Синтаксичний аналіз з використанням бібліотеки Parsec

ВИКОНАВ: СТУДЕНТ ПМ-3 БІКЧЕНТАЄВ М.О.

КЕРІВНИК: К.Ф-М.Н., АСП. ПРОЦЕНКО В.С.

Мета роботи

Метою даної роботи є огляд Parsec – однієї з найпопулярніших бібліотек мови Haskell та аналіз її можливостей. Паралельно з цим, розглядається бібліотека для лексичного аналізу Alex та особливості мови Haskell, які роблять її гарним вибором для вирішення задачі синтаксичного аналізу.

Що таке Parsec?

Parsec – це бібліотека написана мовою Haskell, що використовується для побудови синтаксичних аналізаторів. Процес побудови власного аналізатора складається з комбінування простих синтаксичних аналізаторів. Бібліотеки, що використовують даний принцип, мають назву *комбінатори парсерів (parser combinators)*.

Прості аналізатори

Аналізатори приймають потік символів та повертають значення типу *ParsecT*.

```
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
```

Тип *ParsecT* є результатом роботи кожного аналізатора бібліотеки. *Parsec* – це *синонім типу (type synonym)* для ParsecT.

```
ParsecT s u m a
```

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

Комбінування простих аналізаторів

Для того щоб комбінувати (поєднувати) прості синтаксичні аналізатори найчастіше використовується *do-нотація* мови Haskell.

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

Альтернативою do-нотації є використання *аплікативного стилю.*

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

Обробка помилок. Повідомлення про помилку

Однією з гарних рис бібліотеки Parsec є детальні повідомлення про помилку.

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

Для визначення власних помилок користувач використовує тип *Message*.

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

Обробка помилок. Вивід власної помилки

Генерація повідомлення виду *UnExpect*, використовуючи функцію *unexpected*.

```
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
```

Обробка помилок. Оператор <|>, функція try та комбінатор <?>

Оператор <|> об'єднує аналізатори у ланцюжок. Якщо попередній аналізатор з ланцюжка не зміг обробити символи вхідного потоку, то обробку продовжить наступний. Якщо у аналізаторі з ланцюжку виникає помилка, try її перехоплює та відміняє всі зміни зроблені цим аналізатором.

```
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. Опис знаходить з права від <?>.

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

Керування станом програми

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

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

```
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)
```

Керування станом програми

getState – зчитує дані

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

Аналіз потоку байтів

Функція *pack* перетворює String на ByteString, що є списком, кожний елемент якого займає рівно один байт, при цьому, цей список поділений на частини. Кожна частина займає 64KB.

Після цього ми передаємо ByteString на вхід синтаксичному аналізатору.

```
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
```

Модуль Parsec.Language. Визначення мови

Використовуючи модуль *Parsec.Language* ми можемо генерувати лексичні аналізатори, пишучи мінімум коду.

Для генерації аналізаторів нам потрібно передати визначення мови функції *emptyDef.*

```
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 = words "true false bool int if else while for"
    , Token.reservedOpNames = words "+ - * / < <= > >= := == & | ++"
    }
```

Модуль Parsec.Language. Генерація аналізаторів

```
lexer = Token.makeTokenParser languageDef

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
```

Модуль Parsec.Expr. Таблиця операторів

Parsec.Expr – це модуль який може згенерувати єдиний аналізатор, що буде розпізнавати вирази з усіх потрібних нам операторів. При цьому, цей аналізатор буде враховувати асоціативність та положення цих операторів відносно їх аргументів.

Аналізатор генерується за допомогою функції *buildExpressionParser*, що приймає *таблицю операторів* та аналізатор для лексем, що знаходяться між операторами.

```
operators =
  [ [ Infix
      (reservedOp "*" >> return (\x y -> Op x Times y))
      AssocLeft
    , Infix
      (reservedOp "/" >> return (\x y -> Op x Div y))
      AssocLeft
    ]
    …
  ]
```

Модуль Parsec.Expr. Генерація аналізатора

term – це аналізатор, який розпізнає лексеми, що знаходяться між операторами.

```
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
```

Аналіз готових лексем. Визначення лексем

У визначенні типу ParsecT, на якому ґрунтується бібліотека, тип вхідних даних є параметром, який ми можемо змінювати. Таким чином, замість стрічки ми можемо подавати на вхід бібліотеки список вже готових лексем.

```
data DataToken
  = ArithmOpToken
      { line, column :: Int
      , arithmOperator :: String
      }
  | BoolOpToken
      { line, column :: Int
      , boolOperator :: String
      }
  | AssignOpToken
      { line, column :: Int
      , operator :: String
      }
   …
```

Аналіз готових лексем. Лексичний та синтаксичний аналіз

Лексичний 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 :: Int -> Int -> String -> [DataToken]
```

Прості синтаксичні аналізатори, що розпізнають ту чи іншу лексему у списку лексем.

```
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 ":=")
```

Аналіз готових лексем. Синтаксичний аналіз

Прості аналізатори використовують функцію *satisfyT*, яка приймає логічний вираз і перевіряє чи задовольняє наступна лексема зі списку цьому виразу.

```
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)
```

Аналіз готових лексем. Синтаксичний аналіз

Аналізатор, що розпізнає конструкцію 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
```

Бібліотека Alex

Alex – це бібліотека, що генерує модулі з лексичними аналізаторами. Для генерації модуля Alex використовує *файли лексичної специфікації,* що мають розширення «**.x**».

У цьому файлі 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.

Бібліотека Alex. Файл лексичної специфікації

```
{
module AlexLexer where
}

%wrapper "posn"

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

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 }
…
```

Бібліотека Alex. Генерація та використання аналізатора

Для генерації модуля з аналізатором, нам потрібно викликати Alex та передати йому файл лексичної специфікації. Ми можемо зробити це у терміналі (консолі) наступним чином:

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

Для того, щоб використати згенерований аналізатор, потрібно імпортувати модуль та викликати функцію *alexScanTokens,* яка повертає список лексем.

```
import AlexLexer

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

Висновки

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.

ДЯКУЮ ЗА УВАГУ