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

Робота з даними в форматі JSON

Текстова частина до курсової роботи

за спеціальністю «Інженерія програмного забезпечення»- 121

Керівник курсової роботи
Кандидат фізико-математичних наук,
доцент
Проценко В.С.

_____
(підпис)
“____” _____ 2020 р.

Виконала студентка ІПЗ-3:
Олійник Д.В.
“____” _____ 2020 р.

Київ 2020

Календарний план виконання роботи

Тема: Робота з даними в форматі JSON

№	Назва етапу	Термін виконання	Примітка
1.	Отримання теми курсової	09.10.2019	
2.	Пошук тематичної наукової літератури	20.10.2019	
3.	Ознайомлення з науковою літературою	11.12.2019	
4.	Вивчення інформації про формат даних JSON	25.12.2019	
5.	Повторення синтаксису Haskell	10.01.2020	
6.	Ознайомлення з особливостями роботи з JSON в мові програмування Haskell	30.01.2020	
7.	Реалізація практичної частини	05.02.2020	
8.	Написання першої частини курсової роботи	23.04.2020	
9.	Написання другої частини курсової роботи	30.04.2020	
10.	Написання висновків курсової роботи	01.05.2020	
11.	Перегляд змісту роботи з керівником	08.05.2020	
12.	Внесення змін до роботи	09.05.2020	
17.	Створення презентації	10.05.2020	
18.	Захист роботи	18.05.2020	

Зміст

Вступ

Від самого початку розвитку інформатики зберігання та ефективна передача даних є невід'ємною її частиною. Особливо важливим фактором для ефективної передачі даних являється зручність у використанні та мінімальний об'єм зайвої інформації. На сьогоднішній день одним з найпопулярніших форматів, який відповідає описаним характеристикам, є JSON.

Цей формат особливо поширений при написанні веб-застосунків, саме за умов необхідності зручного способу для обміну даних по мережі він і з'явився. JSON використовують під час обміну даним між сервером та браузером, між серверами, також даний формат часто застосовують для зберігання даних і для створення файлів конфігурації.

Особливістю JSON являється те, що він простий для розуміння людьми і не складний у використанні. Зокрема цей формат доступний для використання на багатьох сучасних мовах програмування, таких як: C, C++, C#, Java, JavaScript, Perl, Python, Haskell, PHP та інших.

Враховуючи актуальність та популярність даного формату за метою роботи являється з'ясування та детальний опис всіх тонкощів формату JSON, а також особливості роботи з ним у функціональній мові програмування Haskell.

Робота складається з трьох розділів.

Перший розділ присвячено детальному опису обраного формату даних. Розглянуто структуру, синтаксис JSON та його особливості. Також надано порівняння даного формату з іншими подібними форматами, зазначення переваг та недоліків JSON у порівнянні з ними. Крім того присутній короткий опис засобів для роботи з JSON у різних мовах програмування.

У другому розділі надано особливості роботи з форматом у функціональній мові програмування Haskell. Зокрема представлено опис трьох бібліотек, які призначені для цього. Також надана загальна характеристика щодо деталей роботи з форматом в Haskell.

Третій розділ присвячено наведенню прикладів для роботи з даними формату JSON у Haskell. Зокрема представлено приклад отримання даних цього формату від API кліматичних даних NOAA та обробка отриманої інформації.

Постановка задачі

Дослідження та вивчення особливостей роботи з даними в форматі JSON, зокрема у функціональній мові програмування Haskell.

Вимоги:

1. Надати детальний опис формату JSON.
2. Виконати аналіз проблем та особливостей використання JSON у функціональній мові програмування Haskell.
3. Надати можливі шляхи для вирішення наведених проблем.
4. Описати бібліотеки для роботи з JSON в Haskell.
5. Продемонструвати роботу з JSON.

Розділ 1: Особливості формату JSON

1.1 Ознайомлення з форматом

JSON (JavaScript Object Notation) — простий формат для обміну даними. З назви можна зрозуміти, що він походить від мови програмування JavaScript. JSON — текстовий формат, який зовсім не залежить від мови реалізації. Також він має досить невеликий та зрозумілий набір правил для форматування.

Даний формат використовує розширення .json, однак, коли він використовується в інших файлових форматах, то може бути представленим у вигляді стрічки або об’єкту.

Метою розробки JSON було створення максимально зручного мінімалістичного текстового формату для обміну даними, який буде легкий у використанні та не складний для розуміння людиною.

Формат JSON широко використовується під час написання веб-застосунків, його застосовують для передачі даних через мережу. Крім того веб-сервіси та API використовують JSON для надання загальнодоступних даних. Також даний формат застосовується для створення файлів конфігурації. До того ж JSON добре підходить для серіалізації складних структур та зберігання даних у реляційних базах даних.

Даний формат може містити чотири базових типи (стрічки, числа, логічні значення (false та true) та null), а також два структурні типи (об’єкти та масиви). Визначення об’єктів (“object”) та масивів (“array”) походять від відповідних термінів з JavaScript (JavaScript Naming Conventions). На верхньому рівні представлення JSON огорнений саме в один із структурних типів. Крім того об’єкти та масиви можуть бути вкладеними. [1]

1.2 Структура та синтаксис JSON

JSON базується на двох структурах даних [1]:

- Колекція пар ключ/значення, дана структура в різних мовах програмування реалізована по-різному: об’єкт, структура, іменований список або асоціативний масив.

  Приклад [2]:

```
{
    "name": "Jack (\"Bee\") Nimble",
    "format": {
        "type":       "rect",
        "width":      1920,
        "height":     1080,
        "interlace":  false,
        "frame rate": 24
    }
}
```

- Впорядкований список значень, який в більшості мов програмування реалізований як масив, вектор, список чи послідовність.

  Приклад [2]:

```
["Sunday", "Monday", "Tuesday", "Wednesday", "Thursday",
"Friday", "Saturday"]
[
    [0, -1, 0],
    [1, 0, 0],
    [0, 0, 1]
]
```

Такі структури даних унікальні тим, що підтримуються в якійсь формі майже у всіх сучасних мовах програмування.

Детальний опис типів даних, які можуть міститися в JSON форматі:

1) Значення

Кожне значення в JSON об’єкті має належати до одного з шести типів даних: стрічка, число, логічний тип даних (true, false), масив, об’єкт чи null. Такі типи як об’єкти та масиви можуть бути вкладеними.

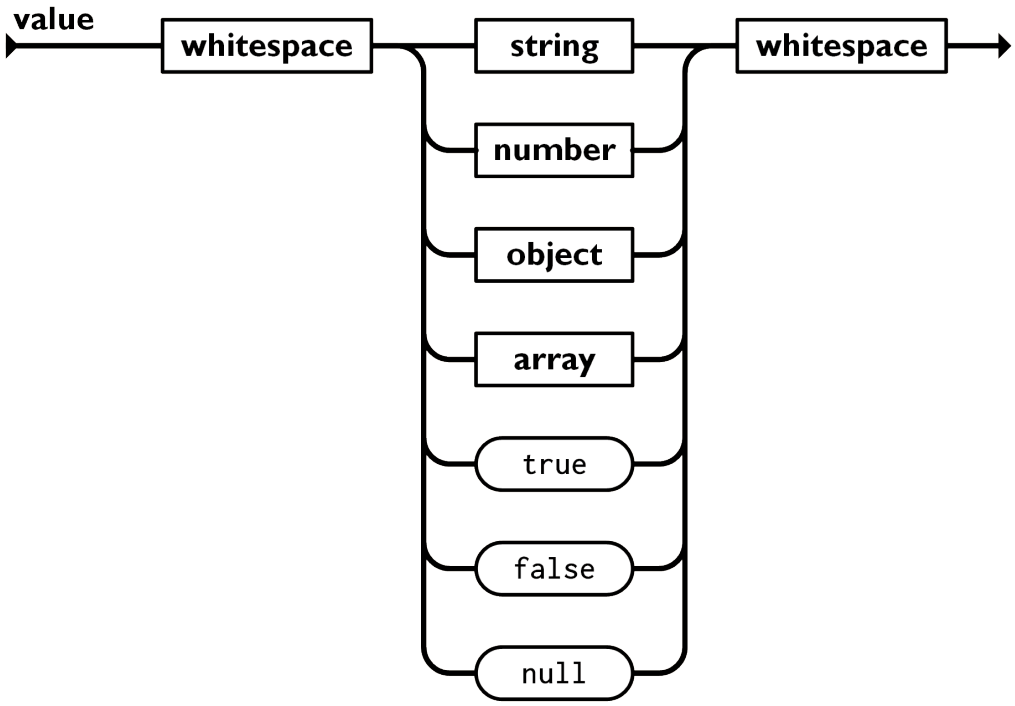

Рисунок 1.1 - Структура значень в JSON [1]

2) Об’єкти

Структура кожного об’єкту являє собою пару фігурних дужок, які огортають одну або кілька пар ключ/значення (об’єкт може містити жодної пари ключ-значення). Пари ключ-значення розділяються комою, ключ та значення — двокрапкою. Ключ знаходиться з правої сторони від двокрапки, значення — з лівої. Ключі об’єкту повинні мати унікальні значення, вони огортаються в лапки та мають формат стрічки.

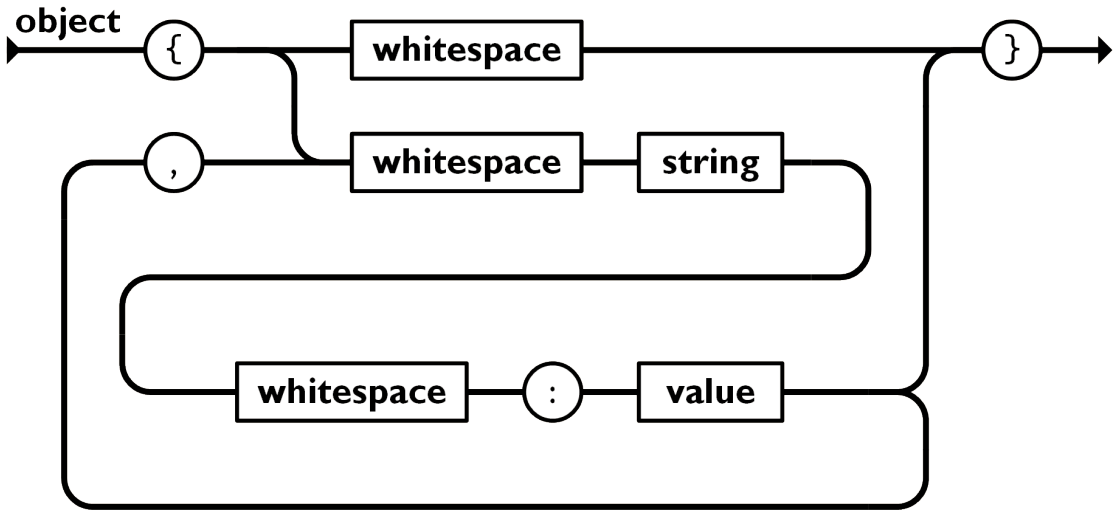

Рисунок 1.2 - Структура об'єктів в JSON [1]

3) Масиви

Масив огортається в квадратні дужки. Він може містити різні типи даних. Елементи масиву мають бути розділені комами.

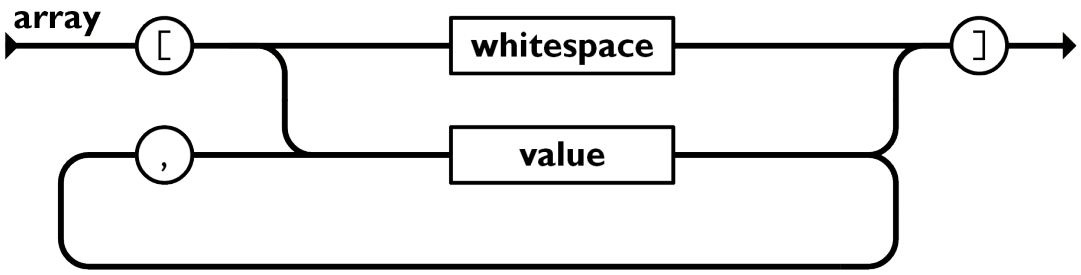

Рисунок 1.3 - Структура масивів в JSON [1]

4) Стрічки

Стрічка являє собою набір символів Unicode, які огортаються в лапки, допускається стрічка, яка не містить символів (“”). Схожий синтаксис використовується в мовах програмування C та Java.

5) Числа

Числа представлені так само, як в C та Java, однак використовується лише десяткова система числення.

1.3 Робота з форматом JSON в різних мовах програмування

Більшість сучасних мов програмування мають засоби для роботи з JSON. Зокрема популярні мови програмування такі як C++, Java, Python, C#, JavaScript підтримують даний формат, адже розуміють його важливість та зручність.

В JavaScript існує об’єкт JSON, який містить методи для роботи з цим форматом. Одним з методів є JSON.stringify(value[, replaser[, space]]) перетворює надане значення у стрічку. Необов’язкові параметри цієї функції: replacer та space — перший з них — функція, яка змінює значення перед його форматуванням в JSON, другий — аргумент, що використовується для контролю над відступами у вихідному рядку. Результат, який ми отримуємо, — стрічка формату JSON, яка називається JSON-форматованим або серіалізованим об’єктом. Отриманий рядок ми можемо відправити по мережі або помістити в базу даних.

Для того щоб із json-стрічки отримати об’єкт (щоб декодувати стрічку) використовують метод JSON.parse(text [, reviver]). Параметр reviver необов’язковий, якщо він вказаний, то кожне значення, яке було отримано з JSON-стрічки, буде перетворено відповідно до функції, вказаної у цьому параметрі. [4]

Крім JavaScript, популярною мовою для створення веб-застосунків являється PHP, тому для неї дуже важливо мати відповідні засоби для роботи з форматом JSON. У цій мові програмування підтримка JSON вбудована в ядро з версії PHP 5.2.0 — це значно полегшує роботу користувачів з форматом. Так, функція json_decode декодує стрічку JSON, а функція json_encode — повертає із заданих значень представлення даних у форматі JSON. Крім того присутні функції json_last_errer_msg та json_last_error за допомогою яких можна дізнатися про помилку, яка виникла піл час

останнього виклику json_encode() або json_decode(). json_last_error повертає саму помилку, а json_last_errer_msg — повідомлення про неї. [5]

З більш детальним переліком доступних засобів для роботи з JSON у різних мовах програмування можна ознайомитися на сайті [1].

1.4 Порівняння JSON з іншими подібними форматами

Крім JSON існують й інші формати, які використовуються для зберігання та передачі даних по мережі. Деякими з них є XML, CSV, YAML. Тому, коли говорять про недоліки та переваги JSON, то найчастіше порівнюють саме з цими форматами:

1) XML — розширювана мова розмітки, розроблена для зберігання даних. Перша відмінність від JSON, яку відразу можна помітити — це об’єм файлу для запису однакової інформації, у JSON він є значно меншим

   Наприклад:

   - вигляд у форматі JSON:

```
{“worker” : [{“name”: “Ivan”, “id”: “1”}, {“name”: “Petro”, “id”: “2”}]}
```

   (для порівняння з іншими форматами також буде використовуватися даний об’єкт)

   - вигляд того ж об’єкту в XML:

```
<?xml version="1.0" encoding="UTF-8" ?>
<root>
   <worker>
        <name>Ivan</name>
        <id>1</id>
   </worker>
   <worker>
        <name>Petro</name>
        <id>2</id>
   </worker>
```

```
</root>
```

Тож, як ми бачимо з прикладу JSON не лише більш компактний, а й зручніший для розуміння людині. Крім візуальної різниці формати мають й інші відмінності:

- дані XML не мають типів, в той час як об’єкти JSON мають;
- XML має можливість відображати дані, JSON її немає;
- JSON менш безпечний у порівнянні з XML;
- XML підтримує різні формати кодування, а JSON лише UTF-8;
- у XML є простір імен, в JSON його немає;
- у JSON дані більш легкодоступні;
- JSON підтримується більшістю браузерів, для XML міжбраузерний аналіз може створити.

2) CSV — файловий формат для представлення табличних даних, для розділення полей використовується кома та символ переходу на новий рядок. У порівнянні з JSON цей формат більш компактний.

   Приклад (вигляд даних, які містить об’єкт з минулого пункту, у CSV форматі):

```
worker/name,worker/id
Ivan,1
Petro,2
```

   Однак CSV має гірше відображення ієрархії між даними, тому часто складно відстежити зв’язки між даними. Також у JSON більш зручний підхід для додавання нових даних та редагування інформації ніж в CSV. В цілому JSON більш універсальний в порівнянні з CSV, адже в CSV форматі записи повинні мати однакові поля, в той час як в JSON об’єкти можуть мати різні. До того ж JSON може містити вкладені структури, значення можуть бути різних типів включаючи такі як масиви та об’єкти, в той час як CSV базується на двовимірному

табличному форматі та не має типів даних. Ще однією відмінністю є те, що CSV обов'язково повинен мати заголовок.

3) YAML — формат серіалізації даних, зручний для розуміння людиною.

Приклад:

```
---
worker:
- name: Ivan
  id: '1'
- name: Petro
  id: '2'
```

Порівняння з JSON:

- Як видно з прикладу YAML як і JSON достатньо зручний формат для розуміння людиною;
- YAML достатньо популярний формат, тому існує багато бібліотек для популярних мов програмування, однак JSON більш поширений;
- YAML підтримує більше типів даних ніж JSON, зокрема дати та посилання;
- На відмінну від JSON YAML підтримує коментарі;
- YAML краще підходить для конфігурації, в той час як JSON — для серіалізації.

Розділ 2: Підтримка формату JSON в функціональній мові програмування Haskell

2.1 Особливості роботи з JSON в Haskell

(Всі наведені фрагменти коду в розділі 2.1 із літературного джерела [6])

Основною проблемою під час роботи з JSON в Haskell являється те, що даний формат підтримує лише декілька простих типів: об’єкти, стрічки, числа, логічний тип та список. Тому однією з головних задач є перетворення більш складних типів з Haskell в JSON.

Для того, щоб використовувати дані з JSON формату в Haskell можна створити конструктор значень, які будуть відповідати типам формату JSON.

```
data JValue = JString String
            | JNumber Double
            | JBool Bool
            | JNull
            | JObject [(String, JValue)]
            | JArray [JValue]
              deriving (Eq, Ord, Show)
```

Щоб мати можливість перетворювати значення з Haskell в JSON слід написати функцію, яка буде це виконувати, наприклад:

```
renderJValue :: JValue -> String

renderJValue (JString s) = show s
renderJValue (JNumber n) = show n
renderJValue (JBool True) = "true"
renderJValue (JBool False) = "false"
renderJValue JNull = "null"

renderJValue (JObject o) = "{" ++ pairs o ++ "}"
     where pairs [] = ""
    pairs ps = intercalate ", " (map renderPair ps)
          renderPair (k,v) = show k ++ ": " ++ renderJValue v

renderJValue (JArray a) = "[" ++ values a ++ "]"
     where values [] = ""
     values vs = intercalate ", " (map renderJValue vs)
```

Однак, існує ще одна проблема, яка ускладнює роботу з JSON: Haskell не підтримує списків, які містять різні значення. Через це, щоб представити JSON об’єкт, який містить різні типи значень, слід кожне значення обернути

конструктором JValue, що значно обмежує гнучкість. Для вирішення цієї проблеми можна скористатися класами типів (typeclasses).

```
class JSON a where
      toJValue :: a -> JValue
      fromJValue :: JValue -> Either JSONError a

instance JSON JValue where
      toJValue = id
      fromJValue = Right
```

Так за допомогою функції toJValue можна обернути значення в JValue, використовуючи fromJValue, навпаки — перетворити JValue в значення бажаного типу, якщо це не вдасться зробити, то виведеться повідомлення про помилку. Наприклад для стрічки ці функції будуть виглядати наступним чином:

```
instance JSON String where
      toJValue = JString

      fromJValue (JString s) = Right s
      fromJValue _                   = Left "not a JSON string"
```

Однак для масивів та об’єктів реалізація мала б такий вигляд:

- для масивів:

```
instance (JSON a) => JSON [a] where
      toJValue = undefined
      fromJValue = undefined
```

- для об’єктів:

```
instance (JSON a) => JSON [(String, a)] where
      toJValue = undefined
      fromJValue = undefined
```

Тут виникає проблема в тому, що компілятор не зможе відрізнити [a] від [(String, a)], тому для її вирішення потрібно огорнути тип списків та об’єктів так, щоб компілятор не сприймав їх як списки:

- для масивів:

```
newtype JAry a = JAry {
          fromJAry :: [a]
     } deriving (Eq, Ord, Show)
```

- для об’єктів:

```
newtype JObj a = JObj {
          fromJObj :: [(String, a)]
     } deriving (Eq, Ord, Show)
```

Відповідно після цього потрібно зробити деякі зміни у визначенні типу JValue:

```
data JValue = JString String
           | JNumber Double
           | JBool Bool
           | JNull
           | JObject (JObj JValue) -- was [(String, JValue)]
           | JArray (JAry JValue) -- was [JValue]
             deriving (Eq, Ord, Show)
```

Залишилося додати необхідні функції для роботи з масивами та об’єктами. Наприклад, для масивів це буде виглядати наступним чином:

```
jaryFromJValue :: (JSON a) => JValue -> Either JSONError (JAry a)

jaryToJValue :: (JSON a) => JAry a -> JValue

instance (JSON a) => JSON (JAry a) where
     toJValue = jaryToJValue
     fromJValue = jaryFromJValue
```

Щоб перетворити стрічку JSON в JValue можна скористатися бібліотекою Parsec. Для початку потрібно зчитати текст наданої стрічки, оскільки на найвищому рівні JSON може бути або об’єктом, або масивом, то аналіз буде виглядати так:

```
p_text :: CharParser () JValue
p_text = spaces *> text
     <?> "JSON text"
     where text = JObject <$> p_object
          <|> JArray <$> p_array
```

Залишається додати функції для аналізу всіх можливих значень JSON (масиви, об’єкти, стрічки, числа, логічні значення та null). Приклад аналізатору значень:

```
p_value :: CharParser () JValue
p_value = value <* spaces
   where value = JString <$> p_string
           <|> JNumber <$> p_number
           <|> JObject <$> p_object
           <|> JArray <$> p_array
           <|> JBool <$> p_bool
           <|> JNull <$ string "null"
           <?> "JSON value"
```

2.2 Бібліотеки для роботи з JSON в Haskell

2.2.1 Text.JSON

Text.JSON — бібліотека для серіалізації значень Haskell в JSON та навпаки. Бібліотека містить відповідні конструктори для роботи з різними типами значень: JSNull, JSBool, JSRational, JSString, JSArray, JSObject. Для того, щоб генерувати правильний JSON, Haskell типи спочатку перетворюють до JSValue [7].

JSON — клас, який містить екземпляри значень для перетворення даних в та з JSON, наприклад: JSON Bool, JSON Char, JSON Int та інші. Він містить методи readJSON (для перетворення JSValue у відповідне значення) та showJSON (перетворює значення в JSValue). Також клас JSON містить відповідні методи для роботи зі списками даних: readJSONs, showJSONs.

Result — тип для граматичного розбору (parsing) JSON даних. Він містить функції encode та decode, які відповідно перетворюють значення класу JSON в стрічку формату JSON та навпаки. Також тут присутні функції encodeStrict та decodeStrint, які виконують на відмінну від попередніх функції працюють лише з верхнім рівнем JSON типів, тобто масивами та об’єктами, в той час як encode та decode дозволяють й інші.

Для кожного з типів JSNull, JSBool, JSRational, JSString, JSArray, JSObject існують відповідні методи для перетворення значень Haskell в них та навпаки. Також присутні функції для запису значень у форматі JSON.

2.2.2 RJson

RJson — бібліотека серіалізації та десеріалізації JSON даних. У цій бібліотеці вкладені типи автоматично перетворюються на відповідні об’єкти JSON та навпаки. Різні аспекти серіалізації та десеріалізації можуть бути налаштовані, однак їх можна застосувати лише до типів Haskell 98. Окрім цього бібліотека надає чітке представлення JSON даних в Haskell. Також присутній безпечний аналізатор та реалізація show (відображення даних).

Для того, щоб скористатися бібліотекою для початку потрібно додати наступні модулі та параметри [8]:

```
{-# OPTIONS_GHC
     -XTemplateHaskell
     -XFlexibleInstances
     -XMultiParamTypeClasses
     -XFlexibleContexts
     -XUndecidableInstances #-}
    import Text.RJson
    import Data.Generics.SYB.WithClass.Basics
    import Data.Generics.SYB.WithClass.Derive
```

Представлення JSON типів в Haskell виглядає наступним чином:

```
data JsonData = JDString String                  |
                    JDNumber Double              |
                    JDArray [JsonData]           |
                    JDBool Bool                  |
                    JDNull                       |
                    JDObject (M.Map String JsonData)
```

Для того, щоб серіалізувати дані ми можемо використати функцію toJSON, після її застосування ми отримаємо JsonData об’єкт, щоб перетворити результат в стрічку можна використати функцію show або toJsonString.

Для десеріалізації об’єкта JsonData застосовується функція fromJson. Можна також використати fromStringJson, що спочатку перетворює рядок на об’єкт JsonData, а потім до результату застосовує fromJson.

RJson також надає можливість реалізувати власну поведінку серіалізації та десеріалізації шляхом додавання екземплярів до класів ToJson та FromJson.

2.2.3 Aeson

Aeson — бібліотека для створення та аналізу даних в форматі JSON. Найбільш загальний спосіб використовувати цю бібліотеку — визначити тип даних відповідний до тих значень JSON з якими планується працювати надалі, а потім визначити екземпляри для FromJSON, щоб мали можливість отримати дані з JSON, та ToJSON, щоб перетворити значення в JSON формат.

Варто зазначити, що прагма LANGUARGE та екземпляр Generic дозволять писати порожні екземпляри FromJSON та ToJSON, для яких компілятор згенерує реалізації за замовчуванням.

Приклад визначення екземплярів ToJSON та FromJSON [9]:

```
{-# LANGUAGE OverloadedStrings #-}
data Person = Person {
      name :: Text
    , age  :: Int
    } deriving Show

instance FromJSON Person where
    parseJSON = withObject "Person" $ \v -> Person
        <$> v .: "name"
        <*> v .: "age"

instance ToJSON Person where
    -- this generates a Value
    toJSON (Person name age) =
```

```
object ["name" .= name, "age" .= age]
```

Можна отримати дані з формату JSON будь-якого екземпляру FromJSON. Крім того існують екземпляри для всіх стандартних типів даних, таких як: Int, String, Double та інші. Процес декодування відбувається наступним чином: біти конвертуються в Value, а потім FromJSON перетворює його у відповідний тип.

Процес створення JSON даних з типів Haskell може відбуватися двома способами:

1) Використовуючи toJSON дані конвертуються до Value, потім можливе додаткове кодування;

2) Пряме кодування шляхом використання toEncoding. Цей спосіб більш ефективний, однак доступний лише у версії aeson 0.10 та новіших.

Функції encode та decode поєднують обидва етапи. decode десеріалізує значення JSON з lazy ByteString, у результаті отримуємо відповідний тип, огорнений в тип Maybe, тобто у разі помилки під час виконання результатом буде Nothing, якщо ж операція проведена успішно отримаємо Just a. Крім decode, для декодування можуть використовуватися й такі функції:

- eitherDecode — у результаті замість Maybe повертається Either, завдяки чому у разі невдалого виконання ми можемо не лише дізнатися про те, що щось пішло не так, а й отримати повідомлення про помилку;
- decodeStrict — функціонує так само як й decode, однак замість lazy ByteString десеріалізує дані з strict ByteString
- decodeFileStrict — для десеріалізації дані зчитуються з файлу, результат огортається в тип Maybe;
- eitherDecodeStrict — функціонує так само як й eitherDecode, однак замість lazy ByteString десеріалізує дані з strict ByteString;

- eitherDecodeFileStrict —дані зчитуються з файлу, результат огортається в тип Either.

encode — серіалізує значення JSON як lazy ByteString. Також присутня функція encodeFile, вона кодує дані та одразу записує результат у файл.

Розділ 3: Приклад використання JSON в функціональній мові програмування Haskell

3.1 Короткий опис проекту

Програма створена для демонстрації роботи з форматом JSON у функціональній мові програмування Haskell.

За запуск проекту відповідає модуль Main саме тут запускаються всі необхідні функції:

```
main :: IO ()
main = do
       getDatasets
       getStations
       getNasa
       printCatFact
       readStations
       readDatasets
       saveData
       saveMainInfo
       printShortCategory
```

Перші три функції відповідають за отримання даних різних типів у форматі JSON від API. Їх результат виконання: у каталозі з’явиться три нових файли з розширенням .json:

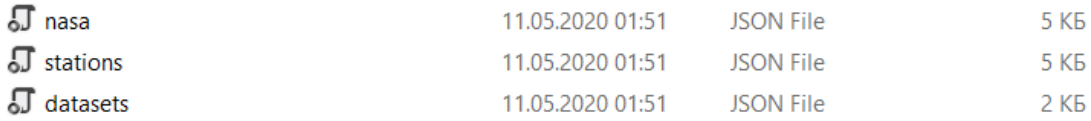

Рисунок 3.1 - Створені файли з отриманими даними

Функція printCatFact також отримує дані у форматі JSON від API, але на відмінну від попередніх одразу виводить інформацію (факт про котів) на екран:

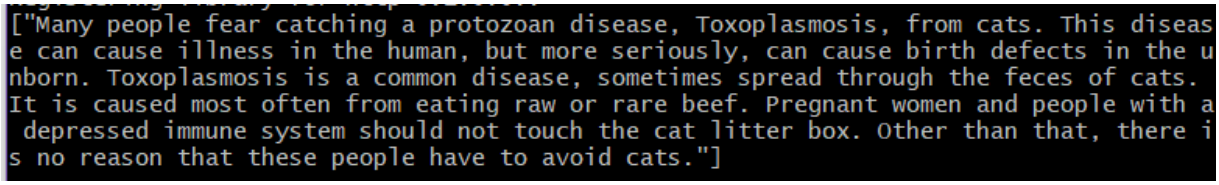

Рисунок 3.2 – Результат виконання функції printCatFact

Наступні дві функції (readDatasets та readStations) з JSON об’єктів, які знаходяться в отриманих файлах, створюють відповідні типи даних та для демонстрації роботи програми виводять поля “name” цих об’єктів:

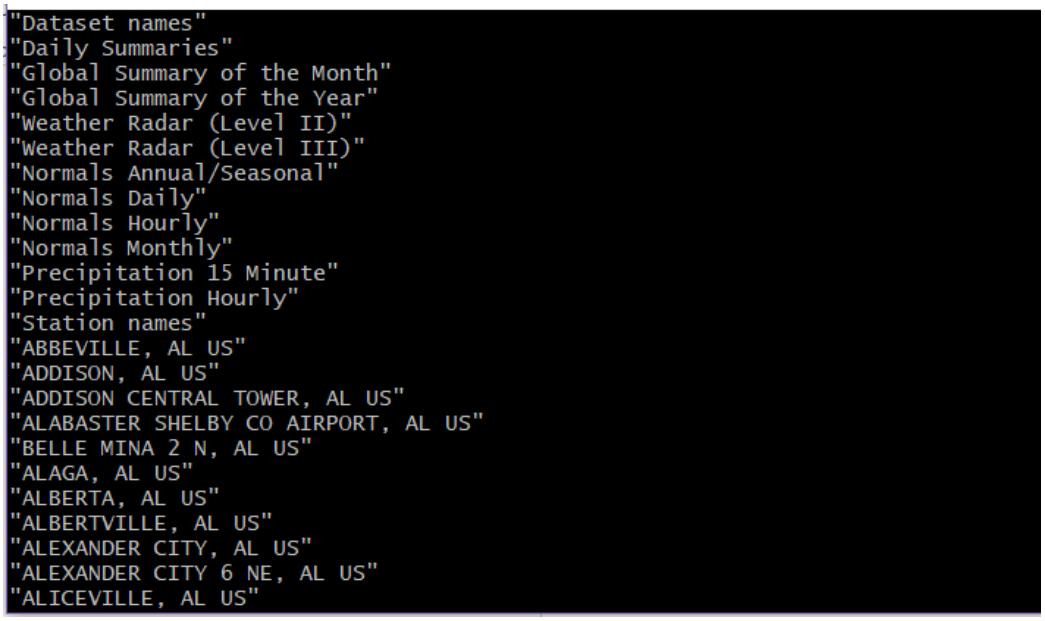

Рисунок 3.3 - Приклад виведення даних

Наступні функції, а саме :saveData та saveMainInfo створюють нові типи даних, використовуючи отримані (saveData оброблює інформацію з файлу datasets.json, а saveMainInfo — з nasa.json) та зберігають нові значення у форматі JSON, у файли datasetsName.json та categories.json відповідно.

Після виконання даної функції в каталозі проекту з’явиться нові файли з розширенням .json:

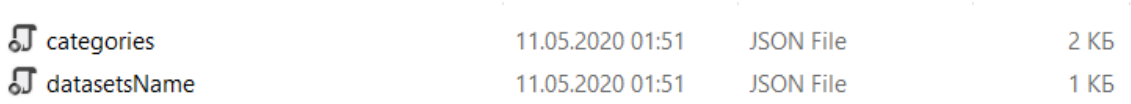

Рисунок 3.4 - Утворений файли, після створення нових даних

Остання функція серед наведених (printShortCategory) зчитує дані з раніше створеного файлу categories.json та виводить її на екран:

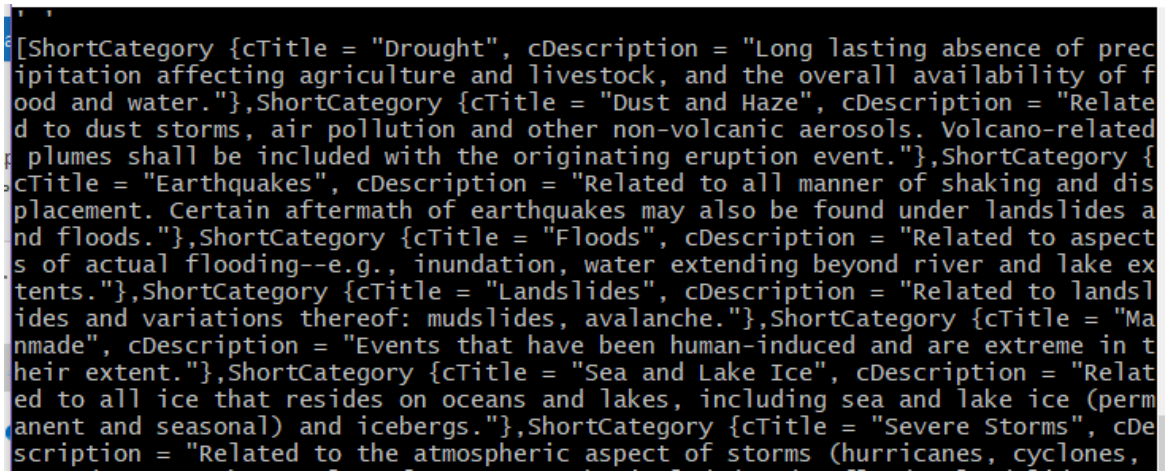

Рисунок 3.5 – Результат виконання функції printShortCategory

3.2 Опис використаних інструментів

Програма написана на функціональній мові програмування Haskell.

Для збірки пакетів було використано Stack. Stack — це кросплатформна програма для розробки проектів на Haskell. Ця програма автоматично встановлює GHC (стандартний компілятор в Haskell), також підключає пакети, необхідні для проекту, крім того вона має засоби для тестування застосунку. Після створення проект за допомогою stack, в каталозі проекту крім необхідних папок, також буде міститися файл з розширенням .cabal, в якому описуються всі додаткові бібліотеки, які потрібні.

Бібліотеки, які використовуються в проекті (частина файлу .cabal):

```
build-depends:
```

```
    base >=4.7 && <5
  , bytestring
  , http-conduit
  , text
  , aeson
```

Base — пакет, який містить Standard Haskell Prelude, генерується автоматично та містить базові інструменти для роботи в Haskell.

Bytestring — бібліотека призначена для роботи з байтовими рядками, зокрема тут рядки представлені як список об’єктів Word8. Не підтримує Unicode. Ця бібліотека зручна для серіалізації, розбору даних (parsing) та передачі інформації по мережі, саме тому вона використовується більшістю бібліотек для роботи з мережею, включаючи http-conduit.

Http-conduit — надає можливість робити HTTP запити. Зокрема використовується модуль Network.HTTP.Simple, який надає зручний та простий інтерфейс для надсилання HTTP запитів та отримання відповіді на них.

Text — бібліотека призначена для безпечної та ефективної роботи з Unicode текстом. Тип тексту представлений у вигляді масиву Word16 (UTF-16), саме це робить його використання більш ефективним ніж String, який представлений як масив Char.

Aeson — пакет, який містить всі необхідні інструменти для роботи з даними у форматі JSON.

JSON дані для обробки отримуються від трьох різний API. Одне з них — NOAA (National Oceanic and Atmospheric Administration) API. Цей API надає дані про погоду та клімат. Для того, щоб мати доступ до даних, потрібно отримати за допомогою електронної адреси унікальний токен. В залежності від кого, яку кінцеву точку буде вказано в посиланні можна отримати різні дані, наприклад [10]:

1) /datasets — усі дані CDO (Climate Data Online) зберігаються у наборах даних, ця кінцева точка покаже які набори даних доступні;
2) /locations — доступні місцеположення (точка довготи та широти);
3) /stations — надає інформацію про доступні станції (платформи для спостереження за погодою);

Інші можливі кінцеві точки можна переглянути на сайті [10].

Наступне API — The Earth Observatory Natural Event Tracker (EONET). Це клієнтський додаток NASA, що надає постійно оновлювані метадані природних подій. У проекті використовується кінцева точка, яка повертає всі можливі категорії таких подій: «/categories». [11]

Третє API, дуже просте, кожен раз повертає випадковий факт про котів.[12]

Варто зазначити, що String, Bytestring та Text — це просто різні представлення рядка. Однак для більшої ефективності в проекті використовується саме Bytestring та Text, а саме: Bytestring для представлення JSON даних, що надає можливість проводити операції над даними в найзручніший та найшвидший спосіб; та Text — для компонентів JSON, тому що цей тип даних призначений для роботи з Unicode (Bytestring не надає такої можливості) та більш ефективний ніж Sting.

3.3 Функціонал програми

Для того, щоб мати дані для обробки, потрібно спочатку їх отримати. Модуль Request відповідає за отримання даних від NOAA API. Для інших API дані отримується в тих модулях, де й оброблюються, а саме: модуль Nasa слугує для роботи з інформацією від EONET API, модуль Cat оброблює дані від API з фактами про котів. В цілому, для всіх трьох випадків процес отримання даних проходить майже однаково, тому для опису деталей обрано

лише модуль Request. HTTP запит створюється, використовуючи модуль Network.HTTP.Simple. Спочатку потрібно вказати всі необхідні змінні:

- Унікальний токен, який надає доступ до даних

```
tokenV :: BC.ByteString
tokenV = "PHcYmkVJMvbQhAXhJeqpxyTnshWdePpK"
```

- Назва хосту (у даному випадку www.ncdc.noaa.gov)

```
hostName :: BC.ByteString
hostName = "www.ncdc.noaa.gov"
```

- Шлях до потрібної категорії даних (в даному випадку datasets)

```
datasetsApiPath :: BC.ByteString
datasetsApiPath = "/cdo-web/api/v2/datasets"
```

Далі слід прописати функцію для створення запитів, вона виглядатиме наступним чином:

```
buildRequest :: BC.ByteString -> BC.ByteString -> BC.ByteString
                -> BC.ByteString -> Request
buildRequest token host method path = setRequestMethod method
                                    $ setRequestHost host
                                    $ setRequestHeader "token"
[token]
                                    $ setRequestPath path
                                    $ setRequestSecure True
                                    $ setRequestPort 443
                                    $ defaultRequest
```

Щоб використовувати цю функцію потрібно вказати token (унікальний токен), host (повне доменне ім’я), method (‘GET’ або ‘POST’) та path (шлях до кінцевої точки). Наприклад, для отримання даних про доступні набори даних від CDO застосовується такий метод:

```
datasetsRequest :: Request
datasetsRequest = buildRequest tokenV hostName "GET"
datasetsApiPath
```

Для того щоб відправити запит використовується функція httpLBS, її результатом є відповідь на запит у вигляді lazy Bytestring. Щоб записати отримані дані у файл, спочатку потрібно впевнитися, що відповідь справді

була отримана і код результату дорівнює 200. Вигляд описаних дій у програмі:

```
getDatasets :: IO ()
getDatasets = do
  response <- httpLBS datasetsRequest
  let status = getResponseStatusCode response
  if status == 200
    then do
         print "saving response about datasets to file"
         let jsonBody = getResponseBody response
         L.writeFile "datasets.json" jsonBody
    else print "datasets request failed with error"
```

Як можна помітити, у разі неуспішної обробки запиту буде виведено повідомлення про помилку, як що ж статус відповіді дорівнює 200, то результат запишеться у файл з назвою “datasets.json”.

Якщо потрібно отримати результати за іншою адресою, то потрібно змінити відповідні параметри. Наприклад, для отримання даних про доступні станції додаємо змінну з відповідним шляхом:

```
stationsApiPath :: BC.ByteString
stationsApiPath = "/cdo-web/api/v2/stations"
```

Так само створюємо нову функцію для побудови запиту для отримання станцій, використовуючи вже існуючу функцію buildRequest:

```
stationsRequest :: Request
stationsRequest = buildRequest tokenV hostName "GET"
stationsApiPath
```

Після успішного виконання запиту для отримання даних про доступні набори даних в папці проекту з’явиться файл “datasets.json” з результатом. Приклад вмісту файлу:

```
{
   "metadata":{
      "resultset":{
         "offset":1,
         "count":11,
         "limit":25
```

```
        }
    },
   "results":[
      {
         "uid":"gov.noaa.ncdc:C00861",
         "mindate":"1763-01-01",
         "maxdate":"2020-05-01",
         "name":"Daily Summaries",
         "datacoverage":1,
         "id":"GHCND"
}, …(інші значення results)... {
         "uid":"gov.noaa.ncdc:C00313",
         "mindate":"1900-01-01",
         "maxdate":"2014-01-01",
         "name":"Precipitation Hourly",
         "datacoverage":1,
         "id":"PRECIP_HLY"
}]
}
```


Аналогічним чином створюється файл з інформацією отриманою від EONET API — nasa.json. Приклад вмісту файлу:


```
{
      "title": "EONET Event Categories",
      "description": "List of all the available event categories
in the EONET system",
      "link":
"https://eonet.sci.gsfc.nasa.gov/api/v2.1/categories",
      "categories": [
            {
                  "id": 6,
                  "title": "Drought",
                  "link":
"https://eonet.sci.gsfc.nasa.gov/api/v2.1/categories/6",
                  "description": "Long lasting absence of
precipitation affecting agriculture and livestock, and the
overall availability of food and water.",
                  "layers":
"https://eonet.sci.gsfc.nasa.gov/api/v2.1/layers/6"
            },
            {
                  "id": 7,
                  "title": "Dust and Haze",
                  "link":
"https://eonet.sci.gsfc.nasa.gov/api/v2.1/categories/7",
                  "description": "Related to dust storms, air
pollution and other non-volcanic aerosols. Volcano-related plumes
shall be included with the originating eruption event.",
                  "layers":
"https://eonet.sci.gsfc.nasa.gov/api/v2.1/layers/7"
            },
            {
                  "id": 16,
                  "title": "Earthquakes",
                  "link":
```


Рисунок 3.6 – Вміст файлу nasa.json

Вигляд даних на сайті для порівняння:

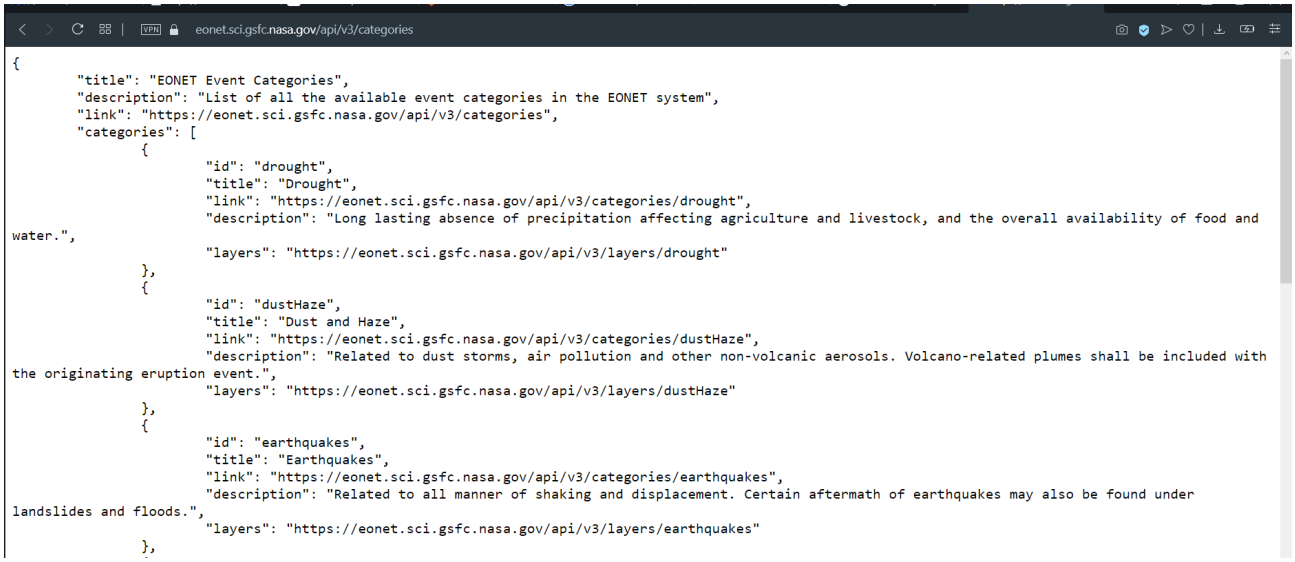

Рисунок 3.7 – Вигляд даних про категорії на сайті [11]

Варто зазначити, що для даних від API з фактами про котів, файли не створюються, вони одразу аналізуються та виводяться на екран:

```
printCatFact :: IO ()
printCatFact = do
  response <- httpLBS catRequest
  let status = getResponseStatusCode response
  if status == 200
    then do
         let jsonBody = getResponseBody response
         let cat = eitherDecode jsonBody :: Either String CatData
         case cat of
              Left l -> print l
              Right r -> print (fact r)
    else print "cat request failed with error"
```

У цій функції дані десеріалізуються за допомогою eitherDecode.

Головна частина роботи програми — саме обробка отриманих даних. Вище було наведено приклад для аналізу даних від одного з API. За обробку інформації від NOAA API відповідає модуль Read. Його функціонал включає в себе, відповідні типи для роботи з отриманими результатами запитів,

функції для виведення деяких даних, також присутні засоби для створення нового об’єкту JSON та запису цього об’єкту у файл.

Тож для того, щоб мати змогу працювати з отриманими даними, спочатку потрібно створити відповідні типи для них. Як можна помітити з прикладу вмісту файлу datasets, ми маємо об’єкт, який містить в собі вкладений об’єкт metadata та вкладений масив results. Тип для роботи з цими даними буде виглядати наступним чином:

```
data DatasetResponse = DatasetResponse
                     { metadata :: Metadata
                     , results :: [DatasetResult]
                     } deriving (Show, Generic)
instance FromJSON DatasetResponse
```

Для цього типу визначено екземпляр FromJSON (надає можливість декодувати відповідні дані з JSON формату), він генерується автоматично, так як вказано, що DatasetResponse є екземпляром касу Generic. ToJSON не визначено оскільки в програма не передбачує перетворення даного об’єкту до формату JSON.

Аналогічним чином визначено й тип для роботи з metadata, який містить вкладений об’єкт resultset, що в свою чергу містить поля offset, count та limit (всі вони можуть відносяться до типу «ціле число»). Значення даних metadata та resultset мають такий вигляд:

```
data Metadata = Metadata
                { resultset :: Resultset
                } deriving (Show, Generic)
instance FromJSON Metadata

data Resultset = Resultset
                 { offset :: Int
                 , count :: Int
                 , limit :: Int
                 } deriving (Show, Generic)
instance FromJSON Resultset
```

Залишилося надати визначення типу даних для роботи з об’єктами, які містяться в масиві results, у визначенні типу DatasetResponce можна помітити, що він матиме назву DatasetResult. Об’єкти масиву results містять поля uid, mindate, maxdate, name, datacoverage та id, однак у типі який створена назви полів дещо відрізняються, зокрема замість mindate використовується minimalDate, а замість maxdate — maximalDate. Тому для цього типу екземпляр FromJSON вказано явно:

```
data DatasetResult = DatasetResult
                   { uid :: T.Text
                   , minimalDate :: T.Text
                   , maximalDate :: T.Text
                   , name :: T.Text
                   , datacoverage :: Double
                   , id:: T.Text
                   } deriving Show

instance FromJSON DatasetResult where
 parseJSON (Object v) =
  DatasetResult <$> v .: "uid"
              <*> v .: "mindate"
              <*> v .: "maxdate"
              <*> v .: "name"
              <*> v .: "datacoverage"
              <*> v .: "id"
```

За таким самим принципом створені типи для роботи з даним, які описують доступні станції:

```
data StantionResult = StantionResult
                   { elevation :: Double
                   , mindate :: T.Text
                   , maxdate :: T.Text
                   , latitude :: Double
                   , stantionName :: T.Text
                   , stantionDatacoverage :: Double
                   , stantionId :: T.Text
                   , elevationUnit :: T.Text
```

```
                    , longitude :: Double
                    } deriving Show

instance FromJSON StantionResult where
 parseJSON (Object v) =
  StantionResult <$> v .: "elevation"
                 <*> v .: "mindate"
                 <*> v .: "maxdate"
                 <*> v .: "latitude"
                 <*> v .: "name"
                 <*> v .: "datacoverage"
                 <*> v .: "id"
                 <*> v .: "elevationUnit"
                 <*> v .: "longitude"
```

Для прикладу роботи з цими даними окремо для станцій та наборів даних є функції, які виводить назви відповідних значень. Принцип дій цих функцій: спочатку зчитуються дані з файлу, потім отримана інформація декодується до відповідних типів, далі за допомогою функції printNames виводяться імена Вигляд однієї з них для наборів даних (datasets):

```
datasetFile :: FilePath
datasetFile = "datasets.json"

readDatasets :: IO()
readDatasets = do
               jsonData <- B.readFile datasetFile
               let response = decode jsonData :: Maybe
DatasetResponse
               case results <$> response of
                    Nothing -> print "error loading data"
                    (Just r) -> printNames (Dataset r)
```

Можна помітити, що тут для десеріалізації використовується decode, в функції для виводу назв станцій (readStations) цей процес виконує decodeStrict．Приклад виводу назв наборів даних:

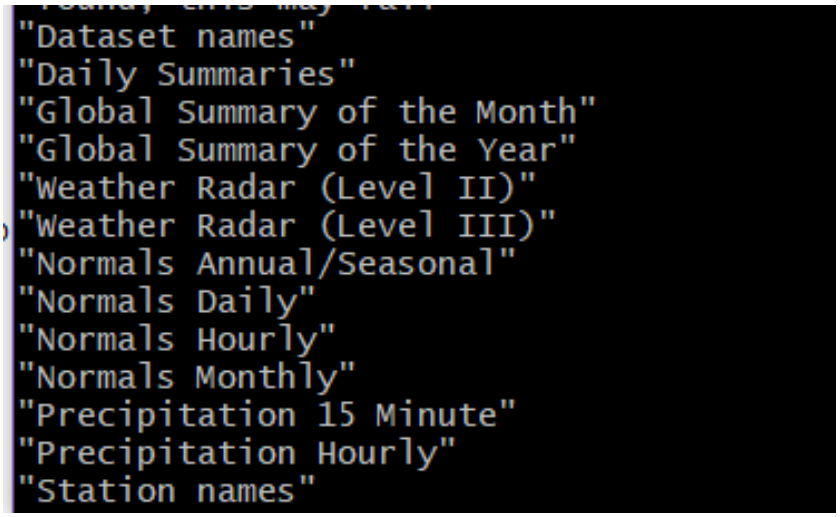

Рисунок 3.8 - Назви наборів даних, виведені з JSON файлу

Ще один приклад роботи з даними — створення даних у форматі JSON та запис результату в файл. Новий тип даних базується на DatasetResult, однак він містить лише поля nameDataset та idDataset:

```
data DatasetsElem = DatasetsElem
                    { nameDataset :: T.Text
                    , idDataset :: T.Text
                    } deriving (Show,Generic)
instance ToJSON DatasetsElem
```

Для того, щоб цей тип даних можна було перетворити у JSON формат, для нього визначається екземпляр ToJSON. Функція saveData відповідає за створення та запис даних у файл:

```
saveData :: IO()
saveData = do
           jsonData <- B.readFile datasetFile
           let response = decode jsonData :: Maybe
DatasetResponse
           let resultsData = results <$> response
           let d = getElems resultsData
           encodeFile "datasetsName.json" d
```

getElems — функція, яка зі списку DatasetResult утворює список потрібного типу, а саме DatasetsElem:

```
getElems :: Maybe [DatasetResult] -> [DatasetsElem]
```

Отримана інформація кодується у JSON формат та записується у файл datasetsName.json, все це відбувається за допомогою функції encodeFile.

Приклад вмісту такого файлу:

```
[{
      "idDataset":"GHCND",
      "nameDataset":"Daily Summaries"
},{
      "idDataset":"GSOM",
      "nameDataset":"Global Summary of the Month"
},{
      "idDataset":"GSOY",
      "nameDataset":"Global Summary of the Year"
}, …]
```

Обробка даних з файлу nasa.json (дані отримані від EONET API, функції для роботи з ними розміщені у модулі Nasa) також включає в себе створення нового типу даних на основі існуючих, а саме ShortCategory, який містить поля для назви та опису категорії (cTitle, cDescription), за допомогою функції saveMainInfo:

```
saveMainInfo :: IO()
saveMainInfo = do
          response <- eitherDecodeFileStrict "nasa.json" :: IO
          (Either String AllCategories)
          case response of
               Left l -> print l
               Right r -> saveC ( categories r)
saveC :: [Category] -> IO()
saveC c = do
          let newC = Prelude.map getData c
          let d = encode newC
          L.writeFile newDataFile d
```

За своїм призначенням наведена функція схожа на описану раніше saveData. Однак, крім того, що функції призначені для аналізу різних даних, їх відмінність, також, полягає в тому, що вони демонструють роботу різних методів серіалізації та десеріалізації JSON даних: в saveData для цих обробки процесів використано decodeFileStrict та encodeFile, а в saveMainInfo —

eitherDecodeFileStrict та encode. Особливості роботи використаних функції для аналізу JSON даних описано в розділі 2.2.3.

Також присутня функція, яка виводить на екран вміст створеного раніше файлу “categories.json” — printShortCategory. В ній дані десеріалізуються за допомогою eitherDecodeStrict:

```
printShortCategory :: IO()
printShortCategory = do
                     jsonData <- B.readFile newDataFile
                     let response = eitherDecodeStrict jsonData
:: Either String [ShortCategory]
                     case response of
                          Left l -> print l
                          Right r -> print r
```

Висновок

Під час написання курсової роботи було детально досліджено формат передачі даних JSON. Він дуже зручний у використанні, не складний для розуміння людиною та підтримується більшістю сучасних мов програмування, саме через ці фактори на даний момент JSON користується значною популярністю, особливо під час створення веб-сторінок.

Крім того було реалізовано проект, який демонструє роботу з форматом JSON у функціональній мові програмування Haskell. Результати програми чітко дають зрозуміти, що даний формат дійсно дуже зручний для передачі даних по мережі. До того ж нескладна та зрозуміла структура дозволяє легко перетворювати об’єкти JSON формату у відповідні типи, структури або класи обраної мови програмування (та навпаки).

Також було проведено порівняння JSON з іншими схожими форматами такими як: CSV, XML, YAML. Після дослідження цього питання можна сказати, що під час вибору формату варто в перш за все звертати увагу на цілі використання. Адже CSV, наприклад, краще підходить для зберігання табличних даних, однак не може зберігати дані зі складною ієрархією. XML — вважається більш безпечним форматом ніж JSON і надає можливість відображення даних, проте JSON більш зручний та зрозумілий. YAML більше підходить для конфігурації даних, JSON — для серіалізації.

Список літератури

1. https://www.json.org/json-en.html
2. http://www.json.org/fatfree.html
3. https://tools.ietf.org/html/rfc4627
4. https://developer.mozilla.org/uk/docs/Web/JavaScript/Reference/Global_Objects/JSON/stringify
5. https://www.php.net/manual/ru/book.json.php
6. Bryan O'Sullivan, John Goerzen, and Don Stewart, "Real World Haskell"
7. https://hackage.haskell.org/package/json-0.10/docs/Text-JSON.html
8. http://hackage.haskell.org/package/RJson
9. https://hackage.haskell.org/package/aeson-1.4.7.1/docs/Data-Aeson.html
10. https://www.ncdc.noaa.gov/cdo-web/webservices/v2
11. https://eonet.sci.gsfc.nasa.gov
12. https://github.com/wh-iterabb-it/meowfacts
13. Уилл К. Программируй на Haskell. ДМК Пресс, М., 2019