Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики

Синтаксичний аналіз знизу-вгору в Haskell. Bottom-up parsing in Haskell

Текстова частина до курсової роботи

за спеціальністю

«Інженерія програмного забезпечення» 121

Керівник курсової роботи

ст. викладач Проценко В.С.

_____

(підпис)

“____” _____ 2024 р.

Виконав студент

Шевченко О. О.

“____” _____ 2024 р.

Київ 2024

Зміст

АНОТАЦІЯ

У роботі розглянуто процес побудови синтаксичного аналізатора знизу-вгору, та певні принципи його роботи. Досліджені інструменти генерації лексичного аналізатора “Alex” та синтаксичного аналізатора “Happy” для функціональної мови програмування Haskell. Результатом роботи є синтаксичний аналізатор для псевдокоду.

Ключові слова: Haskell, синтаксичний аналіз, лексичний аналіз, синтаксичний аналіз знизу-вгору, знизу-вгору, Happy, Alex, формальні граматики, LR граматики, LR, термінал, нетермінал, токен, регулярні вирази.

ВСТУП

Синтаксичний аналіз є одним з напрямків в ком'ютерній лінгвістиці. Синтаксичні аналізатори зокрема синтаксичні аналізатори знизу-вгору часто застосовуються в структурі компіляторів та інтерпретаторів для мов програмування; в текстових редакторах та в IDEs для виокремлення окремих структур тексту; в аналізаторах та оптимізаторах коду; для відловлення помилок при компіляції; для обробки текстів природніх мов; для обробки запитів в базах даних.

Через постійне збільшення об'ємів інформації, зокрема текстової інформації, таких як тексти на формальних мовах (мовах програмування) так і структурованих та частково структурованих текстів на природніх мовах, синтаксичний аналіз текстів буде актуальною проблемою.

Метою роботи було дослідити процес створення синтаксичного аналізатора знизу-вгору в функціональній мові програмування Haskell (далі Haskell). Дослідити інструменти "Alex" та "Happy" для частково автоматизованої побудови лексера та парсера знизу-вгору відповідно. Задачею роботи було написати синтаксичний аналізатор знизу-вгору для псевдомови, яка є спрощеною варіацією мови C.

1 Теорія синтаксичного аналізу

1.1 Формальні граматики

Формальні граматики надають зручний інструментарій для опису та аналізу мов. Формальні граматики - це набір правил виводу, за якими будується речення заданої мови, але вони не можуть перевірити смислове навантаження речення. При описі формальної граматики використовуються такі визначення:

граматика - набір правил, за якими будуються речення мови що обробляється;

нетермінал - символ граматики, який може бути розгорнутий у послідовність терміналів та нетерміналів (далі позначається назвою нетерміналу в кутових дужках);

термінал - виводима частина в мові що обробляється. В формальній граматиці термінали не можуть бути замінені чимось іншим (далі позначається символом або послідовністю символів в одинарних дужках);

продукція - правило граматики, яке описує як замінити символи. Загально можна представити як LEFT -> RIGHT, де в лівій частині обов'язково знаходиться як мінімум один нетермінал (далі позначається (LEFT ::= RIGHT));

похід - послідовність застосувань правил граматики, яка генерує кінцевий рядок терміналів;

початковий символ - в граматиці єдиний нетермінал, з якого походять всі речення;

нульовий символ ε - символ який в представленні мови що обробляється є нічим;

БНФ - спосіб специфікації мов програмування за допомогою формальних граматик та правил продукції з певною формою запису (форма Бекуса-Наура).

Далі буде наведена граматика мови в БНФ яку аналізує розроблена програма.

Але для простоти розгляду потрібно зазначити деякі метасимволи БНФ (позначені в круглих дужках):

(A|B) - вибір одного з варіантів A бо B;

([A]) - A може бути використана 0 або 1 раз;

({A}) - A може бути використана 0 або нескінченно разів.

Граматика мови в БНФ яку аналізує розроблена програма:

<Program> ::= <StatementList>

<StatementList> ::= <Statement> | <StatementList> <Statement>

<Statement> ::= <AssignmentStatement> | <ArrayAssignmentStatement> | <IfStatement> | <WhileLoop> | <FunctionCall> | <Declaration>

<AssignmentStatement> ::= <Identifier> '=' <Expression> ';'

<IfStatement> ::= 'if' '(' <Expression> ')' '{' <StatementList> '}' 'else' '{' <StatementList> '}'

<WhileLoop> ::= 'while' '(' <Expression> ')' '{' <StatementList> '}'

<ForLoop> ::= 'for' '(' [<Expression>] ';' [<Expression>] ';' [<Expression>] ')' '{' <StatementList> '}'

<FunctionCall> ::= <Identifier> '(' <ArgumentList> ')' ';'

<ArgumentList> ::= {<Expression>}

<Declaration> ::= <Type> <Identifier> '[' <Expression> ']' ';' | <Type> <Identifier> ';'

<Type> ::= <Identifier>

<Expression> ::= <Term> | <Expression> '+' <Term> | <Expression> '-' <Term>

<Term> ::= <Factor> | <Term> '*' <Factor> | <Term> '/' <Factor>

<Factor> ::= <Identifier> | <Literal> | '(' <Expression> ')' | <Identifier> '[' <Expression> ']'

<Literal> ::= <Number> | <String>

<Number> ::= {<Digit>}

<String> ::= "" {Character} ""

<Identifier> ::= ('_' | <Letter>) {(<Letter> | <Digit>)}

<Digit> ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

<Character> ::= any printable char

<Letter> ::= any printable letter

1.2 Синтаксичний аналіз

Синтаксичний аналіз - це друга фаза аналізу тексту. Етап після лексичного аналізу, який розділяє тест на узагальнені лесеми (токени). На цій фазі перевіряється синтаксична структура заданого введення, тобто чи введений текст відповідає правильному синтаксису граматики.

При роботі аналізатора будується структура даних, яка називається деревом розбору або синтаксичним деревом. Дерево розбору будується за допомогою попередньо визначеної формальної граматики мови та введеного рядка. Якщо заданий вхідний рядок може бути виведений за допомогою синтаксичного дерева, то буде вважатися що введений рядок має синтаксис заданої граматики. Якщо ні, синтаксичний аналізатор повідомляє про помилку.

Існує кілька типів алгоритмів синтаксичного аналізу:

LL-розбір: це алгоритм розбору зверху-вниз, який починається з кореня дерева розбору і конструює дерево, послідовно розгортаючи нетермінали. LL-розбір відомий своєю простотою і легкістю реалізації.

LR-розбір: це алгоритм розбору знизу-вгору, який починається з листків дерева розбору і конструює дерево, послідовно згортаючи термінали.

LR(1)-розбір: це варіант LR-розбору, який використовує попередній перегляд 1 токена для уточнення граматики.

LALR-розбір: Це варіант LR-розбіру, який використовує зменшений набір символів попереднього перегляду, щоб зменшити кількість станів у LR-аналізаторі.

1.3 LR аналізатор

LR-аналізатор читає вхідний текст зліва направо без повернень назад і створює правосторонній прохід в зворотному напрямку, що дозволяє обробляти лівосторонню рекурсію. Часто назва "LR" супроводжується числовим кваліфікатором, таким як "LR(1)" або іноді "LR(k)". Для уникнення повернень назад або вгадувань, LR-парсер може дивитися вперед на k символів граматики (лексем) у вхідному рядку перед вирішенням, як розбирати попередні символи.

LR-парсери є детермінованими і продукують один правильний розбір без вгадувань чи повернень назад за лінійний час. Це ідеально підходить для мов програмування, але LR-парсери не є найкращим варіантом для природних мов, які вимагають більш гнучких, але відносно повільних методів.

В роботі LR-аналізатора використовується таблиця станів. Це ключовий компонент LR-парсера, який використовується для визначення наступних дій, які парсер повинен виконати залежно від поточного стану та вхідного символу. Ця таблиця визначає, які переходи повинні бути виконані при зчитуванні нових символів та які дії потрібно виконати при кожному кроці аналізу.

Таблиця станів зазвичай представляється у вигляді двовимірного масиву або хеш-таблиці, де рядки представляють стани парсера, а стовпці представляють різні можливі вхідні символи. Для кожної комбінації стану та вхідного символу таблиця містить інформацію про наступний стан та дію, яку потрібно виконати.

Основні типи дій, які можуть бути вказані в таблиці станів, включають наступне:

1. **Зсув (Shift)**: Перехід до нового стану, де один або кілька символів вхідного рядка зчитується у стек аналізатора.
2. **Зведення (Reduce)**: Використання продукції граматики для заміни одного або декількох символів у стеку аналізатора.
3. **Прийняття (Accept)**: Парсер закінчив роботу, оскільки вхідний рядок був успішно розпізнаний.
4. **Помилка (Error)**: Синтаксична помилка, яка виникає, коли немає правила або дії для поточного стану та вхідного символу.

Таблиця станів генерується під час побудови парсера, зазвичай за допомогою алгоритмів побудови LR-автомату. Для різних типів LR-аналізаторів (наприклад, LR(0), SLR, LALR, або LR(1)) можуть використовуватися різні алгоритми для побудови цієї таблиці, що може призводити до різних таблиць станів з різними характеристиками ефективності та потужності.

Створення LR-аналізаторів вручну шляхом побудови таблиці станів є складним завданням, тому зазвичай вони генеруються за допомогою генераторів синтаксичних аналізаторів або компіляторів. Залежно від методу побудови таблиці аналізу, ці аналізатори можуть бути класифіковані як прості LR-аналізатори (SLR), LR-аналізатори з передпереглядом (LALR) або канонічні LR-аналізатори. LALR-аналізатори мають значно більшу потужність у розпізнаванні, ніж SLR-аналізатори. Проте таблиці для SLR-аналізу мають такий же обсяг, що й для LALR-аналізу, тому SLR-аналіз вже застарів. Канонічні LR-аналізатори мають трохи більшу потужність у

розпізнаванні, ніж LALR-аналізатори, але вони вимагають набагато більше пам'яті для таблиць, тому їх використовують дуже рідко.

В роботі буде використовуватися модуль Haskell під назвою “Happy” який є генераторів синтаксичних аналізаторів що дозволяє частково автоматизувати створення LALR аналізатора на Haskell.

2 Імплементація в Haskell

2.1 Лексичний аналізатор, інструмент Alex

Перед тим як почати снтаксичний аналіз спочатку потрібно розбити вхідну стрічку на токени для подальшої обробки. Для спрощеної побудови лексичного аналізатора для Haskell існує інструмент під назвою “Alex”, цей інструмент схожий до інструментів “lex” та “flex” для C та C++ відповідно.

В процесі розбору вхідної стрічки можуть бути певний перетин лексем, наприклад при розпізнаванні стрічки в коді. Для подолання цієї проблеми в в Alex імплементвоаний детермінований скінченний автомат, що дозволить задати стани при розборі стрічки.

Для створення лексичного аналізатора потрібно створити файл з розширенням .x в якому буде використовуватьсяся синтаксис Haskell, який зазначається в фігурних дужках, та свій власний синтаксис Alex. Файл умовно можна поділити на декілька частин:

1 Визначення генеруємого модуля та підключені модулі. В цій частині використовується стандартне для Haskell визначення експортуємого модуля та під’єднанні модулі:

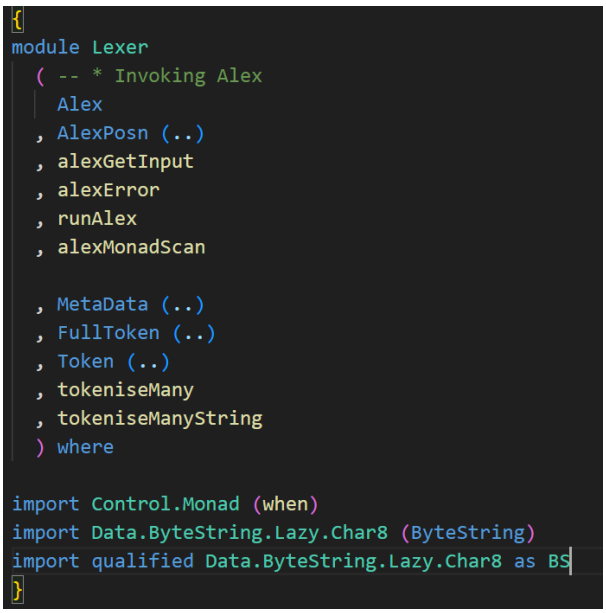

Рис. 1 - Приклад коду для наведеної частини

2 Визначення налаштувань Alex та макроси Regex. Визначення %wrapper зазначає тип коду який буде генерувати Alex: monadUserState - означає генерацію коду при якому в монаді Alex можна буде зберігати додаткову інформацію (додатковий стан) доступну підчас обробки; bytestring - зазначає тип стрічки з якою буде працювати Alex. Щоб задати множину символів для повторного використання використовується шаблон $NAME = CHARACTER_SET; для визначення макроса для Regex використвоується шаблон @NAME = REGEX

Рис. 2 - Приклад налаштування та макроси Alex файлу

3 Визначення токенів та дії яка буде виконуватися при знаходжені. Ця частина визначається позначенням “token:-”. Кожне правило перетворення зазначається по шаблону (<STATE_CODE> LEX { OPERATION }), при відсутності діїї замість {OPERATION } ставиться символ (;). Тепер розберемо цей шаблон - STATE_CODE визначає стан при якому лексер може прочитати лексему та виконати дію; LEX визначає шабллон стрічки яка буде шукатися; OPERATION в цій частині зазначається код на Haskell який буде виконуватися при знаходженні заданого шаблону стрічки, функції якого будуть визначенні далі в файлі, також є функція (andBegin) яка після виконання попередньої функції переведе внутрішній стан скінченого автомату до заданого:

```
tokens :-

<0> $white+ ;

<0>       "/*" { nestMLComment `andBegin` ml_comment }
<0>       "*/" { \_ _ -> alexError "Error: unexpected closing comment" }
<ml_comment> "/*" { nestMLComment }
<ml_comment> "*/" { unnestMLComment }
<ml_comment> .    ;
<ml_comment> \n   ;

<0> "//" .* ;


<0> "if"          { genFullToken IF }
<0> "else"        { genFullToken ELSE }
<0> "while"       { genFullToken WHILE }
<0> "for"       { genFullToken FOR }


<0> "="     { genFullToken ASIG }

<0> "+"     { genFullToken PLUS }
<0> "-"     { genFullToken MINUS }
<0> "*"     { genFullToken MULT }
<0> "/"     { genFullToken DIVIDE }
```

Рис. 3 - Приклад синтаксису побудови структури для аналізу токенів

Рис. 4 - Приклад синтаксису побудови структури для аналізу токенів з використанням макросів

4 Частина функцій обробки. В цій частині зазначаються функції та структури які використовуються при знаходжені токена.

В цій частині кода потрібно визначити структуру даних для збереження токенів. Для подальшої обробки після синтаксичного аналізу може знадобитися певна метаінформація й для прикладу її імплементації було визначенно FullToken який окрім самого типу Token ще має метоінформацію, в якій зберігається початок та кінець в оброблюємій стрічці лексеми

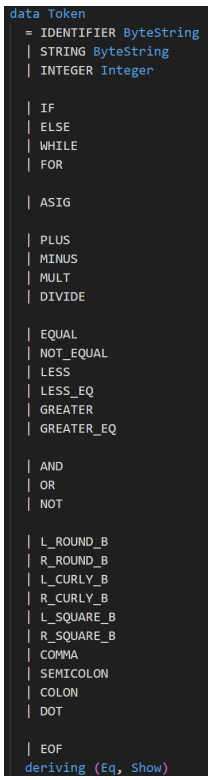

Рис. 5 - Приклад типу для збереження токенів

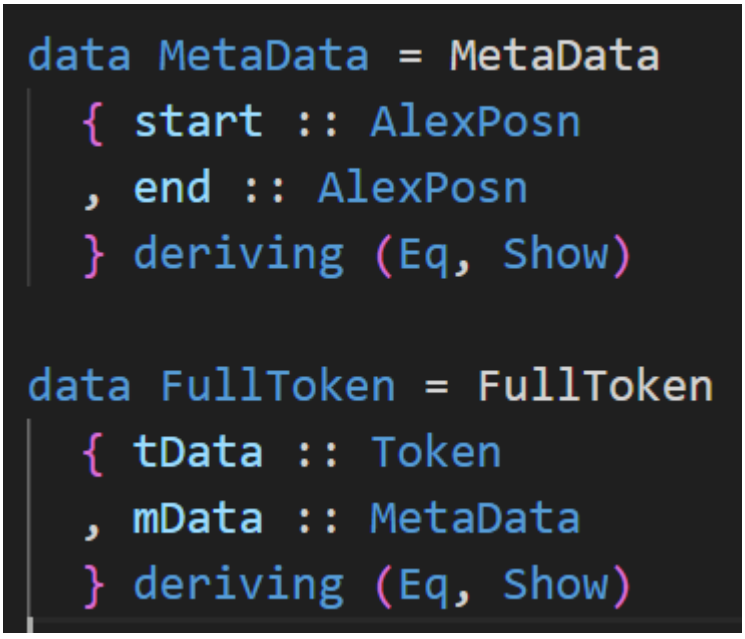

Рис. 6 - Приклад типів для розширеного збереження токенів

Оскільки до цього було зазначено щоб Alex зберігав стан заданий користувачем програмістом, то можна визначити тип в якому він буде зберігатися та початковий стан:

```
data AlexUserState = AlexUserState
  { commentNestLvl :: Int
  , stringStartPos :: AlexPosn
  , stringBufferAcc :: [Char]
  }
```

Рис. 7 - Структури для збереження стану зазначеного користувачем

```
alexInitUserState :: AlexUserState
alexInitUserState = AlexUserState
  { commentNestLvl = 0
  , stringStartPos = AlexPn 0 0 0
  , stringBufferAcc = []
  }
```

Рис. 8 - Функція ініціалізації стану зазначеного користувачем

Кожна функція яка буде використовуватися повина використовувати монаду AlexAction для того щоб мати можливість отримати прочитану стрічку, позицію, довжину прочитаної частини, стан автомату, стан який зазначив користувач програміст.

```
genFullToken :: Token -> AlexAction FullToken
genFullToken tdata inp len =
  pure FullToken
    { tData = tdata
    , mData = genPosMeta inp len
    }
```

Рис. 9 - Прикалад визначення функції для роботи з монадою AlexAction

Використовуючи стан заданий користувачем програмістом та створені функції для доступу до нього можна коректно обробити тип токену String, зберігаючи в буфер поступово зчитаєму стрічку:

```
getState :: Alex AlexUserState
getState = Alex $ \s -> Right (s, alex_ust s)

setState :: AlexUserState -> Alex ()
setState newState = Alex $ \s -> Right (s{alex_ust = newState}, ())

editState :: (AlexUserState -> AlexUserState) -> Alex ()
editState dFunct = Alex $ \s -> Right (s{alex_ust = dFunct (alex_ust s)}, ())
```

Рис. 10 - Функції утиліти для роботи з станом зазначеним користувачем

```
enterString :: AlexAction FullToken
enterString inp@(pos, _, _, _) len = do
  editState $ \state -> state{stringStartPos = pos, stringBufferAcc = '"' : stringBufferAcc state}
  skip inp len

exitString :: AlexAction FullToken
exitString inp@(pos, _, _, _) len = do
  state <- getState
  setState state{stringStartPos = AlexPn 0 0 0, stringBufferAcc = []}
  pure FullToken
    { tData = STRING $ BS.pack $ reverse $ '"' : stringBufferAcc state
    , mData = MetaData (stringStartPos state) (alexMove pos '"')
    }

addToStrAcc :: Char -> AlexAction FullToken
addToStrAcc c inp@(_, _, str, _) len = do
  editState $ \state -> state{stringBufferAcc = c : stringBufferAcc state}
  skip inp len

addCurrentToStrAcc :: AlexAction FullToken
addCurrentToStrAcc inp@(_, _, str, _) len = do
  editState $ \state -> state{stringBufferAcc = BS.head str : stringBufferAcc state}
  skip inp len
```

Рис. 11 Приклад функції обробки складних лексем на прикладі String

Після створення файлу з роширенням .x можна запросити згенерувати .hs файл для отримання готового лексичного аналізатора за допомогою команди (alex NAME.x)

2.2 Синтаксичний аналізатор, інструмент Happy

Для спрощеної побудови синтаксичного аналізатора для Haskell існує інструмент під назвою “Happy”, для його використання потрібно створити файл з роширенням .y з синтаксисом Haskell в фігурних дужках та власним синтаксисом Happy.

Файл можна розбити на декілька частин:

1 Визначення експортуємого модуля та підключені модулі:

```
{
{-# LANGUAGE DeriveFoldable #-}
module Parser
  ( parsePseudoC
  ) where

import Data.ByteString.Lazy.Char8 (ByteString)
import Data.Maybe (fromJust)
import Data.Monoid (First (..))
import qualified Data.ByteString.Lazy.Char8 as BS

import qualified Lexer as L
}
```

Рис. 12 -Приклад визначення модуля та підключених модулів

2 Визначення налаштувань Happy. %name визначає назву функції парсингу та тип який буде повертатися; %tokentype визначає тип токенів які приймаються як листя дерева розбору; %error визначається функція виводу помилок парсингу; %monad визначається спосіб та для роботи в якій монаді буде створений остаточний Haskell код; %lexer визначається звернення до

лексера(токенайзера) та який тип у кінця файлу; %expect 0 визначає очікувати 0 конфліктів:

```
%name parsePseudoC fprogram
%tokentype { L.FullToken }
%error { printError }
%monad { L.Alex } { >>= } { pure }
%lexer { tokeniser } { L.FullToken L.EOF _ }
%expect 0
```

Рис. 13 -Приклад визначення налаштувань файла Happy

3 Визначення синтаксису листків та їх типи. В цій частині за допомогою %token визначається список токенів листків дерева ррозбору отриманих від лексичного аналізатора та їх відповідні типи, запис робиться по шаблону (NAME { HASKELL_TYPE }):

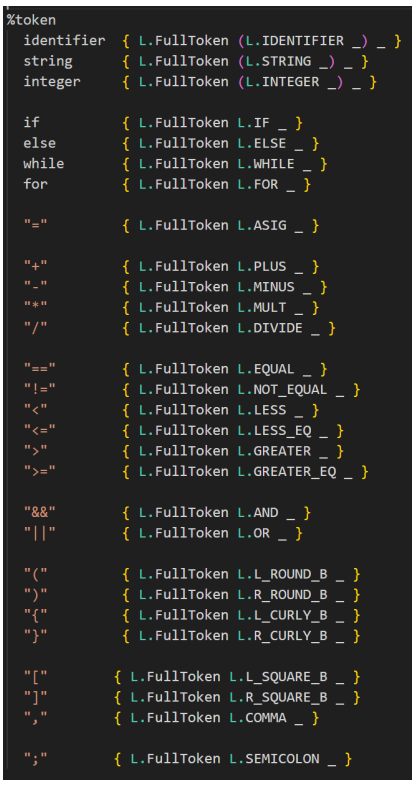

Рис. 14 - Приклад визначення токенів листків дерева розбору

4 Визначення правил асоціативності. В цій частині за допомогою визначень %right %left %nonassoc визначаються правила асоціації які будуть використані при автоматичному спрощенні граматики, яку користувач програміст буде задавати в наступній частині, що є зручном інструментом для розробки граматики. Визначення %% обов’язкове та зазначає перехід до частини задання правил граматики:

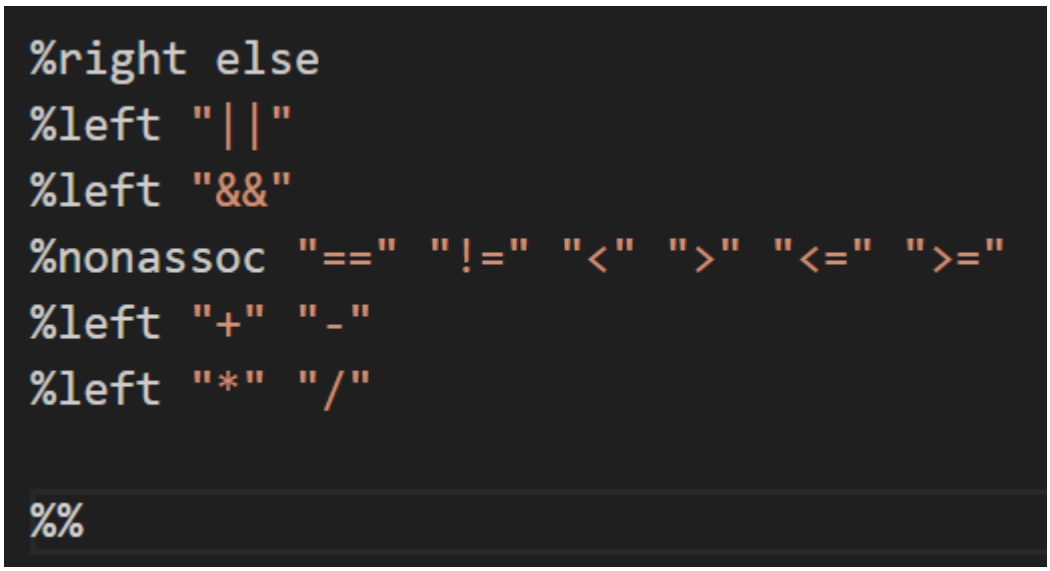

Рис. 15 - Приклад визначення правил асоціативності в файлі Happy

5 Задання правил граматики. В цій частині задається правида граматики та дії які повині бути виконані при згортані, шаблон задання:

(TOKEN_NAME :: { HASKELL_TYPE_OF_TOKEN } : RIGHT { ACTION } ADDITIONALS_RIGHT_PARTS)

Де TOKEN_NAME визначає назву нетерміналу який може бути розкритий в продукцію;

HASKELL_TYPE_OF_TOKEN - це тип Haskell повертаємого токена;

RIGHT - продукція розкриття правила в якій можна використовувати як токени визначені в 3 частині так й нетермінали визначені в поточній;

ACTION - Частина з кодом Haskell яка буде виконуватися при згортані, вона повина повертати тип заданий в HASKELL_TYPE_OF_TOKEN. В цій частині для направлення даних, а саме токенів в частині RIGHT які будуть

згортатися, можна використовувати позначення $N, де N - це послідовний номер токену починаючи з 1ж

ADDITIONALS_RIGHT_PARTS - це додаткові продукції які можна отримати з поточного нетерміналу й мають шаблон (| RIGHT { ACTION })

```
name :: { Name L.MetaData }
  : identifier { convertToken $1 (\meta (L.IDENTIFIER name) -> Name meta name) }

fprogram :: { [Statement L.MetaData] }
  : statementList { $1 }


statementList :: { [Statement L.MetaData] }
  : many(statement) { $1 }

statement :: { Statement L.MetaData }
  : name "=" expression ";"                                                    { AssignmentStatement (mergeMeta (getMeta $1) (L.mData $4)) $1 $3 }
  | name "[" expression "]" "=" expression ";"                                 { ArrayAssignmentStatement (mergeMeta (getMeta $1) (L.mData $7)) $1 $3 $
  | if "(" expression ")" "{" statementList "}"                                { IfStatement (mergeMeta (L.mData $1) (L.mData $7)) $3 $6 }
  | if "(" expression ")" "{" statementList "}" else "{" statementList "}"     { IfElseStatement (mergeMeta (L.mData $1) (L.mData $11)) $3 $6 $10 }
  | while "(" expression ")" "{" statementList "}"                             { WhileStatement (mergeMeta (L.mData $1) (L.mData $7)) $3 $6 }
  | for "(" optional(expression) ";" optional(expression) ";" optional(expression)  ")" "{" statementList "}" { ForStatement (mergeMeta (L.mData $1) (
  | name "(" argumentList ")" ";"                                              { FunctionCall (mergeMeta (getMeta $1) (L.mData $5)) $1 $3 }
  | type name "[" expression "]" ";"                                           { ArrayDeclaration (mergeMeta (getMeta $1) (L.mData $6)) $1 $2 $4 }
  | type name ";"                                                              { VariableDeclaration (mergeMeta (getMeta $1) (L.mData $3)) $1 $2 }

argumentList :: { [Expression L.MetaData] }
  : sepBy(expression, ",") { $1 }
```

Рис. 16 - Приклад визначення правил та продукції в файлі Happy

Також в цій частині можна задати утиліти, які можуть приймати токени як аргументи. Це дозволяє зробити зручнішим задання деяких шаблонів граматики. Були імплементовані утиліти optional, яка дає можливість задати опціональне значення в продукції яке може бути а може й ні (відповідно повертаючи Haskell тип даних асоційований з цим); many_rev та many які дозволяють легко задати лівосторонню рекурсію, й отримати від 0 до нескінченності токенів; sepBy_rev та SepBy утиліти схожі на many але дозволяють ще зазначити токен роздільника між прочитаними токенами

```
optional(p)
  :   { Nothing }
  | p { Just $1 }

many_rev(p)
  :               { [] }
  | many_rev(p) p { $2 : $1 }

many(p)
  : many_rev(p) { reverse $1 }

sepBy_rev(p, sep)
  :                          { [] }
  | sepBy_rev(p, sep) sep p { $3 : $1 }
  | p { [$1] }

sepBy(p, sep)
  : sepBy_rev(p, sep) { reverse $1 }
```

Рис. 17 - Приклад визначення утиліт в файлі Happy

В 4 частині файлу зазначався тип асоціативності це допомагає не створювати додаткових нетерміналів та спростити складання граматики зменшуючи вірогідність помилки. В наведеній частині кода можна побачити що без того визначення було б отримано конфлікт, але з ним його нема й інструментарій Happy сам побудує лівоасоціативне розкриття при спрощені правил

```
expression :: { Expression L.MetaData }
  : readyData                          { ExprReadyData (getMeta $1) $1 }
  | expression "+" expression          { ExprBinOper (mergeMeta (getMeta $1) (getMeta $3)) $1 (Plus (L.mData $2)) $3 }
  | expression "-" expression          { ExprBinOper (mergeMeta (getMeta $1) (getMeta $3)) $1 (Minus (L.mData $2)) $3 }
  | expression "*" expression          { ExprBinOper (mergeMeta (getMeta $1) (getMeta $3)) $1 (Mult (L.mData $2)) $3 }
  | expression "/" expression          { ExprBinOper (mergeMeta (getMeta $1) (getMeta $3)) $1 (Divide (L.mData $2)) $3 }

  | expression "==" expression         { ExprBinOper (mergeMeta (getMeta $1) (getMeta $3)) $1 (Equal (L.mData $2)) $3 }
  | expression "!=" expression         { ExprBinOper (mergeMeta (getMeta $1) (getMeta $3)) $1 (Not_Equal (L.mData $2)) $3 }
  | expression "<" expression          { ExprBinOper (mergeMeta (getMeta $1) (getMeta $3)) $1 (Less (L.mData $2)) $3 }
  | expression "<=" expression         { ExprBinOper (mergeMeta (getMeta $1) (getMeta $3)) $1 (Less_Eq (L.mData $2)) $3 }
  | expression ">" expression          { ExprBinOper (mergeMeta (getMeta $1) (getMeta $3)) $1 (Greater (L.mData $2)) $3 }
  | expression ">=" expression         { ExprBinOper (mergeMeta (getMeta $1) (getMeta $3)) $1 (Greater_Eq (L.mData $2)) $3 }

  | expression "&&" expression         { ExprBinOper (mergeMeta (getMeta $1) (getMeta $3)) $1 (AndOper (L.mData $2)) $3 }
  | expression "||" expression         { ExprBinOper (mergeMeta (getMeta $1) (getMeta $3)) $1 (OrOper (L.mData $2)) $3 }
```

Рис. 18 - Приклад правил та продукції в файлі Happy на які впливає асоціативність

6 Визначення типів нетермінальних токенів та функцій обробки. В цій частині визначаються функції які будуть використані при згортані та Haskell типи токенів

```haskell
printError :: L.FullToken -> L.Alex a
printError _ = do
  (L.AlexPn _ line column, _, _, _) <- L.alexGetInput
  L.alexError $ "Parse error at line " <> show line <> ", column " <> show column

tokeniser :: (L.FullToken -> L.Alex a) -> L.Alex a
tokeniser = (=<< L.alexMonadScan)

convertToken :: L.FullToken -> (L.MetaData -> L.Token -> a) -> a
convertToken (L.FullToken tdata mdata) convRule = convRule mdata tdata

getMeta :: Foldable f => f a -> a
getMeta = fromJust . getFirst . foldMap pure

mergeMeta :: L.MetaData -> L.MetaData -> L.MetaData
mergeMeta (L.MetaData a_startPos _) (L.MetaData _ b_endPos) = L.MetaData a_startPos b_endPos
```

Рис. 19 - Приклад функцій в файлі Happy

```haskell
data Name a = Name a ByteString deriving (Foldable, Show)

data Fprogram a = Fprogram a [Statement a] deriving (Foldable, Show)

data Statement a
  = AssignmentStatement a (Name a) (Expression a)
  | ArrayAssignmentStatement a (Name a) (Expression a) (Expression a)
  | IfStatement a (Expression a) [Statement a]
  | IfElseStatement a (Expression a) [Statement a] [Statement a]
  | WhileStatement a (Expression a) [Statement a]
  | ForStatement a (Maybe (Expression a)) (Maybe (Expression a)) (Maybe (Expression a)) [Statement a]
  | FunctionCall a (Name a) [Expression a]
  | ArrayDeclaration a (Type a) (Name a) (Expression a)
  | VariableDeclaration a (Type a) (Name a)
  | SemicolonLine a
  deriving (Foldable, Show)

data Type a =
  Type a (Name a)
  | BType a (Type a)
  deriving (Foldable, Show)

data ReadyData a
  = Variable a (Name a)
  | Bracket a (Expression a)
  | ArrayAccess a (Name a) (Expression a)
  | DInteger a Integer
  | DString a ByteString
  deriving (Foldable, Show)
```

Рис. 20 - Приклад типів в файлі Happy

Після створення файлу з роширенням .н можна запросити згенерувати .hs файл для отримання готового лексичного аналізатора за допомогою команди (happy NAME.y), також використавши додатковий аргумент -i Happy згенерує ще info файл, де в відносно зручному для розуміння людині форматі буде зазначено всі подробиці аналізатора: граматика, термінали, нетермінали, таблиця станів, використані правила; якщо десь виник конфлікт, або не використаті це буде описано в цьому файлі

Після підключення обох згенерованих .hs файлів можна використати парсер. Для прикладу такий код може обробити розроблений парсер за допомогою поєднання функцій (де TEXT стрічка яка буде оброблюватися, BS.pack потрібно щоб привести String до ByteString для уніфікації)

L.runAlex (BS.pack "TEXT") parsePseudoC

```
/*
comment
/*
in comment
*/
*/
int a;
a = 10;
if (a == (1 + 9)) {
    if (a && 2 * 3){
        fun(1 , a,b , \"Some String\\n\", 1 + 2 * 3 + 4 * 5 + 6);
        char b;
        char arr[10];
        arr[0] = 1 + (3 * 10) * 10;
        b = arr[0] ;
    }
}
else {
    int i;
    for( ; i <= 10; ){

        funny();
    }
}
```

Рис. 21 - Приклад тексту який може обробити розроблений парсер

На виході одержуємо результат синтаксичного аналізу кода:

Рис. 22 - Результат наданий після виклику аналізатора

Висновки

Був досліджений процес роботи синтаксичного аналізатора знизу-вгору; досліджені інструменти для мови програмування Haskell “Alex”, “Happy” для генерації лексичного аналізатора та синтаксичного аналізатора (LALR). Розроблено програму на Haskell для парсингу псевдомови схожої на C, граматика якої зазначена в першому розділі.

Інструмент “Alex” пропонує зручний для користувача спосіб створення лексичного аналізатора. Можливість написати відносно абстрактно спосіб обробки токенів при певних станах, значно спрощує написання коду та зменшує можливу кількість помилок. Певним мінусом є потреба вивчення нового синтаксису з відносно малою документацією.

Інструмент “Happy” пропонує зручний для користувача спосіб створення синтаксичного аналізатора. Можливість налаштувати певні аспекти генеруємого кода та можливість модульного використання лексичних аналізаторів (не підв’язаний під якийсь) робить його універсальним. Можливість задати асоціативність надає можливість полегшити опис граматики формальної мови й прицьому оптимізувати її для лівостороньої рекурсії. Можливість створення утиліт для згортання нетерміналів та направлення токенів у код Haskell також є зручними доповненнями які можуть призвести до оптимізації аналізу (лівостороння рекурсія). Головним мінусом також як й в “Alex” є потреба розібратися з синтаксисом використовуючи малу документацію, але в цей раз кращу.

Список використаної літератури

1. Компиляторы - принципы, технологии и инструментарий / А. В. Ахо та ін. 2018-те вид. 2. 1186 с.

2. Contributors to Wikimedia projects. LR parser - Wikipedia. *Wikipedia, the free encyclopedia*. [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/LR_parser

3. Heitor Toledo Lassarote de Paula. Parsing With Haskell (Part 1): Lexing With Alex. *Parsing With Haskell (Part 1): Lexing With Alex*. [Електронний ресурс] – Режим доступу до ресурсу: https://serokell.io/blog/lexing-with-alex

4. Heitor Toledo Lassarote de Paula. Parsing With Haskell (Part 2): Parsing With Happy. *Parsing With Haskell (Part 2): Parsing With Happy*. [Електронний ресурс] – Режим доступу до ресурсу: https://serokell.io/blog/parsing-with-happy

5. Introduction to Syntax Analysis in Compiler Design - GeeksforGeeks. *GeeksforGeeks*. [Електронний ресурс] – Режим доступу до ресурсу: https://www.geeksforgeeks.org/introduction-to-syntax-analysis-in-compiler-design/

6. Johnson, M., & Zelenski, J. (2012, June 29). CS143 Handout 08: Formal Grammars. Handout presented during the Summer 2012 session. [Електронний ресурс] – Режим доступу до ресурсу: https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/080%20Formal%20Grammars.pdf

7. Alex User Guide – Alex documentation. *Alex User Guide – Alex documentation*. [Електронний ресурс] – Режим доступу до ресурсу: https://haskell-alex.readthedocs.io/en/latest/index.html

8. Welcome to Happy's documentation! – Happy documentation. *Welcome to Happy's documentation! – Happy documentation*. [Електронний ресурс] – Режим доступу до ресурсу: https://haskell-happy.readthedocs.io/en/latest/