Министерство науки и высшего образования Российской Федерации

федеральное государственное бюджетное образовательное учреждение
высшего образования
«Иркутский государственный университет»
(ФГБОУ ВО «ИГУ»)

Институт математики и информационных технологий

Кафедра информационных технологий

ОТЧЕТ

По курсовой работе

ВЫЧИСЛЕНИЕ ВЫРАЖЕНИЙ, ПРЕДСТАВЛЕННЫХ В СТРУКТУРЕ КУРСОВОЙ РАБОТЫ ПЕТУНИНА ДЕНИСА

Студентки 3 курса очного отделения

Группы 01.03.02 «Прикладная математика и информатика»

Барнаковой Веры Андреевны

Руководитель:

к.т.н., доцент

Черкашин Евгений Александрович

Иркутск-2022

Содержание

1. Введение

Haskell -это компьютерный язык программирования. В частности это стандартизированный, ленивый, чистый функциональный язык программирования общего назначения, довольно отличающийся от остальных языков программирования.

Назван в честь Хаскелля Брукса Карри, чьи работы в области математической логики служат основой функциональных языков..

Язык специально разработал, чтобы оперировать широким спектром функций, от численных до символических. Поэтому Haskell имеет ясный синтаксис и богатое разнообразие встроенных типов данных, включая длинные целые и рациональные числа, а также обычные целые числа, числа с плавающей запятой и булевые типы.

2. Реализация вычислений выражений

Рассмотрим реализацию вычисление выражений, генерируемых программой Петунина Дениса в файл вывода(out.txt). Вычисление выполняется в 2 этапа: лексический анализ выполняется *сканером*, затем выход сканера подается на вход *калькулятору*, который выполняет вычисление выражения. Сканер разбивает входную строку символов на список лексем. Лексемы (тип Token) представляют собой объекты, содержащие 3 поля:
1) тип лексемы (представляется перечислимым типом Lexeme) — это 3 знака арифметических операций, открывающая и закрывающая скобки, имя функции, численная константа, неправильная лексема;

2) ее текст (тип String);

3) ее численное значение (тип Double), которое используется для численных констант.
Вот как описываются эти типы:

```
data Lexeme = Number | Name |
            Sum | Prod | Div |
             LPar | RPar | EndLine | Illegal
             deriving (Eq, Ord, Show)

data Token = Token {
   lexeme :: Lexeme,
   text :: String,
   value :: Double
} deriving (Show, Eq, Ord)
```

Далее мы напишем функцию сканнера для того, чтобы полученная на вход текстовая строка, разбивала ее на лексемы и на выходе выдала список лексем. Ее реализация использует хвостовую рекурсию с аккумулятором:

```
scanner :: String -> [Token]
scanner str = scanner' [] str
scanner' :: [Token] -> String -> [Token]
scanner' acc "" = reverse acc
scanner' acc str =
   let (token, rest) = spanToken str in
   if (lexeme token) == EndLine then reverse acc
   else scanner' (token:acc) rest
```

Реализация функции данной функции основана на использовании другой, ранее написанной функции «*spanToken*», которая выделяет лексему и возвращает пару(кортеж пары(tuple)), которая состоит из хеддера в виде выделенной лексему и хвоста в виде оставшийся части списка.

```
spanToken :: String -> (Token, String)
spanToken "" = (Token EndLine "" 0, "")
spanToken str@(h:t)
   | isSpace h        =
     let (u, v) = span isSpace str
     in spanToken v
   | otherwise        =
     if h == "Sum" then (Token Sum [h] 0, t)
     else if h == "Prod" then (Token Prod [h] 0, t)
     else if h == "Div" then (Token Div [h] 0, t)
     else if h == '(' then (Token LPar [h] 0, t)
     else if h == ')' then (Token RPar [h] 0, t)
     else if isAlpha h then
        let (u, v) = span isAlphaNum str
        in (Token Name u 0, v)
     else if isDigit h then
        let (u, v) = span (\ c -> isDigit c || c == '.') str
            n = read u::Double
        in (Token Number u n, v)
     else (Token Illegal "Illegal" 0, t)
```

Вот и все функции, которые нужны нам для лексического анализа, осталось написать функцию, которая будет принимать на вход лексемы, и на основе данных лексем проводить арифметические операции.

```
calculate :: [Token] -> Double
calculate tokens = calculate' [] tokens

calculate':: [Double] -> [Token] -> Double
calculate' stack [] = head stack
calculate' stack (token:rest)
   | isOperation (lexeme token) && length stack >= 2 =
       let
       (y:x:st) = stack
       res = if (lexeme token) == Sum then x + y
             else if (lexeme token) == Prod then x*y
```

```
            else if (lexeme token) == Div then x/y
            else error "Illegal operation"
        in calculate' (res:st) rest
    | isOperation (lexeme token) && length stack < 2 =
        error "Stack underflow"
    | (lexeme token) == Number =
        calculate' ((value token):stack) rest
```

Далее напишем функцию, которая будет передавать отсканированную строку в функцию calculate.

```
calcFormula  :: String -> Double
calcFormula formula =
    calculate $ scanner formula
```

3. Тестирование алгоритма

Теперь протестируем работу данной программы на сгенерированном программой Петунина Дениса файле out.txt. Для этого напишем main, где и вызовем необходимые функции.

```
import Control.Exception
import ExpressionCalc
f::FilePath -> String
f fileOUT =
   line <- readFile fileOUT
main = do
   if null line
   then return ()
   else do
      putStrLn ("X = 2, RESULT = " ++ (show (calcFormula & f "./out.txt")))
`catch` handler

      main
   where
      handler :: SomeException -> IO ()
      handler ex = putStrLn $ "Exception: " ++ show ex
```

File in.txt -> Log (Sum (X) (Coef (2.0)))

Что соответствует формуле: $Ln(x+2)'$

File out.txt -> Div (Coef 1.0) (Sum X (Coef 2.0))

Что соответствует формуле: $\frac{1}{x+2}$

Вывод в консоль, после работы программы, по вычислению выражений:

X = 2, RESULT = 0.25

```
X = 2, RESULT = 0.25
```

ЗАКЛЮЧЕНИЕ

В ходе выполнения данной курсовой работы мы получили реализацию вычисления выражений, представленных в структуре работы Петунина Дениса. Целью данной курсовой работы было показать навыки работы с языком Haskell.