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)

“____” _____ 2021 р.

Виконав студент
Чорнокозинський К. С.

“____” _____ 2021 р.

КИЇВ 2021

ЗМІСТ

АНОТАЦІЯ

У роботі розглянуто процес синтаксичного аналізу тексту. Досліджено та проаналізовано особливості різних бібліотек в мові Haskell, а також використання аналізаторів та їх комбінаторів. Результатом роботи є створення трьох програм на основі таких бібліотек: Regex Applicative, Attoparsec, Parsec.

Ключові слова: Haskell, синтаксичний аналіз, комбінатори, розбір, Parsec, Attoparsec, Applicative parsing, регулярні вирази, формальні граматики, parser combinators, термінал, нетермінал, контекстно-вільна граматика.

ВСТУП

Синтаксичний аналіз є одним з найбільш складних і актуальних напрямків в теорії комп'ютерної лінгвістики. Синтаксичні аналізатори широко застосовуються в таких областях: штучний інтелект (ШІ), створення компіляторів, проектування інтерфейсів баз даних, автоматична обробка текстів (АОТ) для автоматизованих інформаційно-пошукових систем, систем машинного перекладу (МП), перевірки правопису, складання анотації документа.

Зважаючи на стрімке зростання обсягів текстової інформації та складної структурованості природно-мовних текстів, аналіз текстів є актуальною проблемою.

Метою роботи було дослідити процес синтаксичного аналізу в функціональній мові програмування Haskell. Порівняти певні бібліотеки та їх інструментарії шляхом написання однакового додатку на кожній. Задача роботи була написати синтаксичний аналізатор для мови розмітки XML.

Об’єктами дослідження були три бібліотеки Haskell, а саме: Regex applicative parser, Parsec, Attoparsec.

РОЗДІЛ 1. СИНТАКСИЧНИЙ АНАЛІЗ

1.1 Поняття синтаксичного аналізу

Синтаксичний аналіз – це аналіз тексту з метою визначення його синтаксичної структури. В процесі синтаксичного аналізу перевіряється, чи належить вхідний текст до мови, що породжується даною граматикою. Результатом роботи синтаксичного аналізатора є Абстрактне Синтаксичне Дерево (AST).

Перш ніж розв'язувати задачу синтаксичного аналізу, необхідно описати формальну граматику мови. Є багато формальних методів, які задають синтаксичну структуру мов, такі як контекстно-вільні граматики, форми Наура-Бекуса, регулярні граматики і т.і. В цій роботі буде описано дві граматики, а саме контексно-вільна та регулярна, тому для визначення синтаксису мови буде використовуватися форма Наура-Бекуса.

В описі мови використовуються термінали – це символи алфавіту (скінченна множина символів), і нетермінали, які визначають синтаксичні конструкції мови.

Термінали позначають самі себе і записуються в лапках – *'к', 'n', '=', ...*

Нетермінали – це слова в кутових дужках: <*letter*>, <*op* >, <*digit*>.

Опис мови – це скінченна множина правил виду: *нетермінал* ::= *правило*

Правило – слово з терміналів, нетерміналів і метасимволів, яке описує як побудувати всі слова синтаксичної конструкції *нетерміналу*. Для спрощення роботи з формою БНФ використовуються такі метасимволи:

- *a1* | *a2* - вибір одного з варіантів a1 або a2.
- *ε* - позначає порожнє слово «».
- *[a1]* – конструкцію a1 в середині [] потрібно використати 0 або 1 раз.
- *{a1}* – конструкцію a1 в середині {} потрібно використати 0 або багато разів.

1.2. Огляд деяких граматик Хомського. Побудова граматик XML

Формальна граматика Хомського G – (N, T, P, S):

N – множина нетермінальних символів.

T – множина термінальних символів.

P – скінченна множина правил виводу.

S – нетермінальний символ, з якого починаємо опис граматики.

У контекстно-вільній граматиці всі правила мають вигляд: $\alpha \rightarrow \beta$, де $\beta \in (N \cup T)^*$, а $\alpha \in N$, тобто граматика допускає появу в лівій частині правила тільки нетермінального символу.

Регулярні граматики – найпростіші з формальних граматик. Вони є контекстно-вільними, але з обмеженими можливостями. Всі регулярні граматики можуть бути розділені на два еквівалентних класи, які матимуть правила такого вигляду:

- $\alpha \rightarrow \beta\mu$ або $\alpha \rightarrow \mu$, де $\mu \in (T)^*$, а β і $\alpha \in (N)$
- $\alpha \rightarrow \mu\beta$ або $\alpha \rightarrow \mu$, де $\mu \in (T)^*$, а β і $\alpha \in (N)$

The EXtensible Markup Language (розширена мова розмітки або *XML*), один із стандартів для зберігання та передачі даних. Кожен XML-документ – це текстовий рядок певної структури, що включає різні компоненти.

Наступні БНФ повністю описують контекстно-вільну граматику XML документу:

<element> ::= < openTag > < xmls > <endTag>

<openTag> ::= '<' <name> [':' namespace] <space> < attributes > '>'

<selfClosingTag> ::= '<' <name> <space> < attributes > '/' '>'

<attributes> ::= < Attribute > < Attributes > | ε

<attribute> ::= <name> <spaces> '=' <spaces> '"' <val> '"' <spaces>

<endTag>::= '<''/' <name> [':' namespace] '>'

<xmls> ::= <xml> < xmls > | ε

<xml> ::= <Element> | < bodyText >

<document> ::= <spaces> <declaration> <spaces> <element> <spaces> 'eos'

<declaration> ::= <spaces> '<' '?' 'x' 'm' 'l' {<anySym>} '?' '>'

Наступні БНФ описують регулярну граматику XML документу. Але цей опис є неповним, адже мова розмітки XML породжена контекстно-вільною граматикою. Тут визначено таку мову XML, в якій можливе вкладення елементу лише на один рівень в глубину.

<element> ::= <begTag> <xmls> <endTag>

<elementText> ::= <begTag> < bodyText > <endTag>

<openTag> ::= '<' <name> [':' namespace] <space> < attributes> '>'

<attributes> ::= < attribute > < attributes > | ε

<attribute> ::= <name> <spaces> '=' <spaces> '"' <val> '"' <spaces>

<endTag> ::= '<''/' {<anySym>} '>'

<xmls> ::= <xml> < xmls > | ε

<xml> ::= < elementText > | <bodyText>

<document> ::= <spaces> <declaration> <spaces> <element> <spaces> 'eos'

Висновки

В розділі були вказані поняття різних граматик Хомського та поверхнево розглянуто форми Наура-Бекуса, за допомогою яких було створено контекстно-вільна та регулярна граматики мови розмітки XML.

РОЗДІЛ 2. СИНТАКСИЧНИЙ АНАЛІЗ В HASKELL

2.1. Введення до синтаксичного аналізу в Haskell

Мова програмування Haskell чудово підходить для синтаксичного аналізу текстів через наявність вдосконалених типів, узгодження шаблонів («pattern matching»), функцій високого порядку та класів типів, таких як монади або функтори, які роблять мову математичною та простою для описання задач розбору.

В Haskell є декілька бібліотек, які активно використовуються для синтаксичного аналізу різних граматик. В рамках курсової роботи для побудови синтаксичного аналізатора було використано граматику мови розмітки XML.

Для реалізації синтаксичного аналізу було використано бібліотеку Regex Applicative Parser, Parsec та Attoparsec.

2.2. Синтаксичний аналіз в Applicative Parsing

Regex Applicative Parser – це бібліотека Haskell для синтаксичного аналізу за допомогою регулярних виразів.

Синтаксичний аналіз на основі класу типів Applicative, в якому основний тип даних, з яким ми будемо працювати, називається RE. Він визначає синтаксичний аналізатор параметризований двома типами.

```
data RE s a = ...
```

Лістинг 2.2.1. Визначення типу RE

Де параметр s відповідає за тип, який цей синтаксичний аналіз призначений розбирати, а параметр a за те, якого типу буде результат.

Для того щоб працювати з синтаксичним аналізом в бібіліотеці Applicative Parsing важливо оглянути базовий інструментарій, тому більш детально рогзлянуто тип RE та «parser combinators», які входять до неї.

До типу RE входять такі Instances:

Functor (RE s)

Applicative (RE s)

Alternative (RE s)

Filtrable (RE s)

Semigroup a => Semigroup (RE s a)

Monoid a => Monoid (RE s a)

Далі розглянуто класи типів Applicative та Alternative, вони є найважливішими у роботі з синтаксичними аналізаторами у цій бібліотеці.

Почнемо з функцій визначених у Applicative:

```
pure :: a -> RE s a                                                    #
(<*>) :: RE s (a -> b) -> RE s a -> RE s b                             #
liftA2 :: (a -> b -> c) -> RE s a -> RE s b -> RE s c                  #
(*>) :: RE s a -> RE s b -> RE s b                                     #
(<*) :: RE s a -> RE s b -> RE s a                                     #
```

Рис 2.2.1. Клас типів Applicative

Функція pure – додання контексту до значення (аналогічна до return в монадах).

Оператор (<*>) – конкатенація синтаксичних аналізаторів.

А ці функції є допоміжними:

(*>) :: f a -> f b -> f b

(<*) :: f a -> f b -> f a

Вони є спрощеним варіантом оператору (<*>), тобто, хоч вони і виконують дві дії, але повертають результат лише однієї. Дії виконуються зліва направо.

(*>) – повертає значення агументу справа.

(<*) – повертає значення агументу зліва.

Також важливими є функції з класу типів Alternative:

```
empty :: RE s a                                    #
(<|>) :: RE s a -> RE s a -> RE s a                #
some :: RE s a -> RE s [a]                         #
many :: RE s a -> RE s [a]                         #
```

Рис 2.2.2. Клас типів Alternative

Функція empty описує нейтральний елемент для операції альтернативи. У нашому випадку це синтаксичний аналізатор, який ніколи нічого не розбирає, тобто завжди повертає порожній список результатів.

Оператор <|> – Асоціативна бінарна операція. Цей комбінатор реалізує вибір. Синтаксичний аналізатор p <|> q спочатку застосовує p і якщо це вдається, повертається результат роботи синтаксичного аналізатору p. Якщо синтаксичному аналізатору p не вдається розібрати вхідні дані, то те ж саме буде пророблено з синтаксичним аналізатором q, але у випадку невдачі буде повернута помилка.

Також у цьому класі типів реалізовані дві функції, some і many типу f a -> f [a]. Кожна з них може бути виражена через іншу:

- some v = (:) <$> v <*> many v
- many v = some v <|> pure []

Ці функції дозволяють розбирати послідовності даних, якщо відомо як розібрати один елемент даних. Використовуючи some, послідовність повинна бути непорожньою.

2.2.1. Основні синтаксичні аналізатори

Найпростіший синтаксичний аналізатор sym, який порівнює та повертає символ, що передається агрументом s:

```
sym :: Eq s => s -> RE s s
```

Лістинг 2.2.1.1. Визначення аналізатору sym

Щоб використати RE parser, необхідно викликати функцію match або її інфіксний варіант =~. Вона поверне Just зі значенням, якщо синтаксичний аналізатор зміг зіставити вхідний текст з регулярним виразом і повертає Nothing при помилці.

```
match :: RE s a -> [s] -> Maybe a
```

Лістинг 2.2.1.2. Визначення функції match

Приклади:

```
ghci> match parseFour "45"
Nothing

ghci> match parseFour "4"
Just '4'

ghci> "4" =~ parseFour
Just '4'
```

Рис. 2.2.1.1. Приклади використання функції match

Про sym вже було сказано вище, тому перейдемо до близького до нього psym, аргументом якого є предикат.

```
psym :: (s -> Bool) -> RE s s

-- приклад
ghci> "4" =~ psym ('4' ==)
Just '4'

ghci> "5" =~ psym ('4' ==)
Nothing
```

Рис. 2.2.1.2 Типова анотація та приклади використання аналізатору psym

Наступний аналізатор – anySym. Він читає будь-який символ і повертає його.

```
anySym :: RE s s

-- приклад
ghci> "5" =~ anySym
Just '5'

ghci> "" =~ anySym
Nothing
```

Рис. 2.2.1.3 Типова анотація та приклади використання аналізатору anySym

Синтаксичний аналізатор string приймає аргументом стрічку, з якою буде зіставляти вхідний текст.

```
string :: Eq a => [a] -> RE a [a]

-- приклад
ghci> "abcd4" =~ string "abcd"
Nothing

ghci> "abcd" =~ string "abcd"
Just "abcd"
```

Рис. 2.2.1.4 Типова анотація та приклади використання аналізатору String

2.2.2. Написання складних синтаксичних аналізаторів

Знизу приведений приклад синтаксичного аналізатору від'ємних та додатніх чисел.

```
number :: RE Char Int
number = read <$> (numb <|> neg)
  where
    numb = some (psym isDigit)
    neg = (:) <$> sym '-' <*> numb
```

Лістинг 2.2.2.1. Визначення функції number

Спочатку, для з'ясування чи є вхідні дані числом використано psym та предикат IsDigit з модуля Data.Char. Потім використано комбінатор some, який розпізнає одну або більше одиниць тексту, що може зіставити синтаксичний аналізатор. Тому numb розбирає додатнє число.

Тепер дещо складніший аналізатор neg. Спочатку виконується розбір аналізатором sym '-', і застосовується функція функтору <$>, яка повертає функцію в контексті.

```
(:) <$> sym '-' :: RE Char ([Char] -> [Char])
```

Лістинг 2.2.2.2. Пояснення number 1

Потім застосовується функція аплікативного функтору <*>.

```
((:) <$> sym '-' <*>) :: RE Char [Char] -> RE Char [Char]
```

Лістинг 2.2.2.3. Пояснення number 2

В результуючу функцію передається аргументом синтаксичний аналізатор numb. Остання дія – зчитування (виймання з контексту) результату альтернативної комбінації синтаксичних аналізаторів numb і neg.

```
number = read <$> (numb <|> neg)
```

Лістинг 2.2.2.4. Пояснення number 3

2.2.3. Робота з помилками

Бібліотека Regex Applicative Parsing дозволяє здійснювати синтаксичний аналіз регулярних мов, тому відсутні засоби роботи з помилками. Це ускладнює здійснення аналізу. Результат повертається в контексті мона Maybe: у випадку успішного розбору результат повертається у конструкторі Just, а у випадку помилки повертається конструктор Nothing.

2.2.4. Розробка лексичного аналізатору XML

Тепер до розробки. Передусім необхідно імпортувати потрібні бібліотеки.

```
import Text.Regex.Applicative
import Data.Char
```

Лістинг 2.2.4.1. Імпортування модулів

На наступному лістингу імплементовано регулярну граматику XML, яку було описано в попередньому розділі.

```
type AttrName = String
type AttrVal  = String
type Namespace = String
data Attribute = Attribute AttrName Namespace AttrVal deriving (Show)

data XML =  Element String Namespace [Attribute] [XML]
          | Body String
          | Decl String
        deriving (Show)
```

Лістинг 2.2.4.2. Визначення типів

Розробку було почато з написання лексичного аналізатору коментарів.

```
comment :: RE Char [Char]
comment = string "<!--" *> (many anySym) <* string "-->"
```

Лістинг 2.2.4.3. Визначення функції comment

В цій функції використано два базових синтаксичних аналізатори string та anySym. Вони виконуються в тій черзі, в якій вони і написані, але повертається лише результат (many anySym), через те, що були використані спрощені оператори аплікативного функтору.

Наступним кроком імплементовано допоміжну функцію, яка буде приймати синтаксичний аналізатор аргументом і пропускати усі коментарі та пробіли після та перед розбору даних даним аналізатором, та повертати його результат.

```
lexeme :: RE Char a -> RE Char a
lexeme re = spaces *> many comment *> spaces *> re
                   <* spaces <* many comment <* spaces
```

Лістинг 2.2.4.4. Визначення функції lexeme

Наступний синтаксичний аналізатор розбирає декларацію XML файлу.

```
xmlDecl :: RE Char XML
xmlDecl = Decl <$> delcParser
  where delcParser =  string "<?xml" *> many (anySym) <* string "?>"
```

Лістинг 2.2.4.5. Визначення функції xmlDecl

Написання аналізатору аналогічне до функціїї comment, але тепер результат розбору буде типу XML. За допомогою оператору <$> результат аналізатору огортається у конструктор Decl типу XML.

Наступним кроком визначимо синтаксичний аналізатор, який розпізнає атрибут.

```
attribute :: RE Char Attribute
attribute = Attribute <$> name <*> nmspace <*> value
  where
    name = spaces *> many (psym (\x -> (x /= ':') && (x /= ' ')
                                          && (x /= '=')
                                          && (x /= '/')
                                          && (x /= '>')))
    value = (many $ psym (/= '"') ) <* sym '"' <* spaces
    nmspace = lexeme $ (namespace <|> spaces) <* spaces
                                          <* (lexeme $ sym '=') <* sym '"'
```

Лістинг 2.2.4.6. Визначення функції attribute

В блоці where визначено ще три комбінатори:

- name – пропускає пробіли та розбирає послідовність символів, що задовільняють предикат:
  \x -> (x /= ':') && (x /= ' ') && (x /= '=') && (x /= '/') && (x /= '>').
- value – розбирає послідовність символів, що задовільняють предикату (/= "") та повертає результат. Після розбору пропускаються символ " та усі пропуски.
- nmspace – синтаксичний аналізатор
  lexeme $ (namespace <|> spaces) використовує namespace (продемонстровано нижче), якщо вхідний текст можливо зіставити. Інакше буде повернута пуста стрічка. Також до і після розбору пропускаються пробіли. Після цього пропускаються символи " і =.

  ```
  namespace :: RE Char [Char]
  namespace = sym ':' *> many (psym (\x -> (x /= ' ') && (x /= '=')
                                                      && (x /= '/')
                                                      && (x /= '>')))
  ```

Лістинг 2.2.4.7. Визначення функції namespace

Фінальний крок у визначенні функції – групування всіх значень у конструкторі Attribute за допомогою операторів <$>, <*>.

Далі визначено один з головних синтаксичних аналізаторів, який розпізнає елемент XML файлу. Ця функція має параметр типу синтаксичного аналізатору, результатом якого є XML.

Знову ж таки визначаємо допоміжні функції у блоці where.

```
element :: RE Char XML -> RE Char XML
element bodyParser = Element <$> name <*> nmspace <*> attrs
                             <*> (spaces *> many comment *> spaces
                              *> (lexeme $ many $ bodyParser) <* endTag)
  where
    name = lexeme $ sym '<' *> (lexeme $ identifier)
    nmspace = lexeme (namespace <|> spaces)
    attrs = (many attribute) <* sym '>'
    endTag = string "</" *> (many $ psym ( /= '>')) <* sym '>'
```

Лістинг 2.2.4.8. Визначення функції element

Перша name, вона повертає синтаксичний аналізатор, що зіставляє вхідний текст з ідентифікатором, пропускаючи пробіли до і після розбору.

nmspace працює аналогічно до функції, що була визначена у синтаксичному аналізаторі attribute, за винятком, що не пропускаються ніякі символи, окрім пробілів.

attrs – синтаксичний аналізатор, що поверне результат розбору вхідного тексту синтаксичним аналізатором attribute комбінованим комбінатором many, пропускаючи символ > після розбору.

Функція endTag зіставляє вхідний текст зі стрічкою яка починається на "</" і закінчується '>' .

У тілі функції element ми почергово викликаємо кожний синтаксичний аналізатор.

Завершаємо розробку синтаксичного аналізатора функцією document.

```
document :: RE Char [XML]
document = (:) <$> decl <*> xmls
  where
    decl = lexeme $ (xmlDecl <|>  Decl <$> spaces)
    xml = element ((element body) <|> body)
    xmls = (:) <$> xml <*> many empty
```

Лістинг 2.2.4.9. Визначення функції document

Синтаксичний аналізатор document повертає список типу XML, як результат, а саме список з двох елементів: декларації та корневого елементу.

Додамо допоміжні функції для зручності використання зробленого нами синтаксичного аналізатору.

```
docFromFile filepath = do
  xml <- readFile filepath
  return (docFromString xml)

docFromString xml = match (document) xml
```

Лістинг 2.2.4.10. Визначення допоміжних функцій

Протестуємо синтаксичний аналіз, який написаний за допомогою бібліотеки Regex Applicative Parser, на файлі наведеному нижче.

```
<h:table xmlns:h="http://www.w3.org/TR/html4/">

    <h1:td>textBody1</h1:td>

    <h2:td>textBody1</h2:td>

    samleText

</h:table>
```

Рис. 2.2.4.1. Приклад XML

Результат розбору:

```
ghci> docFromFile "regex-xml.xml"
Just [Decl "",Element "h" "table" [Attribute "xmlns" "h" "http://www.w3.org/TR/html4
/"] [Element "h1" "td" [] [Body "textBody1"],Element "h2" "td" [] [Body "textBody1"]
,Body "samleText"]]
```

Рис. 2.2.4.2. Результат розбору XML

2.3. Синтаксичний аналіз в Parsec

Parsec – це бібліотека в Haskell, що використовується для побудови синтаксичних аналізаторів. Процес створення власного синтаксичного аналізатору складається з комбінування більш простих аналізаторів. Цей інструмент називається комбінатором аналізаторів (parser combinators).

Модуль Text.Parsec визначає комбінатори, використовуючи які ми можемо поєднувати синтаксичні аналізатори та визначення типу даних, що є результатом виконання кожного аналізатора бібліотеки.

```
ParsecT s u m a
```

Параметр s – це тип даних, що приймає синтаксичний аналізатор.

Параметр a – це тип даних, який повертає синтаксичний аналізатор.

Параметр u – це тип даних, що бібліотека передає між аналізаторами.

m – монада (контекст), яка буде застосована до результату.

2.3.1. Огляд простих синтаксичних аналізаторів

Доречно розглянути елементарні синтаксічні аналізатори з модуля Text.Parsec.Char, адже на них базуються майже усі комбіновані синтаксичні аналізатори.

Аналізатор oneOf cs досягає успіху, якщо поточний символ є у наданому списку символів cs. Повертає проаналізований символ.

```
oneOf :: Stream s m Char => [Char] -> ParsecT s u m Char
```

Лістинг 2.3.1.1. Типова анотація аналізатору oneOf

Аналізатор noneOf cs досягає успіху, якщо поточного символ немає у наданому списку символів cs. Повертає проаналізований символ.

```
noneOf :: Stream s m Char => [Char] -> ParsecT s u m Char
```

Лістинг 2.3.1.2. Типова анотація аналізатору noneOf

Аналізатор spaces можно уявити як many space, тому він досягає успіху, якщо стрічка на вході пуста або містить пробіли. Повертає проаналізовані символи.

```
spaces :: Stream s m Char => ParsecT s u m ()
```

Лістинг 2.3.1.3. Типова анотація аналізатору spaces

Аналізатор char c досягає успіху, якщо поточний символ дорівнює наданому символу c. Повертає проаналізований символ (аналогічно до sym з Regex Applicative) .

```
char :: Stream s m Char => Char -> ParsecT s u m Char
```

Лістинг 2.3.1.4. Типова анотація аналізатору char

Фундаментальний Аналізатор satisfy p приймає аргументом предикат типу (Char -> Bool). За допомогою нього реалізовані майже всі інші синтаксичні аналізатори. Досягає успіху у випадку якщо поточний символ задовільняє предикату. Повертає проаналізрований символ (аналогічно до psym з Regex Applicative) .

```
satisfy :: Stream s m Char => (Char -> Bool) -> ParsecT s u m Char
```

Лістинг 2.3.1.5. Типова анотація аналізатору satisfy

Аналізатор string cs досягає успіху, якщо вхідний текст дорівнює послідовності символів cs. Повертає проаналізовану стрічку (аналогічно до string з Regex Applicative) .

```
string :: Stream s m Char => String -> ParsecT s u m String
```

Лістинг 2.3.1.6. Типова анотація аналізатору string

2.3.2. Огляд основних комбінаторів

Також доречно розглянути основні комбінатори з модуля Text.Parsec.Combinator, адже вони необхідні для створення комбінацій синтаксичних аналізаторів.

Комбінатор <|> ми зустрічали раніше в бібліотеці Regex Applicative. Насправді, тут він робить абсолютно те ж саме: намагається застосувати синтаксичний аналізатор зліва від оператору і якщо це йому вдається – він повертає результат аналізу, а якщо ні – він намагається застосувати аналізатор справа від оператору. Якщо це в свою чергу вдається, то повертається результат розбору, а інакше – помилка.

```
(<|>) :: ParsecT s u m a -> ParsecT s u m a -> ParsecT s u m a
```

Лістинг 2.3.2.1. Типова анотація комбінатору <|>

Комбінатор p <?> msg поводиться як аналізатор p, але щоразу, коли синтаксичному аналізатору p не вдається розібрати вхідні дані, не прочитавши нічого, він замінює повідомлення про очікувані помилки на msg. Також його називають label.

```
(<?>) :: ParsecT s u m a -> String -> ParsecT s u m a
```

Лістинг 2.3.2.2. Типова анотація комбінатору <?>

Комбінатор try p поводиться як аналізатор p, за винятком того, що він робить відкат при виникненні помилки.

```
try :: ParsecT s u m a -> ParsecT s u m a
```

Лістинг 2.3.2.3. Типова анотація комбінатору try

Комбінатор many дозволяє розбирати послідовності даних, якщо відомо, як розібрати один елемент даних. Вхідні дані можуть бути порожніми (аналогічно до many з Regex Applicative) .

```
many :: ParsecT s u m a -> ParsecT s u m [a]
```

Лістинг 2.3.2.4. Типова анотація комбінатору many

Комбінатор choice дуже схожий до оператору <|>, але коли <|> приймає лише два синтаксичних аналізатори, то комбінатор choice приймає список аналізаторів і обирає перший, який підійшов для розбору вхідних даних.

```
choice :: Stream s m t => [ParsecT s u m a] -> ParsecT s u m a
```

Лістинг 2.3.2.5. Типова анотація комбінатору choice

2.3.3. Обробка помилок

При виникненні помилки при роботі програми, Parsec поверне повідомлення про помилку у конструкторі Left типу класів Either, що буде містити номер рядка та стовпчика, де саме ця помилка сталася, та її причину.

Нижче наведений приклад:

```
ghci> parse (string "example") "" "exampple"

Left (line 1, column 1):
unexpected "p"
expecting "example"
```

Рис. 2.3.3.1. Приклад помилки

2.3.4. Розробка лексичного аналізатору XML

Тепер, коли оглянуто базовий інструментарій, можна приступити до розробки.

Спочатку імпортовано потрібні бібліотеки.

```
import Control.Applicative
import Control.Monad
import Text.Parsec
import Text.Parsec.String
```

Лістинг 2.3.4.1. Імпорт бібліотек

Наступним кроком перенесено формальну граматику, що описана вище, у тип даних в Haskell.

Haskell має особливо потужну систему типів. Тут створено «синоніми типів» для імені та значення атрибуту. Потім використано data constructor для

основних типів: XML та атрибут. Тип Attribute визначається єдиним конструктором Attribute з такими параметрами:

- Ім'я атрибуту.
- Простір імен.
- Значення атрибуту.

А тип Haskell визначається чотирма конструкторами:

- Element, що визначається чотирма параметрами: ім’я, простір імен, список атрибутів, та тіла, що може бути текстом або іншим вкладеним елементом (тип рекурсивно визначає себе). Описує приблизно таку конструкцію:

  ```
  <tag id="foo"> body-text </tag>
  ```

- SelfClosingTag, який містить три параметри: ім’я, простір імен, список атрибутів. Описує, наприклад, таку конструкцію:

  ```
  <tag id="foo" />
  ```

- Decl, який містить лише одне значення типу String – значення декларації XML файлу на його початку. Описує таку конструкцію:

  ```
  <?xml version="1.0" encoding="UTF-8"?>
  ```

- Body, що містить одне значення типу String, яке знаходиться між відкриваючим і закриваючим тегом і не є вкладеним елементом.

Повне визначення типів можна побачити в наступному лістингу:

```
type AttrName = String
type AttrVal  = String
type Namespace = String
data Attribute = Attribute AttrName Namespace AttrVal deriving (Show)

data XML =  Element String Namespace [Attribute] [XML]
          | SelfClosingTag String Namespace [Attribute]
          | Decl String
          | Body String
        deriving (Show)
```

Лістинг 2.3.4.2. Визначення типів

Далі створено синтаксичний аналізатор, що буде робити розбір коментарів.

```
comment :: Parser ()
comment = do
  void $ (try (string "<!--"))
  void $ manyTill anyChar (try $ string "-->")
```

Лістинг 2.3.4.3. Визначення comment

Спочатку синтаксичний аналізатор намагається знайти набір символів початку коментаря «<!--» і якщо знаходить його, то пропускає усі символи до кінця коментаря, що позначається послідовністю «-->».

Наступна функція є допоміжною з метою уникнути постійне дублювання коду.

```
lexeme :: Parser a -> Parser a
lexeme p = p <* spaces <* (many comment) <* spaces
```

Лістинг 2.3.4.4. Визначення lexeme

Функція lexeme приймає параметр типу Parser та повертає певну комбінацію синтаксичних аналізаторів.

Для того, щоб після виклику кожного аналізатору не визивати щоразу функцію spaces та функцію comment використано комбінатор <* з класу типів Applicative, що повертає значення аргументу зліва, виконавши спочатку операцію зліва, а потім справа.

Тобто послідовність виконання тут така :

1. Виконується аналізатор p.
2. Виконується аналізатор spaces.
3. Виконується комбінація аналізаторів many comment.
4. Виконується аналізатор spaces.
5. Повертається результат аналізатору p.

Наступним етапом буде описання кожного аналізатору елементу (продукції формальної граматики). Почнемо з верхньої частини, а саме з XML declaration.

```
xmlDecl ::Parser XML
xmlDecl = do
  void $ string "<?xml"
  decl <- many (noneOf "?>")
  void $ string "?>"
  return (Decl decl)
```

Лістинг 2.3.4.5. Визначення xmlDecl

До цього моменту не було описано функцію void, яка використовується майже в кожному визначенні аналізатору.

```
void :: Functor f => f a -> f ()        -- Defined in `Data.Functor'
```

Ця функція використовується для ігнорування результату синтаксичного аналізатору символів, якщо це значення неважливе. Ви також можете написати функцію без void, але ghc дасть вам попередження, якщо у вони у вас увімкнені.

Наступний крок – синтаксичний розбір елементу такого типу:

```
<f:table>
  <f:name>African Coffee Table</f:name>
  <f:width>80</f:width>
  <f:length>120</f:length>
</f:table>
```

Рис. 2.3.4.1. Приклад XML елементу

В цьому випадку дуже зручно використовувати do-нотацію, яка є, фактично, альтернативним способом запису монад.

```
element::Parser XML
tag = do
  void $ lexeme $ char '<'
  name <- lexeme $ identifier

  nmspace <- lexeme $ option "" namespace
  attr <- lexeme $ many attribute
  close <- lexeme $ try (string "/>" <|> string ">")
  if (length close) == 2 then return (SelfClosingTag name nmspace attr)
                  else do
                         elementBody <- many elementBody
                         void $ lexeme $ endTag (if nmspace == "" then  name
                                                 else name ++ ":" ++ nmspace)
                         return (Element name nmspace attr elementBody)
```

Лістинг 2.3.4.6. Визначення element

Хоча Haskell декларативна мова програмування, але код усередині do-нотації можно читати імперативно, що дуже спрощує нашу задачу. Спершу ми зіставляємо символ «<» з вхідним символом та відкидаємо результат синтаксичного аналізу. Наступним кроком вхідні символи розбираються синтаксичним аналізатором, що описує ідентифікатор, пропускаючи усі пробіли. Результат аналізу записується в уявну змінну name. Аналогічно розбираємо наступний вхідний текст, отримуючи значення простору імен, атрибутів та те, якими символами закривається відкриваючий тег. У випадку, якщо довжина символів, що закривають тег дорівнює двом, стає очевидно, що тег самозакриваючий. Тоді ми повинні повернути XML з конструктором SelfClosingTag, перед тим завернутим у монаду Parsec.

І якщо відкриваючий тег закрився лише одним символом, то ми повинні ще вичленити тіло (паралельно записати його у змінну elementBody) елементу та закриваючий тег перед тим звіривши його ім’я з ім’ям відкриваючого тегу. У результаті повертаємо конструктор типу XML у контексті монади Parsec.

В функції element використано синтаксичний аналізатор elementBody, що повертає результат розбору тіла елементу. Його визначення можете побачити нижче.

```
-- парсинг тіла елементу
elementBody = spaces *> try element <|> text
```

Лістинг 2.3.4.7. Визначення elementBody

Процес розбору виглядає так: спочатку використовуємо синтаксичний аналізатор spaces і відкидаємо його результати. Потім намагаємося зробити синтаксичний аналіз вхідних даних за допомогою element'а (описаний вище) або text'а (описаний нижче). У випадку успішного розбору повертаємо значення, що дає один з синтаксичних аналізаторів, а інакше – помилку і робимо відкат усіх символів, що вони поглинули.

Функція endTag приймає стрічку (ім'я атрибуту чи ім'я атрибуту:простір імен) і повертає синтаксичний аналізатор, який розбирає закриваючий тег з певним іменем та простором імен.

```
-- парсинг закриваючої частини тегу
endTag str = string "</" *> string str <* char '>'
```

Лістинг 2.3.4.8. Визначення endTag

Функція text повертає синтаксичний аналізатор, який розбирає текст тіла елементу і повертає результат у конструкторі Body.

```
-- парсинг тексту в тілі тегу
text = Body <$> many1 (noneOf "><")

-- парсинг неймспейсу
namespace = do
  void $ char ':'
  name <- lexeme $ many1 $ noneOf " = />"
  return name

-- парсинг атрибуту
attribute = do
  name <- lexeme $ many $ noneOf ": = />"
  nmspace <- lexeme $ option "" namespace
  void $ lexeme $ char '='
  void $ lexeme $ char '"'
  value <- lexeme $ many $ noneOf ['"']
  void $ lexeme $ char '"'
  return (Attribute (name, nmspace, value))
```

Лістинг 2.3.4.9. Визначення endTag, namespace, attribute

Передостанній крок – це зібрати усе в одному синтаксичному аналізаторі.

```
document :: Parser [XML]
document = do
  y <- (lexeme $ try xmlDecl <|> element)
  x <- lexeme $ many element
  return (y : x)
```

Лістинг 2.3.4.10. Визначення document

Спочатку проводиться синтаксичний аналіз декларації XML, якщо така є. Потім розбираються усі елементі. Результатом буде масив XML типу. Останнім кроком напишемо функції для спрощення роботи з парсером.

```
docFromFile filepath = do
  xml <- readFile filepath
  return (docFromString xml)

docFromString xml = parse document "" xml
```

Лістинг 2.3.4.11. Визначення допоміжних функцій

Програма закінчена.

Тепер спробуємо протестувати синтаксичний аналіз, який ми написали за допомогою бібліотеки Parsec, на файлі наведеному нижче.

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
<h:table xmlns:h="http://www.w3.org/TR/html4/">
  <h:tr>
    <h:td>Apples</h:td>
    <h:td>Bananas</h:td>
  </h:tr>
</h:table>

<f:table xmlns:f="https://www.w3schools.com/furniture">
  <f:name>African Coffee Table</f:name>
  <f:width>80</f:width>
  <f:length>120</f:length>
</f:table>
</root>
```

Рис. 2.3.4.2. Приклад файлу XML

Результат розбору файлу зазначено нижче:

```
ghci> docFromFile "xml.xml"
Right [Decl " version=\"1.0\" encoding=\"UTF-8\"",Element "root" "" [] [Element "
h" "table" [Attribute "xmlns" "h" "http://www.w3.org/TR/html4/"] [Element "h" "tr
" [] [Element "h" "td" [] [Body "Apples"],Element "h" "td" [] [Body "Bananas"]]],
Element "f" "table" [Attribute "xmlns" "f" "https://www.w3schools.com/furniture"]
 [Element "f" "name" [] [Body "African Coffee Table"],Element "f" "width" [] [Bod
y "80"],Element "f" "length" [] [Body "120"]]]]
```

Рис. 2.3.4.3. Результат розбору файлу XML

2.4. Синтаксичний аналіз в Attoparsec

Attoparsec – це бібліотека комбінаторів аналізаторів типу Parsec. Вона дуже схожа на бібліотеку Parsec, але має дещо простіший API (Application Programming Interface). Attoparsec націлений працювати з типами Char8, ByteString і Text. В цьому розділі демонструється розробка синтаксичного аналізатору за допомогою модуля Data.Attoparsec.Text.

Немає сенсу розбирати цю бібліотеку так глибоко, як дві попередні, оскільки, як було сказано, вона дуже схожа на Parsec. Тому тут будуть вказані лише певні відмінності.

Перша важлива відмінність, яка дуже спрощує роботу – відсутність комбінатору try, він приховано працює при виклику кожного з синтаксичних аналізаторів.

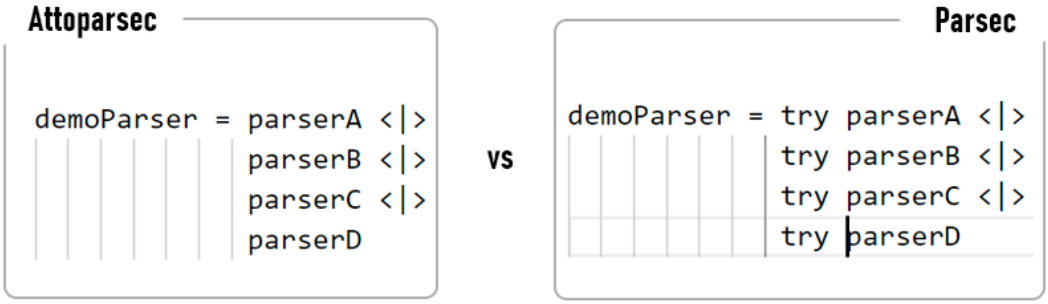

Рис. 2.4.1. Порівняння Parsec`у і Attoparsec`у

Інша відмінність в тому, що введення вхідних даних може відбуватися поступово. Ця особливість дає можливість ефективніше керувати процесом.

У випадку, якщо синтаксичний аналізатор обробив усі дані до кінця, – він поверне конструктор Partial параметризованого типу IResult і r. Якщо користувач хоче закінчити синтаксичний аналіз, то може передати пусту стрічку або mempty (нейтральний елемент) з модулю Text.

```
ghci> A.parse (A.many' A.letter) (T.pack"dDA")
Partial _
```

Рис. 2.4.2. Демонстрація роботи parse

Щоб доповнити дані, необхідно використати функцію feed. Вона приймає два параметри: поточний результат розбору та стрічку типу Text.

```
feed :: IResult i r -> i -> IResult i r
```

Лістинг 2.4.1. Типова анотація функції feed

Приклад використання:

```
feedDemo str1 str2 = A.feed (A.parse (A.many' A.letter) (T.pack str1)) (T.pack str2)

//тестування
ghci> feedDemo "let" "t3rs"
Done "3rs" "lett"
```

Рис. 2.4.3. Демонстрація роботи feed

Доречно розглянути тип, що пропонує бібліотека для обертання результату.

```
data IResult i r = Fail i [String] String
                 | Partial (i -> IResult i r)
                 | Done i r
```

Лістинг 2.4.2. Визначення типу IResult

Де параметр i – ще не оброблений текст, а r – результат вдалого розбору.

Демонстрація виводу:

```
//Результат вдачно аналізу
ghci> A.parse (A.many' A.letter) (T.pack".dDA")
Done ".dDA" ""
//Результат аналізу з помилкою
ghci> A.parse ( A.letter) (T.pack".dDA")
Fail ".dDA" ["letter"] "Failed reading: satisfy"
```

Рис. 2.4.4. Демонстрація конструкторів типу IResult

Є ще інша функція синтаксичного аналізу parseOnly. Вона працює аналогічно до parse у бібліотеці Parsec, тобто не змушує аналізатор проходити увесь рядок. Результат повертає у вигляді монади Either.

```
parseOnly :: Parser a -> Text -> Either String a
```

Лістинг 2.4.3. Типова анотація функції parseOnly

Приклад використання:

```
parseOnlyDemo str = A.parseOnly (A.many1 A.letter) (T.pack str)

//тестування
ghci> parseOnlyDemo "d3asdsa"
Right "d"
```

Рис. 2.4.5. Демонстрація роботи feed

2.4.1. Обробка помилок

Помилки в Attoparsec є менш інформативними ніж в Parsec. В залежності від того, яка функція використовується: parse чи parseOnly, виводи помилок будуть різними.

Використовуючи parse, помилка будет повертатися у конструкторі Fail, де буде вказана частина вхідного тексту, яка не була прочитана, список контекстів, в яких відбулась помилка, та текст помилки, якщо такий є.

```
A.parse (A.letter <* A.letter)  (T.pack "a3bc")

Fail "3bc" ["letter"] "Failed reading: satisfy"
```

Рис. 2.4.1.1. Помилка при роботі з parse

А використовуючи parseOnly, помилка буде повертатися у конструкторі Left типу Either, де буде вказаний синтаксичний аналізатор, при розборі якого виникла помилка та текст помилки.

```
A.parseOnly (A.many1 A.letter) (T.pack "3we")

Left "letter: Failed reading: satisfy"
```

Рис. 2.4.1.2. Помилка при роботі з parseOnly

2.4.2. Розробка лексичного аналізатору XML

Розробка проводилася аналогічно до застосунку написаному на Parsec, лише з певними редагуваннями.

Спочатку треба імпортувати потрібні бібліотеки.

```
import qualified Data.Attoparsec.Text as A
import qualified Data.Text as T
```

Лістинг 2.4.2.1. Імпорт бібліотек

Представлення граматики залишилось таким самим як у прикладі з Parsec.

Наступним кроком потрібно було прибрати усі комбінатори try, адже вони вже приховано реалізовані.

Комбінатор many змінив назву на many’, тому усюди треба було його замінити. І останній крок – переписати допоміжну функцію для роботи з користувачем.

```
docFromString :: String -> Either String [XML]
docFromString xml = A.parseOnly document (T.pack xml)
```

Лістинг 2.4.2.2. Визначення допоміжної функції

Тепер, перед тим як проводити синтаксичний аналіз, потрібно конвертувати стрічку до типу Text функцією pack з модуля Data.Text.

Перевіряємо додаток на тому самому файлі XML.

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
<h:table xmlns:h="http://www.w3.org/TR/html4/">
  <h:tr>
    <h:td>Apples</h:td>
    <h:td>Bananas</h:td>
  </h:tr>
</h:table>

<f:table xmlns:f="https://www.w3schools.com/furniture">
  <f:name>African Coffee Table</f:name>
  <f:width>80</f:width>
  <f:length>120</f:length>
</f:table>
</root>
```

Рис. 2.4.2.1. Файл XML

Результат:

```
ghci> docFromFile "xml.xml"
Right [Decl " version=\"1.0\" encoding=\"UTF-8\"",Element "root" "" [] [Element "h" "table" [Attribute "xmlns" "h" "h
ttp://www.w3.org/TR/html4/"] [Element "h" "tr" [] [Element "h" "td" [] [Body "Apples"],Element "h" "td" [] [Body "Ban
anas"]]],Element "f" "table" [Attribute "xmlns" "f" "https://www.w3schools.com/furniture"] [Element "f" "name" [] [Bo
dy "African Coffee Table"],Element "f" "width" [] [Body "80"],Element "f" "length" [] [Body "120"]]]]
```

Рис. 2.4.2.2. Результат розбору файлу XML

ВИСНОВКИ

Метою роботи було дослідити процес синтаксичного аналізу у мові програмування Haskell. Було оглянуто три бібліотеки: Regex applicative parser, Parsec, Attoparsec.

Бібліотека Regex applicative parser пропонує лише аплікативні парсери, якими не можна керувати як монадами, що досить сильно ускладнює розробку. Вона досить добре проводить синтаксичний аналіз регулярних мов, але опрацювати складніші граматики неможливо. Також великим недоліком цієї бібліотеки є дуже низька інформативність помилок.

Наступною була розглянута бібліотека Parsec. Вона може працювати з будь-якими граматиками, на відміну від бібліотеки Regex applicative parser. Одна з головних переваг – це повідомлення помилок: воно найдетальніше серед усіх оглянутих бібліотек у цій роботі. Також важливим аспектом є те, що усі синтаксичні аналізатори у Parsec'і є монадами, що дуже спрощує розробку, наприклад, є можливість використовувати do – нотацію.

Останньою з трьох бібліотек була Attoparsec, яка базується на Parsec`і . Вона пропонує швидкий аналіз для типів Text, Char8 та ByteString. Вона найкраще підходить для аналізу даних, з якими люди не повинні взаємодіяти (наприклад, JSON, бінарні протоколи тощо). Attoparsec не є найкращим вибором для аналізу мови програмування, оскільки він навіть не повідомляє вам місць помилок, коли вони трапляються. Головною перевагою є продуктивність: вона може бути вищою за Parsec до десяти раз. Також має вбудований бектрекінг, що означає, що вам не потрібно використовувати комбінатор try. Хоча ця особливість досить зручна при написанні коду, але може бути ситуація , коли потрібно відключити бектрекінг, але в цій бібліотеці такої можливості немає.

Отже, бібліотека Regex applicative parser дуже обмежена і не є монадною, тому причин її використовувати немає у відмінності від потужних Parsec та Attoparsec.

Attoparsec рекомендовано використовувати для роботи з бінарними даними та у випадках, коли ви маєте справу з обмеженнями у часі або великими обсягами даних, які зазвичай не пишуться безпосередньо людиною.

Parsec оптимально підходить для роботи з мовами програмування, файлами конфігурації та введеннями користувача. Такі дані зазвичай пишуться людиною, тому аналізаторам не потрібно масштабуватися, але вони повинні вміти добре повідомляти про помилки.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Hackage: The Haskell Package Repository [Електронний ресурс] – Режим доступу до ресурсу: https://hackage.haskell.org/.
2. MONDAY MORNING HASKELL [Електронний ресурс] – Режим доступу до ресурсу: https://mmhaskell.com/.
3. Intro to Parsing with Parsec in Haskell [Електронний ресурс] – Режим доступу до ресурсу: https://jakewheat.github.io/intro_to_parsing/#getting-started.
4. Making a naïve XML parser [Електронний ресурс] – Режим доступу до ресурсу: https://charlieharvey.org.uk/page/naive_xml_parser_with_haskell_parsec_and_perl_regexen_part_one_haskell.
5. Аплікативні парсери на Haskell [Електронний ресурс] – Режим доступу до ресурсу: https://habr.com/ru/post/436234/.
6. Attoparsec Tutorial Part 1 [Електронний ресурс] – Режим доступу до ресурсу: http://www.mchaver.com/posts/2016-05-09-attoparsec-tutorial-1.html.
7. The Haskell Programming Language [Електронний ресурс] – Режим доступу до ресурсу: https://wiki.haskell.org/Haskell.
8. Applicative Regular Expressions using the Free Alternative [Електронний ресурс] – Режим доступу до ресурсу: https://blog.jle.im/entry/free-alternative-regexp.html.
9. Regular expression [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Regular_expression#Formal_language_theory.
10. Functors, Applicative Functors and Monoids [Електронний ресурс] – Режим доступу до ресурсу: http://learnyouahaskell.com/functors-applicative-functors-and-monoids#applicative-functors.

11. Syntax Analysis: Compiler Top Down & Bottom Up Parsing Types [Електронний ресурс] – Режим доступу до ресурсу: https://www.guru99.com/syntax-analysis-parsing-types.html#:~:text=Syntax%20Analysis%20is%20a%20second,the%20programming%20language%20or%20not.
12. Контекстно-вільні граматики, висновок, ліво- і правобічний висновок, дерево розбору [Електронний ресурс] – Режим доступу до ресурсу: https://cutt.ly/SbZZQpW
13. Parsing [Електронний ресурс] – Режим доступу до ресурсу: https://guide.aelve.com/haskell/parsing-lnwybqv9.