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

# ВИКОРИСТАННЯ ТИПІВ ДАНИХ BYTESTRING І TEXT МОВОЮ ПРОГРАМУВАННЯ HASKELL

**Текстова частина до курсової роботи**
**за спеціальністю «Інженерія програмного забезпечення» 121**

Керівник курсової роботи
доц. Проценко В. С.

_____
*(підпис)*
“____” _____ 2020 р.

Виконав студент Владимирська А. О.
“____” _____ 2020 р.

Київ 2020

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ

Зав. кафедри інформатики,

доц., С. С. Гороховський

_____

*(підпис)*

“____” _____ 2020 р.

**ІНДИВІДУАЛЬНЕ ЗАВДАННЯ**

на курсову роботу

студенту 4 курсу факультету інформатики

Владимирській Анастасії Олександрівні

**Тема:** Використання типів даних ByteString і Text

Вихідні дані:

- Аналіз принципів роботи ByteString і Text та їхніх переваг
- Програма мовою Haskell для демонстрації особливостей роботи з попередньо зазначеними типами даних

**Зміст ТЧ до курсової роботи:**

Вступ
1. Текстові типи даних в Haskell
2. Проектування програми з використанням Text і ByteString
Висновки
Список літератури
Додатки

Дата видачі “____” _____ 2020 р. Керівник _____

*(підпис)*

Завдання отримав _____

*(підпис)*

**Тема: Використання типів даних ByteString і Text**

**Календарний план виконання роботи:**

| № | Назва етапу курсового проекту (роботи) | Термін виконання етапу | Примітка |
|---|---|---|---|
| 1. | Отримання завдання на курсову роботу. | 29.10.2019 | |
| 2. | Огляд технічної літератури по мові програмування Haskell. | 20.12.2019 | |
| 3. | Огляд статей та документації на тему текстових типів даних в Haskell. | 09.01.2020 | |
| 4. | Проектування програми для демонстрації використання текстових типів даних. | 25.01.2020 | |
| 5. | Застосування розробленого плану для поступової розробки програми. | 15.02.2020 | |
| 6. | Написання пояснювальної роботи. | 12.03.2020 | |
| 7. | Створення слайдів презентації та написання доповіді. | 01.04.2020 | |
| 8. | Аналіз отриманих результатів з керівником. | 09.04.2020 | |
| 9. | Корегування роботи за відгуком наукового керівника. | 10.04.2020 | |
| 10. | Остаточне оформлення презентації роботи | 11.04.2020 | |
| 11. | Здача курсової роботи | 19.04.2020 | |

Студент _____

Керівник _____

**“____”_____ 2020 р.**

**Зміст**

**Анотація**

У роботі розглянуто типи даних для роботи з текстом в мові програмування Haskell: String, Text та ByteString. Розібрано їхні відмінності, способи використання та поєднування, сильні та слабкі сторони та можливості. Для ілюстрації роботи з типами даних Text і ByteString спроектовано по окремій програмі на кожен тип даних. Призначенням кожної з цих двох програм є відобразити особливості та спеціальні можливості розглянутих типів даних.

У першому розділі описані теоретичні нюанси використання текстових типів, способи їхнього використання, а також переваги та недоліки кожного. В другому розділі описано створення програми з використанням текстових типів даних Text та ByteString для ілюстрації їхніх можливостей.

## Вступ

Стандарт мови програмування Haskell надає для роботи з текстом тип даних String. Він працює в стилі самого Haskell, а тобто реалізований як список і є «лінивим». На жаль, такий підхід не є ефективним в плані місця в пам’яті та швидкодії. А для повноцінної роботи з текстовими документами зазвичай саме ці два фактори є вкрай важливими. Для вирішення проблеми низької ефективності базового типу даних, було створено додаткові модулі з його замінниками.

Існує декілька модулів, що надають можливість працювати з текстом без використання звичайного типу даних String. Строго кажучи, таких модулів є два: Data.Text і Data.ByteString. В свою чергу, дані модулі містять у собі ще декілька, які дещо змінюють базову поведінку типу даних. Основними є підмодулі Lazy, що надають тексту звичної для Haskell «лінивої» поведінки. Тож, в мові програмування Haskell виділяють п’ять текстових типів даних: String, Text, Lazy Text, ByteString, Lazy ByteString. Кожен з них має свої особливості та переваги, і, звісно, рекомендовані випадки для використання.

Метою даної роботи є розглянути особливості основних текстових типів даних мови програмування Haskell та галузі їхнього використання. За завдання поставлено створення програми для роботи з файлами розширення epub та обробка їхнього вмісту за допомогою типів даних Text і ByteString. Методом дослідження є аналіз існуючої літератури про Haskell та текстові типи даних цієї мови, а також практичне втілення програми.

Джерелами дослідження слугували книги з вивчення мови Haskell, інтернет-статті про функціональне програмування та навчальні матеріали з мови програмування Haskell.

Результати дослідження можуть бути використані як навчальний матеріал, на прикладі якого можна розглянути відмінності між текстовими типами даних та можливості їхнього використання.

Робота складається з двох розділів.

В першому розділі проаналізовано наявні в мові Haskell текстові типи даних з різних модулів, а також зазначено їхні відмінності між собою, визначено їхні властивості та сфери застосування.

Другий розділ присвячено огляду програмного рішення. Розглянуто окремі модулі програми та їхній функціонал. Проаналізовано використані типи даних та їхню взаємодію між собою, а також зовнішнє представлення.

Постановка задачі:

1. Розглянути текстові типи даних мови програмування Haskell.
2. Проаналізувати відмінності між типами даних та можливості їхнього використання.
3. Запроектувати програму, яка б використовувала одразу якомога більше з розглянутих раніше текстових типів даних, для максимально детального їхнього представлення.

# 1. ТЕКСТОВІ ТИПИ ДАНИХ В HASKELL

## 1.1 Стандартний тип String

Для відображення рядків, стандартом мови програмування Haskell визначено спеціальний тип даних – String. Тобто, літерал виду “text” інтерпретатор, якщо не надано ніяких додаткових визначень, розпізнає як String. Всередині цей тип є простим синонімом до списку символів ([Char]). Що цілком очікувано, оскільки списки є найбільш вживаною структурою даних в мові Haskell. Таким чином, у базовій програмній реалізації будь-чого на Haskell весь текст представлено саме як списки символів.

Така стандартна реалізація є зручною і звичною для усвідомлення. За допомогою String можна працювати з текстом як і з будь-яким іншим типом даних, упакованим в список. Усі функції, що сприймають на вхід цю структуру, доступні для використання. Оскільки String визначено стандартом мови, він не потребує імпортування сторонніх пакетів, або прописування додаткових директив для інтерпретатора. String, як і майже все мовою Haskell, має «ліниву» реалізацію. Це означає, що рядок сприймається як потік і не буде обраховано увесь одразу. [1]

Проте, базова реалізація текстових рядків має суттєві недоліки. Представлення тексту у вигляді списку є неефективним через багаторазове засмічення пам’яті. Цей тип є незмінним, а це означає, що для будь-якої модифікації рядка буде створено нову копію. Крім того, саме зберігання тексту в такому форматі веде до великих накладних витрат. Вони можуть сягати до чотирьох машинних слів на символ. Сам по собі символ важить два машинних слова, що дорівнює чотирьом або восьми байтам, в залежності від архітектури. Таке надмірне використання пам’яті також негативно впливає на швидкість.

Отже, String є першим типом даних, що спадає на думку, коли говоримо про текст. Він продиктований стандартом мови Haskell і підтримує функції для роботи зі списками. Але його представлення у пам’яті є неефективним та повільним, тож для обробки великих текстових даних він підходить погано.

## 1.2 Тип Text

В більшості мов програмування стрічки тексту зберігаються у вигляді масиву. Ця структура даних є швидкою та займає менше місця, у порівнянні зі зв’язним списком. Саме так у Haskell реалізовано внутрішнє представлення типу даних Text. Для його використання необхідно імпортувати модуль Data.Text. Зазвичай, для зручності, використовують іменований імпорт однією літерою (див. схему 1.1). Іменований імпорт також необхідний для уникання конфліктів зі стандартними функціями Prelude — більшість функцій цього модуля їх імітують, тобто використовують те саме ім’я для зручності переходу на тип даних Text.

```
import qualified Data.Text as T
```

1.1

Саме Text є пріоритетним вибором типу даних для орієнтованих на текст програм. У порівнянні зі стандартним String, він виграє як у швидкодії, так і за займаємим місцем у пам’яті. Накладні витрати для зберігання одного рядка у вигляді Text становлять шість машинних слів.

Окрім того, модуль Data.Text надає «жорстку» реалізацію рядка, на відміну від «лінивого» String. Це означає, що рядок буде оброблено весь повністю, що зазвичай є кориснішим в контексті роботи з текстом. Але це також означає, що в пам’яті буде знаходитись одночасно цілий рядок. Обробка не може початись до того, як буде прочитано весь файл, що передано на вхід, наприклад. Тож потоковий стиль може бути кращим у випадку, коли ми знаємо наперед, що знадобиться лише частина рядка. У такому разі в нагоді стане пакет Data.Text.Lazy — цей модуль надає ті ж самі можливості, що і Data.Text, але з «лінивою» реалізацією механізму роботи. За необхідності використання обох модулів одночасно їх іменований імпорт виглядає як на схемі 1.2. Строго кажучи, Text.Lazy є окремим типом даних. [2]

```
import qualified Data.Text as TS

import qualified Data.Text.Lazy as TL
```

1.2

Однією з головних переваг типу даних Text над String є те, що функції його модулю мають можливість «зливатись» між собою. Це означає, що на кожне перетворення рядка компілятор не буде створювати новий алокатор, а оптимізує витрати та проведе всі маніпуляції над одним об’єктом, не засмічуючи пам’ять. [3]

Проблеми можуть виникнути тільки зі стандартними функціями Prelude, які приймають на вхід (чи повертають) саме тип String. Але частина з цих функцій може бути замінена на аналогічні з модуля Data.Text. Для всіх інших використовують спеціальні утиліти pack та unpack (див. схему 1.3). Як видно із сигнатури цих функцій, перша з них перетворює String в Text, а друга навпаки, відповідно.

```
T.pack :: String -> T.Text

T.unpack :: T.Text -> String
```

1.3

Конвертація одного типу даних в інший є ресурсоємною та не являється бажаною. Тож варто уникати цих операцій, якщо можна. Одна з базових речей, що може знадобитись при роботі з текстом — вивід його на екран, чи будь-який інший пристрій виводу, для користувача. Зазвичай в Haskell використовується пакет IO для подібних дій, який, в свою чергу, працює зі String. Аби уникнути необхідності перетворень Text у String щоразу, варто використати пакет Data.Text.IO, що містить ті самі утиліти, що і стандартний IO, з єдиною відмінністю у використанні Text замість String.

Вище вже було зазначено, що Prelude сприймає будь-який літерал виду «text» як сутність з типом даних String. Це означає, що не можна створити змінну типу Text і надати їй значення, просто написавши щось у лапках. Це

буде розпізнано як список символів, що в жорстко типізованій мові Haskell не може бути підставлено на місце типу Text. Для вирішення цієї проблеми потрібно застосувати деякі розширення мови, а саме OverloadedStrings. Це одна з найпоширеніших розширень, яке використовують під час програмування на Haskell для практичного застосування. Воно робить усі текстові типи — String, Text, ByteString, — поліморфними. А отже дає можливість ініціалізувати їх за допомогою рядків тексту в лапках, так само, як можна ініціалізувати усі числові типи цифрами.

Для використання розширень мови існує два способи. Перший — під час компіляції. Тобто, безпосередньо під час роботи з GHC застосувати директиву. Другий — записати прагму прямо в програмному коді. Цей метод є більш пріоритетним, оскільки зручніший для використання. Не потрібно пам’ятати про всілякі директиви кожен раз при компілюванні чи завантажені коду. Обидва способи зображені на схемі 1.4, номери 1 та 2 відповідно.

```
$ ghc program.hs -XOverloadedStrings     (1)

{-# LANGUAGE OverloadedStrings #-}       (2)
```

1.4

Ще однією перевагою типу Text є те, що він підтримує Unicode. Звісно, в загальному випадку можна сказати, що String теж це може. Просто стандартний модуль від Prelude не слідкує за всіма нюансами символів, що може зіпсувати текст якоюсь специфічною мовою. String очікує, що всі символи всередині його списку будуть поводитись однаково. Незалежно від змін, що до них застосовано. Але в контексті Unicode це неправда, тож може призвести до помилок та непорозумінь. В свою чергу, Text має більш обережний підхід до кожного окремого символу. Це дозволяє уникнути подібних помилок, але також впливає на швидкість обрахунку довжини рядка: оскільки кожен символ має різну довжину, необхідно перевіряти кожен окремо, щоб не помилитись в оцінці загального розміру. Крім того, String не слідкує, чи валідний кожен окремий символ всередині нього. Таким чином є

вірогідність отримати помилку вже після усіх розрахунків. Тип Text більш уважний до речей подібного виду, тож помилка буде видна одразу, як тільки виникне, а не коли настане час презентувати результат. Ця опція корисна для відлагодження великих програм.

Отже, тип даних Text має ряд переваг, що в основному виражені в його швидкодії та кількості місця в пам'яті під нього. Цей тип є рекомендованим для використання в практичних застосунках під час роботи з текстом.

## 1.3 Тип ByteString

Наступний тип даних, якщо говорити строго, не зовсім текстовий. ByteString призначений для роботи з байтами, що очевидно з назви. Всередині ця структура є списком об'єктів типу Word8, що, в свою чергу, є восьмибітним числовим представленням символу Unicode. Для використання цього типу даних необхідно імпортувати модуль Data.ByteString. Зазвичай це роблять іменовано, однією літерою, як зображено на схемі 1.5. Так само як і в ситуації з Data.Text, більшість функцій цього модуля мають такі ж самі імена, як і їхні аналоги з базового Prelude. Звісно, це робить зручнішим перехід від String до ByteString під час написання коду, але може викликати конфлікти. Тож саме іменований імпорт необхідний для уникання проблеми з однаковими іменами функцій в різних модуля. [4]

```
import qualified Data.ByteString as B
```

1.5

Головна перевага ByteString в тому, що така репрезентація є найближчою до машинного коду. Тобто це найнижчий рівень операцій. В цьому плані даний тип даних, так само як і Text, є ефективнішим за звичайний String. ByteString також і швидший за Text, оскільки другий витрачає час на правильну обробку символів Unicode. А ByteString, в свою чергу, просто оперує байтами інформації, без потреби знати, що саме вони кодують.

За потреби записати літерал цього типу, варто використовувати розширення мови OverloadedStrings, так само як і у випадку з Text. Тільки це розширення необхідне для запису літералів усіх текстових типів даних в Haskell. [5]

```
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude> import qualified Data.ByteString.Char8 as BS
Prelude BS> import qualified Data.Text.IO as T
Prelude BS T>  :set -XOverloadedStrings
Prelude BS T> putStrLn "hello, world"
hello, world
Prelude BS T> BS.putStrLn "hello, world"
hello, world
Prelude BS T> T.putStrLn "hello, world"
hello, world
Prelude BS T>
```

1.6

Тип даних ByteString не найкращий вибір саме для обробки тексту, оскільки не слідкує за зберіганням правильності кодування символів. Але, зважаючи на його швидкість та зручність для обробки великих масивів даних, він дуже добре підходить для серіалізації, обміну даних за мережею та парсингу.

Як і у випадку з модулем Data.Text, для ByteString існує його «лінивий» аналог. «Строгий» варіант цього типу даних зберігає стрічку як один великий масив. Це добре підходить для передачі даних, між C та Haskell, наприклад. «Лінивий» ByteString використовує всередині себе лінивий список зі строгих шматків, що робить його зручним для операцій вводу та виводу, як потік даних. ByteString.Lazy, по аналогії з Text. Lazy, вважається окремим типом даних, з огляду на його відмінну від базового типу поведінку.

ByteString має інтерфейс зв’язного списку, так само як і базовий текстовий тип String. Це означає, що його можна використовувати з усіма функціями, призначеними для списків, яких мовою Haskell більшість. А ще це робить перехід від String до ByteString швидким і майже непомітним, в плані коду.

Інтерфейс списку надає типу даних ByteString можливість нормально взаємодіяти зі стандартним вводом та виводом. Тобто немає необхідності імпортувати додаткові модулі для IO, як для відображення для користувача типу Text. ByteString буде поводити себе з IO від Prelude так само, як і звичайний String. Але, на відміну від нього, цей тип може містити не тільки текстові дані, що дозволяє оперувати не тільки текстом для вводу та виводу, а й іншими файлами, наприклад, зображеннями.

Проте, уже було зазначено, що модуль Data.ByteString не є найкращим рішенням для роботи з текстовими даними. Це дуже суттєво під час розпаковування рядка з ByteString у String — зазвичай інформація буде зіпсована через відсутність порозумінь у кодуванні. Щоб уникнути цієї помилки, варто використовувати модуль Data.ByteString.Char8 (схема 1.7).

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

1.7

Модуль Char8 надає символьний вид для стрічки байтів. Він робить її придатною для обробки змішаного контенту — як восьмибітних символів, так і двійкових даних. Така суміш є доволі поширеним способом зберігання інформації в багатьох файлових форматах та мережевих протоколах. Char8 також доступний і для Data.ByteString.Lazy, лише потрібен імпорт Data.ByteString.Lazy.Char8.

Але, Char8, як може бути очевидно з назви, підтримує саме восьмибітні символи. Що означає незапобіжні проблеми з обробкою символів Unicode. Великі за обсягом символи будуть розбиті на шматки по вісім біт, що, очікувано, зламає кінцеве представлення символу для користувача. Для роботи з Unicode варто звернутись до модуля UTF8.

Існує також підтип даних ShortByteString. Він має менші накладні витрати для зберігання. Модуль Data.ByteString.Short містить спеціальні функції для перетворення з ShortByteString на звичайний ByteString та навпаки. Проте, більшість інших операцій базового модуля він не підтримує. Він також не має

розширення для роботи з символами, такого як Char8 для звичайного та «лінивого» ByteString. Але цей тип даних може стати у нагоді задля зберігання багатьох коротких рядків даних в пам’яті.

Отже, тип даних ByteString створений для операцій з байтами. Цей підхід можна також використовувати і для роботи з текстом. Таке представлення є низькорівневим, близьким до машинного коду, а від цього швидким. Може бути корисним для серіалізації, обміну даних та парсингу. Але на байтовому рівні важко слідкувати за кодуванням, а отже використання цього типу даних для безпосередньої роботи з текстом не вважається найкращим вибором.

## 1.4 Можливості роботи з декількома текстовими типами одночасно

Головним питанням при роботі з текстом в мові програмування Haskell є вибір типу даних для його обробки. Наразі наявно п’ять основних типів: String, Text, Text.Lazy, ByteString, ByteString.Lazy. Кожен з цих типів має свої особливості, які було розглянуто раніше. У проектах, призначених для практичного застосування, часто може виникати потреба використовувати одночасно декілька з них. Використання може бути як паралельним, тобто незалежним один від одного, так і перетинатись. У другому випадку необхідно застосувати перетворення типів. Модулі Data.Text та Data.ByteString містять всередині себе усі необхідні утиліти.

Розглянемо перетворення базового типу даних String. Для конвертування String в Text Data.Text має спеціальну функцію pack. Зворотнє перетворення також втілено в цьому модулі, така функція називається unpack. Сигнатури цих двох функцій вже були наведені раніше, на схемі 1.3. Модуль Data.Text. Lazy містить аналогічні функції. Проте, у випадку з типом даних ByteString можуть виникнути проблеми, оскільки базовий ByteString має внутрішнє представлення як масив машинних слів (схема 1.8). При спробі «розпакувати» його в String відбудеться помилка.

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

1.8

Але для роботи з текстом завжди рекомендовано використовувати модуль Data.ByteString.Char8, що має внутрішнє представлення як масив символів. Саме функція unpack з нього перетворить ByteString на String.

Ще однією важливою конвертацією є перетворення «лінивого» типу даних в «строгий» і навпаки. Бібліотеки Data.Text та Data.ByteString мають для цього аналогічні функції (схема 1.9). Тип даних ByteString, окрім цього, ще має можливість переходу між типами за допомогою функцій fromChunks і toChunks.

```
toStrict :: TL.Text -> TS.Text

fromStrict :: BS.ByteString -> BL.ByteString
```

1.9

Перетворення ж типу даних ByteString на Text має певні підводні камені. Для коректного представлення тексту потрібно знати, в якому кодуванні цей текст було збережено в масиві байтів. Усі можливі функції декодування надає модуль Data.Text.Encoding (або Data.Text.Lazy.Encoding). Найпоширенішим залишається кодування UTF-8, тож, скоріш за все, буде використана функція decodeUtf8, що на схемі 1.10. Зворотнє перетворення є простішим, оскільки коли ми кодуємо інформацію, ми початково знаємо, в якому кодуванні збираємось зберігати текст. В цілому модуль Data.Text.Encoding надає можливості роботи з UTF-8, UTF-16, UTF-32, також враховуючи різницю little та big endian для останній двох.

```
decodeUtf8 :: ByteString -> Text

encodeUtf8 :: Text -> ByteString
```

1.10

Отже, бібліотеки містять усі необхідні утиліти для взаємодії між собою, тож їхнє одночасне вживання не є проблемою. Варто пам’ятати, що операції перетворення типів є ресурсоємними, тож їх варто уникати за можливості.

# 2. ПРОЕКТУВАННЯ ПРОГРАМИ З ВИКОРИСТАННЯМ ТЕКСТОВИХ ТИПІВ ДАНИХ

## 2.1 Загальний опис програми

Для демонстрації роботи з текстовими типами даних була спроектована програма, що оброблює файли. На вхід отримуємо розташування файлу з розширенням EPUB. Цей формат є обгорткою над zip-архівом, тобто, якщо в цьому архіві запаковані певні файли, ми можемо їх дістати.

Програма дістає XML контейнер з архіву EPUB та оброблює метадані з нього. Ці дані будуть збережені у вигляді файлу з форматуванням виду Markdown. Окрім цього, з архіву EPUB розархівовуються усі зображення і зберігаються в окрему теку.

В даному проекті продемонстровано використання типу даних ByteString для обробки архівованих файлів, читання та запису файлів зображень. Тип даних Text використано для обробки текстової інформації та запису готового результату в файл.

## 2.2 Робота з файлами

Першим етапом роботи програми є читання файлу. Як вже було зазначено вище, програма приймає на вхід файл з розширенням EPUB, опрацьовує його зміст та зберігає текстову частину у вигляді файлу з оформленням виду Markdown. Перш ніж почати обробку вмісту файлу, необхідно правильно прочитати цей файл. Як відомо, файли з розширенням EPUB являють собою zip-архіви. Для опрацювання нас цікавить файл XML, що міститься всередині архіву.

Тож, читаємо вміст архіву (схема 2.1). Читання файлів можу викликати помилки, тож їхня обробка зазначена в сигнатурі функції. В даному випадку BS є іменем для імпорту модуля Data.ByteString.Char8. Це «строгий»

ByteString, що зберігає дані у вигляді одного загального масиву символів. Саме з його допомогою ми отримуємо файл архіву для подальшої обробки.

```
getPkgPathXmlFromZip :: (MonadError String m, MonadIO m) =>
                            FilePath -> m (FilePath, T.Text)
getPkgPathXmlFromZip zipPath = do
                      zipFileBytes <- liftIO $ BS.readFile zipPath
                      getPkgPathXmlFromBS zipFileBytes
```

2.1

Як видно зі схеми 2.1, відбувається виклик функції getPkgPathXmlFromBS. Саме ця функція робить основну роботу — виділяє для нас шлях до файлу та його вміст (схема 2.2). На вхід тут ми отримуємо «строгий» ByteString. Для видобування ж файлу з архіву нам необхідно перетворити цей тип даних на «лінивий», задля роботи в потоковому стилі. Робимо це за допомогою функції fromChunks.

```
getPkgPathXmlFromBS :: (MonadError String m, MonadIO m) =>
                          BS.ByteString -> m (FilePath, T.Text)
getPkgPathXmlFromBS strictBytes = do
   -- converting strict byte string into a lazy one
   let lazyBytes = fromChunks [strictBytes]
   result <- liftIO $ ( try $ evaluate
      (toArchive lazyBytes) :: IO (Either SomeException Archive) )
   archive <- either (throwError . show) return result

   containerDoc <- fileFromArchive containerPath archive

   rootPath <- locateRootFile containerPath containerDoc
   rootContents <- fileFromArchive rootPath archive
   return (rootPath, rootContents)
```

2.2

У функції для отримання файлу з архіву ми зберігаємо текстовий результат. Для кращої його обробки це зроблено за допомогою типу даних Text. Перетворення з ByteString на Text відбувається за допомогою функції decode, яку надає модуль Data.Text.Encoding. Використано кодування UTF-8, що є наразі найбільш розповсюдженим та достатнім для задовільних результатів на тестовому прикладі.

```
-- Extract a file from a zip archive
fileFromArchive :: MonadError String m => FilePath -> Archive -> m T.Text
fileFromArchive filePath archive = do
                let mbEntry = findEntryByPath filePath archive
                maybe
                   (throwError $ "Unable to locate file " ++ filePath)
                   (return . decodeUtf8 . B.toStrict . fromEntry) mbEntry
```

2.3

Основна робота з файлом відбувається у функції convertFile (схема 2.4). Як видно з сигнатури функції, першим параметром є функція, що приймає на вхід тип даних Text і повертає його ж. Саме за допомогою цього параметру в програмі буде відбуватись перетворення файлу виду XML на Markdown. Параметр markdownOutput як раз відповідає за утримання обробленого вмісту файлу. Ця частина програми буде розглянута в розділі 2.3. Наразі розглянемо механізм збереження файлів.

```
convertFile :: (T.Text -> T.Text) -> FilePath -> FilePath -> T.Text -> IO()
convertFile converterFunc rootDir filePath file = do
            let relativePath = makeRelative rootDir filePath
            let outputFilePath = markdownFilePath relativePath
            let markdownOutput = converterFunc file
            saveFile outputFilePath markdownOutput
```

2.4

Окрім параметру markdownOutput, призначення якого вже було зазначено, маємо ще параметр outputFilePath. Це шлях, куди буде збережено файл результату. Отримуємо його простим перейменуванням вхідного файлу, для більшої зручності (схема 2.5). Для цього найбільш суттєвим є зміна розширення файлу, в нашому потрібно «.md». Це виконується бібліотечною функцією з модуля System.FilePath — replaceExtension.

```
markdownFilePath :: FilePath -> FilePath
markdownFilePath path = replaceExtension (joinPath [outputDir, path]) ".md"
               where
                 endsWithSlash = last path == '/'
                 rootDirName = if endsWithSlash
                               then takeDirectory path
                               else takeFileName path
                 outputDirName = rootDirName ++ "-md"
                 replaceRootNameWith = if endsWithSlash
                                       then replaceDirectory path
                                       else replaceFileName path
                 outputDir = replaceRootNameWith outputDirName
```

2.5

Головна робота по збереженню файлу виконана з використанням функцій модуля Data.Text, а точніше Data.Text.IO (в проекті імпортовано іменовано як IOText). Цей модуль надає всі потрібні функції для роботи з вводом та виводом, в тому числі і для читання та запису з файлів. В нашому випадку для збереження текстового контенту використовуємо функцію IOText.writeFile (схема 2.6).

```
saveFile :: FilePath -> T.Text -> IO ()
saveFile filepath content = do
        let directory = takeDirectory filepath
        createDirectoryIfMissing True directory
        IOText.writeFile filepath content
```

2.6

Отже, для роботи з файлами було використано одночасно і Data.Text, і Data.ByteString. Їхнє місце використання відрізняється між собою: ByteString використано для читання файлу архіву, а Text для кінцевого представлення тексту та його запису в файл. Також було використано перетворення звичайного ByteString на «лінивий» задля поступової потокової обробки архівного файлу. Повний код модуля для роботи з файлами наведено в додатку A.1.

## 2.3 Обробка вмісту документу з розширенням EPUB

Другим етапом роботи програми є обробка файлу, а точніше перетворення XML документу на текст з Markdown-форматуванням. Для цього відбувається заміна тегів, притаманних формату XML, на теги, які використовуються в форматуванні Markdown. В попередньому розділі для перетворення вмісту файлу використовувалась спеціальна функція перетворень (схема 2.4). В проекті в якості даної функції використовується xhtmlToMarkdown, що зображена на схемі 2.7. Як видно зі схеми, ця функція є композицією двох інших: tagsToMarkdown та parseHtmlContent. Розглянемо їх почергово.

```
xhtmlToMarkdown :: T.Text -> T.Text
xhtmlToMarkdown = tagsToMarkdown . parseHtmlContent
```

2.7

Функція parseHtmlContent є простим застосуванням функції parseTags з модуля Text.HTML.TagSoup. Цей модуль надає зручні інструменти для роботи з HTML, і, що головне, тип даних Tag a, яким ми і будемо оперувати. В нашому випадку використовуємо Tag T.Text, для представлення вмісту тегів. (T тут — відображення іменованого імпорту модуля Data.Text).

Функція ж tagsToMarkdown виконує основну роботу. Тобто, конвертує тег стилю XML в тег стилю Markdown. Використано співставлення за зразком для обробки різних типів тегів (TagText і TagOpen), а також передбачено отримання тегу з непідходящою конструкцією (схема 2.8). Кожен іменований тег оброблюється, і буде перетворено у відповідну йому Markdown конструкцію, за допомогою функції convertTag.

```
tagsToMarkdown :: [Tag T.Text] -> T.Text
tagsToMarkdown [] = T.empty
tagsToMarkdown (TagText text:xs) = T.concat [cleanText text, tagsToMarkdown xs]
tagsToMarkdown (TagOpen name attributes:xs) =
                  T.concat [convertTag name attributes tagInnerContent,
                  "\n", tagsToMarkdown remainingTags]
         where (tagInnerContent, remainingTags) = getInnerContent name xs
tagsToMarkdown (x:xs) = tagsToMarkdown xs
```

2.8

Отже, вміст файлу буде перетворено на такий, що задовольняє стандартам стилю Markdown. Це виконано за допомогою рекурсивної обробки вмісту та типу даних Tag, який надає модуль Text.HTML.TagSoup. Розглянути код обробки детальніше можна у додатку A.2, де наведено модуль для парсингу.

## 2.4 Виділення файлів зображень з архіву EPUB

Наступним етапом в програмі є вилучення заборажень та збереження їх в окремій теці. Для цього ми в першу чергу читаємо архів EPUB та шукаємо в ньому файли з потрібним розширенням (схема 2.9). Функції для роботи з

архівами отримані з модуля Codec.Archive.Zip, що спрощує для нас внутрішнє представлення даних.

```
-- A list of all images
getAllImages :: FilePath -> IO [FilePath]
getAllImages rootDir = do
             archive <- readArchive rootDir
             return $ filter (isSuffixOf ".jpg") $ filesInArchive archive
```

2.9

За допомогою функції getAllImages ми отримали список назв зображень, а точніше, їхні файлові шляхи. Цю інформацію передаємо в наступну функцію, що на схемі 2.10.

```
-- Get path and content for image
getImgFromZip :: (MonadError String m, MonadIO m) =>
                   FilePath -> FilePath -> m (FilePath, BC.ByteString)
getImgFromZip rootDir zipPath = do
              archive <- liftIO $ readArchive rootDir
              content <- imageFromArchive zipPath archive
              return (zipPath, content)
```

2.10

Як видно з сигнатури функції, ми приймаємо на вхід тільки один шлях до файлу. Це означає, що для обробки всіх зображень потрібно буде виконати мапування цієї функції по масиву шляхів. Загалом, вона повертає пару зі шляху до зображення на першому місці та самого зображення на другому. Вміст картинки отримується за допомогою функції imageFromArchive (схема 2.11).

```
-- Extract an image from a zip archive
imageFromArchive :: MonadError String m =>
                      FilePath -> Archive -> m BC.ByteString
imageFromArchive filePath archive = do
                 let mbEntry = findEntryByPath filePath archive
                 maybe
                   (throwError $ "Unable to locate file " ++ filePath)
                   (return . BL.toStrict . fromEntry) mbEntry
```

2.11

Ця функція має аналогічну поведінку з функцією fileFromArchive, яка вже була наведена раніше. Суттєвою відмінністю є те, що для збереження зображення нам необхідно використовувати ByteString, а не Text, тож в цьому

випадку ми не декодовуємо ByteString, а лише перетворюємо його на «строгий».

Так само і з функцією saveImage — вона є аналогом saveFile, тільки оперуємо не типом даних Text, а ByteString.

```
saveImage :: FilePath -> BC.ByteString -> IO ()
saveImage filepath content = do
          let directory = takeDirectory filepath
          createDirectoryIfMissing True directory
          BC.writeFile filepath content
```

2.12

Обробка і збереження зображень відбувається у функції processImage, що на схемі 2.13. Тут ми зберігаємо зображення, яке раніше видобули з архіву, в окрему теку. Вона буде знаходитись поруч з базовим файлом, який ми обробляємо.

```
processImage :: FilePath -> FilePath -> BC.ByteString -> IO()
processImage rootDir filePath image = do
                    let relativePath = makeRelative rootDir filePath
                    let outputFilePath = joinPath ["img/", relativePath]
                    saveImage outputFilePath image
```

2.13

Таким чином, на виході маємо окрему теку, куди ми розархівували всі зображення, які містились в файлі EPUB.

## Висновки

В результаті виконаної роботи можна оцінити різницю між текстовими типами даних в мові програмування Haskell. Детальний аналіз продемонстрував сильні та слабкі сторони кожного з типів даних, а також можливості для їхнього ефективного використання. Грамотне поєднання типів може допомогти розробнику досягти максимальної ефективності у роботі з текстом, якою б не була його проектна задача. А оскільки розглянуті типи даних можна конвертувати один в інший, розробник має можливість легко між ними переключатись за потреби.

В рамках практичної частини було спроектовано програму для обробки файлів з розширенням EPUB. Програма отримує з файлу метадані та перетворює їх на файл з форматуванням Markdown. Окрім цього, програма вилучає з архівованого EPUB всі зображення та розархівовує їх в окрему теку. Цей функціонал досягнуто за допомогою використання одночасно типів даних Text та ByteString у різних частинах програми.

**Список використаних джерел**


1. Alexey Shmalko: Haskell String Types [Електронний ресурс] – Режим доступу: https://www.alexeyshmalko.com/2015/haskell-string-types/
2. Will Kurt, Get Programming With Haskell — Manning Publications, 2018 — 274 с.
3. Monday Morning Haskell, Untangling Haskell's Strings [Електронний ресурс] — Режим доступу: https://mmhaskell.com/blog/2017/5/15/untangling-haskells-strings
4. Miran Lipovača, Learn You a Haskell for Great Good! A Beginner`s Guide – San Francisco, 2011 – 154 с.
5. Will Kurt, Get Programming With Haskell — Manning Publications, 2018 — 296 с.
6. Hackage :: [Package]: bytestring: Fast, compact, strict and lazy byte strings with a list interface [Електронний ресурс] – Режим доступу: http://hackage.haskell.org/package/bytestring
7. Hackage :: [Package]: text: An efficient packed Unicode text type. [Електронний ресурс] – Режим доступу: http://hackage.haskell.org/package/text

## Додаток А. Лістинг програмного коду

1. Модуль для роботи з файлами

```
{-# LANGUAGE FlexibleContexts #-}

module IO
   ( getPkgXmlFromZip
   , getPathXmlFromZip
   , getImgFromZip
   , convertFile
   , getAllImages
   , processImage
   )
   where

import qualified Data.Text as T
import qualified Data.Text.IO as IOText
import qualified Data.ByteString as B
import qualified Data.ByteString.Lazy as BL
import qualified Data.ByteString.Char8 as BC
import qualified Data.ByteString.Lazy.Char8 as BLC
import Data.Text.Encoding
import Codec.Archive.Zip
import Control.Exception
import Control.Monad.Except
import Data.List
import System.Directory
import System.FilePath
import Control.Arrow.ListArrows ( (>>>), deep )
import Data.ByteString.Lazy ( fromChunks )
import Text.XML.HXT.Arrow.ReadDocument ( readString )
import Text.XML.HXT.Arrow.XmlArrow ( getAttrValue, hasName, isElem
)
```

```
import Text.XML.HXT.Arrow.XmlState ( no, runX, withValidate )


locateRootFile :: (MonadIO m, MonadError String m) => FilePath ->
T.Text -> m FilePath
locateRootFile containerPath' containerDoc = do
               result <- liftIO $ runX
                  (readString    [withValidate    no]    (T.unpack
containerDoc)
                  >>> deep (isElem >>> hasName "rootfile")
                  >>> getAttrValue "full-path")
               case result of
                  (p : []) -> return p
                           -> throwError $ "ERROR: rootfile full-
path missing from " ++ containerPath'


-- Extract a file from a zip archive
fileFromArchive :: MonadError String m => FilePath -> Archive ->
m T.Text
fileFromArchive filePath archive = do
                let mbEntry = findEntryByPath filePath archive
                maybe
                   (throwError  $  "Unable  to  locate  file  "  ++
filePath)
                   (return . decodeUtf8 . BL.toStrict . fromEntry)
mbEntry


containerPath :: FilePath
containerPath = "META-INF/container.xml"


getPkgPathXmlFromBS  ::  (MonadError  String  m,  MonadIO  m)  =>
BC.ByteString -> m (FilePath, T.Text)
getPkgPathXmlFromBS strictBytes = do
                   -- converting strict byte string into a lazy
one
                   let lazyBytes = fromChunks [strictBytes]
                   result <- liftIO $ ( try $ evaluate
```

```
                        (toArchive lazyBytes) :: IO (Either
SomeException Archive) )
                    archive <- either (throwError . show) return
result


                    containerDoc <- fileFromArchive containerPath
archive


                    rootPath <- locateRootFile containerPath
containerDoc
                    rootContents <- fileFromArchive rootPath
archive
                    return (rootPath, rootContents)


-- Get path and content
getPkgPathXmlFromZip :: (MonadError String m, MonadIO m) =>
FilePath -> m (FilePath, T.Text)
getPkgPathXmlFromZip zipPath = do
                     zipFileBytes <- liftIO $ BC.readFile zipPath
                     getPkgPathXmlFromBS zipFileBytes


-- Get only path
getPathXmlFromZip :: (MonadError String m, MonadIO m) => FilePath
-> m FilePath
getPathXmlFromZip zipPath = fst `liftM` getPkgPathXmlFromZip
zipPath


-- Get only content
getPkgXmlFromZip :: (MonadError String m, MonadIO m) => FilePath
-> m T.Text
getPkgXmlFromZip zipPath = snd `liftM` getPkgPathXmlFromZip
zipPath


saveFile :: FilePath -> T.Text -> IO ()
saveFile filepath content = do
         let directory = takeDirectory filepath
```

```
          createDirectoryIfMissing True directory
          IOText.writeFile filepath content

markdownFilePath :: FilePath -> FilePath
markdownFilePath path = replaceExtension (joinPath [outputDir, path]) ".md"
               where
                 endsWithSlash = last path == '/'
                 rootDirName = if endsWithSlash
                               then takeDirectory path
                               else takeFileName path
                 outputDirName = rootDirName ++ "-md"
                 replaceRootNameWith = if endsWithSlash
                                       then replaceDirectory path
                                       else replaceFileName path
                 outputDir = replaceRootNameWith outputDirName

convertFile :: (T.Text -> T.Text) -> FilePath -> FilePath -> T.Text -> IO()
convertFile converterFunc rootDir filePath file = do
            let relativePath = makeRelative rootDir filePath
            let outputFilePath = markdownFilePath relativePath
            let markdownOutput = converterFunc file
            saveFile outputFilePath markdownOutput

-- Extract an image from a zip archive
imageFromArchive :: MonadError String m => FilePath -> Archive -> m BC.ByteString
imageFromArchive filePath archive = do
                 let mbEntry = findEntryByPath filePath archive
                 maybe
                   (throwError $ "Unable to locate file " ++ filePath)
                   (return . BL.toStrict . fromEntry) mbEntry
```

```
-- Read a zip Archive from disk
readArchive :: FilePath -> IO Archive
readArchive = fmap toArchive . BL.readFile

-- Get path and content for image
getImgFromZip :: (MonadError String m, MonadIO m) => FilePath ->
FilePath -> m (FilePath, BC.ByteString)
getImgFromZip rootDir zipPath = do
              archive <- liftIO $ readArchive rootDir
              content <- imageFromArchive zipPath archive
              return (zipPath, content)

saveImage :: FilePath -> BC.ByteString -> IO ()
saveImage filepath content = do
          let directory = takeDirectory filepath
          createDirectoryIfMissing True directory
          BC.writeFile filepath content

-- A list of all images
getAllImages :: FilePath -> IO [FilePath]
getAllImages rootDir = do
             archive <- readArchive rootDir
             return $ filter (isSuffixOf ".jpg") $ filesInArchive
archive

processImage :: FilePath -> FilePath -> BC.ByteString -> IO()
processImage rootDir filePath image = do
                    let relativePath = makeRelative rootDir
filePath
                    let outputFilePath = joinPath ["img/",
relativePath]
                    saveImage outputFilePath image
```

2. Модуль для обробки тексту

```
{-# LANGUAGE OverloadedStrings #-}

module Parse
    ( xhtmlToMarkdown) where

import qualified Data.Text as T
import Text.HTML.TagSoup

-- spaces before a list item, depending on its level
spacesForLevel:: Int -> T.Text
spacesForLevel 0 = T.empty
spacesForLevel level = T.concat $ replicate (level * 2) " "

parseHtmlContent :: T.Text -> [Tag T.Text]
parseHtmlContent xhtmlString = parseTags xhtmlString

convertList :: [Tag T.Text] -> Int -> T.Text
convertList [] _ = T.empty
convertList (TagOpen "li" _ : TagOpen "ul" _: xs) level =
                T.concat [convertList nestedListInnerXhtml
(level + 1), convertList otherTags level ]
                where (nestedListInnerXhtml, otherTags) =
getInnerContent "ul" xs
convertList (TagOpen "li" _ : TagOpen "ol" _: xs) level =
                T.concat [convertList nestedListInnerXhtml
(level + 1), convertList otherTags level ]
                where (nestedListInnerXhtml, otherTags) =
getInnerContent "ol" xs
convertList (TagOpen "li" _ : xs) level =
                T.concat [spacesForLevel level, "- ",
tagsToMarkdown listItemInnerXhtml, "\n", convertList xs level]
                where (listItemInnerXhtml, otherTags) =
getInnerContent "li" xs
```

```
convertList (x:xs) level = convertList xs level


convertTag :: T.Text -> [Attribute T.Text] -> [Tag T.Text] ->
T.Text
convertTag tagName _ innerTags
          | isEmpty markdownContent = T.empty
-- headings
          | tagName == "h1" = T.concat ["# ", markdownContent,
"\n\n"]
          | tagName == "h2" = T.concat ["## ", markdownContent,
"\n\n"]
          | tagName == "h3" = T.concat ["### ",
markdownContent, "\n\n"]
          | tagName == "h4" = T.concat ["#### ",
markdownContent, "\n\n"]
          | tagName == "h5" = T.concat ["##### ",
markdownContent, "\n\n"]
          | tagName == "h6" = T.concat ["###### ",
markdownContent, "\n\n"]
-- paragraph
          | tagName == "p" = T.concat [markdownContent, "\n\n"]
-- bold, italic and code
          | tagName == "strong" = T.concat ["**",
markdownContent, "**"]
          | tagName == "em" || tagName == "i" = T.concat ["*",
markdownContent, "*"]
          | tagName == "code" = T.concat ["`", markdownContent,
"`"]
-- lists
          | tagName == "ul" || tagName == "ol" = T.concat
[convertList innerTags 0, "\n"]
-- default
          | otherwise = markdownContent
          where
            markdownContent = tagsToMarkdown innerTags
            isEmpty = T.null . T.strip
```

```
cleanText :: T.Text -> T.Text

cleanText " " = " "

cleanText text = if T.null trimmed

                 then T.empty

                 else text

          where trimmed = T.strip text


getInnerContent' :: T.Text -> [Tag T.Text] -> [Tag T.Text] ->
Int -> ([Tag T.Text], [Tag T.Text])

getInnerContent' _ innerTags [] _ = (innerTags, [])

getInnerContent' tagName innerTags (TagClose tagCloseName:xs)
level

               | tagCloseName == tagName && level == 0 =
(innerTags, xs)

               | tagCloseName == tagName && level > 0 =
getInnerContent' tagName (innerTags ++ [TagClose tagCloseName])
xs (level - 1)

               | otherwise = getInnerContent' tagName (innerTags
++ [TagClose tagCloseName]) xs level

getInnerContent' tagName innerTags (TagOpen tagOpenName
attrs:xs) level

               | tagOpenName == tagName = getInnerContent'
tagName (innerTags ++ [TagOpen tagOpenName attrs]) xs (level +
1)

               | otherwise = getInnerContent' tagName (innerTags
++ [TagOpen tagOpenName attrs]) xs level

getInnerContent' tagName innerTags (x:xs) level =
getInnerContent' tagName (innerTags ++ [x]) xs level


getInnerContent :: T.Text -> [Tag T.Text] -> ([Tag T.Text], [Tag
T.Text])

getInnerContent tagName tags = getInnerContent' tagName [] tags
0


tagsToMarkdown :: [Tag T.Text] -> T.Text

tagsToMarkdown [] = T.empty

tagsToMarkdown (TagText text:xs) = T.concat [cleanText text,
tagsToMarkdown xs]
```

```
tagsToMarkdown (TagOpen name attributes:xs) =
                  T.concat [convertTag name attributes
tagInnerContent,
                  "\n", tagsToMarkdown remainingTags]
         where (tagInnerContent, remainingTags) =
getInnerContent name xs
tagsToMarkdown (x:xs) = tagsToMarkdown xs


xhtmlToMarkdown :: T.Text -> T.Text
xhtmlToMarkdown = tagsToMarkdown . parseHtmlContent
```

3. Головний модуль виклику програми

```
import Parse
import IO
import Control.Monad.Except
import GHC.IO.Encoding
import System.Win32.Console

main :: IO ()
main = do
     setLocaleEncoding utf8
     setConsoleOutputCP 65001

     result <- runExceptT $ do

        let filePath =
"E:/Stasia_Uni/Coursework/EpubParse/book.epub"
        file <- getPkgXmlFromZip filePath
        root <- getPathXmlFromZip filePath
        liftIO $ convertFile xhtmlToMarkdown root filePath file
        images <- liftIO $ getAllImages filePath
        imagesData <- mapM (getImgFromZip filePath) images
```

```
        liftIO $ mapM_ (\(a,b) -> processImage filePath a b)
imagesData


     either putStrLn return result
```