Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики

СИНТАКСИЧНИЙ АНАЛІЗАТОР НА HASKELL

Текстова частина до курсової роботи

за спеціальністю «Інженерія програмного забезпечення» 121

Керівник курсової роботи

ст. викладач Проценко В.С.

_____

(підпис)

“____” _____ 2020 р.

Виконав студент

Ільченко Т. Р.

“____” _____ 2020 р.

Київ 2020

ЗМІСТ

.

Вступ

Синтаксичний аналіз є досить популярним зараз. Будь-який синтаксис може бути проаналізований. Починаючи від мов програмування і закінчуючи мовами, якими ми розмовляємо. Кожен програміст, як початківець так і програміст з досвідом, стикався з задачами синтаксичного аналізу. У світі програмістів існує безліч бібліотек, які можуть стати в нагоді при побудові синтаксичного аналізатора. Адже синтаксичний аналіз є одним з важливих елементів при побудові інтерпретаторів. Граматика кожної мови має свої особливості, тому важливо це враховувати та при побудові синтаксичного аналізатора обирати бібліотеки, як вдало підійдуть для цих цілей.

Мова Haskell є чудовим вибором для побудови синтаксичного аналізатора (парсера). Вона пропонує ряд потужних бібліотек таких, як Parsec, Attoparsec та Megaparsec. Враховуючи функціональні особливості мови Haskell і її здатність працювати з монадами, а також функціонал вище наведених бібліотек, процес створення синтаксичного аналізатора стає легшим.

Метою моє роботи було дослідити бібліотеки Attoparsec та Megaparsec та написати синтаксичний аналізатор для мови MiniJava, використовуючи якусь із них.

1. ОСНОВНА ЧАСТИНА

1.1. Поняття синтаксичного аналізу

Синтаксичний аналіз - це аналіз тексту з метою визначення його синтаксичної структури. Він є наступним кроком після лексичного аналізу, чия задача полягає в тому, щоб зчитати вхідний рядок символів і згрупувати їх в лексеми. Лексемами можуть бути якісь зарезервовані слова, ідентифікатори, літерали, константи, коментарі, тощо. В процесі синтаксичного аналізу перевіряється, чи породжується даний рядок лексем даною граматикою. Відомо, що будь-яку граматику можна проаналізувати за допомогою синтаксичного аналізатора. Проте найбільш застосовуваним та ефективним цей аналіз є для контекстно-вільних граматик. Результатом роботи синтаксичного аналізатора є дерево розбору.

1.2. Контекстно-вільна граматика

Контекстно-вільна граматика G – це четвірка (N, T, P, S):

N – алфавіт нетермінальних символів

T – алфавіт термінальних символів

P – правила виводу

S – нетермінальний символ, з якого починаємо опис граматики (аксіома)

У контекстно-вільній граматиці всі правила мають вигляд:

$$\alpha \to \beta$$
$$\alpha \in N, \beta \in (T \cup N)^*$$

Розглянемо приклад граматики:

G = ({5,7},{B,C}, P, S}, тоді:

N = {B,C}

T = {5,7}

P складається з таких правил:

S → 5B7

5B → 55B7

B → ג, де ג - пустий ланцюг, у якому немає жодного символа

1.3. Види виводу

Вивід – це процес заміни нетермінала на правило. Існують два види виводу: лівосторонній та правосторонній.

Лівосторонній вивід – це певна послідовність кроків виводу, де на кожному кроці береться крайній лівий нетермінал.

Правосторонній вивід – це певна послідовність кроків виводу, де на кожному кроці береться крайній правий нетермінал.

Розглянемо граматику та побудуємо лівосторонній та правосторонній вивід для ланцюжка x+y+x:

A → B

A → B + A

B → x

B → y

Лівосторонній:

A → B + A → x + A → x + B + A → x + y + A → x + y + B → x + y + x

Правосторонній:

A → B + A → B + B + A → B + B + B → B + B + x → B + y + x → x + y + x

Отже, бачимо, що для контекстно-вільних граматик можна побудувати, як лівосторонній, так і правосторонній вивід.

1.4. Дерево розбору

Для більш наглядного представлення структури програми замість кроків виводу використовують дерево розбору, або дерево виводу – це графічне представлення кроків виводу у вигляді зв’язного орієнтованого графа. Коренем дерева є початковий символ. Заключні вершини – це термінальні символи і, відповідно, незаключні – це нетермінальні. Побудуємо дерево розбору для прикладу, наведеного вище (Рисунок 1):

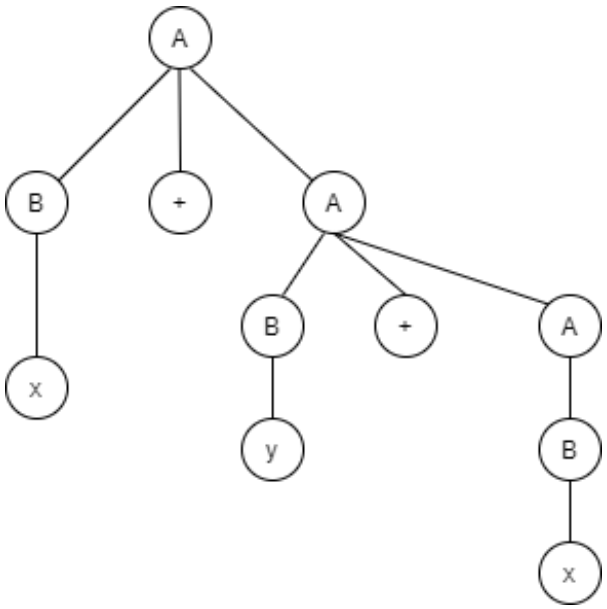

Рисунок 1. Дерево розбору

Дане дерево будується однаково для обох варіантів виводу. Проте це зовсім не означає, що завжди якомусь певному виводу відповідає лише одне дерево розбору, їх може бути декілька.

1.5. Неоднозначність граматики

Можемо говорити про неоднозначність граматики, якщо для якогось ланцюжка існує декілька дерев розбору. Розглянемо граматику G:

S → S + S

S → S*S

S → a

S → b

S → c

Побудувавши вивід для ланцюжка a*b+c, можемо пересвідчитись, що їх декілька:

S → S*S → S*S+S → a*S+S → a*b+S → a*b+c

S → S+S → S*S+S → a*S+S → a*b+S → a*b+c

Відповідно і відрізнятимуться дерева розбору (Рисунок 2):

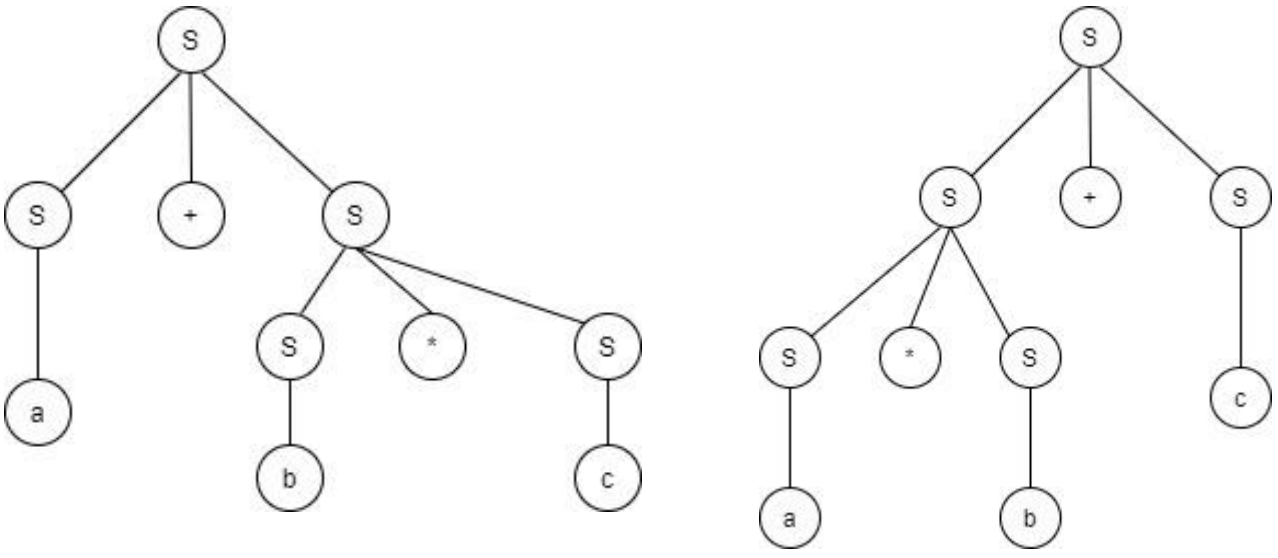

Рисунок 2. Дерева розбору неоднозначної граматики

Неоднозначна граматика є однією з проблем синтаксичного аналізу, адже невідомо, в якому порядку застосовувати правила виводу і яке саме дерево розбору ми отримаємо в кінці. Тому важливо позбуватись неоднозначності. Для цього до граматики додають деякі нетермінали. Наприклад, граматика, наведена вище, набуде вигляду:

S → S + A | A

S → A

A → B

A → A*B | B

B → a

B → b

B → c

Тоді для ланцюжка a*b+c існує наступний єдиний вивід:

S → S+A→ A+A→ A*B+A→ B*B+A→ B*B+A→ a*B+A→ a*b+A→ a*b+c

1.6. Рекурсивні граматики

Граматика є рекурсивною, якщо для неї існує правило вигляду A → αAβ. Якщо ж β → ג, де ג - порожній символ, то правило набуває вигляду A → αA, і тоді така граматика є праворекурсивною. Якщо ж для граматики існує правило вигляду A → Aβ (у цьому випадку α → ג), то вона називається ліворекурсивною. Достатньою умовою ліворекурсивності є наявність хоча б одного ліворекурсивного нетермінала. Ліворекурсивні граматики є однією з

проблем синтаксичного аналізу. Тому перед тим, як розпочати синтаксичний аналіз, потрібно позбутись ліворекурсивності. Для цього використовують наступний алгоритм:

1. Спочатку записують всі правила виводу для нетермінала A: $A \rightarrow A\beta1 \mid A\beta2 \mid \ldots \mid A\beta n \mid \alpha1 \mid \alpha2 \mid \ldots \mid \alpha m$
2. Далі переписують правила виводу наступним чином:

$$B \rightarrow \alpha1B \mid \alpha2B \mid \ldots \mid \alpha mB \mid \alpha1 \mid \alpha2 \mid \ldots \mid \alpha m$$

На цьому етапі вводять новий нетермінал B, для якого граматика тепер виглядає так:

$$B \rightarrow \beta1B \mid \beta2B \mid \ldots \mid \beta nB \mid \beta1 \mid \beta2 \mid \ldots \mid \beta n$$

При чому, початкова ліворекурсивна граматика та граматика, яку ми отримали, позбувшись ліворекурсивності, є рівнозначними.

1.7. Класифікація за методами синтаксичного розбору

Оскільки граматики можуть бути різними, то до кожної варто застосовувати оптимальний підхід. Для роботи з граматиками відомі декілька таких методів: висхідний, низхідний та комбінований. Проте найбільш вживаними та часто застосовуваними є перші два. Ці підходи відрізняють тим, яким чином буде побудоване дерево розбору. Так, при низхідному підході починаємо аналіз та побудову з кореня (початкового символа). Далі застосовуємо правило, яке породжується початковим нетерміналом. Процес відбувається доти, доки ми не встановимо зв’язки між коренем дерева та усіма символами (листками дерева), що є елементами вхідного ланцюжка.

Основна ідея - це застосувати правила граматики, починаючи з початкового символа, щоб в кінці аналізу у нас побудувався вхідний рядок.

При висхідному підході побудова дерева розпочинається з терміналів (листків) і відбувається доти, доки коренем дерева не виявиться початковий нетермінальний символ, а всі символи вхідного ланцюжка не стануть листками. Тут основна ідея - рухатись з вхідного ланцюжка до початкового символа.

Розглянемо приклад висхідного синтаксичного аналізу для ланцюжка abbcde граматики вигляду:

S → aPTe

P → Pbc

P → b

T → d

Спочатку створюємо листки для терміналів вхідного ланцюжка (Рисунок 3).

Рисунок 3

Далі проходимось зліва направо і шукаємо правило, яке можна застосувати до якогось з терміналів. Підходить правило P → b (Рисунок 4):

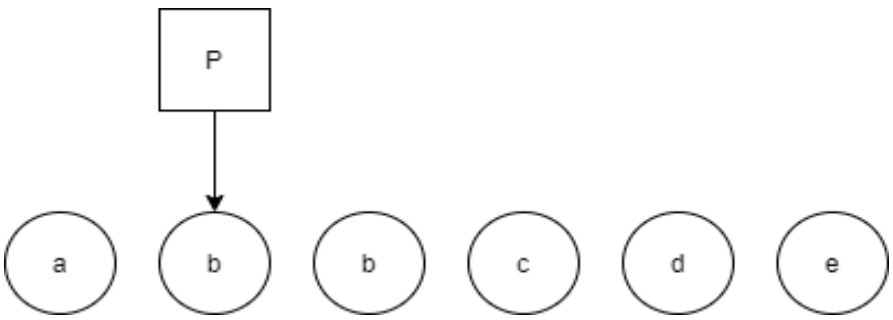

Рисунок 4

Тепер P теж вважається листком дерева розбору, тому враховуємо це і шукаємо відповідне правило. Підходить P → Pbc (Рисунок 5):

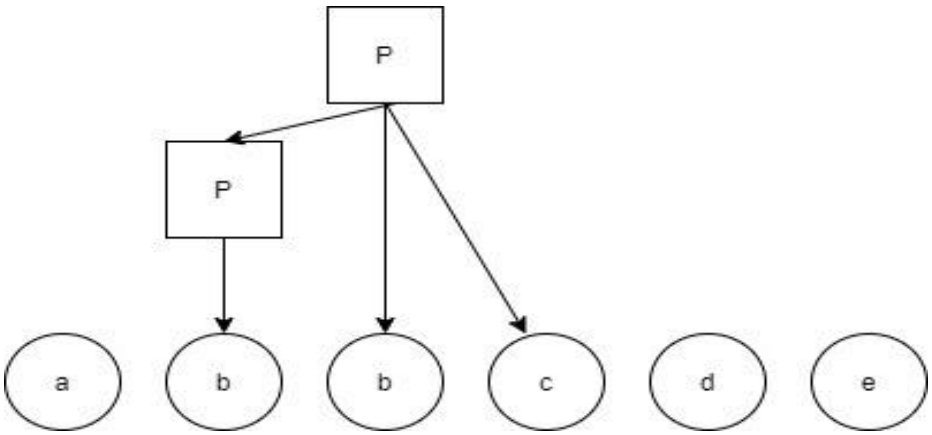

Рисунок 5

Оскільки ми все ще проходимось зліва направо, то наступне правило, яке нам підходить - це T → d (Рисунок 6):

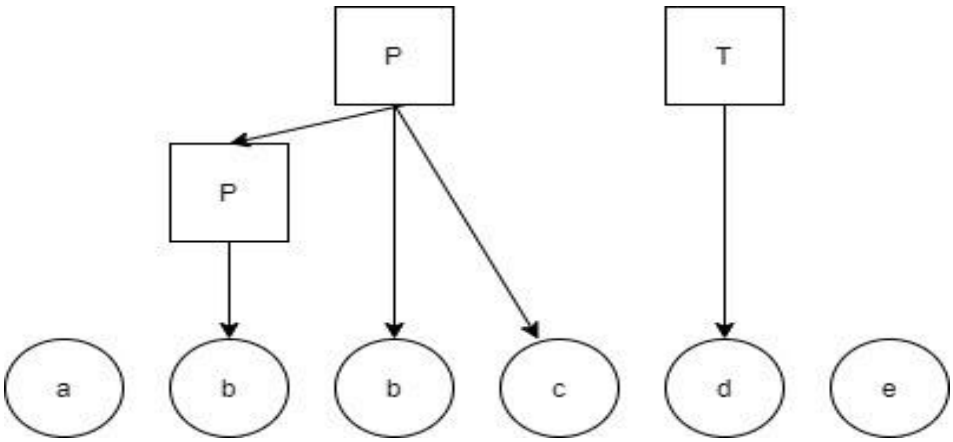

Рисунок 6

Для термінала e у нас немає правила, тому вертаємось на початок і шукаємо правило, яке б задовільняло листкам a, P, T, e. S → aPTe (Рисунок 7):

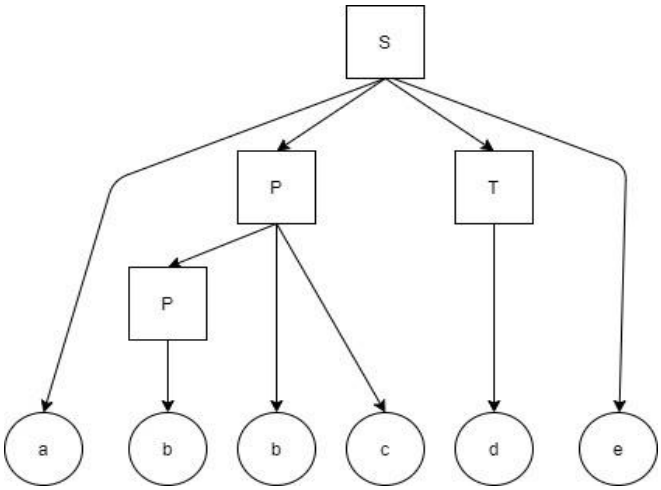

Рисунок 7

В результаті бачимо, що корінь дерева - це початковий символ S, а всі символи вхідних даних - це листки.

1.8. LR(k) граматики

LR(k) - це граматика, яка належить до контекстно-вільних граматик. В граматиці даного виду ланцюжок зчитується зліва направо і вивід буде правосторонній. Символ k означає кількість символів попереднього перегляду. Для аналізу таких граматик дуже часто застосовують shift-reduce метод.

Цей алгоритм може бути застосований до довільної КВ-граматики. Shift-reduce метод використовує стек, МП-автомат, а також таблицю, на основі якої визначається, яку операцію застосувати: shift чи reduce. МП-автомат - це так званий автомат з магазинною пам’яттю. Його використовують для того, щоб запам’ятати поточний стан стеку, адже нам при проходженні стрічки потрібно запам’ятовувати і десь зберігати символи, які ми вже пройшли.

Сам принцип виглядає наступним чином:

1. Зчитуємо перший термінальний символ та заносимо його в стек. За це відповідає операція shift. Це відбувається доти, доки ми не знайдемо підланцюжок чи символ, який відповідатиме правій частині якогось з даних нам правил.
2. Якщо ми знаходимо такий підланцюжок, то ми його вилучаємо з стеку, і на його місце ставимо ліву частину відповідного правила. За це відповідає операція reduce.

Якщо в результаті у нас на вершині стеку залишається перший нетермінал, а сама стрічка є порожньою, то результат роботи є успішним.

Для того, щоб shift-reduce метод коректно виконував свою роботу, граматика повинна бути однозначною. В протилежному випадку виникатимуть конфлікти, відомі як shift/reduce - коли одночасно можуть бути застосовані дві операції, і shift/shift - коли двічі застосовуємо shift.

1.9. LL(k) граматики

Цей тип граматики теж належить до контекстно-вільних. Хоча ланцюжок тут зчитується зліва направо, вивід будується лівосторонній, на відміну від LR(k) граматик. Синтаксичний аналіз граматик LL(k) відноситься до низхідних. До таких граматик дуже часто застосовують метод рекурсивного спуску, який є досить простим алгоритмом. За допомогою нього легко можна розпарсити якийсь арифметичний вираз. Оскільки він добре реалізується для LL(1) граматик, то нам важливо перед цим позбутись рекурсії. Метод рекурсивного спуску використовує рекурсивні процедури.

Основна ідея:

1. З вказаного місця вхідного ланцюжка знайти підланцюжок, який може виводитись з цього нетермінала.
2. Під час пошуку процедура може рекурсивно викликати сама себе для пошуку потрібних нетерміналів.
3. Якщо процедура знайшла відповідність до нетермінала, то вона завершується успішно і повертається на точку виклику.
4. Якщо ж знайти потрібний підланцюжок для нетермінала знайти не вдалось, то процедура завершується аварійно і може вивести повідомлення про помилку.

2. СИНТАКСИЧНИЙ АНАЛІЗ В ХАСКЕЛІ

Функціональні мови, а Haskell є однією з них, чудово підходять для реалізації синтаксичних аналізаторів, адже пропонують великий спектр можливостей для полегшення роботи. Дуже часто виникає потреба розпарсити файл чи дані різних типів. Хоча є варіант використовувати регулярні вирази, проте такий підхід все частіше відходить на другий план. Бо у випадку з складними типами даних регулярні вирази не можуть та ефективно впоратись.

В мові Haskell є декілька бібліотек, які активно використовуються для синтаксичного аналізу різного роду граматик. В рамках моєї курсової роботи для побудови синтаксичного аналізатора було використано граматику мови MiniJava, яка є підмножиною мови Java (додаток 1). Відмінність цих двох мов полягає в тому, що в MiniJava немає перегрузки, оператор System.out.println() друкує лише цілі числа, а вираз a.length застосовується тільки до масивів цілих чисел. Для реалізації парсера було використано бібліотеку Attoparsec.

2.1. Attoparsec

Attoparsec - це бібліотека парсерних комбінаторів. Комбінатори дозволяють нам складати функції вищого порядку для того, щоб згенерувати парсер. При використанні бібліотеки Attoparsec, побудова синтаксичного аналізатора буде відбуватись в стилі монад. Це значно полегшує роботу та сприйняття на відміну від аплікативного підходу. Attoparsec націлений більше на роботу з мережевими протоколами та файлами двійкового і текстового формату. Відповідно він працює з типами ByteString, Char8 і Text. Надалі опис буде відбуватись зсилаючись на Модуль Data.Attoparsec.Text, який створює парсери, що працюють з типом Text.

Бібліотека пропонує декілька типів (Рисунок 8):

```
type Parser = Parser Text

type Result = IResult Text

data IResult i r
```

Рисунок 8

Attoparsec дуже схожий на бібліотеку Parsec, проте дещо відрізняється. До прикладу, в Attoparsec введення вхідних даних відбувається поступово, тобто на вхід можна передати спочатку лише частину даних. Такий підхід дозволяє користувачеві ефективніше керувати процесом.

Якщо парсер розпарсив вже кінець стрічки і може отримати на вхід решту даних, то він поверне Partial. Partial входить в констуктор IResult, який є результатом парсингу вхідних даних. Якщо ж на вхід подати наступну частину даних, то Partial продовжить з тої точки, де він зупинився, а наступна частина даних буде новими вхідними даними в кінці попередніх. Якщо ж на вхід більше немає чого подати, то просто подаємо на вхід порожній рядок. У випадку, якщо поступове подання даних на вхід парсеру не є потрібним, то для запуску парсера використовують функцію parseOnly замість функції parse.

Помилки в Attoparsec є менш ефективними, ніж в Parsec, що пришвидшує його роботу. Така висока ефективність Attoparsec ще спричинена двома парсерами: string та takeWhile.

В практичній частині моєї курсової роботи було використано тільки монаду Parser. Він повертає список можливих результатів розбору і порожній список у випадку невдачі. Як вже було сказано раніше, монади значно полегшують роботу. Наприклад, ми можемо вирішувати, які саме дії робити на поточному стані, беручи до уваги попередні результати. Аплікативний підхід цього робити не може, нам потрібно наперед визначати, які операції варто зробити, щоб отримати бажаний результат.

2.2. Деякі особливості монад

Однією з функцій, яка належить класу монад, є функція return, вона є аналогом функції pure, що належить класу аплікативних функторів. Тип функції return виглядає так: (Monad m) => a → m a. Тобто вона приймає на вхід якесь значення та обгортує його в монаду. Важливо розуміти, що вона не є завершенням функції, її основна мета - це помістити значення в контекст.

Важливим поняттям є do-нотація, яку придумали для того, щоб спростити не тільки запис та читабельність коду, а і надати змогу легше підставляти значення в функції з декількома змінними. Вирази записуються порядково, і тут важливо дотримуватись послідовного запису, адже кожне наступне значення залежить від результатів попереднього (Рисунок 9). В do нотації також можна вводити локальні змінні за допомогою let.

```
parseFormalVar:: Parser FormalVar
parseFormalVar = do
                 varType <- getType
                 varId <- getClassId
                 return (varType, varId)
```

Рисунок 9

2.3. Комбінатори синтаксичного аналізу

Для того, щоб з якихось базових парсерів, як наприклад числові чи рядкові, побудувати більш складні та більші, використовуються комбінатори, які можуть об’єднати декілька парсерів. Загалом використання комбінаторних монадних парсерів - це досить типовий функціональний підхід для синтаксичного аналізу. Один з таких комбінаторів - це комбінатор вигляду < | > - це так звана диз’юнкція двох парсерів. Тобто, якщо перший з парсерів не видає успішний результат, то буде застосовано наступний (Рисунок 10).

```
expHelper:: Parser Exp
expHelper = try(createArray)
          <|> try(createObject)
          <|> try(thisExp)
          <|> try(falseExp)
          <|> try(trueExp)
          <|> try(number)
          <|> try(notOp)
          <|> getClassId
```

Рисунок 10

Проте, у цього комбінатора є певна особливість. Якщо ж все-таки при проходженні операндів на якомусь з них парсер зазнає невдачі, то комбінатор не повернеться на початок. Тому часто з комбінатором < | > використовують ще один популярний комбінатор try, який дозволяє повернутись на початок. Try - це бектрекінг оператор, який добре підходить для парсингу неоднозначних виразів.

Ще одним базовим комбінатором являється комбінатор вигляду <*>. Його тип визначається так: f (a → b) → f a → f b. За допомогою цього комбінатора функції можна передавати більше аргументів. Тобто він витягує функцію та застосовує до неї один з аргументів. Більш того, його можна застосовувати до кожного аргумента функції з декількома аргументами. Обидва комбінатори належать до класу типу Applicative. Загалом Applicative та Monad дуже тісно зв'язані. Тобто кожне значення типу монад є одночасно значенням типу аплікативних функторів.

Варто ще згадати про такі комбінатори, як sebBy, skipMany, manyTill. Комбінатор sepBy призначений для синтаксичного аналізу деяких лексем, які розділені певним розділювачем. Наприклад, на Рисунку 11 видно, що sepBy застосовується до формальних змінних, розділених між собою комою.

```
parseFormalList:: Parser  FormalList
parseFormalList = sepBy parseFormalVar (string ",");
```

Рисунок 11

А skipMany, в свою чергу, пропускає нуль або більше екземплярів. Його можна викоистовувати, наприклад, якщо нам потрібно зчитати декілька

пробілів підряд. На Рисунку 12 реалізовано функцію, яка парсить стрічку з пробілами, використовуючи комбінатор skipMany.

Комбінатор manyTill застосовує певну дію допоки не закінчиться успіхом. Його часто використовують для аналізу коментарів.

```
parseStringWithSpaces:: String -> Parser String
parseStringWithSpaces str = do
                            skipmany space
                            string str
                            skipmany space
                            return str
```

Рисунок 12

2.4.Реалізація

Граматика, наведена в додатку 1, підходить для аналізу зверху-вниз, а саме так працює Attoparsec. Для того, щоб розпочати процес синтаксичного аналізу, потрібно на основі граматики задати структуру програми. Так, на основі структури даних, синтаксичний аналізатор зможе побудувати дерево виводу, що і очікується в результаті. Ось, наприклад, як визначена структура виразів можна подивитись на рисунку 13.

```
data Exp = OpExp Op Exp Exp
         | ArrayLp Exp Exp
         | ArrayLength Exp
         | CallMethod Exp Id [Exp]
         | TrueExp
         | FalseExp
         | ThisExp
         | Number Exp
         | CreateObject Type
         | IdExp Id
         | NotOperator Exp
         | CreateArray Type Exp
        deriving(Show)
```

Рисунок 13

Оскільки Attoparsec працює за принципом низхідних аналізаторів, то для побудови дерева виводу ми будемо рухатись від кореня до листків. У випадку з нашою граматикою коренем, тобто початковим нетерміналом, буде сама програма (Рисунок 14).

```
data Program1 = Program Decl [Decl] deriving (Show)
```

Рисунок 15

Загалом ми визначаємо парсери для всіх структур граматики. Основна ідея полягає в тому, щоб з парсерів вищого порядку, як Program, дійти до парсерів нижчого - базових простих парсерів, як наприклад парсер операторів чи літералів. Оскільки у нас на виході повинно утворитись

дерево, то типи визначених структур взаємопов’язані. Поглянемо на реалізацію парсера, який парсить символи від одного до дев’яти (Рисунок 16).

```
parseDigits:: Parser Char
parseDigits = satisfy isDigit
            where isDigit d = d >= '1' && d <= '9'
```

Рисунок 16

Тут бачимо використання функції satisfy, яка визначає, чи даний символ збігається з предикатом, тобто чи є символ числом, буквою чи навіть конкретним словом.

Далі потрібно символи цифр перевести в літерали. Літерал - це числове представлення наших цифр. Приклад реалізації наведено на Рисунку 17.

```
digitsIntoLitterals:: Parser Integer
digitsIntoLitterals = try (do {l <- parseDigits;lt <- many digits;return (read (l:lt)::Integer})
                    <|> do {_ <- char '0';return 0} -- повертає літерал символа
```

Рисунку 17

Варто сказати про функцію many, яка досить часто використовується при синтаксичному аналізі. Вона застосовує якийсь аргумент до функції декілька разів поки не отримає помилку, а потім повертає зібрані дані до моменту помилки. Далі функція digitsIntoLitterals буде використана для реалізації парсера літеральних числових виразів Рисунку 18.

```
number:: Parser Exp
number = do
         skipMany space
         numberExp <- digitsIntoLitterals
         skipMany space
         return (Number numberExp)
```

Рисунок 18

Відповідно функція number буде використана в функції mainExp, де зібрані всі реалізовані функції для виразів нашої граматики Рисунок 19.

```
mainExp:: Parser Exp
mainExp = try(createArray)
        <|> try(createIdExp)
        <|> try(createObject)
        <|> try(thisExp)
        <|> try(falseExp)
        <|> try(trueExp)
        <|> try (callMethod)
        <|> try (arrayLength)
        <|> try (arrayLp)
        <|> try(number)
        <|> try(notOp)
        <|> getClassId
```

Рисунок 19

Також потрібно буде парсити оператори такі, як: додавання (+), віднімання (-), множення (*), порівняння (<, >), а також оператор &.

Перш ніж говорити про асоціативність, важливо визначити пріоритет кожного оператора. Пріоритет - це порядок в якому виконуватимуться оператори. У випадку декількох операторів з однаковим пріоритетом за

порядок виконання операцій відповідає асоціативність. Більшість операторів ліво асоціативні.

Корисним буде використати модуль Data.Attoparsec.Expr, який допомагає розпарсити вирази досить легким записом. Він будує парсер по заданій таблиці операторів, яка реалізована в порядку спадання пріоритетності, та їх асоціативності (ліва асоціативність, права асоціативність та без асоціативності). Наприклад, операція множення є найбільш пріоритетною та ліво асоціативною.

Модуль пропонує функцію, яка на основі таблиці операторів та термів (у нашому випадку - це парсер mainExp) парсить вирази (Рисунок 20).

```
getExp:: Parser Exp
getExp = buildExpressionParser ops mainExp
```

Рисунок 20

Отже, далі рухаючись каскадно і розбиваючи складні парсери на парсери більш нижчого порядку в результаті повинно утворитись дерево виводу, яке і є очікуваним успішним результатом роботи синтаксичного аналізатора.

2.5.Megaparsec

Megaparsec є теж досить популярною бібліотекою, яку також можна використати для побудови синтаксичного аналізатора. Megaparsec також відноситься до бібліотек парсерних комбінаторів. Бібліотека має дуже багато спільного з Attoparsec, проте його структура дещо складніша. У Megaparsec є два типи параметрів e та s, а також трансформер монад ParsecT (Рисунок 21).

```
data ParsecT e s m a

type Parsec e s = ParsecT e s Identity
```

Рисунок 21

Тип e дозволяє нам дозволяє надавати парсеру деякі дані про помилки. Тип s відповідає за вивід, наприклад String, Text, ByteString - типи, з якими по дефолту працює Megaparsec. Тип m - це внутрішня монада, а тип a - результат парсингу, тобто саме значення монади. Оскільки структура складна, то можна визначити свій тип Parser (Рисунок 22).

```
type Parser = Parsec Void Text
```

Рисунок 22

Визначивши таку структуру, надалі можна писати Parser Exp чи Parser Program1, ну і результат тоді буде повертатись відповідно до типів. Void - це компонент для помилок, Text - це тип вхідних даних.

У Megaparsec є тип даних ParseErrorBundle, який дозволяє працювати з багатьма повідомленнями про помилку та одночасно їх виводити, замість одного повідомлення.

В Megaparsec теж пропонує такі комбінатори, як takeWhile і tokens, що значно пришвидшує його роботу. Однією з переваг Megaparsec при роботі з помилками є комбінатори withRecovery та observing. WithRecovery дозволяє відновити роботу при виникненні помилки та продовжити її, а в кінці виводить всі знайдені помилки.

Загалом бібліотека Megaparsec - це поєднання швидкості та якості помилок і дуже вдале поєднання бібліотек Parsec та Attoparsec.

2.6 Parsec

Бібліотека Parsec є не менш важливою для використання при побудові синтаксичного аналізатора. Вона є основою для бібліотек Attoparsec та Megaparsec і також вважається бібліотекою парсерних комбінаторів. У ній є дуже багато модулів для роботи з Text I ByteString і не менш важливими модулями є модулі ParserCombinators. Як вже було сказано раніше, при роботі з Parsec, бібліотека пропонує дуже інформативні повідомлення про помилки.

Parsec також містить монаду Parser. Як і в Megaparsec, тут є трансформер монад ParsecT (Рисунок 23).

```
data ParsecT s u m a

type Parsec s u = ParsecT s u Identity
```

Рисунок 23

Параметр s означає тип, параметр u – це стан користувача, m – це монада, параметр a – це тип, який повертається.

3. Висновок

Отже, під час роботи над курсовою роботою було досліджено дві бібліотеки: Attoparsec та Mageparsec. Проаналізовано їхні переваги та недоліки. Кожна бібліотека призначена для своїх цілей. Проте обидві дуже добре справляються з синтаксичним аналізом. Досить легко було знайти опис потрібного комбінатора чи структури, адже у кожної бібліотеки є гарно прописана документація. Більш того, для бібліотеки Megaparsec створено гайд по вивченню.

В результаті аналізу та ознайомлення з бібліотеками було реалізовано синтаксичний аналізатор для мови MiniJava. Хаскель - є тою мовою, яка завжди залишатиметься актуальною для такого роду завдань. Адже надає широкий спектр інструментів для програміста: функціональність, монади, вбудовані функції, можливість написання чистого та зрозумілого коду.

ДОДАТОК 1

```
program  =  mainClass { classDecl }
mainClass  =  "class" id
                "{" "public" "static" "void" "main" "(" "String" "[""]" id ")"
                    "{"  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  = "(" expr ")" | number | id | "true" | "false" | "this" |
          | "new" id "(" ")" | "new" "int" "[" exp "]"
factList  = exp { "," exp}
```

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. https://hackage.haskell.org/

2. https://mmhaskell.com/

3. Erik Meijer. Advanced Functional Programming Techniques

4. https://markkarpov.com/tutorial/megaparsec.html

5. Роман Душкин. Функциональное программирование на Хаскель

6. Andrew W. Appel and Jens Palsberg. Modern Compiler Implementation in Java .

7. Bryan O'Sullivan, Don Stewart, and John Goerzen. Real World Haskell

8. Miran Lipovaca. Learn you a Haskell for great gob! A beginner’s guide

9. Graham Hutton. Programming in Haskell