

Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и технологий
Высшая школа программной инженерии

Курсовой проект по дисциплине «Функциональное
программирование»
Взаимодействие с драйверами устройства

Выполнил студент гр. 3530904/80001

Хисматуллин К. И.

Руководитель

Лукашин А. А.

1 Задание

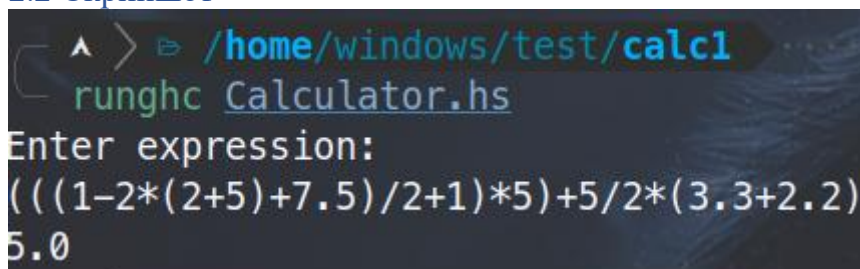
Калькулятор, поддерживающий простые арифметические операции над вещественными числами, приоритеты и скобки.

2 Ход работы

2.1 Алгоритм решения

Выражение, заданное в инфиксной нотации, фильтруется на допустимые символы (цифры, скобки, точка и *, /, +, -). Далее из исходного выражения формируется массив операторов и операнд, при встрече открывающейся скобки формируется подмассив из аналогичной конструкции. Производится поиск унарных минусов и вычитания, заменяется умножением на (-1) и сложением обратного числа соответственно. Операции умножения и деления выносятся в отдельные подмассивы для приоритета операций. В дальнейшем данный массив рекурсивно вычисляется (операнд, оператор, операнд). Код программы приведен в приложении.

2.2 Скриншот



```
^ > /home/windows/test/calcl1
runghc Calculator.hs
Enter expression:
(((1-2*(2+5)+7.5)/2+1)*5)+5/2*(3.3+2.2)
5.0
```

3 Вывод

В ходе работы был изучен функциональный подход к программированию, который значительно отличается от стандартного императивного подхода. Изучены некоторые основные алгоритмы, используемые в функциональном программировании и произведена работа с ними.

Приложение

Calculator.hs

```
calculator :: (Read a, Num a, Fractional a) => String -> a
calculator expression = evaluate $ restructure $ parse $ clean expression
```

```
data Operators = Plus
               | Minus
               | Multiply
               | Division
               deriving (Show, Eq)
```

```
data SyntacticalElement a = Operator Operators
                          | Operand a
                          | SubList [SyntacticalElement a]
                          deriving (Show)
```

```
number = ['0'..'9'] ++ ['.']
operator = ['+', '-', '*', '/']
open_brackets = ['(', '[']
close_brackets = [')', ']']
brackets = open_brackets ++ close_brackets
allowed_chars = number ++ operator ++ brackets
```

```
clean expression = filter (\c -> any (==c) allowed_chars) expression
```

```
parse :: (Read a, Num a, Fractional a) => String -> [SyntacticalElement a]
parse "" = []
parse expression = element : (parse rest)
    where (element, rest) = get_next_element expression
```

```
get_next_element :: (Read a, Num a, Fractional a) => String -> (SyntacticalElement a, String)
get_next_element s@(first:_)
    | is_open_bracket first = (to_sublist content, rest_b)
    | is_operator first     = (to_operator operator, rest_o)
    | is_number first       = (to_number number, rest_n)
    | is_close_bracket first = error "Unexpected closing bracket!"
    | otherwise             = error $ "Invalid Expression: \"\" ++ s ++ \"\""
    where (number, rest_n) = span is_number s
          (operator, rest_o) = span is_operator s
          (content, rest_b) = parse_bracket s
```

```
is_operator char = any (==char) operator
is_number char = any (==char) number
is_open_bracket char = any (==char) open_brackets
is_close_bracket char = any (==char) close_brackets
```

```
parse_bracket expression = (remove_brackets content, rest)
    where content = bracket_content expression 0
          rest = drop (length content) expression
```

```

bracket_content [] 0 = []
bracket_content (first:rest) counter
  | is_open_bracket first = first : bracket_content rest (counter + 1)
  | is_close_bracket first = first : bracket_content rest (counter - 1)
  | counter == 0 = []
  | otherwise = first : bracket_content rest counter
bracket_content _ _ = error "No closing bracket!"

remove_brackets s = tail $ init s

to_operator "+" = Operator Plus
to_operator "-" = Operator Minus
to_operator "*" = Operator Multiply
to_operator "/" = Operator Division
to_operator s = error $ "Unknown operator: " ++ s
to_number s = Operand (read s)
to_sublist s = SubList $ parse s

restructure list = group_by_precedence $ resolve_prefix_minus list

resolve_prefix_minus (a:(Operator Minus):xs) = resolve_prefix_minus $ a : (Operator Plus) :
(Operand (-1)) : (Operator Multiply) : xs
resolve_prefix_minus ((Operator Minus):xs) = (Operand (-1)) : (Operator Multiply) :
resolve_prefix_minus xs
resolve_prefix_minus ((SubList l):xs) = (SubList $ resolve_prefix_minus l) :
resolve_prefix_minus xs
resolve_prefix_minus (x:xs) = x : resolve_prefix_minus xs
resolve_prefix_minus [] = []

group_by_precedence list = group_by_operators (\o -> o == Multiply || o == Division) list

group_by_operators :: (Num a, Fractional a) => (Operators -> Bool) -> [SyntacticalElement a] ->
[SyntacticalElement a]
group_by_operators _ [] = []
group_by_operators f [SubList [a, o, b]] = [SubList $ (group_by_operators f [a]) ++ [o] ++
(group_by_operators f [b])]
group_by_operators f (a:(Operator o):b:rest)
  | f o = group_by_operators f $ (SubList [a, Operator o, b]) : group_by_operators f rest
group_by_operators f ((SubList a):rest) = (SubList $ group_by_operators f a) :
group_by_operators f rest
group_by_operators f (a:rest) = a : group_by_operators f rest

evaluate :: (Num a, Fractional a) => [SyntacticalElement a] -> a
evaluate (a:(Operator o):rest)
  | o == Plus = (c + d)
  | o == Minus = (c - d)
  | o == Multiply = (c * d)
  | o == Division = (c / d)
  where c = evaluate [a]
        d = evaluate rest
evaluate [(Operand n)] = n
evaluate [(SubList l)] = evaluate l

```

```
evaluate _ = error "Evaluation failed!"
```

```
main :: IO()
```

```
main = do
```

```
    putStrLn $ "Enter expression:"
```

```
    str <- getLine
```

```
    print ( calculator str )
```