ODWROTNA NOTACJA POLSKA

REVERSE POLISH NOTATION

Emil Śmiech, Amelia Królczyk

Spis treści

Czym jest odwrotna notacja polska?

Odwrotna notacja polska została opracowana przez australijskiego naukowca Charlesa Hamblina jako „odwrócenie” beznawiasowej notacji polskiej Jana Łukasiewicza na potrzeby zastosowań informatycznych.
W ONP znak wykonywanej operacji umieszczony jest po operandach, a nie pomiędzy nimi jak w konwencjonalnym zapisie algebraicznym lub przed operandami jak w zwykłej notacji polskiej. Zapis ten pozwala na całkowitą rezygnację z użycia nawiasów w wyrażeniach, jako że jednoznacznie określa kolejność wykonywanych działań. Np.: działanie:

10 - (4 + 3) * 2,

zapiszemy w ONP jako:

10 4 3 + 2 * -.

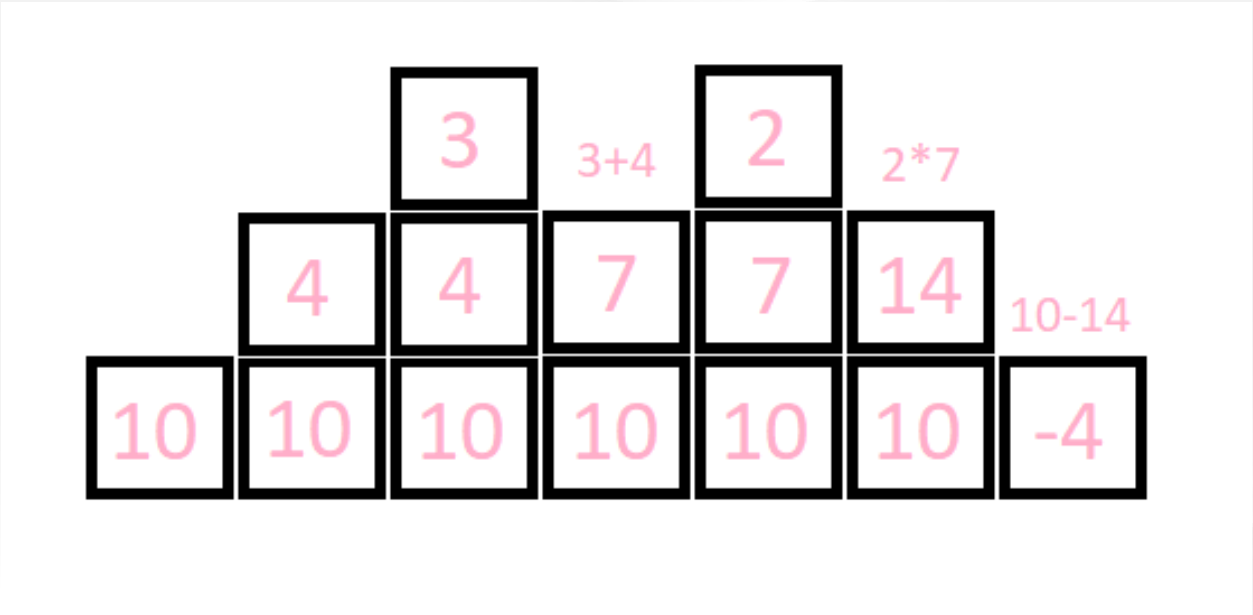
Algorytm obliczania wyrażeń w ONP

Wyrażenia w tej notacji omawiamy od ich lewej strony do prawej. Aby łatwiej zrozumieć obliczanie w ONP, wyobraźmy sobie stos liczb.
Za każdym razem gdy napotykamy liczbę, odkładamy ją na stos. Gdy napotykamy operator, stosujemy go na dwóch ostatnich dołożonych liczbach.

* 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.

Algorytm obliczania wyrażeń w ONP

Jako przykład weźmy sobie wspomniane już wyrażenie: 10 4 3 + 2 * -.



Algorytm obliczania wyrażeń w ONP

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.

Tworzenie funkcji ONP w Haskell'u

1. Funkcja powinna przyjmować wyrażenie "10 4 3 + 2 * -" jako parametr i zwrócić nam wynik.
2. Chcemy, aby wynik był liczbą zmienno przecinkową o podwójnej precyzji, ponieważ chcemy też zawrzeć również podział.

```
solveRPN::String -> Double
```

Należy pamiętać, aby wszystkie elementy naszego ciągu potraktować jako oddzielne pozycje, czyli musimy rozdzielić je spacją.
3. Kolejne stany stosu będziemy przedstawiać jako listę. Np. stos 10, 4, 3 przedstawimy jako [3,4,10]. Poszczególne elementy będziemy wpisywać w odwrotnej kolejności.

Tworzenie funkcji ONP w Haskell'u

```
solveRPN :: String -> Double
solveRPN = head . foldl foldingFunction [] . words
    where  foldingFunction (x:y:ys) "*" = (y * x):ys
           foldingFunction (x:y:ys) "+" = (y + x):ys
           foldingFunction (x:y:ys) "-" = (y - x):ys
           foldingFunction xs numberString = read numberString:xs
```

Dodawanie większej liczby operatorów

Zawarty wcześniej kod można zmodyfikować w taki sposób, aby obsługiwał też inne operatory:

```
solveRPN :: String -> Double
solveRPN = head . foldl foldingFunction [] . words
    where  foldingFunction (x:y:ys) "*" = (y * x):ys
           foldingFunction (x:y:ys) "+" = (y + x):ys
           foldingFunction (x:y:ys) "-" = (y - x):ys
           foldingFunction (x:y:ys) "/" = (y / x):ys
           foldingFunction (x:y:ys) "^" = (y ** x):ys
           foldingFunction (x:xs) "ln" = log x:xs
           foldingFunction xs "sum" = [sum xs]
           foldingFunction xs numberString = read numberString:xs
```

Nasza wersja kodu

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

Nasza wersja kodu

```haskell
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

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)
```

DZIĘKUJEMY ZA UWAGĘ.