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

Кафедра інформатики факультету інформатики

СИНТАКСИЧНИЙ АНАЛІЗ З ВИКОРИСТАННЯМ БІБЛІОТЕКИ PARSEC

Текстова частина до курсової роботи
за спеціальністю «Прикладна математика» 113

Керівник курсової роботи
к.ф-м.н., асп. Проценко В.С.
(прізвище та ініціали)
_____
(підпис)
“_—_” _____ 2020 р.

Виконав студент
3-го року навчання
спеціальності «Прикладна математика»
Бікчентаєв М.О.
(прізвище та ініціали)
“_—_” _____ 2020 р.

Київ 2020

Тема: Синтаксичний аналіз з використання бібліотеки Parsec
Календарний план виконання роботи:

№ п/п	Назва етапу курсової	Термін виконання етапу	Примітка
1.	Отримання теми курсової	10.10.2019	
2.	Ознайомлення з темою курсової	15.11.2019	
3.	Ознайомлення з літературою	30.12.2019	
4.	Розробка структури роботи	15.03.2020	
5.	Написання першого розділу	01.04.2020	
6.	Написання другого та третього розділів	10.04.2020	
7.	Написання четвертого та п’ятого розділів	19.04.2020	
8.	Попередній аналіз курсової. Виправлення помилок	04.05.2020	
9.	Оформлення презентації	09.05.2020	
10.	Захист курсової роботи		

Зміст

Анотація

У даній роботі розглядається бібліотека для синтаксичного аналізу Parsec, її можливості та застосування. Розглядаються стандартні функції та модулі цієї бібліотеки, а також досліджується її співпраця з бібліотекою для лексичного аналізу Alex.

Вступ

Розробка кожної мови програмування потребує вирішення задачі синтаксичного аналізу, що робить її завжди актуальною, адже нові мови програмування з’являються досить часто. Тільки за період з 2010 року по 2020 рік з’явилось безліч нових мов програмування, таких як Rust, Kotlin, Swift, тощо. Ці мови стали достатньо популярними щоб на них можна було натрапити у тому чи іншому медіа присвяченому програмуванню. Варто звернути увагу і на велику кількість бібліотек присвячених синтаксичному аналізу. Наприклад для мови Java ми маємо Jparsec, Parboiled, Jacc, Grammatica, тощо. Для Python маємо Canopy, Tatsu, PlyPlus і под.

Haskell не є винятком з цього списку, адже гнучка система типів, функціональна природа цієї мови та велика кількість бібліотек робить цю мову одним з найкращих варіантів для використання у синтаксичному аналізі.

Таким чином, метою даної роботи є огляд Parsec – однієї з найпопулярніших бібліотек мови Haskell та аналіз її можливостей. Паралельно з цим, ми розглянемо бібліотеку для лексичного аналізу Alex та особливості мови Haskell, які роблять її гарним вибором для рішення задачі синтаксичного аналізу.

1.Опис процесу синтаксичного та лексичного аналізу. Підготовка до синтаксичного аналізу

1.1 Мета синтаксичного та лексичного аналізу

Комп'ютерні програми зазвичай пишуться на високорівневих мовах програмування, що є зручними для сприйняття людьми, проте зовсім не пристосовані до обробки комп'ютером. Тому, перед тим як написаний нами текст програми буде виконано, він проходить попередню обробку *компілятором* чи *інтерпретатором*[1, 4-5].

Компілятор перетворює програму, написану на високорівневій мові програмування, у еквівалентну програму на низькорівневій мові (байт-код, асамблерний код і под.), що є близькою до машинного коду, або ж у програму на машинному коді. У будь-якому з цих представлень програма може бути виконана комп'ютером без повторного виклику компілятора[2].

Інтерпретатор, у свою чергу, читає текст програми рядок за рядком. Кожен рядок перетворюється у машинний код (або у інше, близьке до нього представлення) та відразу виконується. Для повторного виконання комп'ютером програми, інтерпретатор повинен її повторно обробити, адже він не генерує проміжного представлення програми як це робить компілятор[2].

У роботі компілятора та інтерпретатора важливими етапами є *лексичний* і *синтаксичний аналіз* (варто зазначити, що існують підходи до побудови компіляторів та інтерпретаторів, в яких лексичний аналіз явно не виділений та є частиною синтаксичного аналізу)[3, 4]. Під час лексичного аналізу, текст програми, який є послідовністю символів, перетворюється на послідовність *лексем (tokens)* [1, 5-8]. На етапі синтаксичного аналізу, на основі цієї послідовності, утворюється внутрішнє представлення програми, що має назву *абстракте синтаксичне дерево (abstract syntax tree – AST)* та використовується для подальшого аналізу та обробки програми[1, 5-8].

1.2 Процес синтаксичного та лексичного аналізу

Нехай, наша програма містить наступний рядок:

```
b := 6 + 10;
```

У результаті лексичного аналізу цей рядок буде перетворено на послідовність лексем приблизно наступного вигляду:

```
<VarName "b">
<Assign ":=">
<I 6>
<Plus "+">
<I 10>
<DelimiterSymbol ";">
```

Лістинг 1.1 Послідовність лексем

Кожна лексема має наступну структуру: *<клас лексеми значення лексеми>*. Значення лексеми – це послідовність символів, з якої лексема була утворена. Класи лексем, що наведені у лістингу 1.1, мають наступні значення:

a) *VarName* – ідентифікатор змінної;
b) *Assign* – оператор присвоєння;
c) *I* – числовий літерал;
d) *Plus* – арифметичний оператор «+»;
e) *DelimiterSymbol* – символ-роздільник.

Під час синтаксичного аналізу, отримана послідовність лексем зіставляється з граматикою мови програмування, на якій був написаний текст програми. Це дозволяє нам отримати відповідь на питання чи належить текст програми мові з якою ми його зіставляємо, чи ні[3, 4-5]. Також, на етапі синтаксичного аналізу, (з лексем) будується абстрактне синтаксичне дерево[1, 7-8]. Дерево, побудоване з лексем лістингу 1.1, буде мати наступний вигляд:

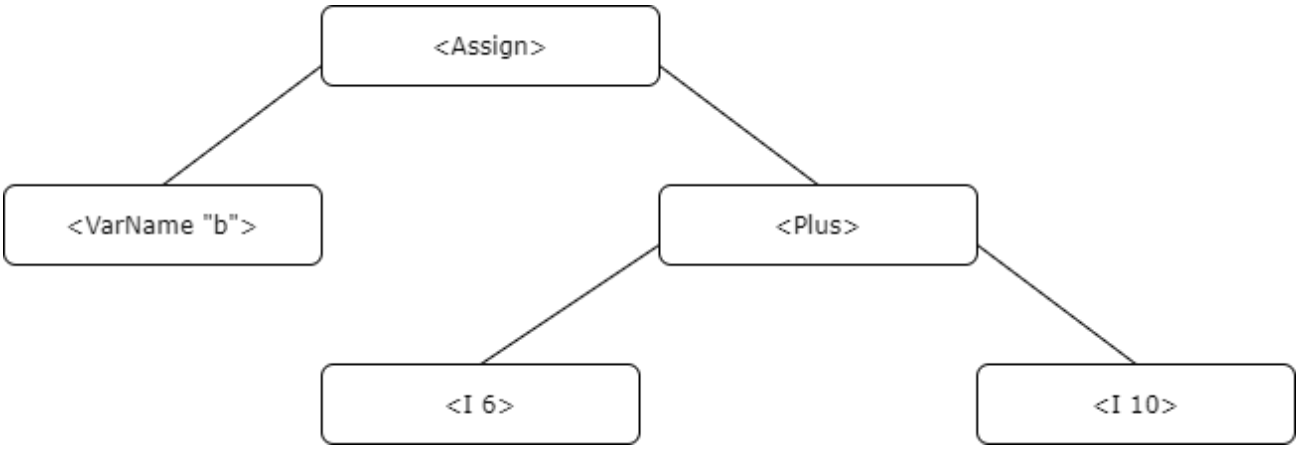

Рисунок 1.1 Абстрактне синтаксичне дерево

Примітка – лексеми VarName та DelimiterSymbol наведені для наочності. У даній роботі, під час лексичного аналізу, ці лексеми зазвичай є складовими частинами інших лексем та не існують окремо від них. Враховуючи це, лексема

<Assign> на рисунку 1.1 мала б вигляд <Assign "b">, а лексема <VarName "b"> була б відсутня.

Абстрактне синтаксичне дерево використовується для подальшого аналізу та обробки програми. Наприклад, воно використовується для *семантичного аналізу* програми[1, 8]. Частиною семантичного аналізу програми є перевірка типів, одночасно з якою може бути виконане їх зведення[1, 8].

1.3 Визначення граматики мови

Граматика мови визначає набір правил, що описують послідовності символів, які є правильно оформленою програмою або ж її частиною[4]. Наприклад, граматика мови Haskell визначає конструкцію if…else наступним чином:

```
if <умова> then <значення1> else <значення2>,
```

тобто конструкція if…else – це ключове слово «if», умова, ключове слово «then», значення, що повертається, коли умова істинна, слово «else» та значення, що повернеться, коли умова хибна.

Граматика використовується лексичним аналізатором для виділення лексем та синтаксичним аналізатором для перевірки синтаксису програми та побудови абстрактного синтаксичного дерева. Нижче наведене визначення граматики невеликої мови програмування у формі *розширеної нотації Бекуса-Наура*. На прикладі цієї мови, у даній роботі, будуть розглянуті можливості бібліотек Parsec та Alex.

```
symbol = ';' | '{' | '}' | '(' | ')'
idenName=  char {digit | char}
keyword= "int" | "bool" | "if" | "while" | "for" | "else" | "true" | "false"
iden   =  idenName .... not "int" "bool" "if" "while" "for" "else" "true" "false"
number = digit { digit }.
mulOrDivOp  = "*" | "/".
addOrSubOp  = "+" | "-".
relOp  = "<" | "<=" | ">" | ">=" | "=="
andOp  = "&"
orOp  = "|"
idenType  = "int" | "bool"

factor = '(' expr ')' | number | "true" | "false" | iden
term   = factor { mulOrDivOp factor }
relat  = term { addOrSubOp term }
conj   = relat [relOp relat]
disj   = conj { orOp conj}
expr   = disj { andOp disj}
```

```
stmt   = "for" forSt | "while" whileSt | "if" ifSt
       | iden assSt | blockSt
forSt  = '(' stmt ';' expr ';' stmt ')' stmt
whileSt= '(' expr ')' stmt
ifSt   = '(' expr ')' stmt "else" stmt
assSt  = "++" | ":=" expr

blockSt= '{' {defin} listSt '}'
defin  = type iden ';'
listSt = stmt {';' stmt}
program= stmt eos
```

1.4 Визначення необхідних типів

Як зазначалося у підрозділі 1.2, після того як лексичний аналізатор перетворить вхідний текст програми у послідовність токенів, синтаксичний аналізатор почне перетворювати цю послідовність токенів у абстрактне синтаксичне дерево[1, 8]. Для представлення цього дерева, у мові програмування Haskell, ми будемо використовувати *рекурсивні типи даних*, тобто типи даних, конструктори яких приймають значення цього ж типу[5], та звичайні (нерекурсивні) типи даних.

```haskell
data Value
  = I Int
  | B Bool
  deriving (Show, Eq)

data Exp
  = Var String
  | Const Value
  | Op Exp Bop Exp
  deriving (Show, Eq)

data Bop
  = Plus
  | Minus
  | Times
  | Div
  | Gt
  | Ge
  | Lt
  | Le
  | Eql
  | Ba
```

```
  | Bo
  deriving (Show, Eq)

data Stmt
  = Assign String Exp
  | Incr String
  | If Exp Stmt Stmt
  | While Exp Stmt
  | For Stmt Exp Stmt Stmt
  | Block [(String, Type)] [Stmt]
  deriving (Show, Eq)

data Type
  = It
  | Bt
  deriving (Show, Eq)
```

Значення цих типів наступне:

a) *Value* позначає числовий або булевий літерал;

b) *Expr* позначає арифметичний (логічний) вираз, оголошення змінної або ж числовий (булевий) літерал;

c) *Bop* позначає арифметичні та логічні оператори;

d) *Stmt* позначає операцію присвоєння, конструкцію if…else, цикли з передумовою та блок коду;

e) *Type* визначає тип змінної, де *It* позначає число, а *Bt* булевий літерал.

Вираз *deriving (Show, Eq)*, що записаний після визначення кожного типу, є вказівкою забезпечити ці типи стандартними функціями для виводу та порівняння[6, 140].

2. Бібліотека Parsec

2.1 Опис принципу роботи бібліотеки

Parsec – це бібліотека написана мовою Haskell, що використовується для побудови синтаксичних аналізаторів[7]. Процес побудови власного аналізатора складається з комбінування простих синтаксичних аналізаторів[10, 423]. Бібліотеки, що використовують даний принцип, називаються *комбінаторами парсерів (parser combinators)*[8, 34].

Отримавши на вхід текст програми, Parsec передає цей текст синтаксичним аналізаторам у вигляді потоку символів. Аналізатори проводять

обробку символів з потоку по черзі, один за одним, намагаючись виділити у ньому послідовності символів, що будуть відповідати правилам, на основі яких ці синтаксичні аналізатори були побудовані[9].

Кожен аналізатор «забирає» з потоку символи, які він успішно обробив[9]. Тому, символи, з якими працює наступний аналізатор, відрізняються від тих, з якими працював попередній.

Для роботи з бібліотекою необхідні два модулі, імпорт яких наведений нижче.

```
import qualified Text.Parsec as Parsec
import Text.ParserCombinators.Parsec
```

Модуль *Text.Parsec* містить визначення *функцій-комбінаторів*, використовуючи які ми можемо поєднувати синтаксичні аналізатори між собою, та визначення типу даних, що є результатом виконання кожного аналізатора бібліотеки[7]. Цей тип має наступний вигляд:

```
ParsecT s u m a
```

Лістинг 2.1 Тип ParsecT

Параметр *s* позначає тип даних, що приймає синтаксичний аналізатор, *a* позначає тип даних, який повертає синтаксичний аналізатор. Параметр *u* – це тип даних, що бібліотека передає між аналізаторами[7]. Якщо *u* має тип масиву, то це буде масив, доступний кожному синтаксичному аналізатору під час їх роботи. При цьому, кожен аналізатор зможе цей масив змінити.

Останній параметр, що не був розглянутий – це *m*. *m* – монада яка буде застосована до результату роботи синтаксичного аналізатора. Таким чином, тип ParsecT є *монадним трансформером (monad transformer)*[10, 469].

Проте, у сигнатурах функцій, що будуть виконувати синтаксичний аналіз, ми будемо використовувати більш стислий тип:

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

Тип Parsec є *синонімом типу (type synonym)*[6, 121] для ParsecT. Фактично, це просто інша назва типу ParsecT, параметру *m* якого присвоєна монада *Identity*, що є монадою, яка нічого не робить.

Модуль *Text.ParserCombinators.Parsec* містить визначення простих синтаксичних аналізаторів[7].

2.2 Розгляд простих аналізаторів бібліотеки Parsec.

Розглянемо декілька простих синтаксичних аналізаторів, сигнатури яких наведено нижче.

```
char :: (Stream s m Char) => Char -> ParsecT s u m Char
digit :: (Stream s m Char) => ParsecT s u m Char
string :: (Stream s m Char) => String -> ParsecT s u m String
oneOf :: (Stream s m Char) => [Char] -> ParsecT s u m Char
```

Лістинг 2.2 Прості синтаксичні аналізатори

char приймає один аргумент, який є символом. Отримуючи на вхід потік символів, він намагається співставити аргумент із першим символом у потоці. Якщо в нього не виходить це зробити, виникає помилка. У іншому випадку повертається розпізнаний символ[7]. За цим принципом працюють усі прості аналізатори. Приклад використання *char* у середовищі GHCI, разом з результатом:

```
ghci> parse (char 'H') "" "Hello"
Right 'H'
```

Лістинг 2.3 Використання char

parse – це функція, що приймає три аргументи: синтаксичний аналізатор, файл для виводу помилок та текст до якого потрібно застосувати аналізатор. Результатом виконання функції є монада *Either ParseError a*, де *a* – це параметр, що відповідає типу, який є результатом роботи синтаксичного аналізатора[7].

У разі успішного завершення повертається значення вигляду *Right a*, інакше – *Left ParseError*, яке позначає помилку та описує причину її виникнення[7]. У випадку лістингу 2.3, значення, що повернулося, має тип *Right Char*.

digit розпізнає цифру на початку потоку символів та повертає її[7]. *string* приймає стрічку з символів та намагається розпізнати відповідну стрічку на початку потоку символів[7]. Важливо зазначити, що string розпізнає всі символи зі стрічки по черзі, тобто функція спочатку намагається розпізнати у потоці перший символ, потім другий і так далі.

oneOf приймає стрічку з символів та намагається співставити один з цих символів символу на початку потоку[7]. Приклади використання цих аналізаторів наведено у лістингу 2.4.

```
ghci> parse digit "" "13"
Right '1'
ghci > parse (string "Hello") "" "Hello"
Right "Hello"
ghci > parse (oneOf "abcd") "" "axzs"
Right 'a'
```

Лістинг 2.4 Використання digit, string та oneOf

2.3 Поєднання простих аналізаторів

Parsec, будучи комбінатором парсерів, дозволяє нам поєднувати прості аналізатори, на кшталт тих, що наводилися раніше, для побудови більш складних аналізаторів. Такі аналізатори називаються *комбінаторами*[8, 34].

Оскільки кожен простий аналізатор повертає значення типу ParsecT, що є монадою, для їх комбінування ми будемо користуватися *do-нотацією*. do-нотація – це спеціальний синтаксис для монад, що спрощує їх використання[6, 254-255, 387-391]. Приклад застосування цієї нотації:

```
number :: Parsec.Parsec String [(String, SourcePos)] Int
number = do
  s <- string "-" <|> return []
  cs <- many1 digit
  return $ read (s ++ cs)
```

Лістинг 2.5 Застосування do-нотації

Альтернативою do-нотації є використання *аплікативного стилю*[9]. Код з лістингу 2.5 у аплікативному стилі має наступний вигляд:

```
number :: Parsec.Parsec String [(String, SourcePos)] Int
number = (\a b -> read (a ++ b) :: Int)
            <$> (Parsec.string "-" <|> return [])
            <*> many1 digit
```

Комбінатор *number* розпізнає числа у вхідному потоці символів. Він складається з простих аналізаторів string та digit, що розглядалися раніше, та ще одного комбінатора *many1*.

many1 приймає один аргумент, який може бути або простим синтаксичним аналізатором, або ж комбінатором, та застосовує його до вхідного потоку символів допоки він не зможе обробити символи з цього

потоку[7]. Причому, цей аналізатор повинен спрацювати як мінімум один раз. Інакше виникне помилка[7].

Таким чином, для побудови аналізатора мови, ми спочатку будуємо аналізатори для простих лексем, наприклад чисел, назв змінних, арифметичних операторів, тощо. Після чого, з їх використанням, будуємо аналізатори для більш складних конструкцій, таких як арифметичні вирази, операції присвоєння, т.ін.

Наприклад, аналізатор, що розпізнає булеві оператори, може мати наступний вигляд:

```
oprtr :: String ->
         Bop ->
         Parsec.Parsec String [(String, SourcePos)] Bop
oprtr str bop = do
  void $ string str
  whitespaces
  return bop

relOp :: Parsec.Parsec String [(String, SourcePos)] Bop
relOp =
  try (oprtr ">=" Ge) <|>
  try (oprtr ">" Gt)  <|>
  try (oprtr "<=" Le) <|>
  try (oprtr "==" Eql)<|>
  try (oprtr "<" Lt)
```

Лістинг 2.6 Аналізатор для булевих операторів

Аналізатор *oprtr*, наведений у лістингу 2.6, приймає символьне представлення оператора та тип даних, яким цей оператор представлений у програмі, після чого він розпізнає цей оператор у вхідному потоці символів, ігноруючи довільну кількість проміжків після нього. У свою чергу, *relOp* (лістинг 2.6) використовує *oprtr* для того, щоб розпізнати всі можливі булеві оператори.

2.4 Обробка помилок

При виникненні помилки у ході виконання програми, Parsec виведе докладне повідомлення про помилку, що буде містити номер рядка та стовпчика, де ця помилка сталася, та її причину[7]. Приклад такого повідомлення:

```
ghci> parse (string "apple") "" "pineapple"
Left (line 1, column 1):
unexpected "p"
expecting "apple"
```

Єдиним недоліком цих повідомлень є те, що вони стосуються тільки тих ситуацій, в яких аналізатор не може розпізнати потрібну послідовність символів у вхідному потоці. Для інших випадків у бібліотеці визначено тип даних під назвою *Message*[7], що наведений нижче:

```
data Message = SysUnExpect String
             | UnExpect String
             | Expect String
             | Message String
```

Тип Message представлений чотирма варіантами[7]:

a) *SysUnExpect* повідомляє про неправильні вхідні дані. Згенерувати його можна за допомогою функції *satisfy*;

b) *UnExpect* найчастіше використовують для повідомлень про синтаксичні помилки. Згенерувати його можна за допомогою функції *unexpected*;

c) *Expect* повідомляє про елемент вхідного потоку, що очікується аналізатором. Генерується за допомогою комбінатора <?>;

d) *Message* використовують для довільних повідомлень. Генерується функцією *fail*.

Приклад генерування одного з цих повідомлень, а саме повідомлення типу UnExpect, наведено нижче на прикладі аналізатора, що розпізнає ім’я змінної.

```
iden :: Parsec.Parsec String [(String, SourcePos)] String
iden =
  try $ do
    name <- idenName
    if name `elem`
       ["int", "bool", "if", "while", "for", "else", "True", "False"]
     then unexpected ("use of reserved word " ++ show name)
     else return name
```

Для обробки неочікуваних ситуацій Parsec також надає[7] функцію *choice*, оператор <|> та комбінатор <?>, який згадувався вище.

Оператор <|> є інфіксним варіантом choice. Він об'єднує аналізатори у ланцюжок. Якщо попередній аналізатор з ланцюжка не зміг обробити символи

вхідного потоку, то обробку продовжить наступний аналізатор з того місця, де закінчився попередній[7]. Приклад застосування цього оператора наведено у лістингу 2.6.

Важливо зазначити один нюанс у роботі оператора <|>: процес з ланцюжком, описаний вище, використовується для вибору аналізатора, який буде застосований до вхідного потоку символів. Обраним буде тільки той аналізатор, що успішно обробить символи на початку потоку. Якщо у ньому пізніше виникне помилка, то програма завершить виконання з відповідним повідомленням[9]. Це показано у наступному лістингу:

```
ghci> parse (string "hello" <|> string "howdy") "" "howdy"
Left (line 1, column 1):
unexpected "o"
expecting "hello"
```

Лістинг 2.7 Помилка у програмі

У лістингу 2.7, за допомогою оператора <|> був обраний перший аналізатор string «hello», оскільки він успішно обробив першу літеру вхідного тексту «howdy». Проте, пізніше виникла помилка, тому що обраний аналізатор очікував зустріти літеру «e», а отримав «o».

Для запобігання цьому ми можемо використати функцію під назвою *try*. try приймає синтаксичний аналізатор та застосовує його до вхідного потоку символів. Якщо у аналізаторі виникає помилка, try її перехоплює та відміняє всі зміни зроблені цим аналізатором[7]. Таким чином, програма може продовжити своє виконання.

Приклад застосування функції:

```
ghci> parse (try (string "hello") <|> string "howdy") "" "howdy"
Right "howdy"
```

Комбінатор <?> застосовує аналізатор, що стоїть зліва від нього. Якщо у цьому аналізаторі виникає помилка, то комбінатор виводить цю помилку разом з описом того, які елементи цей аналізатор очікував отримати[7]. Опис знаходиться з права від <?>.

Приклад застосування комбінатора:

```
ghci> parse (string "hello" <?> "a greeting") "" "wrong"
Left (line 1, column 1):
unexpected "w"
expecting a greeting
```

Якщо комбінатор <?> буде стояти останнім у ланцюжку оператора <|>, то він виведе помилку тільки у тому випадку, якщо жоден аналізатор з ланцюжку не зможе обробити вхідний потік символів[9]. Приклад такої ситуації:

```
ghci> parse (string "dog" <|> string "cat" <?> "cat or dog") "" "bat"
Left (line 1, column 1):
unexpected "b"
expecting cat or dog
```

2.5 Керування станом програми

Оскільки Haskell – це чиста мова програмування, функції у наших програмах не можуть впливати на дані, що знаходяться поза їх областю видимості. Вони можуть тільки приймати значення, робити обрахунки та повертати результат. Проте, оскільки деякі програми вимагають такої можливості, у Haskell з'явилась монада під назвою *MonadState*[11] і так вийшло, що ParsecT, тип на якому ґрунтується бібліотека Parsec, «наслідує» цю монаду[7].

Таким чином, у нас є можливість передавати певні дані між всіма аналізаторами, причому кожен аналізатор може їх змінювати (для типу цих даних у визначенні ParsecT, що наведено у лістингу 2.1, відведений параметр u).

Для роботи з цими даними у бібліотеці є три функції[7]:

a) *getState* – зчитує дані;

b) *putState* – видаляє дані, що існують на даний момент та записує на їх місці нові;

c) *modifyState* – приймає функцію та змінює дані, що збережені у даний момент за допомогою цієї функції.

Нижче наведений аналізатор, який за допомогою цих функцій будує *символьну таблицю*[1, 4-6].

```
defin :: Parsec.Parsec String [(String, SourcePos)] (String, Type)
defin = do
  pos <- Parsec.getPosition
  tp <- idenType
  idn <- idenName
  Parsec.modifyState (\state -> state ++ [(idn, pos)])
  symbol ';'
  return (idn, tp)
```

Аналізатор *defin*, розпізнає тип та назву змінної, після чого зберігає назву змінної та її позицію у символьну таблицю, що представлена масивом кортежів. У лістингу нижче показано як ми цю таблицю зчитуємо.

```
parseSPL :: String -> Either ParseError (Stmt, [(String, SourcePos)])
parseSPL =
  Parsec.runParser
    (do pr <- program
        st <- Parsec.getState
        return (pr, st)) [] ""
```

Функція *parseSPL* запускає весь процес синтаксичного аналізу. У разі успішного завершення повертає абстрактне синтаксичне дерево та таблицю символів.

2.6 Аналіз потоку байтів

За замовченням бібліотека приймає на вхід String, що є псевдонімом для списку символів. З типом String зручно працювати, проте він має один недолік: швидкість роботи.

Символи (Char) з яких складається String не мають фіксованого розміру, оскільки якщо це символ Unicode, для його представлення нам потрібно декілька байтів, скільки саме ми наперед не знаємо. Також, String, будучи списком, є «лінивим», тобто значення у ньому оцінюються тільки коли вони дійсно потрібні. Це дає можливість працювати зі String як з потоком символів, проте негативно впливає на швидкість програми, особливо, коли ми працюємо з великими файлами[12, 198-199].

Більш ефективним у порівнянні зі String є тип ByteString, що є списком кожен елемент якого займає рівно один байт[12, 198-199], при цьому, цей список поділений на частини. Кожна частина займає 64KB. Як тільки один байт з цієї частини буде оцінений, то буде оцінена і вся частина[12, 198-199].

Parsec працює з ByteString так само як і з String, тому все, що нам потрібно зробити для того, щоб наші аналізатори використовували ByteString замість String – це перетворити текст на ByteString та передати його на вхід. Приклад наведено нижче.

```
import qualified Data.ByteString.Char8 as BC

startParser :: String -> Either ParseError Stmt
startParser text =
  case parseSPLByteString (BC.pack text) of
    Right res -> Right res
    Left err -> Left err
```

Функція *pack* перетворює String на ByteString[6, 306], після чого ми передаємо ці дані на вхід синтаксичному аналізатору. Не дивлячись на те, що ми змінили тип вхідних даних, усі аналізатори, що згадувалися раніше, залишаються незмінними.

3. Використання Parsec.Language та Parsec.Expr

3.1 Створення лексичних аналізаторів за допомогою Parsec.Language

Під час синтаксичного аналізу ми працюємо тільки з лексемами, тому нам доводиться доволі часто займатися лексичним аналізом. У Parsec не існує уніфікованих лексичних аналізаторів, що правильно б розпізнали будь-які лексеми, оскільки для кожної мови програмування лексеми можуть відрізнятися. Таким чином у нас є два варіанти:

a) написати всі аналізатори самостійно;

b) згенерувати потрібні аналізатори за допомогою бібліотеки.

Перший варіант є досить зручним коли у мові програмування, з якою ви працюєте, зовсім небагато лексем. Тоді, використовуючи стандартні аналізатори Parsec, деякі з яких розглядались у підрозділі 2.2, ви зможете доволі швидко написати все потрібне. Наприклад, ось аналізатор, що розпізнає ідентифікатор змінної:

```
idenName :: Parsec.Parsec String [(String, SourcePos)] String
idenName = do
  whitespaces
  a <- letter
  cs <- many (letter <|> digit)
  return (a : cs)
```

Проте, якщо у вашій мові програмування багато лексем та, можливо, є коментарі, які треба ігнорувати під час лексичного аналізу, задача може зайняти

більше часу. Саме тоді нам приходить на допомогу Parsec, який має модуль під назвою *Parsec.Language*[7], використовуючи який ми зможемо згенерувати потрібні лексичні аналізатори, пишучи мінімум коду.

Для цього нам потрібно спочатку передати визначення мови функції *emptyDef*, яка є конструктором, що повертає значення типу *GenLanguageDef* , згідно з яким пізніше будуть згенеровані потрібні нам аналізатори[7]. Як саме ми визначаємо мову та передаємо її у emptyDef показано нижче.

```
rNames :: [String]
rNames = words "true false bool int if else while for"

opNames :: [String]
opNames = words "+ - * / < <= > >= := == & | ++"

languageDef :: GenLanguageDef String u Data.Functor.Identity.Identity
languageDef =
  emptyDef
    { Token.commentStart = "/*"
    , Token.commentEnd = "*/"
    , Token.commentLine = "//"
    , Token.identStart = letter
    , Token.identLetter = alphaNum <|> char '_'
    , Token.reservedNames = rNames
    , Token.reservedOpNames = opNames
    }
```

Лістинг 3.1 Визначення мови

Для визначення мови ми використовуємо *синтаксис записів (record syntax)*[6, 127-130] мови Haskell. Ми вказуємо як виглядають коментарі, багато строкові та одно строкові, ідентифікатори змінних (причому ми окремо вказуємо з чого повинен починатися ідентифікатор), зарезервовані імена та оператори.

Далі, ми передаємо значення, отримане від emptyDef, функції *makeTokenParser*, яка, в свою чергу, повертає значення типу *GenTokenParser*[7]. Це значення буде містити лексичні аналізатори, що розпізнають всі лексеми наведені у визначенні мови, причому, вони за замовченням ігнорують коментарі та проміжки між лексемами[7]. Використання makeTokenParser наведено нижче.

```
lexer = Token.makeTokenParser languageDef
```

Лістинг 3.2 Використання makeTokenParser

Для того щоб використати ці аналізатори їх спочатку потрібно дістати зі значення (у лістингу 3.2 воно має назву *lexer*), що ми отримали від makeTokenParser. Для цього існують окремі функції, які фактично є звичайними гетерами. Наприклад, якщо нам потрібен аналізатор, що розпізнає ідентифікатори – викликаємо функцію *identifier*. Якщо нам потрібен аналізатор, що розпізнає число – викликаємо функцію *integer*[7].

Використання цих та інших функцій для отримання аналізаторів можна побачити нижче.

```
identifier = Token.identifier lexer

reserved = Token.reserved lexer

reservedOp = Token.reservedOp lexer

parens = Token.parens lexer

integer = Token.integer lexer

semi = Token.semi lexer

whiteSpace = Token.whiteSpace lexer

symbol = Token.symbol lexer
```

3.2 Аналіз виразів за допомогою Parsec.Expr

Під час синтаксичного аналізу нам доволі часто доводиться працювати з виразами, тому у Parsec є допоміжний модуль під назвою *Parsec.Expr*[7], який може згенерувати нам єдиний аналізатор, що буде розпізнавати вирази з усіх потрібних нам операторів. При цьому, аналізатор буде враховувати асоціативність та положення цих операторів відносно їх аргументів.

Аналізатор генерується за допомогою функції *buildExpressionParser*, що приймає два аргументи[7]:

а) *table – таблиця операторів*, або ж значення типу *OperatorTable*. OperatorTable є псевдонімом для матриці, кожний елемент якої має тип *Operator*. Цей тип описує оператор та містить інформацію про його асоціативність, позицію відносно аргументів (інфіксна, постфіксна чи префіксна) та аналізатор, який розпізнає символьне представлення цього оператора. Оператори, що знаходяться в одному рядку матриці мають

однаковий пріоритет. Оператори, що знаходяться у першому рядку, мають вищий пріоритет за ті, що знаходяться у другому і т. д.;

b) *term* – аналізатор для лексем, що знаходяться між операторами.

Приклад таблиці операторів наведений у лістингу 3.3. Приклад використання функції buildExpressionParser наведено у лістингу 3.4.

```
operators =
  [ [ Infix
      (reservedOp "*" >> return (\x y -> Op x Times y))
      AssocLeft
    , Infix
      (reservedOp "/" >> return (\x y -> Op x Div y))
      AssocLeft
    ]
  , [ Infix
      (reservedOp "+" >> return (\x y -> Op x Plus y))
      AssocLeft
    , Infix
      (reservedOp "-" >> return (\x y -> Op x Minus y))
      AssocLeft
    ]
  , [ Infix
      (reservedOp "==" >> return (\x y -> Op x Eql y))
      AssocNone
    , Infix
      (reservedOp "<" >> return (\x y -> Op x Lt y))
      AssocNone
    , Infix
      (reservedOp "<=" >> return (\x y -> Op x Le y))
      AssocNone
    , Infix
      (reservedOp ">" >> return (\x y -> Op x Gt y))
      AssocNone
    , Infix
      (reservedOp ">=" >> return (\x y -> Op x Ge y))
      AssocNone
    ]
  , [ Infix
      (reservedOp "&" >> return (\x y -> Op x Ba y))
      AssocLeft
    , Infix
      (reservedOp "|" >> return (\x y -> Op x Bo y))
      AssocLeft ] ]
```

Лістинг 3.3 Таблиця операторів

```
term :: Parser Exp
term =
  parens expr <|> (Const . I . fromIntegral) <$> integer <|>
  (reserved "true" >> return (Const (B True))) <|>
  (reserved "false" >> return (Const (B False))) <|>
  liftM Var identifier

expr :: Parser Exp
expr = buildExpressionParser operators term
```

Лістинг 3.4 Використання buildExpressionParser

Аналізатор виразів, який нам повертає buildExpressionParser, у лістингу 3.4 має назву *expr*. Для обробки лексем, що стоять між операторами ми використовуємо функцію *term*, яка в свою чергу викликає expr, оскільки в якості лексем, що стоять між операторами можуть виступати як літерали так і вирази з інших операторів.

Нижче наведено ще одне застосування expr на прикладі аналізатора *whileStmt*, що розпізнає оператор циклу while.

```
whileStmt :: Parser Stmt
whileStmt = do
  reserved "while"
  cond <- parens expr
  While cond <$> stmt
```

4. Синтаксичний аналіз тексту програми у вигляді готових лексем

4.1 Визначення лексем

У базовому випадку, текст програми, що ми обробляємо, подається на вхід Parsec у вигляді значення типу String, проте, ми не мусимо використовувати тільки String для подання вхідних даних, адже у визначенні типу ParsecT (лістинг 2.1), на якому ґрунтується бібліотека, тип вхідних даних є параметром, який ми можемо змінювати[13] (варто зазначити[7], що тип, який ми вирішимо використовувати замість String, повинен бути членом *класу типів*[6, 133-134] *Stream*. Членами класу Stream, є, наприклад, усі списки*)*.

Наприклад, ми можемо замість списку символів подавати на вхід бібліотеки список вже готових лексем. Таким чином ми зможемо чіткіше розмежувати процеси лексичного та синтаксичного аналізу, і, якщо це потрібно, використати для лексичного аналізу окрему бібліотеку, відмінну від Parsec. Почнемо з визначення лексем:

```
data DataToken
  = ArithmOpToken
      { line, column :: Int
      , arithmOperator :: String
      }
  | BoolOpToken
      { line, column :: Int
      , boolOperator :: String
      }
  | AssignOpToken
      { line, column :: Int
      , operator :: String
      }
  | KeywordToken
      { line, column :: Int
      , reservedWord :: String
      }
  | VarTypeToken
      { line, column :: Int
      , varType :: String
      }
  | VarNameToken
      { line, column :: Int
      , varName :: String
      }
  | NumConstToken
      { line, column :: Int
      , numValue :: Int
      }
  | BoolConstToken
      { line, column :: Int
      , boolValue :: Bool
      }
  | DelimiterSymbolToken
      { line, column :: Int
      , customSymbol :: String
      }
```

Лістинг 4.1 Визначення лексем

Усі лексеми належать типу *DataToken*, який має 10 конструкторів для кожної лексеми, що може нам знадобитись. Конструктори ми визначаємо за допомогою синтаксису записів. Цей синтаксис дає нам можливість вказати типи

даних, які приймає конструктор, та назви функцій, що, отримавши на вхід значення, створене цим конструктором, повертають дані, якими ми користувались для його створення[6, 127-130].

Наприклад, конструктор лексеми під назвою *KeywordToken* позначає ключове слово, таке як «else». Він приймає два числа, що є номерами рядка та стовпчика, де у вхідному тексті починається послідовність символів, з якої ця лексема була утворена, та саму послідовність символів. Водночас, ми визначаємо три функції, що приймають значення утворене KeywordToken, та дають нам можливість отримати дані, зазначені вище:

a) *line* – повертає номер рядка;

b) *column* – повертає номер стовпчика;

c) *reservedWord* – повертає послідовність символів.

Невеликий приклад використання конструктора KeywordToken, та функцій з переліку, в середовищі GHCI:

```
ghci> lexem = KeywordToken 12 17 "else"
ghci> :type lexem
lexem :: DataToken
ghci> line lexem
12
ghci> column lexem
17
ghci> reservedWord lexem
"else"
```

Для того, щоб ми мали можливість вивести лексеми на екран, потрібно зробити тип DataToken членом класу типів[6, 133-134] під назвою *Show*. Show – це клас, членів якого можна вивести на екран[6, 139]. Для того щоб DataToken належав цьому класу нам потрібно визначити функцію *show* для кожного конструктора цього типу. Це можна зробити наступним чином:

```
instance Show DataToken where
  show ArithmOpToken {arithmOperator = op} = "ArithmOpToken " ++ show op
  show BoolOpToken {boolOperator = op} = "BoolOpToken " ++ show op
  show DelimiterSymbolToken {customSymbol = sym} =
    "DelimiterSymbolToken " ++ show sym
  show KeywordToken {reservedWord = rw} = "KeywordToken " ++ show rw
  show VarNameToken {varName = vn} = "VarNameToken " ++ show vn
  show VarTypeToken {varType = vt} = "VarTypeToken " ++ show vt
  show AssignOpToken {operator = op} = "AssignOpToken " ++ show op
  show NumConstToken {numValue = val} = "NumConstToken " ++ show val
  show BoolConstToken {boolValue = val} = "BoolConstToken " ++ show val
```

Лістинг 4.2 DataToken як член класу Show

4.2 Написання лексичного аналізатора

Тепер потрібно написати лексичний аналізатор, що буде розпізнавати лексеми (наведені в лістингу 4.1), у вхідному тексті програми.

Почнемо з допоміжних функцій.

```
parsePrefix ::
     Int -> Int -> String -> [String] -> Maybe (String, Int, Int, String)
parsePrefix lin col str operators =
  case oprtrtIndex of
    Just index ->
      let oprtrt = (xOperators !! index)
       in Just
            ( oprtrt
            , lin
            , col + length oprtrt
            , fromJust (stripPrefix oprtrt str))
    Nothing -> Nothing
  where xOperators = operators
        presentOperator = map (`isPrefixOf` str) xOperators
        oprtrtIndex = elemIndex True presentOperator
```

Функція *parsePrefix* приймає на вхід номери рядка та стовпчика, що позначають позицію у вхідному тексті, починаючи з якої ми здійснюємо обробку тексту, вхідний текст у вигляді стрічки та список стрічок, одну з яких parsePrefix повинен розпізнати. Функція повертає кортеж, що містить номери рядка та стовпчика, де закінчується розпізнана стрічка, саму стрічку та вхідний текст, який залишився. Інакше функція повертає *Nothing*, що є значенням типу *Maybe*, яке позначає «нічого»[6, 215-219]. parsePrefix ми будемо використовувати у функціях, які розпізнають окремі лексеми. Наприклад:

```
boolOperators :: [String]
boolOperators = ["&", "|", "==", "<=", ">=", "<", ">"]

arithmeticOperators :: [String]
arithmeticOperators = ["/", "*", "-", "++", "+"]

parseArithmOp :: Int -> Int -> String -> Maybe (DataToken, Int, Int, String)
parseArithmOp lin col str =
  case parsePrefix lin col str arithmeticOperators of
    Just (op, newLin, newCol, rest) ->
      Just (ArithmOpToken lin col op, newLin, newCol, rest)
```

```
    Nothing -> Nothing

parseBoolOp :: Int -> Int -> String -> Maybe (DataToken, Int, Int, String)
parseBoolOp lin col str =
  case parsePrefix lin col str boolOperators of
    Just (op, newLin, newCol, rest) ->
      Just (BoolOpToken lin col op, newLin, newCol, rest)
    Nothing -> Nothing
```

Лістинг 4.3 Функції parseArithmOp та parseBoolOp

Функція *parseArithmOp* розпізнає арифметичні оператори. Вона приймає ті ж параметри, що і parsePrefix, за винятком списку стрічок, та повертає лексему виду ArithmOpToken. *parseBoolOp* аналогічна parseArithmOp. Вона розпізнає булеві оператори та повертає лексему виду BoolOpToken.

Функції, що розпізнають інші лексеми відрізняються від тих, що наведені у лістингу 4.3, тільки лексемами, які вони повертають. Проте, є і винятки. Наприклад *parseVarName*, що розпізнає ідентифікатори змінних:

```
parseVarName :: Int -> Int -> String-> Maybe (DataToken, Int, Int, String)
parseVarName lin col str =
  case span isAlphaNum str of
    ([], _) -> Nothing
    (name, rest) ->
      Just (VarNameToken lin col name, lin, col + length name, rest)
```

Загалом, лексичний аналізатор буде виглядати наступним чином:

```
lexer :: Int -> Int -> String -> [DataToken]
lexer _ _ [] = []
lexer lin col input@(ch:chrs)
  | any (`isPrefixOf` input) arithmeticOperators =
    case parseArithmOp lin col input of
      Just (tok, newLin, newCol, rest) ->
        case rest of
          ('-':remains) ->
            case parseNumLiteral newLin newCol remains of
              Just (NumConstToken l c n, newerLin, newerCol, remainder) ->
                tok :
                NumConstToken l c (n * (-1)) : lexer newerLin newerCol remainder
              Just (_, _, _, _) -> lexer lin col input
              Nothing -> lexer lin col input
          _ -> tok : lexer newLin newCol rest
      Nothing -> lexer lin col input
```

```
| any (`isPrefixOf` input) boolOperators =
  case parseBoolOp lin col input of
    Just (tok, newLin, newCol, rest) -> tok : lexer newLin newCol rest
    Nothing -> lexer lin col input
| any (`isPrefixOf` input) [":="] =
  case parseAssignmentOp lin col input of
    Just (tok, newLin, newCol, rest) -> tok : lexer newLin newCol rest
    Nothing -> lexer lin col input
| any (`isPrefixOf` input) ["true", "false"] =
  case parseBoolLiteral lin col input of
    Just (tok, newLin, newCol, rest) -> tok : lexer newLin newCol rest
    Nothing -> lexer lin col input
| any (`isPrefixOf` input) varTypes =
  case parseVarType lin col input of
    Just (tok, newLin, newCol, rest) -> tok : lexer newLin newCol rest
    Nothing -> lexer lin col input
| any (`isPrefixOf` input) reservedWords =
  case parseReservedWord lin col input of
    Just (tok, newLin, newCol, rest) -> tok : lexer newLin newCol rest
    Nothing -> lexer lin col input
| any (`isPrefixOf` input) delimitingSymbols =
  case parseDelimiterSymbol lin col input of
    Just (tok, newLin, newCol, rest) -> tok : lexer newLin newCol rest
    Nothing -> lexer lin col input
| isDigit ch =
  case parseNumLiteral lin col input of
    Just (tok, newLin, newCol, rest) -> tok : lexer newLin newCol rest
    Nothing -> lexer lin col input
| "\n" `isPrefixOf` input || "\r\n" `isPrefixOf` input =
  case stripPrefix "\n" input of
    Just rest -> lexer (lin + 1) 1 rest
    Nothing ->
      case stripPrefix "\r\n" input of
        Just rest -> lexer (lin + 1) (col + 2) rest
        Nothing -> lexer lin col input
| isSpace ch = lexer lin (col + 1) chrs
| isAlpha ch =
  case parseVarName lin col input of
    Just (tok, newLin, newCol, rest) -> tok : lexer newLin newCol rest
    Nothing -> lexer lin col input
| otherwise = error
              ("lexer: unexpected character: " ++
              show ch ++ " at line " ++ show lin ++ " column " ++ show col)
```

Аналізатор має назву *lexer*. Він приймає на вхід текст, який потрібно розбити на лексеми, та номери рядка і стовпчика, що позначають початок цього тексту. Результатом роботи функції є список лексем.

lexer є скінченим автоматом, кожний стан якого представлений аналізатором, що розпізнає ту чи іншу лексему. Умови переходу в кожний зі станів визначені за допомогою *охоронних виразів (guards)*[6, 191-192] мови Haskell.

Охоронні вирази є аналогом оператора розгалуження if then…else. Вони виглядають наступним чином: «| *умова = блок коду*» (зверніть увагу на вертикальну риску на початку виразу). Якщо умова істинна – виконується блок коду, інакше – виконується наступний охоронний вираз.

Це є важливим моментом, оскільки охоронні вирази повинні бути спроможними обробити кожне значення, що передається у функцію. Якщо умова попереднього виразу була хибною, а наступного виразу немає – функція завершиться аварійно. Наприклад, маємо функцію *animalSounds*:

```
animalSounds :: String -> String
animalSounds str
    | str == "cat" = "Meow"
    | str == "dog" = "Bark"
```

Коли ми викличемо її з аргументом, що не врахований в умові жодного охоронного виразу, отримаємо наступне:

```
ghci> animalSounds "parrot"
"*** Exception: examples.hs:(137,1)-(139,27): Non-exhaustive patterns in
function animalSounds
```

Тому дуже важливо визначати групи охоронних виразів так, щоб для кожного аргументу функції знайшовся вираз, умова якого істинна. Таким чином, гарною практикою вважається наявність наприкінці групи – охоронного виразу, що має змінну *otherwise* у своїй умові. *otherwise* – це змінна, що рівна True. Мотивацією до її використання замість звичайного True є покращення зовнішнього вигляду програми.

Важливо зазначити, що розташування цього виразу саме наприкінці групи не є випадковим, адже як тільки знайдеться охоронний вираз з істинною умовою, усі інші після нього не будуть розглянуті.

4.3 Написання синтаксичного аналізатора

Синтаксичний аналізатор буде використовувати Parsec та працювати зі списком лексем, проте, за своєю структурою він буде мало чим відрізнятись від того, що обробляє список символів.

Перед тим як почати обробляти список, нам потрібно зробити тип DataToken (лістинг 4.1), який мають всі наші лексеми, членом класу типів під назвою *Eq*. Eq – це клас, членів якого можна порівнювати між собою за допомогою операцій «==» та «/=»[6, 137]. Для того щоб тип DataToken належав цьому класу нам достатньо визначити тільки операцію «==» для кожного конструктора цього типу.

```
instance Eq DataToken where
  (==)
    ArithmOpToken {arithmOperator = ao1}
    ArithmOpToken {arithmOperator = ao2}
    =
    ao1 == ao2
  (==)
    BoolOpToken {boolOperator = bo1}
    BoolOpToken {boolOperator = bo2}
    =
    bo1 == bo2
  (==)
    DelimiterSymbolToken {customSymbol = sym1}
    DelimiterSymbolToken {customSymbol = sym2}
    =
    sym1 == sym2
  (==)
    KeywordToken {reservedWord = rw1} KeywordToken {reservedWord = rw2} =
    rw1 == rw2
  (==) VarTypeToken {varType = vt1} VarTypeToken {varType = vt2} = vt1 == vt2
  (==) VarNameToken {} VarNameToken {} = True
  (==) AssignOpToken {operator = op1} AssignOpToken {operator = op2} =
    op1 == op2
  (==) NumConstToken {} NumConstToken {} = True
  (==) BoolConstToken {boolValue = val1} BoolConstToken {boolValue = val2} =
    val1 == val2
  (==) _ _ = False
```

Лістинг 4.4 DataToken як член класу Eq

Оскільки ми тепер маємо можливість порівнювати лексеми між собою, ми можемо написати прості аналізатори, що будуть розпізнавати ту чи іншу лексему у списку. Ось декілька таких функцій:

```
matchKeyword :: String -> Parsec.Parsec [DataToken] () DataToken
matchKeyword keyword = satisfyT (== KeywordToken 0 0 keyword)

matchArithmOp :: String -> Parsec.Parsec [DataToken] () DataToken
matchArithmOp op = satisfyT (== ArithmOpToken 0 0 op)

matchAssignOp :: Parsec.Parsec [DataToken] () DataToken
matchAssignOp = satisfyT (== AssignOpToken 0 0 ":=")

matchDelimiter :: String -> Parsec.Parsec [DataToken] () DataToken
matchDelimiter delimiter =
  satisfyT (== DelimiterSymbolToken 0 0 delimiter)
```

Лістинг 4.5 Аналізатори, що розпізнають лексеми

Аналізатори використовують функцію *satisfyT*, яка приймає логічний вираз і перевіряє чи задовольняє наступна лексема зі списку цьому виразу. Вона має наступний вигляд:

```
satisfyT ::
     (Monad m)
  => (DataToken -> Bool)
  -> Parsec.ParsecT [DataToken] u m DataToken
satisfyT f =
  tokenPrim
    show
    updatePos
    (\c ->
       if f c
         then Just c
         else Nothing)
```

Така ж сама функція визначена у бібліотеці Parsec[7] та використовується у таких аналізаторах як *digit* (лістинг 2.2), проте для нашого випадку вона не підходить, оскільки працює тільки з символами.

Всередині satisfyT використовується функція з Parsec під назвою *tokenPrim*. Вона приймає[13]:

а) функцію для виводу лексеми на екран (це потрібно для повідомлень про помилку);

b) функцію яка оновлює позицію (номери рядка та стовпчика) у вхідному тексті, починаючи з якої ми здійснюємо обробку тексту (у нас ця інформація зберігається у лексемах). Вона приймає поточну позицію, лексему та список лексем, що йдуть після неї, та повертає позицію наступної лексеми;

c) функцію яка визначає чи є лексема зі списку підходящою чи ні. Вона повинна повернути Nothing, якщо ні, або ж *Just* разом з цією лексемою.

Функції з лістингу 4.5 та подібні до них ми будемо використовувати для написання синтаксичних аналізаторів, що розпізнають логічні та арифметичні оператори, вирази, тощо. Наприклад, ось аналізатори, що розпізнають арифметичні оператори:

```
arithmOpBop :: String -> Bop -> Parsec.Parsec [DataToken] () Bop
arithmOpBop op bop = do
  void $ matchArithmOp op
  return bop

mulOrDivOp :: Parsec.Parsec [DataToken] () Bop
mulOrDivOp = arithmOpBop "*" Times <|> arithmOpBop "/" Div

addOrSubOp :: Parsec.Parsec [DataToken] () Bop
addOrSubOp =
  try (arithmOpBop "+" Plus)
  <|>
  try (arithmOpBop "-" Minus)
```

Ось аналізатор, що розпізнає конструкції if, while, for, присвоєння та послідовність виразів у фігурних дужках:

```
stmt :: Parsec.Parsec [DataToken] () Stmt
stmt =
  (do void $ matchKeyword "for"
      forSt) <|>
  (do void $ matchKeyword "while"
      whileSt) <|>
  (do void $ matchKeyword "if"
      ifSt) <|>
  (do var <- iden
      assignSt var) <|> (blockSt <?> "statement")
```

А ось аналізатори, що використовуються у stmt з попереднього лістингу та розпізнають окремо конструкції if, while та for:

```
forSt :: Parsec.Parsec [DataToken] () Stmt
forSt = do
  void $ matchDelimiter "("
  st1 <- stmt
  void $ matchDelimiter ";"
  ex <- expr
  void $ matchDelimiter ";"
  st2 <- stmt
  void $ matchDelimiter ")"
  For st1 ex st2 <$> stmt

whileSt :: Parsec.Parsec [DataToken] () Stmt
whileSt = do
  void $ matchDelimiter "("
  ex <- expr
  void $ matchDelimiter ")"
  While ex <$> stmt

ifSt :: Parsec.Parsec [DataToken] () Stmt
ifSt = do
  void $ matchDelimiter "("
  ex <- expr
  void $ matchDelimiter ")"
  st1 <- stmt
  void $ matchKeyword "else"
  If ex st1 <$> stmt
```

Для того, щоб почати процес синтаксичного аналізу нам потрібно викликати функцію parse, передавши їй на вхід замість списку символів, список лексем:

```
runLexer :: String -> [DataToken]
runLexer = lexer 1 1

startParser :: String -> Either ParseError Stmt
startParser programText = parse stmt "parameter" (runLexer programText)
```

5. Лексичний аналіз з використанням бібліотеки Alex

5.1 Опис бібліотеки Alex

Як зазначалось на початку підрозділа 4.1, бібліотека Parsec дозволяє нам змінювати тип її вхідних даних, що робить можливим використання інших бібліотек поряд з Parsec. Однією з таки бібліотек є Alex.

Alex – це бібліотека, що робить фактично те саме, що ми робили у підрозділі 4.2: вона створює лексичні аналізатори, що повертають список готових лексем[14, 4]. Використовуючи її поряд з Parsec ми звільнюємо себе від необхідності писати лексичні аналізатори, що дає нам можливість повністю зосередитись на синтаксичному аналізі.

Для генерації лексичних аналізаторів Alex використовує *файли лексичної специфікації*, що мають розширення «**.x**» та містять описи лексем у вигляді регулярних виразів[14, 4]. Результатом роботи Alex є *модуль*, що містить всі необхідні для лексичного аналізу функції та функцію *alexScanTokens*, що приймає String та повертає список лексем[14, 4].

```
alexScanTokens :: String -> [Token]
```

Лістинг 5.1 Функція alexScanTokens

5.2 Визначення лексем у файлі лексичної специфікації Alex. Генерація лексичного аналізатора

У файлі лексичної специфікації Alex ми описуємо потрібні лексеми у вигляді регулярних виразів, а також маємо можливість, у фігурних дужках, написати код на мові Haskell, який потім буде скопійовано у згенерований Alex модуль[14, 7-10]. У файлах специфікації цей код може бути тільки на початку та вкінці файлу, при цьому він є необов’язковим.

На початку нашого файлу ми визначимо ім’я, що буде мати згенерований модуль:

```
{
module AlexLexer where
}
```

Код у фігурних дужках буде скопійовано на початок модуля, тому у ньому можна визначити ім’я модуля чи додати необхідні імпорти. Не рекомендується визначати типи чи функції, адже після того як Alex скопіює цей

код у модуль, він додасть після нього власні імпорти[14, 8]. Якщо там буде, наприклад, визначення функції, то виникне помилка.

Після того як ми визначили ім'я модуля нам потрібно визначити *обгортку*, яка теж не є обов'язковою. Обгортка визначає інтерфейс, яким Alex забезпечить згенерований аналізатор, зокрема, визначення обгортки – це найпростіший спосіб отримати функцію alexScanTokens, з лістингу 5.1, у згенерованому модулі[14, 15]. Ми будемо використовувати обгортку під назвою «*posn*», яка, на відміну від базової, відслідковує позиції початку лексем[14, 15].

```
%wrapper "posn"
```

Далі, визначимо декілька макросів, що стануть нам у нагоді, коли ми будемо визначати лексеми.

```
$digit = 0-9
$alpha = [a-zA-Z]
$white = [\ \t \n]
```

Назва макросу починається з символу «$». Після назви, через «=», ми вказуємо регулярний вираз[14, 8]. Таким чином, макрос *$digit* визначає цифру від нуля до дев'яти, *$alpha* визначає великі та малі літери латинського алфавіту, *$white* – символи проміжків та нового рядка.

Далі нам потрібно визначити список правил за якими згенерований лексичний аналізатор буде розпізнавати потрібні лексеми. Список правил починається з рядка вигляду «*ідентифікатор* :-». Ідентифікатор є необов'язковим, головне щоб символи «:-» були наявні на початку списку правил[14, 8].

Кожне правило складається з регулярного виразу, який розпізнає послідовність символів, та функції на мові Haskell яка приймає цю послідовність та повертає лексему[14, 8]. Функції можуть приймати і інші параметр, що залежить від обгортки, яку ми використовуємо. Оскільки ми обрали обгортку posn, функція приймає ще й позицію початку послідовності символів[14, 15].

Визначення правил наведено нижче. Кожна функція, повертає лексему типу DataToken (лістинг 4.1). Таким чином, у результаті роботи згенерованого лексичного аналізатора ми отримаємо список лексем типу DataToken.

```
tokens :-

$white+                         ;
"\r\n"                          ;
[\+ \- \* \/]|"++"              { \(AlexPn _ lin col) op -> ArithmOpToken lin col op }
[\| \& \< \>]|"=="|"<="|">="    { \(AlexPn _ lin col) op -> BoolOpToken lin col op }
":="                            { \(AlexPn _ lin col) op -> AssignOpToken lin col op }
```

```
"true"                          { \(AlexPn _ lin col) _ -> BoolConstToken lin col True }
"false"                         { \(AlexPn _ lin col) _ -> BoolConstToken lin col False }
"bool"|"int"                    { \(AlexPn _ lin col) vType -> VarTypeToken lin col vType }
"if"|"else"|"while"|"for"       { \(AlexPn _ lin col) kWord -> KeywordToken lin col kWord }
[\( \) \{ \} \;]                { \(AlexPn _ lin col) dlmSym -> DelimiterSymbolToken lin col dlmSym }
$digit+                         { \(AlexPn _ lin col) nConst -> NumConstToken lin col (read nConst) }
$alpha[$alpha $digit \_ \']*    { \(AlexPn _ lin col) vName -> VarNameToken lin col vName }
```

Крапка з комою після регулярно виразу є командою ігнорувати, розпізнану виразом, послідовність.

У кінці файлу лексичної специфікації, у фігурних дужках, ми визначаємо тип DataToken, та робимо його членом класів типів Show та Eq. Код у цих дужках повторює код з лістингів 4.1, 4.2 та 4.4.

Після того, як ми завершили написання файлу специфікації, для генерації модуля з аналізатором, нам потрібно викликати Alex та передати йому цей файл. Ми можемо зробити це у терміналі (консолі) наступним чином:

```
C:\>alex AlexLexer.x -o AlexLexer.hs
```

«AlexLexer.x» – це назва файлу лексичної специфікації. «-o AlexLexer.hs» є вказівкою записати код згенерованого модуля у файл «AlexLexer.hs».

Таким чином, ми отримали модуль з готовим лексичним аналізатором, який аналогічно до лексичного аналізатора, який ми написали у підрозділі 4.2, повертає список лексем типу DataToken. Для того щоб викликати згенерований аналізатор, ми імпортуємо модуль та використовуємо функцію з лістингу 5.1, передавши їй на вхід текст, що потрібно розбити на лексеми. Результат виконання ми передаємо Parsec на синтаксичний аналіз.

```
import AlexLexer

startParser :: String -> Either ParseError Stmt
startParser programText = parse stmt "parameter" (alexScanTokens programText)
```

Процес синтаксичного аналізу, є аналогічним тому, що був наведений у підрозділі 4.3, адже ми тільки змінили лексичний аналізатор.

Висновки

У даній роботі було досліджено використання мови Haskell у контексті синтаксичного аналізу та розглянуто її бібліотеки Parsec та Alex, включно з оглядом їх сумісного використання.

Haskell – це не найпопулярніша мова, проте вона не стоїть на місці та постійно розвивається. З часом з'являється все більше проектів написаних повністю на Haskell, наявність інструменту для побудови проектів (Stack) та системи для управління залежностями проекту (Cabal) тільки сприяють цьому. Бібліотеки мають детальну документацію, легко доступні та швидко працюють.

Окрім розглянутих бібліотек для синтаксичного аналізу Haskell пропонує безліч інших, наприклад trifecta, uu-parsinglib, gll, тощо. Деякі з цих бібліотек можуть бути дуже швидкими, проте працюватимуть тільки з текстом, деякі сконцентровані на обробці помилок. Таке різноманіття дає нам можливість знайти бібліотеку, яка буде задовольняти нашим потребам.

Отже, Haskell є гарним вибором для рішення задачі синтаксичного аналізу, оскільки природа цієї мови дозволяє писати досить складні аналізатори використовуючи мінімум коду, при цьому вона має широкий вибір бібліотек, що знайдуть своє застосування під час вирішення великої кількості задач, пов'язаних з синтаксичним аналізом.

Список літератури

1. Compilers, Principles, Techniques and Tools (Second Edition) / [Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman] – Pearson Education, Inc, 2007 – 1-10 с.
2. Компиляторы, интерпретаторы и байт-код [Електронний ресурс] / Алан Джок – 2001 – https://www.osp.ru/cw/2001/06/9339/
3. Основы синтаксического разбора, построение синтаксических анализаторов / Д.И. Соломатин, А.В. Копытин, А.И. Другалев – Воронеж : Воронежский государственный университет, 2014 – 4-6 с.
4. Let's Build A Simple Interpreter. Part 4. [Електронний ресурс] / Ruslan Spivak – 2015 – https://ruslanspivak.com/lsbasi-part4/
5. Recursive Data Structures in Haskell [Електронний ресурс] / Michael Tuttle – 2013 – https://tuttlem.github.io/2013/01/04/recursive-data-structures-in-haskell.html
6. Will Kurt. Get Programming with Haskell / Will Kurt – Shelter Island, NY : Manning Publications Co., 2018 – 121-391 с.
7. parsec: Monadic parser combinators [Електронний ресурс] – 2019 – http://hackage.haskell.org/package/parsec
8. Stephen Diehl. Write You a Haskell – 2015 – 34 с.
9. An introduction to parsing text in Haskell with Parsec [Електронний ресурс] / Nick Chung – 2017 – https://www.cnblogs.com/ncore/p/6892500.html
10. Bryan O'Sullivan. Real World Haskell / Bryan O'Sullivan, John Goerzen, and Don Stewart – O'Reilly Media, Inc., 2008, 1st ed. – 423 с., 469 с.
11. Control.Monad.State.Lazy [Електронний ресурс] – https://hackage.haskell.org/package/mtl-2.2.2/docs/Control-Monad-State-Lazy.html
12. Miran Lipovaca. Learn You a Haskell for Great Good! – 2011 – 198-199 с.
13. Parsec Generally [Електронний ресурс] / Albert Y. C. Lai – https://www.vex.net/~trebla/haskell/parsec-generally.xhtml
14. Alex User Guide / Chris Dornan, Isaac Jones, Simon Marlow – 4-15 с.