Funkcjonalne Rozwiązywanie Problemów
Odwrotna notacja polska (ONP)

Emil Śmiech, Amelia Królczyk

December 2020

Wstęp

W naszej pracy spróbujemy rozwiązać problem związany z odwrotną notacją polską, używając technik programowania funkcyjnego.

Zwykle, gdy pracujemy z wyrażeniami algebraicznymi, piszemy je w sposób wrostkowy. Na przykład piszemy $10-(4+3)\cdot 2$. Dodawanie, mnożenie i odejmowanie są operatorami wrostków, podobnie jak wrostkowe funkcje w Haskellu (+ ‘elem‘, i tak dalej). Taką formę możemy łatwo przeanalizować w głowie. Wadą jest to, że musimy używać nawiasów aby oznaczać pierwszeństwo.

Innym sposobem zapisywania wyrażeń algebraicznych jest użycie odwrotnej notacji polskiej - ONP. W ONP operatory występują po liczbach - nie są wciśnięte między nie. Zamiast pisać $4+3$, piszemy 4 3+. Ale jak piszemy wyrażenia zawierające kilka operatorów? Na przykład, jak napisalibyśmy wyrażenie, które dodaje 4 i 3, a następnie mnoży je o 10? To proste: 4 3 + 10·. Ponieważ 4 3+ to 7,więc dalsza część wyrażenia to to samo co 7 10·.

Algorytm obliczania wyrażeń w ONP

Wyrazenia w tej notacji omawiamy od ich lewej strony do prawej. Aby łatwiej zrozumiec obliczanie w ONP, możemy wyobrazić sobie stos liczb. Za kazdym razem gdy napotykamy liczbe, odkładamy ją na stos. Gdy napotykamy operator, stosujemy go na dwóch ostatnich dołożonych liczbach.

Struktura algorytmu

* Dla wszystkich symboli z wyrażenia ONP:
    * jeśli i-ty symbol jest liczbą, to odkładamy go na stos,
    * jeśli i-ty symbol jest operatorem to:
        * zdejmujemy ze stosu jeden element (ozn. a),
        * zdejmujemy ze stosu kolejny element (ozn. b),
        * odkładamy na stos wynik działania operatora na oba elementy.
    * jeśli i-ty symbol jest funkcją to:
        * zdejmujemy ze stosu oczekiwaną liczbę parametrów (ozn. $a_1...a_n$)
        * odkładamy na stos wynik funkcji dla parametrów $a_1...a_n$
* Zdejmujemy ze stosu wynik.

Działanie algorytmu na przykładowym równaniu

Dla wyrażenia 10 4 3 + 2 * -
1. Liczbę 10 wkładamy na stos, więc składa się on teraz z 10.
2. 4 również wkładamy na stos. Mamy teraz: 10, 4.

3. Robimy to samo z 3, stos wynosi teraz: 10, 4, 3.
4. Napotykamy operator: +. Zdejmujemy dwie najwyższe liczby ze stosu (więc zostaje tylko 10), dodajemy te liczby do siebie i odkładamy wynik na stos. Składa się teraz z 10, 7.
5. Wkładamy 2 do stosu i wygląda on tak: 10, 7, 2.
6. Napotykamy operator mnożenia. Zdejmujemy ze stosu 7 i 2, mnożymy je i umieszczamy wynik na stosie. Mnożenie 7 i 2 daje 14 - stos wynosi teraz 10, 14.
7. Na koniec mamy minus. Zdejmujemy 10 i 14 ze stosu, odejmujemy 14 od 10, i wkładamy z powrotem.
8. Liczba na stosie wynosi teraz -4. Ponieważ nie ma więcej liczb i operatorów w naszym wyrażeniu, jest to już wynik końcowy.

Program do obliczania metodą ONP w Haskellu

Kod programu

```haskell
{-# LANGUAGE BangPatterns #-}
import Data.String
import System.IO

data Token = TNum Int | TOp Operator
data Operator = Add | Sub | Mul | Div

main :: IO ()
main = do
    line <- getLine
    let tokens      = tokenise line
        (numc, opc) = countTok tokens
        !junk       =
            if numc == opc + 1
                then ()
                else error "Not a correct expression."
    print $ eval [] tokens


tokenise :: String -> [Token]
tokenise = map str2tok . words

eval :: [Int] -> [Token] -> Int
eval (s:_) []                   = s
eval stack (TNum t:ts)          = eval (t : stack) ts
eval (x:y:stacknoxy) (TOp t:ts) = eval (applyOp t y x : stacknoxy) ts

str2tok :: String -> Token
str2tok tkn@(c:_)
    | c `elem` ['0'..'9'] = TNum (read tkn :: Int)
    | otherwise = TOp $ case tkn of
        "+" -> Add
        "-" -> Sub
        "*" -> Mul
        "/" -> Div
        _   -> error $ "No such operator " ++ tkn
```

```haskell
applyOp :: Operator -> Int -> Int -> Int
applyOp Add a b = a + b
applyOp Sub a b = a - b
applyOp Mul a b = a * b
applyOp Div a b = a `div` b

countTok :: [Token] -> (Int, Int)
countTok [] = (0, 0)
countTok (t:ts) =
    let (x, y) = case t of
            TNum _ -> (1, 0)
            _      -> (0, 1)
    in (x, y) `addPair` countTok ts

addPair :: (Num a, Num b) => (a, b) -> (a, b) -> (a, b)
addPair (x, y) (z, w) = (x + z, y + w)
```

Działanie programu na przykładzie

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\Admin\Desktop\ONP> ghc ONP.hs
[1 of 1] Compiling Main             ( ONP.hs, ONP.o )
Linking ONP.exe ...
PS C:\Users\Admin\Desktop\ONP> ghci *
GHCi, version 8.10.2: https://www.haskell.org/ghc/  :? for help
Warning: ignoring unrecognised input `ONP.exe'
Warning: ignoring unrecognised input `ONP.hi'
Ok, one module loaded.
Prelude Main> :l ONP.hs
Ok, one module loaded.
Prelude Main> main
10 4 3 + 2 * -
-4
Prelude Main>
```