Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики

Використання типів даних ByteString і Text

Текстова частина до курсової роботи
за спеціальністю „Інженерія програмного забезпечення” 121

Керівник курсової роботи
доц. Проценко В. С.

_____

(підпис)

“____” _____ 2021 р.

Виконав студент
Рибак В. Я.

_____

(підпис)

“____” _____ 2021 р.

Київ 2020

Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мережних технологій факультету інформатики

ЗАТВЕРДЖУЮ

Зав. Кафедри інформатики,
доц., Гороховський С. С.

_____
(підпис)

“____” _____ 2021 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на курсову роботу

Рибаку Володимиру Ярославичу, студентці 4 курсу факультету інформатики
Тема: Використання типів даних ByteString і Text

Вихідні дані:
Аналіз особливостей використання типів даних ByteString і Text
Приклади використання вище зазначених типів з поясненнями
Зміст ТЧ до курсової роботи:
Вступ
1: Теоретичні відомості про текстові типи даних
1.1: String
1.2: Text
1.3: ByteString
2: Приклади використання з поясненнями
Висновки
Список використаної літератури
Глосарій
Додатки

Дата видачі „___” _____ 2020 р. Керівник _____
(підпис)
Завдання отримав _____
(підпис)

Зміст

Анотація

У роботі представлено аналіз основних текстових типів даних присутніх у мові програмування Haskell: String, Text, ByteString. Розглянуті переваги та недоліки кожного з них, разом з особливостями використання та додано інформацію про взаємодію між вище зазначеними типами.

Для покращення розуміння було наведено декілька прикладів використання з поясненнями.

У першому розділі наведено теоретичні відомості та особливості використання кожного з типів. У другому наведені приклади використання з поясненнями, які націлені на закріплення теоретичної частини.

Вступ

Текстові типи даних є одними з основних в будь-якій мові програмування, найчастіше вони реалізовані як незмінний масив символів, що є доволі швидким та економним прикладом, але в Haskell стандартний тип для роботи з тестом String є «лінивим» списком символів. Така реалізація має ряд недоліків пов’язаних з накладними витратами пам’яті на підтримання структури даних та малою продуктивністю. Хоча рішення було прийнято не безпідставно, оскільки списки в Haskell є фундаментальною структурою. Деякі розробники навіть вважають Prelude, стандартний модуль Haskell, не доречним для комерційної розробки через залежність від вище зазначеного типу даних.

Щоб справитись з даною проблемою було створено декілька модулів з аналогічним API як і у String:

1. Data.Text – найбільш використовуваний текстовий тип даних для комерційної розробки
2. Data.ByteString – тип призначений для оброки двійкових даних, але з ним можна обробляти й текст

Вони розроблені з можливістю повної заміни стандартного типу String в програмах. Дані модулі є «строгими» для попередження бездумного використання «лінивих» вираховувань, але за потреби можна звернутись до підмодуля Lazy, який надасть цим типам стандартної для Haskell «лінивої» поведінки.

Ціллю даної роботи є дослідження вже згаданих типів даних та огляд основних задач, націлених на використання особливостей кожного з типів. Робота складається з вступу, основної частини, у якій розказано почергово про кожен тип окремо, заключних прикладів, що мають дати можливість закріпити теоретичний матеріал, та висновків. Для дослідження був вибраний метод аналізу існуючої

літератури, основним джерелом вибрана книга «Програмуй на Haskell» Курт У., додатковими джерелами є статті з мережі Інтернет та документація.

Розділ 1. Теоретичні відомості про текстові типи даних

1.1 String

Стандартним типом для обробки текстових даних в мові програмування Haskell є тип String, тобто без деяких розширень, про які піде мова далі, літерал виду “string” буде сприйматись як String. В структурі даного типу представлений список символів, а не масив як зазвичай, але це є доволі зрозумілим рішенням, так як списки є одною з фундаментальних структур даних в Haskell.

Завдяки такому рішенню, над типом String можна працювати як над списком, що надає велику кількість корисних функцій для використання. Вони знаходяться в модулі Data.List найпростіші з них: head та tail, які повертають перший елемент та всі, крім першого, відповідно.

Як і у звичній для більшості мов програмування реалізації, списки в Haskell також є незмінними, тому замість додавання символу до рядку, буде створено новий з відповідними значеннями, але на відміну від масиву, списки потребують значно більше пам’яті на підтримання самої структури, що є значним недоліком такого рішення.

Також String підтримує «ліниві» вираховування, дана функціональність є зручною наприклад для дій вводу та виводу, коли не можна наперед сказати як багато тексту потрібно буде обробити. Такий випадок можна розглянути на функції, що буде виводити текст у нижньому регістрі по мірі його надходження:

```
print . map toLower =<< getContents
```

Завдяки цьому не потрібно використовувати буфер для зберігання тексту, а дана функція буде виконувати дії на «льоту».

Хоч «ліниве» вираховування є достатньо зручним, у нього є великі проблеми з продуктивністю, пам’яттю та можливістю переповнення стеку, так як для такого процесу потрібно зберігати ще й інформацію про розвертання даної функції.

Таким чином, не правильне використання даної властивості може призвести до значних втрат продуктивності або навіть до зупинки програми з повідомленням, що говорить про переповнення стеку:

Stack space overflow: current size 8388608 bytes.

Якщо це все підсумувати, то можна прийти до висновку, що стандартний тип для текстових даних, який надає Haskell, є достатньо хорошим прикладом для вивчення роботи з списками, але його використання в комерційній розробці може призвести до великих втрат пам’яті та продуктивності.

1.2 Text

Як вже було зазначено, стандартний тип, представлений у Haskell для тестову, насправді має багато недоліків. Неприязнь до String у комерційних розробників іноді доходить до того, що деякі вважають навіть Prelude, стандартний модуль Haskell, не придатним для розробки, спираючись на глибокі залежності в ньому від даного типу. Щоб вирішити дану проблему, майже кожен пропонує для роботи з текстом використовувати тип Text, що знаходиться у модулі Data.Text. Даний модуль майже завжди підключають кваліфіковано, тобто щоб викликати функцію з нього, потрібно вказати ім’я модулю або його псевдонім.

import qualified Data.Text as T

Такий варіант підключення зумовлений наявністю функцій з однаковою назвою в даному та стандартному модулях.

В основу типу Text закладений масив, так само як і в більшості мов програмування, така реалізація є більш ефективною в багатьох операціях над стрічками та зменшує кількість пам’яті, потрібної для збереження тексту, в порівнянні з стандартним типом.

Також до теми пам’яті потрібно додати одну з основних переваг даного типу – функції в модулі Data.Text підтримують «злиття». «Злиття» - спеціальний спосіб оптимізації в Haskell, ціллю якого є позбавитись від проміжних структур даних.

Варто додати що тип Text не використовує ліниві вираховування, так як на практиці вони можуть призвести до втрати продуктивності, тобто замість потокового варіанту, потрібно спочатку вичитати весь текст в пам’ять і тільки потім обробляти його. Оскільки ліниві вираховування можуть бути корисними, «лінивий» варінт типу Text все таки є, він знаходиться у модулі Data.Text.Lazy, інтерфейс якого збігається з оригінальним, в його основу закладена вже складніша структура даних – список з строгих шматків, масивів. В інтерфейс типу Lazy додано 2 функції доволі важливі функції:

1) fromStrict – зводить тип «строгого» Text до «лінивого»
2) toStrict – зводить тип «лінивого» Text до «строгого»

Дані функції мають таку ж саму складність як і pack/unpack, тому їх використання варто уникати, особливо при роботі з великою кількістю інформації.

Перейдемо до використання типу Text. В даному модулі присутні 2 функції призначені для перетворення типів String -> Text та Text -> String їх назви pack і unpack відповідно. Для того, щоб зрозуміти про що мова піде далі, потрібно подивитись на даний приклад.

1) justString :: String
   justString = “string”
2) stringToText :: T.Text
   stringToText = T.pack justString
3) textToString :: String
   textToString = T.unpack stringToText

В першому пункті визначено константу, яка задається літералом і потім перетворюється на екземпляр типу Text. Дані операції є дорогими у

вираховуванні, тому їх краще не використовувати. В даному випадку зручним було визначення константи одразу в типі Text.

```
justText :: T.Text

justText = “text”
```

Але цього зробити не можна. Помилка отримана в результаті таких дій:

```
Couldn`t match expected type `T.Text` with actual type `[Char]`
```

Прочитавши дане повідомлення, зрозуміло що проблема в тому, що літерал “text” має тип String при стандартних налаштуваннях. Щоб впоратись з даною проблемою потрібно втрутитись у спосіб, яким GHC читає файл. Вирішенням стануть мовні розширення, точніше розширення під назвою OverloadedStrings.

Існує два способи застосувати мовне розширення:

1) Запустити файл на компіляцію з прапорцем -X, тобто
   $ ghc file.hs -XOverloadedStrings
2) На початку файлу вказати директиву LANGUAGE
   {-# LANGUAGE OverloadedStrings #-}

Перевагу варто надати другому варіанту, так як про виставлення прапорцю можна легко забути. З такими налаштуваннями тип Text здається хорошим рішенням для переходу від не ефективного String.

Залишилась остання проблема, вона полягає в тому, що більшість функцій для обробки тексту підтримують тільки тип String, конвертувати Text в String і потім назад звісно є найгіршим варіантом. В такому випадку потрібно розібратись які можливості надає модуль Data.Text і чи покриває він функціонал представлений для типу String. Насправді майже всі корисні функції для роботи з текстом були перенесені у модуль Data.Text. Після використання кваліфікованого імпорту даного модулю, викликати потрібні функції можна за допомогою назви модуля/його псевдоніма та назви функції, розділених крапкою:

1) Data.Text.lines – розділяє стрічку по символам нового рядка на список стрічок
2) Data.Text.words – аналогічна lines, різниця в тому, що розділення виконується не тільки по символу нового рядка, але й по іншим видам пробільних символів
3) Data.Text.splitOn – поділяє стрічку по заданому тексту, спочатку слід задати текст по якому потрібно розбити стрічку, а вже потім саму стрічку
4) Data.Text.unlines – зворотна функція до lines, тобто об'єднує стрічки зі списку в одну розділену символами нового рядка
5) Data.Text.unwords – зворотна функція до words, встановлює пробіли між стрічками в списку
6) Data.Text. intercalate – зворотна функція до splitOn, параметри встановлюються аналогічно

Основна втрата функціональності спостерігається при згадці про те, що Text в своїй структурі є масивом, а не списком, тому така операція як ++, комбінація списків, не може бути використана у даному контексті. Але таку функціональність можливо отримати за допомогою класів типів Semigroup або Monoid, завдяки тому, що Data.Text надає екземпляри даних класів типів.

Щоб скористатись Semigroup потрібно включити його до програми:

```
import Data.Semigroup
```

Він надає одну важливу операцію «<>», комбінування двох елементів одного типу, тепер варто замінити «++» на «<>» і зручний спосіб комбінування стрічок буде отриманим. Для класу типу Monoid не потрібно нічого додатково включати у програму, потрібно просто використати функцію mconcat [“this”, “text”].

Таким чином, хоч для використання Text потрібно зробити немало додаткових маніпуляцій, вони варті того, щоб отримати можливість використовувати

більш швидкий та економічний для пам’яті текстовий тип даних в порівнянні з стандартним String.

1.3 ByteString

Даний тип даних насправді створений не для зберігання тексту, хоч і має такий самий інтерфейс як і Text. Незважаючи на те, що в його назві другою частиною є «String», варто придати більше уваги саме першій частині – «Byte». ByteString – це масив байтів, представлених типом Word8, що робить даний тип зручним для обміну між C та Haskell. Оскільки він зберігає не символи, які потребують додаткових перевірок, даний тип є швидшим за Text, але зважаючи на те, що кожен байт може представляти певний символ з таблиці ASCII, є можливість використовувати його для стрічок, хоч і з обмеженими можливостями. Даний модуль, як і Text, потрібно включати до програми кваліфіковано, щоб назви функцій не збігались.

```
import qualified Data.ByteString as BS
```

Як вже було сказано, даний тип може представляти стрічку, тому не дивно що при використанні мовного розширення OverloadedStrings значення для типу ByteString можна задати за допомогою літералу виду “ByteString”.

```
{-# LANGUAGE OverloadedStrings #-}

bString :: BS.ByteString

bString = “ByteString text”
```

Хоч таке представлення і можливе, все таки не варто сприймати даний тип як стрічку, наприклад звести значення даного типу до стандартного String не є можливою операцією.

```
toString :: String
```

```
toString = BS.unpack bString
```

Даний код призведе до помилки компіляції, все стане зрозумілішим, якщо глянути на типову анотацію функції unpack.

```
BS.unpack :: BS.ByteString -> [GHC.Word.Word8]
```

Замість зведення до типу String ми бачимо перетворення до списку байтів типу Word8. Якщо ж користувачу потрібно працювати з ByteString як з масивом восьмибітових символів, варто звернутись до Data.ByteString.Char8

```
import qualified Data.ByteString.Char8 as BSC
```

Який призначений саме для цього. Щоб показати різницю між цими модулями, подивимось на типову анотацію unpack з Data.ByteString.Char8.

```
BSC.unpack :: BSC.ByteString -> [Char]
```

Вся різниця полягає у тому, що замість думки що ByteString це просто набір байтів, Data.ByteString.Char8 вважає що це набір восьмибітних символів і надає такі самі функції для роботи з текстом, як і Text.

ByteString має «ліниву» реалізацію, її можна знайти в Data.ByteString.Lazy, структурне рішення таке ж, як і у Text – список «строгих» фрагментів. Така реалізація добре підходить для використання в потоках вводу та виводу. Так само як і для Text, ми маємо функції для перетворення з «лінивого» на «строгий» і навпаки. Слід знати що, наприклад операція toStrict, спочатку викачує у пам’ять весь «лінивий» ByteString і потім копіює його в «строгий», тому для використання даної функції мають бути важливі причини.

Підсумувавши дану інформацію зрозуміло, що ByteString призначений для роботи над байтами. Така реалізація є швидшою за Text, але не призначена для тексту в Unicode. Зважаючи на переваги та недоліки, даний тип найкраще використовувати для обміну даними або серіалізації.

Пов’язане використання обговорених типів даних

Наразі зрозумілі всі переваги та недоліки кожного з типів, ми можемо з точністю сказати який з типів варто використовувати для тої чи іншої задачі, але що до їх взаємодії між собою. В комерційних проектах нерідко можуть знадобитись декілька з даних типів, а можливо навіть перетворення одного типу в інший. Вже було зазначено функції pack і unpack, застосувавши які можна зробити перетворення пов’язані з типом String, але не було вказано як можна з екземпляру типу Text отримати ByteString і навпаки. Для таких цілей існує модуль Data.Text.Encoding для «строгих» і Data.Text.Lazy.Encoding для «лінивих». З даною функціональністю є деякі проблеми, як ми знаємо, тип ByteString не призначений для зберігання саме тексту, тому дані у ньому мають бути представлені як набір байтів, таким чином перетворення пов’язані з типом Text можуть бути використані тільки з вказанням кодування. У даних модулях присутні такі варіанти кодування:

1) UTF-8
2) UTF-16 big endian
3) UTF-16 little endian
4) UTF-32 big endian
5) UTF-32 little endian

Вони всі супроводжуються відповідними функціями, таким чином є можливість перетворити тип Text у потрібному кодуванні в ByteString, або знаючи кодування привести байти з ByteString до саме текстового типу даних Text.

Як ми бачимо нічого не заважає нам влаштувати взаємодію між всіма типами про які було розказано, але дані перетворення є доволі складними, тому зловживання ними призведе до значної втрати продуктивності, щоб уникнути дану проблему варто завчасно визначити які типи і де потрібні та зменшити використання таких складних операцій.

Розділ 2. Приклади використання з поясненнями

Тип String не має переваг над іншими, крім можливості використовувати над ним стандартні функції, які в достатній кількості перенесені до Text та ByteString, тому було вирішено, що доречніше за допомогою даного типу поглянути на зручність потоків вводу та виводу у кооперації з «лінивими» обчисленнями.

```
main :: IO ()
main = do
content <- getContents
let fileName = lines content
mapM_ (\x -> writeFile ("pathToFile\\" ++ x ++ ".txt") "") fileName
```

Приклад 1 лініві вираховування

Main приймає з потоку вводу та виводу стрічку і розділяє її по символам нового рядка. Якби Haskell був «строгий» довелось би спочатку вичитати весь потік до символу закінчення файлу, а вже потім виконувати наступні дії, але у нашому випадку, замість вичитування всього тексту, утворюється стек з «обіцянок», тобто ще не вирахуваних значень. Саме завдяки цим «обіцянкам» ми можемо спостерігати створення файлів «на льоту», тобто до написання символу закінчення файлу.

1)

Наступним прикладом буде робота з Unicode. Як вже було зазначено, тип ByteString може бути використаний для обробки стрічок, символи яких є восьмибітними, тому його використання для обробки Unicode символів не є можливим. Для прикладу можна використати функції pack та unpack для перетворення з ByteString у Text.

```haskell
import qualified Data.ByteString.Char8 as BSC
import qualified Data.Text as T
import qualified Data.Text.IO as TIO

someByteString :: BS.ByteString
someByteString = "やめてください"
```

Приклад 2.1 Unicode в ByteString

```haskell
textPacking :: T.Text
textPacking =  (T.pack . BSC.unpack) someByteString
```

Приклад 2.2 Unicode в ByteString

```
*Main> TIO.putStrLn textPacking

fO`UD
*Main>
```

Приклад 2.3 Unicode в ByteString

Таким чином дані Unicode втрачаються, зрозуміло чому таке перетворення не є правильним. ByteString зберігає стрічку як набір байтів, а остільки символи Unicode не можуть бути представлені в такому виді, тому даний тип їх розділяє без можливості відновлення. Звісно є можливість помістити Unicode символи до ByteSting.

```haskell
import qualified Data.ByteString as BS
import qualified Data.ByteString.Char8 as BSC
import qualified Data.Text as T
import qualified Data.Text.Encoding as TE
import qualified Data.Text.IO as TIO

someText :: T.Text
someText = "やめてください"

toByteString8 :: BS.ByteString
toByteString8 = TE.encodeUtf8 someText

toByteString16 :: BS.ByteString
toByteString16 = TE.encodeUtf16LE someText

toByteString32 :: BS.ByteString
toByteString32 = TE.encodeUtf32LE someText
```

Приклад 3.1 Правильне перетворення типів

```
*Main>
*Main> (TIO.putStrLn . TE.decodeUtf8) toByteString8
やめてください
*Main>
```

Приклад 3.2 Правильне перетворення типів

У даному варіанті дані не були втрачені, тому при необхідності можна використовувати даний функціонал, не забуваючи про те, що такі перетворення є ресурсоємними.

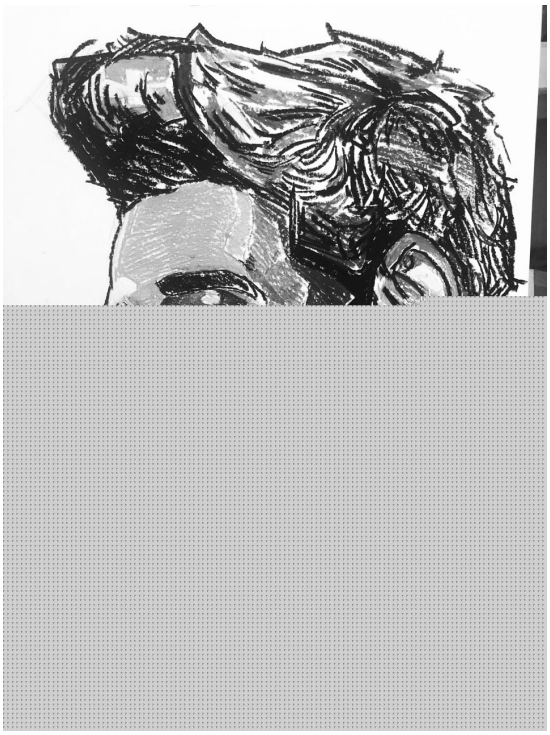
Основною задачею ByteSting є робота з байтами як зі стрічками, тому одним з варіантів використання може бути спеціальне спотворення картинки для отримання цікавого результату.

```haskell
module Main where
  import qualified Data.ByteString as BS

  sortPartInTheMiddle :: BS.ByteString -> BS.ByteString
  sortPartInTheMiddle picture = first <> BS.sort second <> third
                    where len = div (BS.length picture) 2
                          (first, tail) = BS.splitAt len picture
                          (second, third) = BS.splitAt 40 tail

  main :: IO ()
  main = do
  content <- BS.readFile "pathTo\\colorface.jpg"
  BS.writeFile "pathTo\\colorface2.jpg" (sortPartInTheMiddle content)
```

Приклад 4.1 навмисне псування фотографії

Даний приклад достатньо добре показує яким чином потрібно використовувати ByteString, спочатку зчитується файл, а потім за допомогою функцій представлених у даному модулі, які є схожими до операцій над стрічками, відбувається обробка байтів.

Приклад 4.2 навмисне псування фото. Оригінальне фото



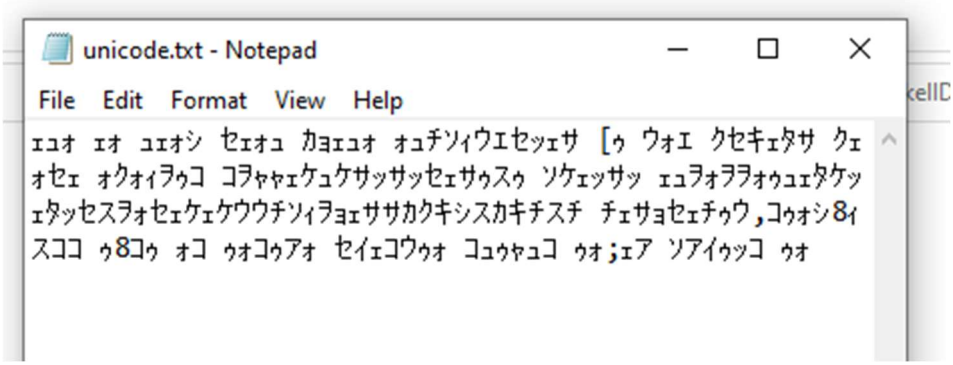
Приклад 4.3 навмисне псування фотографії. Фото після псування

Результат є доволі цікавим, приблизно третя частина картинки збереглась без змін, а інша її частина є настільки зіпсованою що зовсім не відображається.

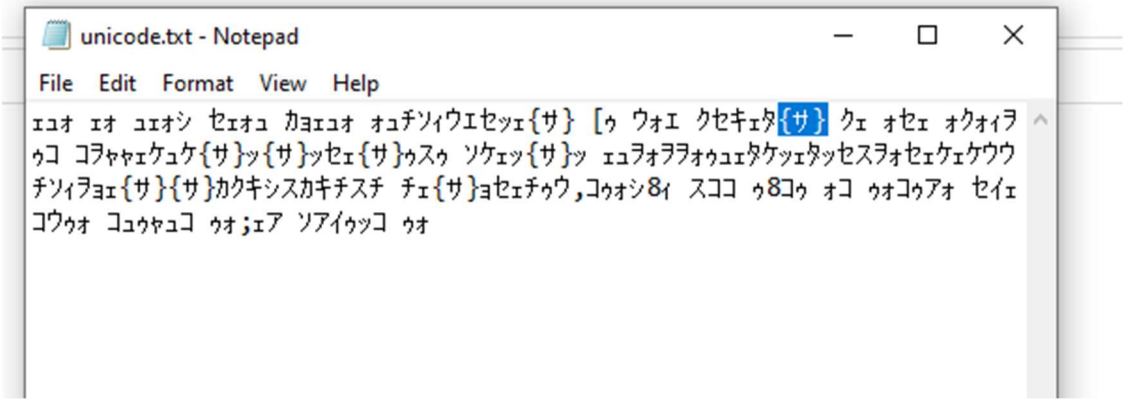
Останнім прикладом є робота з Text, даний тип є найбільш корисним при обробці тексту в Unicode, тому даний приклад направлений саме на це

```haskell
{-# LANGUAGE OverloadedStrings #-}
module Main where
  import qualified Data.Text as T
  import qualified Data.Text.IO as TIO

  findHieroglyph :: T.Text -> T.Text -> T.Text
  findHieroglyph hieroglyph text = T.intercalate wrappedHieroglyph splitedText
                where
                  splitedText = T.splitOn hieroglyph text
                  wrappedHieroglyph = mconcat ["{", hieroglyph,"}"]


  main :: IO ()
  main = do
  content <- TIO.readFile "pathTo\\unicode.txt"
  TIO.putStrLn (findHieroglyph "ㅐ" content)
```

Приклад 5.1 Виділення ієрогліфу



Приклад 5.2 Виділення ієрогліфу. Файл до опрацювання



Приклад 6.3 Виділення ієрогліфу. Файл після опрацювання

Програма, що перед вами, зчитує файл, виділяє потрібний ієрогліф і записує результат назад до файлу. Це був доволі легкий приклад, але головна його задача показати як легко можна працювати з типом Text для обробки тексту в Unicode.

Висновки

У результаті даного дослідження можна з точністю сказати який тип даний, розглянутих у роботі, потрібно використовувати в тій чи іншій ситуації. Були з’ясовані сильні та слабкі сторони кожного з типів і продемонстровано можливість розширення деякої функціональності, яка дозволить користуватись типами Text та ByteString так само зручно, як і стандартним String. Було розказано про переваги та недоліки «лінивих» вираховувань і застережено він бездумного їх використання. Дана робота пояснила особливості перетворення між типами, оскільки вони є достатньо різними та обдумане використання їх взаємодії може бути основним фактором написання ефективної програми.

Також було розглянуто декілька прикладів для закріплення знань та остаточного розуміння проблеми вибору. Вони націлені на зменшення непорозумінь і наглядно показують деякі варіанти використання даних типів.

Список використаної літератури

1. Will Kurt, Get Programming With Haskell — Manning Publications, 2018 — 296 с.
2. Data.Text [Електронний ресурс] Режим доступу: https://hackage.haskell.org/package/text-1.2.4.1/docs/Data-Text.html
3. Data.ByteString [Електронний ресурс] Режим доступу: https://hackage.haskell.org/package/bytestring-0.11.1.0/docs/Data-ByteString.html
4. Data.ByteString.UTF8 [Електронний ресурс] Режим доступу: http://hackage.haskell.org/package/utf8-string-1.0.2/docs/Data-ByteString-UTF8.html
5. Data.ByteString.Lazy [Електронний ресурс] Режим доступу: https://hackage.haskell.org/package/bytestring-0.11.1.0/docs/Data-ByteString-Lazy.html
6. Data.String [Електронний ресурс] Режим доступу: https://hackage.haskell.org/package/base-4.12.0.0/docs/Data-String.html
7. Data.Text.IO [Електронний ресурс] Режим доступу: http://hackage.haskell.org/package/text-1.2.4.1/docs/Data-Text-IO.html
8. Data.Text.Lazy [Електронний ресурс] Режим доступу: https://hackage.haskell.org/package/text-1.2.4.1/docs/Data-Text-Lazy.html
9. Untangling Haskell's Strings [Електронний ресурс] Режим доступу: https://mmhaskell.com/blog/2017/5/15/untangling-haskells-strings

Додатки

Додаток А. Лістинг програмного коду

Приклад номер 1

```
{-# LANGUAGE OverloadedStrings #-}
module Main where
  import qualified Data.Text as T
  import qualified Data.Text.Encoding as TE
  import qualified Data.Text.IO as TIO

  findHieroglyph :: T.Text -> T.Text -> T.Text
  findHieroglyph hieroglyph text = T.intercalate wrappedHieroglyph splitedText
            where
              splitedText = T.splitOn hieroglyph text
              wrappedHieroglyph = mconcat ["{", hieroglyph,"}"]


  main :: IO ()
  main = do
  let filePath = "pathToFile\\unicode.txt"
  content <- TIO.readFile filePath
  TIO.writeFile filePath (findHieroglyph "𓃰" content)
```

Приклад номер 2

```
module Main where
  import qualified Data.ByteString as BS

  sortPartInTheMiddle :: BS.ByteString -> BS.ByteString
  sortPartInTheMiddle picture = first <> BS.sort second <> third
                where len = div (BS.length picture) 2
```

```
                (first, tail) = BS.splitAt len picture
                (second, third) = BS.splitAt 40 tail

main :: IO ()
main = do
content <- BS.readFile "pathToFile\\colorface.jpg"
BS.writeFile "pathToFile\\colorface2.jpg" (sortPartInTheMiddle content)
```

Приклад номер 3

```
{-# LANGUAGE OverloadedStrings #-}
module Main where
  import qualified Data.ByteString as BS
  import qualified Data.ByteString.Char8 as BSC
  import qualified Data.Text as T
  import qualified Data.Text.Encoding as TE
  import qualified Data.Text.IO as TIO
  import System.Environment
  import Data.Char
  import Data.Semigroup

  someString :: String
  someString = "やめてください"

  someText :: T.Text
  someText = "やめてください"

  someByteStringAscii :: BS.ByteString
  someByteStringAscii = "text"

  someByteString :: BS.ByteString
  someByteString = "やめてください"
```

```
toByteString8 :: BS.ByteString
toByteString8 = TE.encodeUtf8 someText

toByteString16 :: BS.ByteString
toByteString16 = TE.encodeUtf16LE someText

toByteString32 :: BS.ByteString
toByteString32 = TE.encodeUtf32LE someText



fromByteString8 :: T.Text
fromByteString8 = TE.decodeUtf8 toByteString8

fromByteString16 :: T.Text
fromByteString16 = TE.decodeUtf16LE toByteString16

fromByteString32 :: T.Text
fromByteString32 = TE.decodeUtf32LE toByteString32

textPacking :: T.Text
textPacking =  (T.pack . BSC.unpack) someByteString

main :: IO ()
main = do
content <- getContents
let fileName = lines content
mapM_ (\x -> writeFile ("pathTo\\students\\" ++ x ++ ".txt") "") fileName
```