік символів, який подається безпосередньо, але це значно сповільнює його роботу. [8]
- Alex – генератор лексичних аналізаторів. Ним можна скористатися у випадку, якщо у вас немає бажання створювати свій власний лексичний аналізатор для Happy. Для розпізнавання йому надається опис лексем у вигляді регулярних виразів. [9]
- Regex-Applicative – бібліотека для здійснення синтаксичного аналізу. Головна її мета – ефективний аналіз регулярних виразів. Аналізатори можна будувати за допомогою інтерфейсу Applicative. [10]

Це не весь перелік доступних бібліотек в Haskell для синтаксичного та лексичного аналізу. Є й інші відомі, такі як aeson (аналіз JSON даних), cassava (CSV дані), frown, polyparser, frisby та інші.

Розділ 3. Реалізація синтаксичного аналізу на функціональній мові програмування Haskell

3.1 Вибір граматики

Для реалізації синтаксичного аналізу мовою програмування Haskell я обрала таку мову як MiniJava. Java – одна із найпопулярніших мов програмування на даний момент, яка досить добре відома більшості програмістам. MiniJava – це її підмножина. На ній можна чудово продемонструвати можливості бібліотек Haskell для здійснення синтаксичного аналізу.

У додатку А розміщено граматику MiniJava [11]. Id – ідентифікатор, перший символ якого – літера, решта – множина символів, яка складається із літер, цифр або ж символу '_' (множина може бути і порожня). INTEGER LITERAL – ціле число (додатне або від'ємне). Op – бінарна операція, одна із "&&", "<", "+", "–", "*". Коментарі в MiniJava можуть бути між двома лексемами. Існують двох видів:

- починається "/*" та закінчується "*/";
- починається "//" і продовжується до кінця рядка.

Граматики MiniJava належить до такого типу граматик, як контекстно-вільна. Контекстно-вільна граматика, як і будь-яка інша, складається із нетермінальних символів, термінальних, початкового символу та множини правил.

Термінальні символи – базові символи, які утворюють рядки. У нашій граматиці (додаток А) до такого типу символів належать, наприклад, зарезервовані слова такі як "if", "else", "while" та розділови знаки такі як "{", "}" та інші.

Нетермінали – це синтаксичні змінні, які позначають множину рядків. У нашій граматиці (додаток А) до терміналів належать “Statement”, “Exp” та інші.

Початковий символ – один із нетерміналів даної граматики. Множина рядків, які він позначає – це і є мова, яку породжує дана граматика. У поданій граматиці (додаток А) початковим символом є “Program”.

Множина правил визначає спосіб, за якого нетермінальні та термінальні символи поєднуються для створення граматики. Кожне правило складається із нетерміналу, який виступає в якості заголовку (ліва частина), символу “->” (::=) та тіла (права частина) – або порожня множина, або певна послідовість терміналів та нетерміналів.

Варто також врахувати, що дана граматика – неоднозначна (ambiguous). Нагадаю, що неоднозначна граматика – граматика, яка для одного і того ж вхідного рядка видає більше одного дерева розбору [12]. Наприклад, нехай задано наступний рядок: “1 + 2 * 3”. За даною граматикою можна побудувати два дерева (див. рис. 3.1).

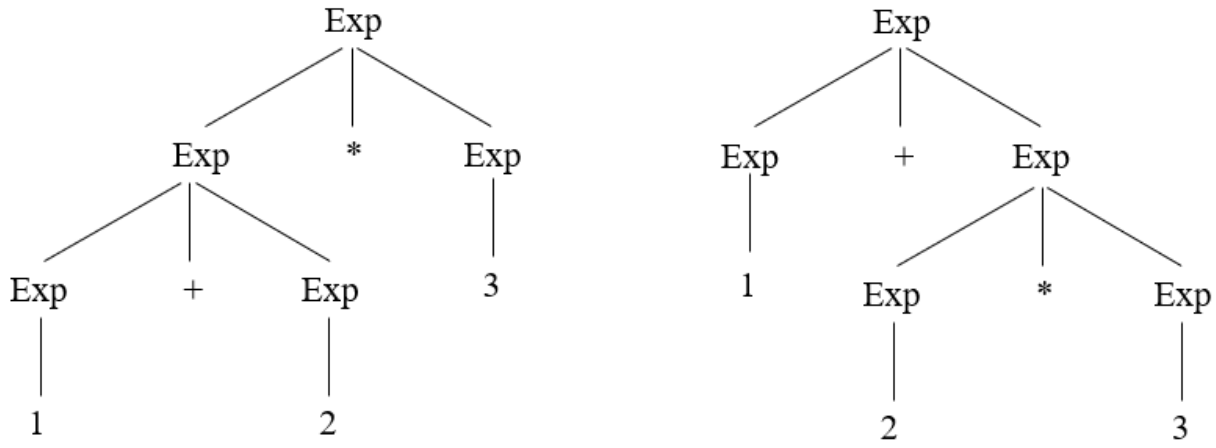

*Рис. 3.1 Дерева розбору у неоднозначній граматиці для "1 + 2 * 3"*

Так як наша граматика неоднозначна то її варто використовувати із правилами усунення неоднозначностей (disambiguating rules). До того ж в такий спосіб ми зможемо врахувати пріоритети операцій.

Крім того, дана граматика – ліворекурсивна, як видно з рисунку 3.2. І цього теж потрібно позбутися.

```
Exp        -> Exp op Exp | Exp [ Exp ] | Exp . "length" | Exp . id ( ExpList )
           | INTEGER LITERAL | "true" | "false" | id | "this" | "new" "int" [ Exp ]
           | new" id () | ! Exp | ( Exp )
```

Рис. 3.2 Ліво рекурсивна частина граматики

Отже, перед тим як починати писати код для синтаксичного аналізу, ми дещо змінимо поточну граматику. Для початку потрібно позбутися лівої рекурсії, потім визначити пріоритети операцій.

В результаті ми отримаємо таку граматику, яка зображена у додатку Б. Такий її вигляд більш зручний для аналізу. Варто зауважити, що у додатку Б використовується ітераційна форма граматик, де |, [,], {, } – метасимволи.

3.2 Побудова структури даних

Перед тим, як почати писати код, потрібно створити структуру даних відповідно до заданої граматики. В результаті отримаємо код, розміщений у додатку В.

3.3 Реалізація синтаксичного аналізу з використанням бібліотеки Applicative Parsing

Перед тим, як розпочати роботу з бібліотекою Regex Applicative Parsing, варто уточнити деякі моменти. Перш за все, необхідно зазначити, що ця бібліотека дозволяє нам виконувати синтаксичний аналіз будь-якої регулярної мови без використання монад. А так як MiniJava – це нерегулярна мова, тому в Applicative Parsing можна реалізувати лише деякі її фрагменти.

Насправді кожна конструкція, яка описана регулярним виразом, може бути описана і за допомогою граматики, але не навпаки. Інакше кажучи, кожна регулярна мова – контекстно-вільна, а навпаки – ні [5].

3.3.1 Тип даних

Як уже було сказано вище, використовуючи цю бібліотеку ми не будемо працювати із монадами, натомість активно використовуватимемо такий тип класу як Applicative. Основний тип даних, який ми будемо використовувати – це RE: data RE s a. Тип s – це фундаментальна одиниця, яка використовується у синтаксичному аналізі. Так як вхідні дані – це стрічка (String), то в якості s у нас буде символ (Char). Тип a – це тип результату, який ми отримаємо внаслідок синтаксичного аналізу.

3.3.2 Написання функцій

Давайте розглянемо, як можна у даному випадку скористатися цією бібліотекою, щоб розпізнати число (лістинг 3.1).

```
numb :: RE Char Expr
numb = (NConst . read) <$> (neg <|> num)
 where
  num = some (psym isNumber)
  neg = (:) <$> sym '-' <*> num
```

Лістинг 3.1 Функція розпізнавання цілого числа

Число – слово, яке складається із символів від 0 до 9. Для розпізнавання таких символів використаємо функцію isNumber. Якщо символ – цифра, то дана функція поверне значення True, в іншому випадку – False. У даній бібліотеці є функція psym, яка приймає параметром функцію із типом s -> Bool

(це і є наша функція isNumber) і як результат повертає символ, якщо він відповідає предикату. Для того, щоб розпізнати один або більше символів, варто скористатися функцією some (варто зауважити, що якщо нам потрібно розпізнати нуль або більше символів, то ми можемо скористатися функцією many). Таким чином, ми розпізнаємо додатне число. Але у нас можуть бути і від'ємні числа. Перед від'ємним варто розпізнати символ '–'. Це можна зробити, написавши наступний код: neg = (:) <$> sym '-' <*> num.

У даному випадку, ми використовуємо такі аплікативні оператори як (<$>) та (<*>). Оскільки кожен оператор повертає свій результат, нам потрібно поєднати їх. Ці оператори надають таку можливість. (<$>) слідує одразу після функції, яка поєднує результати. Ми бачимо, що у нас два аргументи. Для їх комбінації ми використовуємо оператор (<*>). Тобто у даній частині коду спочатку ми розпізнаємо знак '–', потім множину цифр, далі поєднуємо ці дві розпізнані частини у одну.

У даному прикладі є ще одна функція, за яку ми не згадали – (.). Ця функція має тип (b -> c) -> (a -> b) -> a -> c. Тобто ми розпізнали число, але воно ще має тип String. Нам потрібно перетворити його у Int (функція read), щоб застосувати конструктор NConst Int та повернути тип Expr. Ці дії ми виконуємо, використовуючи функцію (.).

Повернемося до аплікативних функцій. Іноді виникають ситуації коли нам потрібно розпізнати певний набір символів або ж просто символ, але далі його не використовувати. У такому випадку ми можемо скористатися іншим оператором, різновидом оператора (<*>), який називається (<*). Розглянемо приклад (лістинг 3.2).

```
lexem :: RE Char a -> RE Char a
lexem p = p <* many (psym isSpace)
```

*Лістинг 3.2 Функція lexem. Приклад застосування оператора <**

В якості параметра дана функція приймає певний аналізатор. Ми виконуємо аналіз, потім розпізнаємо пробіли. Але нам важливе лише те значення, яке розпізнав аналізатор, переданий у якості аргументу. Пробіли нас не цікавлять. Тому ми використовуємо оператор (<*) і повертаємо значення, яке розпізнав аналізатор p. Ще одним різновидом оператора (<*>) є (*>). Його застосовуємо, якщо спочатку нам потрібно розпізнати певний набір символів або символ, але не використовувати його в подальшому. Приклад застосування даного оператора зображено на лістингу 3.3.

```
-- base = number | "true" | "false" | "this" | "new" id
"(" ")" | id
base :: RE Char Expr
base = nmb <|> bool <|> this <|> newId <|> idExpr
 where
  nmb = lexem numb
  bool = BConst <$> (true <|> false)
  true = reserved "true" *> pure True
  false = reserved "false" *> pure False
  this = reserved "this" *> pure This
  idExpr = Var <$> (lexemIden)
  newId = NewId <$> (reservedSp "new" *> lexemIden <*
  symbol '(' <* symbol ')')
```

*Лістинг 3.3 Функція base. Приклад застосування оператора *>*

Даний приклад ілюструє функцію base, яка розпізнає ціле число, зарезервоване слово: true, false this, конструкцію для створення об’єкта класу – new id (), ідентифікатор. Тут теж можна побачити уже знайомі нам оператори (<$>), (<*>), (*>), (<*). Як бачимо, вони застосовуються доволі часто.

Варто також згадати за ще одну аплікативну функцію, яка використовується в останньому прикладі – це (<|>). Таким чином ми визначаємо альтернативний аналізатор у випадку, якщо попередній видав помилку та не спрацював. Тобто в даному прикладі, якщо нам не вдалося розпізнати число, ми пробуємо розпізнати значення типу Boolean. У випадку невдачі далі намагаємося розпізнати зарезервоване слово “this” і т. д.

Варто зауважити, що у функції bool теж застосовується (<|>). Якщо спіткала невдача у випадку розпізнавання зарезервованого слова “true”, ми пробуємо розпізнати інше зарезервоване слово – “false”.

3.3.3 Тестування. Робота з помилками

Як уже було згадано, бібліотека Applicative Parsing надає можливість здійснювати синтаксичний аналіз будь-якої регулярної мови, тому відсутні засоби роботи з помилками. Це ускладнює здійснення аналізу. Наприклад, важко перевірити, чи ідентифікатор – це зарезервоване слово.

Щоб виконати тестування функцій, необхідно скористатися функцією match або ж =~. Якщо аналіз буде вдалим, ми отримаємо (Just значення). В іншому випадку Nothing. Давайте поглянемо на приклади. Спробуємо викликати функцію із рисунку 3.3.

```
*ApplicativeParsing> match base "10"
Just (NConst 10)
*ApplicativeParsing> match base "thue"
Just (Var "thue")
*ApplicativeParsing> match base "true"
Just (BConst True)
*ApplicativeParsing> match base "this"
Just This
*ApplicativeParsing> match base "new A"
Nothing
*ApplicativeParsing> match base "new A()"
Just (NewId "A")
*ApplicativeParsing> "10" =~ base
Just (NConst 10)
*ApplicativeParsing> "-d10" =~ base
Nothing
*ApplicativeParsing> "a)" =~ base
Nothing
```

Рис. 3.3 Результат тестування функції base

Варто зазначити, що функція base не розпізнає ("(" exp ")") та ("new" "int" "[" exp "]") так як бібліотека Applicative Parsing створена для розпізнавання

регулярних виразів, а ми працюємо із нерегулярною мовою. Якщо прописати розпізнавання даних конструкцій, виникає зациклювання.

3.3.4 Висновок

Отже, так як основна мета даної бібліотеки – синтаксичний аналіз регулярних виразів, то вона чудово підходить саме для такого аналізу. MiniJava – нерегулярна мова, тому краще реалізовувати синтаксичний аналіз для неї за допомогою інших бібліотеки.

3.4 Реалізація синтаксичного аналізу з використанням бібліотеки Attoparsec

Attoparsec – наступна бібліотека, яку ми застосуємо під час синтаксичного аналізу мови MiniJava.

Тут ми уже можемо не використовувати аплікативний стиль. Натомість можемо застосовувати монади. Такий підхід зустрічається доволі часто. Код, написаний використовуючи монади, простіше читати та розуміти.

3.4.1 Тип даних

Як і у випадку із попередньою бібліотекою варто розповісти, який основний тип даних ми будемо використовувати. У випадку аплікативного розпізнавання ми використовувати RE Char a, який належить до класу типу Applicative. У даному випадку ми будемо використовувати тип Parser a, де a – тип, який повернеться в результаті виконання аналізу.

3.4.2 Написання функцій

Давайте розглянемо знову функцію, яка дозволить розпізнати число (лістинг 3.4).

```
numb :: Parser Expr
numb = do { n <- signed decimal; return $ NConst n }
```

Лістинг 3.4 Функція розпізнавання цілого числа, написана із використанням do-нотації

В даному випадку ми використовуємо do-нотацію. Крім того, варто зауважити, що ми все ще можемо використовувати аплікативний стиль. На лістингу 3.5 зображено функцію, яка розпізнає теж значення, що і функція, зображена на лістингу 3.4. Відмінність полягає лише у стилі написання. На лістингу 3.5 ми використовуємо аплікативний стиль.

```
numb' :: Parser Expr
numb' = NConst <$> (signed decimal)
```

Лістинг 3.5 Функція розпізнавання цілого числа, написана із використанням аплікативного стилю

Що ж робить дана функція? Розпізнає число. Як уже було сказано в попередньому підрозділі, нам потрібно розпізнати додатне або від'ємне число. Саме число – це множина символів від 0 до 9.

Які ж функції з бібліотеки ми тут використовуємо?

- decimal – розпізнає ціле число без знаку.
- signed – розпізнає число зі знаком '+' або '-'.

3.4.3 Робота з Text та ByteString

Особливістю даної бібліотеки є те, що вона працює із Text або ByteString. Ця особливість сприяє швидкій обробці вхідного рядка, так як

attoparsec працює із даними у бінарному вигляді. Для того, щоб використовувати Text, потрібно імпортувати модуль Data.Attoparsec.Text. Для використання ByteString – модуль Data.Attoparsec. Я у своїй роботі обрала перший варіант.

3.4.4 Тестування. Робота з помилками

Для здійснення аналізу потрібно викликати функцію із назвою parse. Вона приймає два аргументи: перший – функція для здійснення синтаксичного аналізу, другий – стрічка, яку потрібно розпізнати (типу Text). Повертає дана функція один із наступних конструкторів:

- Done i r, де i – це значення, яке ще не було проаналізовано, r – це результат уже проаналізованої частини вхідного рядка.
- Fail i [String] String повертається у випадку помилки. Перший аргумент – стрічка, яка ще не була оброблена, другий – список із контекстом, де було знайдено помилку, третій – повідомлення із описом помилки.
- Partial (i -> IResult i r). Даний конструктор показує одну особливість attoparsec, а саме те, що він аналізує рядки поступово. Якщо attoparsec недостатньо даних для здійснення аналізу, він повідомляє про це і ви можете надати ще дані, використовуючи функцію feed.

Для запуску синтаксичного аналізу можна використати різновид функції parse – parseOnly, який повертає або невдачу, або результат внаслідок вдалого аналізу, використовуючи при цьому Either. Саме цю функцію я використала для запуску.

```
parseStr :: String -> Either String Program
parseStr str = parseOnly program (pack str)
```

Лістинг 3.6 Функція запуску програми

Варто додати, що у даному лістингу коду (лістинг 3.6) ми використовуємо функцію pack. Вона необхідна для того, щоб конвертувати значення типу String у значення типу Text.

Давайте спробуємо протестувати синтаксичний аналіз, написаний з використанням бібліотеки attoparsec, на простій програмі, зображеній на рисунку 3.4. Результат виконання аналізу зображено на рисунку 3.5.

```
class Factorial {
    public static void main(String[] a) {
         System.out.println(new Fac().ComputeFac(10));
    }
}
class Fac {
    public int ComputeFac(int num) {
        int num_aux;
        if (num < 1)
            num_aux = 1;
        else
            num_aux = num * (this.ComputeFac(num-1));
        return num_aux;
    }
}
```

Рис. 3.4 Проста програма, написана на MiniJava

```
*AttoparsecParsing> parseStr prog1
Right (Program {maincls = MainClass {className = "Factorial", argName = "a", mai
nVars = [], mainStmt = [Print (CallM (NewId "Fac") "ComputeFac" [NConst 10])]},
othercls = [ClassDecl {classId = "Fac", extendsClass = Nothing, varsDecl = [], m
ethodsDecl = [MethodDecl {methodType = In, methodId = "ComputeFac", methodParam
= [(In,"num")], methodVars = [(In,"num_aux")], methodStmt = [If (BinOp Less (Var
 "num") (NConst 1)) (Assign "num_aux" (NConst 1)) (Assign "num_aux" (BinOp Mul (
Var "num") (CallM This "ComputeFac" [BinOp Sub (Var "num") (NConst 1)])))], meth
odRes = Var "num_aux"}]}]})
```

Рис. 3.5 Результат синтаксичного аналізу

Варто звернути увагу на ще одну особливість Attoparsec. Хоч він і видає інформацію про помилку, але описання помилки реалізовано не надто добре. Ось що ми отримаємо (рисунок 3.6), якщо в якості вхідних даних подамо наступне значення із синтаксичною помилкою (рисунок 3.7). Як бачимо, виведено інформацію про помилку, але пояснення, де вона виникла та що її спричинило, немає. Але така спрощена робота із помилками сприяє швидкому процесу синтаксичного аналізу.

```
*AttoparsecParsing> parseStr prog1
Left "endOfInput"
```

Рис. 3.6 Результат синтаксичного аналізу на даних, які містять помилку

```
class Factorial {
    public static void main(String[] a) {
         System.out.println(new Fac().ComputeFac(10));
    }
}
class Fac {
    public int ComputeFac(int num) {
        int num_aux;
        if (num < 1);
            num_aux = 1;
        else
            num_aux = num * (this.ComputeFac(num-1));
        return num_aux;
    }
}
```

Рис. 3.7 Проста програма на MiniJava, що містить синтаксичну помилку

3.4.5 Висновок

Хоч і робота із помилками реалізована не надто добре, Attoparsec має багато інших переваг. Одна із них – він дуже швидкий, що робить його чудовим вибором для синтаксичного аналізу бінарних файлів.

3.5 Особливості реалізації синтаксичного аналізу з використанням бібліотеки Parsec

Бібліотека Parsec дещо схожа на Attoparsec, але має свої відмінності. Перш за все, вона теж дозволяє працювати із нерегулярною мовою, тому синтаксичний аналіз для MiniJava можна реалізувати повністю.

3.5.1 Тип даних

Ми будемо використовувати тип Parser a, де a – тип, який повертаємо внаслідок здійснення синтаксичного аналізу.

3.5.2 Написання функцій. Комбінатор try

У випадку як з Attoparsec функції ми можемо писати, використовуючи як і аплікативний стиль, так і do-нотацію.

Перша особливість – комбінатор try. Його часто використовують із оператором альтернативи. Ця практика називається backtracking (зворотнє відстежування) [13]. Давайте поглянемо на приклад на лістингу 3.7.

```
reserved :: String -> Parser ()
reserved str =
 try (lexem $ string str *>
 notFollowedBy alphaNum *> return ())
```

Лістинг 3.7 Функція reserved. Приклад використання try

Нам потрібно прочитати зарезервоване слово (наприклад “if”, “else” або будь-яке інше). Дана функція робить це, причому ми перевіряємо, щоб після розпізнаного слова не було інших символів алфавіту або чисел. Також ми видаляємо усі пробіли, коментарі. Але якщо нам не вдалося прочитати

зарезервоване слово, то вхідні дані залишаються такими ж, як і до виконання цієї функції. Якби ми не використовувати try, то при невдалому розпізнавані зарезервованого слова, ми втрачали б вхідні дані. А це негативно вплинуло б на подальше виконання синтаксичного аналізу: виводило б помилку в подальшому або ж неправильно розпізнавало вхідні дані.

Варто зазначити, що try дорогий у використанні, тому що ми зберігаємо вхідні дані. Таким чином, тут дуже добре показано метод рекурсивного спуску, який використовується і в попередніх бібліотеках.

3.5.3 Тестування. Робота з помилками

Ще одна особливість даної бібліотеки – виведення інформації про наявність помилки. Parsec напевно найкраще підходить у такому випадку, адже він досить добре сповіщає не лише про існування помилки, але й вказує рядок та місце. Спробуємо запустити програму для перевірки коду, зображеного на рисунку 3.7.

```
*ParsecParsing> parseStr prog1
*** Exception: (line 9, column 13):
unexpected ";"
expecting space, "//", "/*", "{", "if", "while", "System" or letter
CallStack (from HasCallStack):
  error, called at ParsecParsing.hs:289:16 in main:ParsecParsing
```

Рис. 3.8 Виведення помилки в Parsec

Як результат (рис. 3.8) дійсно бачимо виведення інформації про помилку із вказанням місця, де саме вона була знайдена.

До того ж ми самі можемо створювати помилки в коді. Наприклад зарезервоване слово не може бути ідентифікатором. Отже, після того, як ми прочитали ідентифікатор, потрібно перевірити, чи це не зарезервоване слово. Це можна зробити наступним чином (лістинг 3.8).

```
iden :: Parser String
iden =
 do l <- letter
    lrest <- many (alphaNum <|> char '_')
    return (l : lrest)

identifier :: Parser Id
identifier = try (do
    idn <- iden
    if elem idn reservedL
    then unexpected ("reserved word " ++ show idn)
    else return idn)

reservedL :: [String]
reservedL =
  ["class", "public", "static", "void",
   "main", "String","extends",
   "return", "int", "boolean", "if",
   "else", "while", "System",
   "out", "println", "length", "true",
   "false", "this", "new"]
```

Лістинг 3.8 Розпізнавання ідентифікатора

У випадку, якщо ми використаємо зарезервоване слово, то програма поверне помилку (рис. 3.9).

```
*ParsecParsing> parse identifier "" "if"
Left (line 1, column 3):
unexpected reserved word "if"
expecting letter or digit or "_"
```

Рис. 3.9 Сповіщення про помилку в ідентифікаторі

Як видно із рисунку, для того, щоб виконати функцію синтаксичного аналізу, потрібно скористатися функцією parse, яка має три параметри: перший – функція для здійснення аналізу, другий – може бути порожнім рядком, використовується лише в повідомленнях про помилку, третій – вхідні дані, які потрібно проаналізувати. Повертає дана функція значення типу Either ParseError a.

Що ми отримаємо в результаті помилки, ми побачили. А що ж буде, якщо подати на вхід коректні дані (рис. 3.4). Відповідь на це питання на рисунку 3.10.

```
*ParsecParsing> parseStr prog1
Program {maincls = MainClass {className = "Factoria
l", argName = "a", mainVars = [], mainStmt = [Print
 (CallM (NewId "Fac") "ComputeFac" [NConst 10])]},
thercls = [ClassDecl {classId = "Fac", extendsClass
 = Nothing, varsDecl = [], methodsDecl = [MethodDec
l {methodType = In, methodId = "ComputeFac", method
Param = [(In,"num")], methodVars = [(In,"num_aux")]
, methodStmt = [If (BinOp Less (Var "num") (NConst
)) (Assign "num_aux" (NConst 1)) (Assign "num_aux"
(BinOp Mul (Var "num") (CallM This "ComputeFac" [Bi
nOp Sub (Var "num") (NConst 1)]))))], methodRes = Va
r "num_aux"}]}]}
```

Рис. 3.10 Результат виконання синтаксичного аналізу із використанням Parsec

3.5.4 Додаткові модулі

Варто ще згадати про деякі модулі з бібліотеки Parsec, які дозволять спростити створення лексичного та синтаксичного аналізу для певної мови програмування. Це модулі Text.ParserCombinators.Parsec.Language, Text.Parsec.Prim та Text.ParserCombinators.Parsec.Token. Імпортувавши їх, потрібно створити визначення мови. Для цього треба викликати конструктор emptyDef та присвоїти відповідні значення (лістинг 3.9):

- вказати типи коментарів та прописати, як ідентифікувати їх початок та кінець;
- вказати зарезервовані слова – список типів String, зарезервовані операції – теж у вигляді списку;
- описати ідентифікатор – вказати, яким повинен бути перший символ, та описати решту символів, які входять до складу ідентифікатора.

```
languageDef :: GenLanguageDef String u
  Data.Functor.Identity.Identity
languageDef =
 emptyDef
  { Token.commentStart    = "/*"
  , Token.commentEnd      = "*/"
  , Token.commentLine     = "//"
  , Token.identStart      = letter
  , Token.identLetter     = digit <|> char '_' <|> letter
  , Token.reservedNames   =
    [ "class", "public", "static", "void"
    , "main", "String", "extends", "return"
    , "int", "boolean", "if", "else", "while"
    , "System", "out", "println", "length"
    , "true", "false", "this", "new" ]
  , Token.reservedOpNames =
    ["+", "-", "*", "=", "<", "&&", "!"]
   }
```

Лістинг 3.9 Застосування languageDef

Дані модулі полегшують обробку індентифікаторів та зарезервованих слів, адже в такому випадку нам не потрібно прописувати їх розпізнавання вручну. Крім того можна скористатися уже готовими функціями, які є у цих модулях, для розпізнавання цілих чисел, знаків (наприклад, “,”, “;”, “.”), різних видів дужок.

3.5.5 Висновок

Як бачимо, дана бібліотека Parsec чудово підходить для синтаксичного аналізу нерегулярної мови. Вона досить добре сповіщає про помилки та має багато своїх особливостей, які вирізняють її з поміж інших бібліотек. Але при цьому треба бути уважним у застосуванні try.

Розділ 4. Порівняння використаних бібліотек

Усі три використані мною бібліотеки (а саме Applicative Parsing, Attoparsec та Parsec) мають як переваги, так і недоліки. Наприклад, Applicative Parsing чудово підходить для роботи із регулярними виразами, але для нерегулярної мови її використовувати не можна. Або навіть якщо і використовувати, то лише для аналізу певної частини. Крім того, так як Applicative Parsing працює із регулярними виразами, то у цієї бібліотеки немає засобів для роботи із помилками.

Attoparsec та Parsec схожі між собою. Насправді, код, написаний з використанням Applicative Parsing, можна майже не змінювати для того, щоб він працював і з використанням Attoparsec або ж Parsec. Більше того, дані бібліотеки можна використовувати для написання синтаксичного аналізу для контекстно-вільної граматики. Варто додати, що ці дві бібліотеки все ж мають і відмінності – свої нюанси, на які потрібно звернути увагу.

Attoparsec чудово підходить для роботи із бінарними даними, тому що у процесі створення коду ви працюєте із такими типами як Text або ByteString, в залежності від модулю, який імпортуєте. Ця особливість значно пришвидшує синтаксичний аналіз. До того ж у Attoparsec вбудована здатність backtracking, тобто не потрібно використовувати try для того, аби не втратити вхідні дані при аналізі. Але як і у випадку із Applicative Parsing не дуже добре надані можливості обробки помилок та створення повідомлень про їх існування.

Якщо говорити про Parsec, то він чудово підходить для роботи із помилками. При введені некоректних даних, він сповіщає про те, де знаходиться помилка, та який символ став її причиною. Крім того Parsec працює із більш поширеним типом даних, таких як String. Але у нього є і недолік – доводиться застосовувати функцію try для забезпечення backtracking.

Отже, кожна бібліотека має свої переваги та підходить для різних видів завдань. Тому перед тим, як обрати, яку бібліотеку краще застосувати, варто ознайомитися із мовою (її граматикою), для якої планується написання синтаксичного аналізу, та розставити перед собою пріоритети: на що варто звертати більше уваги при здійсненні синтаксичного аналізу (швидкість, робота з помилками і т. п.).

Висновки

Отже, після детального ознайомлення із процесом синтаксичного аналізу, можу сказати, що це дуже важливий етап процесу компіляції, адже він перевіряє вхідний код на наявність синтаксичних помилок. Іноді відбувається перевірка і лексичних помилок, якщо лексичний та синтаксичний аналізатор поєднані у один.

В процесі створення курсової роботи було проаналізовано та вивчено особливості використання різних бібліотек Haskell (а саме Applicative Parsing, Attoparsec та Parsec), які надають можливість здійснювати синтаксичний аналіз. Як результат було створено три програми, які використовували функціонал вище згаданих бібліотек.

Варто зазначити, що Haskell є однією із найкращих мов програмування для здійснення синтаксичного аналізу, завдяки простому синтаксису, можливості поділити основне завдання на менші (легші), а потім поєднати їх між собою. Тому і створено так багато бібліотек для здійснення як синтаксичного, так і лексичного аналізу. Деякі з них реалізовані настільки добре, що з’явилися аналоги і для інших мов програмування. Крім того, завдяки такій різноманітності, можна підібрати будь-яку із цих бібліотек відповідно до поставленого завдання та розставлених пріоритетів.

Список літератури

1. Parsing with Haskell [Електронний ресурс] – Режим доступу до ресурсу: https://mmhaskell.com/parsing.
2. Allen C. Haskell programming from first principles / C. Allen, J. Moronuki // Haskell programming from first principles / C. Allen, J. Moronuki.. – C. 882–883, 899–900.
3. Глибовець М. М., Проценко В. С., Кирієнко О. В. Курс “Моделі обчислень у програмній інженерії” (НаУКМА)
4. Жежерун О. П. Курс “Системне програмування” (НаУКМА)
5. Syntax Analysis / A. Aho, M. Lam, R. Sethi, J. Ullman // Compilers. Principles, Techniques & Tools. Second Edition / A. Aho, M. Lam, R. Sethi, J. Ullman.. – C. 191–286.
6. megaparsec: Monadic parser combinators [Електронний ресурс] – Режим доступу до ресурсу: https://hackage.haskell.org/package/megaparsec.
7. attoparsec: Fast combinator parsing for bytestrings and text [Електронний ресурс] – Режим доступу до ресурсу: https://hackage.haskell.org/package/attoparsec.
8. Happy. The Parser Generator for Haskell [Електронний ресурс] – Режим доступу до ресурсу: https://www.haskell.org/happy/.
9. Alex: A lexical analyser generator for Haskell [Електронний ресурс] – Режим доступу до ресурсу: https://www.haskell.org/alex/.
10. regex-applicative: Regex-based parsing with applicative interface [Електронний ресурс] – Режим доступу до ресурсу: https://hackage.haskell.org/package/regex-applicative.
11. Appel A. Appendix A: MiniJava Language Reference Manual / A. Appel, J. Palsberg // Modern Compiler Implementation in Java, Second Edition / A. Appel, J. Palsberg.. – C. 518–520.

12. Basten H. Ambiguity in context-free grammars / H.J.S. Basten // Ambiguity Detection Methodsfor Context-Free Grammars / H.J.S. Basten.. – С. 16–17.
13. O'Sullivan B. Real World Haskell [Електронний ресурс] / B. O'Sullivan, D. Stewart, J. Goerzen. – 2008. – Режим доступу до ресурсу: http://book.realworldhaskell.org/read/using-parsec.html.
14. Mena A. Beginning Haskell A Project-Based Approach / Alejandro Serrano Mena // Beginning Haskell A Project-Based Approach / Alejandro Serrano Mena.. – С. 235–236, 241–250.
15. Diehl S. Parsing / Stephen Diehl // What I Wish I Knew When Learning Haskell / Stephen Diehl.. – С. 311–327.
16. Parsing a simple imperative language [Електронний ресурс] – Режим доступу до ресурсу: https://wiki.haskell.org/Parsing_a_simple_imperative_language.
17. Проценко В. С. Курс “Функціональне програмування” (НаУКМА)

Додаток А

(обов’язковий)

Граматика MiniJava

```
Program       -> MainClass ClassDecl*
MainClass     -> "class" id { "public" "static" "void" "main" ( "String" [] id )
                    { VarDecl* Statement* }}
ClassDecl     -> "class" id { VarDecl* MethodDecl* }
              -> "class" id "extends" id { VarDecl* MethodDecl* }
VarDecl       -> Type id ;
MethodDecl    -> "public" Type id ( FormalList )
                        { VarDecl* Statement* "return" Exp ;}
FormalList    -> Type id FormalRest*
              ->    // список формальних параметрів може бути порожнім
FormalRest    -> , Type id
Type          -> "int" []
              -> "boolean"
              -> "int"
              -> id
Statement     -> { Statement* }
              -> "if" ( Exp ) Statement "else" Statement
              -> "while" ( Exp ) Statement
              -> "System.out.println" ( Exp ) ;
              -> id = Exp ;
              -> id [ Exp ] = Exp ;
Exp           -> Exp op Exp
              -> Exp [ Exp ]
              -> Exp . "length"
              -> Exp . id ( ExpList )
```

-> INTEGER LITERAL

-> "true"

-> "false"

-> id

-> "this"

-> "new" "int" [Exp]

-> "new" id ()

-> ! Exp

-> (Exp)

ExpList -> Exp ExpRest*

-> // список фактичних параметрів може бути порожнім

ExpRest -> , Exp

Додаток Б

(обов’язковий)

Граматика MiniJava, отримана в результаті усунення неоднозначностей (ітераційна форма)

```
program     =  mainClass { classDecl }
mainClass   =  "class" id
              "{" "public" "static" "void" "main" "(" "String" "[""]" id ")"
                   "{" {varDecl} {statement} "}" "}"
classDecl   =  "class" id ["extends" id ]
                    "{"  {varDecl} {methodDecl} "}"
varDecl     =  type id ";"
methodDecl =  "public" type id "(" formalList ")"
                    "{" {varDecl} {statement} "return" exp ";" "}"
formalList  =  [type id {"," type id }]
type        =  "int" ["[" "]"] | "boolean" | id
statement   = "{" {statement} "}" | "if" "(" exp ")" statement "else" statement
                    | "while" "(" exp ")" statement
                    | "System.out.println" "(" exp ")" ";"  | id ["[" exp "]"] = exp ";"
exp         = full {"&&" full}
full        = simple ["<" simple]
simple      = term {addOp term}
term        = fact { "*" fact}
fact        = { "!"}  access
access      = base {"[" exp "]"  | "." "length" | "." id "(" factList ")"}
base        = "(" exp ")" | number | id | "true" | "false" | "this" |
                     | "new" id "(" ")" | "new" "int" "[" exp "]"
factList    = [exp { "," exp}]
```

Додаток В

(обов'язковий)

Структура даних для виконання синтаксичного аналізу MiniJava

```
type Id = String

data Expr =
  NConst Int | Var Id | BConst Bool | This | NewId Id | NewInt Expr |
  Index Expr Expr | Lng Expr | CallM Expr Id [Expr] |
  BinOp Bop Expr Expr | NotOp Expr deriving (Show)

data Bop = Add | Sub | Mul | Less | And  deriving (Show)

data Stmt =
  StmtList [Stmt]    |  If Expr Stmt Stmt |  While Expr Stmt    |
  Print Expr         |  Assign Id Expr     |  AssignA Id Expr Expr
   deriving (Show)

data Type = In | InAr | Bln | Class Id
   deriving (Show)

type FormatList = [(Type, Id)]

type VarDecl = (Type, Id)

data MethodDecl = MethodDecl
 {
   methodType  :: Type,
   methodId    :: Id,
   methodParam :: FormatList,
   methodVars  :: [VarDecl],
   methodStmt  :: [Stmt],
   methodRes   :: Expr
 }
   deriving (Show)

data ClassDecl = ClassDecl
 {
   classId      :: Id,
   extendsClass :: Maybe Id,
   varsDecl     :: [VarDecl],
   methodsDecl  :: [MethodDecl]
 }
   deriving (Show)

data MainClass = MainClass
 {
   className :: Id,
   argName   :: Id,
   mainVars  :: [VarDecl],
   mainStmt  :: [Stmt]
 }
   deriving (Show)

data Program = Program
 {
   maincls  :: MainClass,
   othercls :: [ClassDecl]
 }
   deriving (Show)
```