Raport Haskell Words Counting

Mikołaj Wielebnowski Krzysztof Kubień

9 stycznia 2021

1 Wstęp

W niniejszym raporcie znajduje się opis prac przeprowadzonych w celu stworzenia programów w językku Haskell . Projekt został oparty o poradnik i został stworzony w celu zaliczenia projektu na przedmiot *"Programowanie Funkcyjne"*.

2 Opis projektu

Etapy prac:

1. Wstęp.
2. Przygotowanie środowiska Haskell.
3. Wyświetlanie liczb w innej bazie.
4. Odczytywanie liczb z innej bazy.
5. Znajdowanie ciągów znaków używając Data.ByteString.
6. Znajdowanie ciągu znaków używając algorytmu Boyer-Moore-Horspool.
7. Poszukiwanie ciągu znaków używając algorytmu Rabin-Karpa.
8. Rozdzielenie ciągu znaków.
9. Znajdowanie najdłuższej wspólnej sekwencji.
10. Kod fonetyczny.
11. Przeliczanie dystansu edycji.
12. Przeliczanie dystansu Jaro-Winkler pomiędzy dwoma ciągami.
13. Znajdowanie ciągu z jednym dystansem.
14. Naprawianie błędów w zapisie.
15. Podsumowanie.

2.1 Wstęp.

Haskell to czysto funkcyjny język programowania nazwany na cześć Haskella Curry'ego. Był on początkowo intensywnie rozwijany wokół ośrodka University of Glasgow, popularny kompilator tego języka to Glasgow Haskell Compiler (GHC) kompilujący szybki kod maszynowy porównywalny w szybkości wykonania do kodów z GCC.

Rysunek 1: Logo Haskell

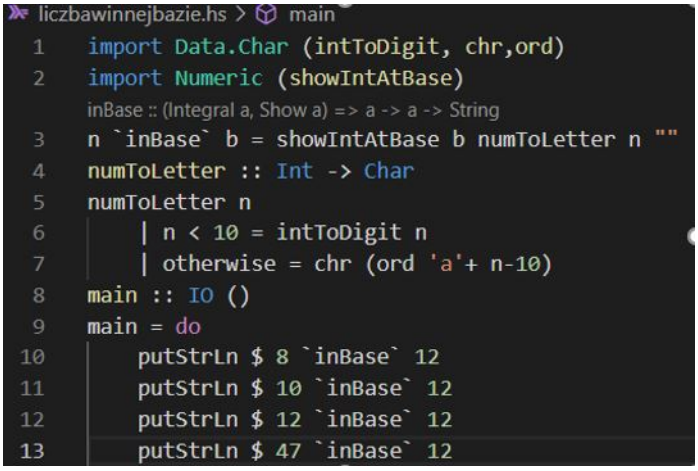
Programy w projekcie pokazują podstawowe sposoby interpretacji ciągów znaków oraz zmiany na nie liczb, co po rozwinięciu pozwala na modyfikację oraz analizę dużej ilości tekstu. Takie techniki w dzisiejszych czasach są bardzo istotne i przyczyniają się zarówno do analiz big data, jak i technologiami AI i badaniem tekstów takich jak dzieła literackie.

2.2 Przygotowanie środowiska Haskell.

Przed rozpoczęciem pracy nad projektem zespół musiał przygotować środowisko programistyczne dla języka Haskell. W tym celu postanowiono posłużyć się edytorem Visual Studio Code. Następnie użyto zakładki extensions w celu dodania środowiska Haskell, oraz poprzez instalację pakietu Chocolatey. Zespół korzystał z kompilatora ghci.
Kolejnym krokiem zespołu było zapoznanie się z treścią opracowania zawartego w pliku WordsCounting.pdf

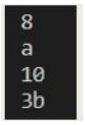
2.3 Wyświetlanie liczb w innej bazie.

Ten program pozwala na wczytanie liczby dziesiętnej int n i otrzymanie jako output liczby w postaci stringu w zadanej bazie b. Funkcja showIntAtBase bierze za bazę wybraną liczbę, a następnie przypisuje ją do odpowiedniego dla bazy znaku z przedziału 0, 1 ,2, ... , a ,b , ... , z. Aby funkcja mogła zadziałać poprawnie wymaga zamiany liczby na ciąg cyfr, tą funkcje pełni tutaj numToLetter



Rysunek 2: liczbawinnejbazie.hs

Podane dla przykładu liczby 8, 10 ,12 oraz 47, po przekonwertowaniu ich na bazę 12 dają wynik:



Rysunek 3: Wynik programu.

2.4 Odczytywanie liczb z innej bazy.

W tym programie staramy się osiągnąć odwrotność poprzedniej operacji, czyli przekonwertować liczbę w postaci stringa zapisanej w systemie liczbowym o danej bazie na liczbę dziesiątkową typu int. Program dla zadanego stringa stosuje funkcję readInt wczytuje liczbę i konwertuje ją na daną bazę. Aby funkcja zadziałała poprawnie należy zamienić typ zmiennych ze stringów na int (letterToNum) i sprawdzić czy otrzymany znak jest zgodny (isValidDigit).

odczytzbazy.hs > ...

```haskell
import Data.Char (ord, digitToInt, isDigit)
import Numeric (readInt)
base :: Num a => String -> a -> [(a, String)]
str `base` b = readInt b isValidDigit letterToNum str
letterToNum :: Char -> Int
letterToNum d
    | isDigit d = digitToInt d
    | otherwise = ord d - ord 'a' + 10

isValidDigit :: Char -> Bool
isValidDigit d = letterToNum d >= 0
main :: IO ()
main = do
    print $ "8" `base` 12
    print $ "a" `base` 12
    print $ "10" `base` 12
    print $ "3b" `base` 12
```

Rysunek 4: odczytzbazy.hs

Po podstawieniu przykładowych liczb 8, a, 10, 3b w bazie 12 otrzymujemy następujący output:

```
[(8,"")]
[(10,"")]
[(12,"")]
[(47,"")]
```

Rysunek 5: Wynik programu.

2.5 Znajdowanie ciągów znaków używając Data.ByteString.

W tym programie pobieramy ciąg znaków, przy pomocy jednego z dostępnych algorytmu dzielącego ciągi znaków z biblioteki Data.ByteString, który nadaję się do używania na dużej ilości danych ze względu na swoją efektywność. Program otrzymując dwa string konwertuje je przy pomocy funkcji Char8 na 8 bitowe, następnie następnie przy pomocy algorytmu breakSubstring poszukuje zadanego w pierwszej części (query) inputu ciągu w drugim ciągu. Gdy ciąg zostanie znaleziony program wyrzuca wartości logiczne True lub False, w zależności czy został on znaleziony czy nie.

databyte.hs > main

```haskell
import Data.ByteString (breakSubstring)
import qualified Data.ByteString.Char8 as C
substringFound :: String -> String -> Bool
substringFound query str =
    (not . C.null . snd) $
    breakSubstring (C.pack query) (C.pack str)
main :: IO ()
main = do
    print $ substringFound "scraf" "swedish scraf mafia"
    print $ substringFound "flute" "swedish scraf mafia"
```

Rysunek 6: databyte.hs

3 Znajdowanie ciągu znaków używając algorytmu Boyer-Moore-Horspool.

Algorytm Boyer'a-Moor'a-Horspool'a to algorytm doskonale nadający się do znajdowania ciągów w innych ciągach. Ten wyjątkowo dobrze sprawdza się długich ciągów i poszukiwania ich w bardzo dużych ciągach, ponieważ algorytm jest w stanie efektywnie pomijać części tekstu, dla których porównania już zostały wykonane. W tym programie zastosowana jest uproszczona wersja określana jako algorytm Horspool'a, która nie wymaga postprocessingu i jednocześnie traci znacznie na efektywności na dużych ciągach, ale jest nie jest tak „przeciążony" na początku.
Program podobnie jak w poprzednim poszukuje ciągu z pierwszej części inputu w drugiej. Algorytm w uproszczonej formie zastosowanej tutaj tworzy „okno", o wielkości równej zadanemu ciągowi i porównuje znak po znaku następnie poruszając „okno" z prawej do lewej.

boyermoore.hs > bmh'

```haskell
import Data.Map (fromList, (!), findWithDefault)
indexMap :: (Ord k, Num k, Enum k) => [a] -> Data.Map.Internal.Map k a
indexMap xs = fromList $ zip [0..] xs
revIndexMap :: (Ord k, Num a, Enum a) => [k] -> Data.Map.Internal.Map k a
revIndexMap xs = fromList $ zip (reverse xs) [0..]
bmh :: Ord a => [a] -> [a] -> Maybe Int
bmh pat xs = bmh' (length pat - 1) (reverse pat) xs pat
bmh' :: Ord a => Int -> [a] -> [a] -> [a] -> Maybe Int
bmh' n [] xs pat = Just (n + 1)
bmh' n (p:ps) xs pat
    | n >= length xs   = Nothing
    | p == (indexMap xs) ! n = bmh' (n - 1) ps xs pat
    | otherwise              = bmh' (n + findWithDefault
    (length pat) (sMap ! n) pMap)
    (reverse pat) xs pat
    where sMap = indexMap xs
          pMap = revIndexMap pat
main :: IO ()
main = print $ bmh "Wor" "Hello World"
```

Rysunek 7: boyermoore.hs

W tej implementacji wynik programu pokazuje ostatnie miejsce przed rozpoczęciem pierwszego wystąpienia zadanego stringu stąd dla przykładu wynik to:

```
Just 6
```

Rysunek 8: Wynik programu.hs

4 Poszukiwanie ciągu znaków używając algorytmu Rabin-Karpa.

Kolejnym algorytmem, który stosuje się do poszukiwania do poszukiwania ciągu w ciągu jest algorytm Rabina-Karpa, ten algorytm porównuje „okienko" z tekstem, ale nie znak po znaku, a przez numer o bazie 26+, identyfikujący zestaw znaków. Algorytm ten nie jest zbyt szybki dla pojedynczego przeszukania, jednak zaczyna być bardzo sprawny, gdy zapytań jest więcej, ponieważ po pierwszym przejściu przez tekst jest w stanie szybko znajdować kolejne.

Określenia haystack (stog siana) oraz needle (igła) stosowane są w angielskim w celu określenia odpowiednio przeszukiwanego tekstu i zapytania.

Po zainstalowaniu modułu cabala stringsearch, algorytm Rabina-Karpa można zaimplementować z biblioteki ByteString. Program zamienia litery z naszego zapytania używając miejsca na którym znajduje się znak jako potęgę bazy i funkcji ord do zamiany znaku na liczbę jej odpowiadającą. I po raz kolejny program porusza „oknem", tym razem lewej do prawej, porównując wartości identyfikujące znaki i w ten sposób znajdując dopasowania.

rabinkarp.hs > ...

```haskell
{-# LANGUAGE OverloadedStrings #-}
import Data.ByteString.Search.KarpRabin (indicesOfAny)
import qualified Data.ByteString as BS
main = do
    let needles = [ "preparing to go away"
                  , "is some letter of recommendation"]
    haystack <- BS.readFile "big.txt"
print $ indicesOfAny needles haystack
```

Rysunek 9: rabinkarp.hs

Po wykonaniu programu otrzymujemy listę, która składa się z numeru określającego położenie i liczbę w nawiasie określającą numer zapytania.

```
[(3738968,[1]),(5632846,[0]),(5714386,[0])
```

Rysunek 10: Wynik programu.

5 Rozdzielenie ciągu znaków.

Problem ze użytecznymi danymi jest taki, że trzeba je w jakiś sposób rozdzielić i do tego idealna jest funkcja splitOn , która pochodzi z modułu cabala split. Dzieli ona dane na osobne ciągi jak i usuwa separator. Ten program ma za zadanie wczytać dane z pliku input.txt, a następnie podzielić zawartość na różne sposoby poprzez linie, spacje, przecinki lub dowolne znaki odpowiednio dla każdej linijki z pliku.

Program jest banalnie prosty wystarczy wczytać plik i użyć splitOn z odpowiednim separatorem podanym jako pierwszy argument, gdzie wyjątkiem jest nowa linia, ponieważ tak zaimportowany plik Haskell dzieli automatycznie.

splitting.hs

```haskell
import Data.List.Split (splitOn)
main = do
    input <- readFile "input.txt"
    let ls = lines input
    print $ ls
    let ws = words $ ls !! 2
    print ws
    let cs = splitOn "," $ ls !! 3
    print cs
    let ds = splitOn "an" $ ls !! 4
    print ds
```

Rysunek 11: splitting.hs

W pierwszym przykładzie dzielimy po linijce:

```
first line
second line
words are split by space
comma,separated,values
or any delimiter you want
```

Rysunek 12: Wynik pierwszego przykładu

W drugim po spacji

```
words,are,split,by,space
```

Rysunek 13: Wynik drugiego przykładu

W trzecim po przecinku:

```
comma,separated,values
```

Rysunek 14: Wynik trzeciego przykładu

W czwartym po dowolnym znaku w tym wypadku „e”:

```
or any d,limit,r you want
```

Rysunek 15: Wynik czwartego przykładu

6 Znajdowanie najdłuższej wspólnej sekwencji.

Jednym ze sposobów na znajdowanie podobieństw w ciągach znaków jest znajdowanie ich najdłuższej wspólnej sekwencji. Sposób ten jest użyteczny w znajdowaniu mutacji danych. Algorytm ten jest zaimplementowany z zapamiętaniem rekursywnych przywołań. Jeśli dwa pierwsze składniki listy są takie same, wtedy najdłuższą wspólną sekwencją jest funkcja lcs zastosowana do pozostałej części listy. W przeciwnym przypadku najdłuższą wspólną sekwencją jest ta, która jest najdłuższy z możliwych. Do usprawnienia programu została użyta funkcja memorize aby zapamiętać wcześniej przeliczone wartości.

common.hs

```haskell
import qualified Data.MemoCombinators as Memo
memoize :: (String -> String -> r) -> String -> String -> r
memoize = Memo.memo2
    (Memo.list Memo.char) (Memo.list Memo.char)
lcs :: String -> String -> String
lcs = memoize lcs'
  where lcs' xs'@(x:xs) ys'@(y:ys)
      | x == y = x : lcs xs ys
      | otherwise = longer (lcs xs' ys) (lcs xs ys')
    lcs' _ _ = []
  longer as bs
      | length as > length bs = as
      | otherwise = bs
main :: IO ()
main = do
    let xs = "find the lights"
    let ys = "there are four lights"
    print $ lcs xs ys
```

Rysunek 16: common.hs

Po wykonaniu programu otrzymujemy następujący wynik:

```
the lights
```

Rysunek 17: Wynik programu

7 Kod fonetyczny.

Kiedy mamy do czynienia z korpusem angielskich słów to możemy je podzielić na fonetyczny kod żeby zobaczyć jak podobnie brzmią. Kod fonetyczny działa dla dowolnego alfabetycznego ciągu znaków. Będziemy korzystać z pakietu Text.PhoneticCode. Program wykonuje manipulacje ciągiem bazując na heurystyce wzorców języka angielskiego.

phonetic.hs

```haskell
import Text.PhoneticCode.Soundex (soundexNARA, soundexSimple)
import Text.PhoneticCode.Phonix (phonix)
ws = ["haskell", "hackle", "haggle", "hassle"]
main :: IO ()
main = do
    print $ map soundexNARA ws
    print $ map soundexSimple ws
    print $ map phonix ws
```

Rysunek 18: phonetic.hs

Po wykonaniu otrzymujemy wynik:

```
H240,H240,H240,H240][H240,H240,H240,H240][H82,H2,H2,H8]
```

Rysunek 19: wynik programu

8 Przeliczanie dystansu edycji.

Dystans edycji lub dystans Levenshteina jest minimalną liczbą prostych operacji na ciągu wymaganej do konwersji jednego ciągu znaków na drugi.

$$\operatorname{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j) = 0, \\ \min \begin{cases} \operatorname{lev}_{a,b}(i-1,j) + 1 \\ \operatorname{lev}_{a,b}(i,j-1) + 1 \\ \operatorname{lev}_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases}$$

Rysunek 20: Formuła

Kod będzie bezpośrednią interpretacją powyższej formuły.
Algorytm rekurencyjnie próbuje usuwać, dodawać oraz zamieniać na wszelki sposób do czasu aż znajdzie minimalny dystans z jednego ciągu do drugiego.

editdist.hs

```haskell
import qualified Data.MemoCombinators as Memo
lev :: Eq a => [a] -> [a] -> Int
lev a b = levM (length a) (length b)
  where levM = memoize lev'
        lev' i j
          | min i j == 0 = max i j
          | otherwise    = minimum
          [ ( 1 + levM (i-1) j )
          , ( 1 + levM i (j-1) )
          , ( ind i j + levM (i-1) (j-1) ) ]
ind i j
    | a !! (i-1) == b !! (j-1) = 0
    | otherwise = 1
memoize = Memo.memo2 (Memo.integral) (Memo.integral)
main = print $ lev "mercury" "sylvester"
```

Rysunek 21: editdist.hs

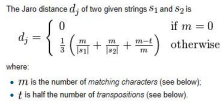
Po wykonaniu trzymujemy poniższy wynik:

```
8
```

Rysunek 22: Wynik programu

9 Przeliczanie dystansu Jaro-Winkler pomiędzy dwoma ciągami.

Dystans Jaro-Winkler'a określa podobieństwo pomiędzy ciągami za pomocą liczby z zakresu 0 i 1, gdzie 0 to brak podobieństwa a 1 to pełne podobieństwo. Algorytm wyraża się za pomocą matematycznego równania przedstawionego poniżej:



Rysunek 23: Formuła

Gdzie s1 i s2 to dwa ciągi, m to liczba identycznych znaków w maksymalnie połowieodlgełosci długości ciągu dłuższego, t to połowa ilości identycznych znaków, ale o różnych pozycjach.

```haskell
jarowinkler.hs > jaro
import Data.List (elemIndices)
jaro :: Eq a => [a] -> [a] -> Double
jaro s1 s2
    | m == 0    = 0.0
    | otherwise = (1/3) * (m/ls1 + m/ls2 + (m-t)/m)
    where ls1 = toDouble $ length s1
          ls2 = toDouble $ length s2
          m' = matching s1 s2 d
          d = fromIntegral $
           max (length s1) (length s2) `div`  2 - 1
          m = toDouble m'
          t = toDouble $ (m' - matching s1 s2 0) `div` 2
toDouble :: Integral a => a -> Double
toDouble n = (fromIntegral n) :: Double
matching :: Eq a => [a] -> [a] -> Int -> Int
matching s1 s2 d = length $ filter
    (\(c,i) -> not (null (matches s2 c i d)))
    (zip s1 [0..])
matches :: Eq a => [a] -> a -> Int -> Int -> [Int]
matches str c i d = filter (<= d) $
    map (dist i) (elemIndices c str)
    where dist a b = abs $ a - b
main :: IO ()
main = do
    print $ jaro "marisa" "magical"
    print $ jaro "haskell" "hackage"
```

Rysunek 24: jarowinkler.hs

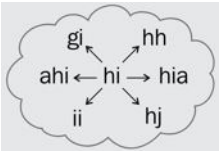
Powyższy program po zaimplementowaniu algorytmu wyrzuca w wyniku dwie wartości dla przykładu marisa i magical oraz haskell i hackage, przypisując im poniższe wartości:

```
0.746031746031746
0.7142857142857142
```

Rysunek 25: Wynik programu

10 Znajdowanie ciągu z jednym dystansem.

Ten program znajduje ciągi, które różnią się od zadanego ciągu o jedną długość edycji, czyli szuka wszystkich elementów, na których trzeba wykonać tylko jedną akcję, aby przekształcić go do zadanego ciągu, tworząc rodzinę dla wybranego słowa jak na poniższym rysunku:



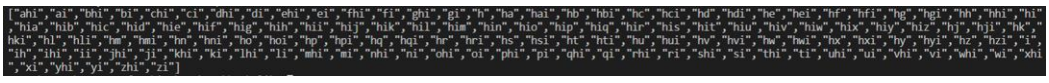
Rysunek 26: Rysunek

```haskell
import Data.Char (toLower)
import Data.List (group, sort)
edits1 :: String -> [String]

edits1 word = unique $
              deletes ++ transposes ++ replaces ++ inserts
  where splits   = [ (take i word', drop i word') |
              i <- [0..length word']]
        deletes = [ a ++ (tail b) |
            (a,b) <- splits, (not.null) b]
        transposes = [a ++ [b!!1] ++ [head b] ++ (drop 2 b) |
            (a,b) <- splits, length b > 1 ]
        replaces  = [ a ++ [c] ++ (drop 1 b)
            | (a,b) <- splits
            , c <- alphabet
            , (not.null) b ]
        inserts = [a ++ [c] ++ b
            | (a,b) <- splits
            , c <- alphabet ]
        alphabet = ['a'..'z']
        word' = map toLower word
unique :: [String] -> [String]
unique = map head.group.sort
main :: IO ()
main = print $ edits1 "hi"
```

Rysunek 27: oneedit.hs

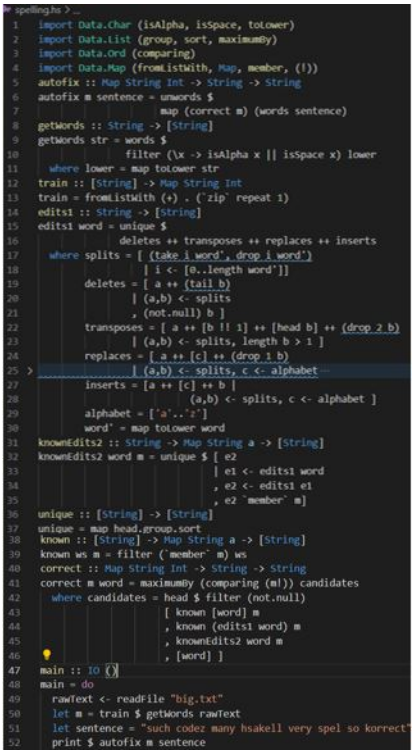
Dla wybranego przykładu "hi" z dużej bazy słów 1, 2 i 3 literowych otrzymujemy poniższy wynik:



Rysunek 28: Wynik programu

11 Naprawianie błędów w zapisie.

Ten program korzystają z algorytmu Petera Norvig'a oraz bazy słów poprawnie zapisanych. Algorytm porównuje dane słowa dzieląc je na ciągi i tworzy rodzinę ze słowami z bazy tak jak w poprzednim przykładzie. Algorytm zakłada, że literówki występuje dla pojedynczego lub podwójnego dystansu edycji, a następnie wybiera słowo z rodziny, korzystając z hierarchii opartej na częstości ich używania, zastępując badany ciąg. Ten algorytm jest dość prosty, jednak dzięki temu działa bardzo szybko, a wykonuje zadanie, które zakrawa o uczenie maszynowe.



Rysunek 29: spelling.hs

Jako przykład zastosowano „such codez many hsakell very spel so korrect", w którym znajdują się literówki w odpowiednich dystansach edycji, dzięki czemu program wyrzuca poprawiony teskt:

```
"such code many hsakell very spell so correct"
```

Rysunek 30: Wynik programu

12 Podsumowanie

Podsumowując prace związane z powyższymi programami zespół stwierdził, że język Haskell jest bardzo użyteczny jeśli chodzi o operacje na dużych ciągach znaków, a nawet korekcję błędów. Warto dodać, że strona Facebook używa właśnie języka Haskell do korekcji błędów literowych.
Jednakże zespół napotkał poważny problem w importowaniu niektórych bibliotek. Po długim poszukiwaniu powodu owego błędu odkryto, że winą była niekompletna instalacja pakietu cabal, przez co dodatkowe importowanie bibliotek poleceniem cabal było przeprowadzane ze skutkiem negatywnym.

Literatura

[1] WordsCounting.pdf

[2] $http://norvig.com/spell-correct.html$

[3] $https://hoogle.haskell.org$