**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ**
**«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ**
**імені ІГОРЯ СІКОРСЬКОГО»**

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

Кафедра системного програмування і спеціалізованих комп'ютерних систем

«До захисту допущено»

Завідувач кафедри

_____ Віталій РОМАНКЕВИЧ
(підпис) (ініціали, прізвище)

"___" __2021р.

# Дипломний проєкт

**на здобуття ступеня бакалавра**

зі спеціальності **123 «Комп'ютерна інженерія»**

на тему: Система підтримки роботи з JSON-файлами для програмістів на мові Haskell

Виконав:

студент IV курсу, групи КВ-71

Хомутник Дмитро Юрійович _____
(підпис)

Керівник доц. каф. СП і СКС, к.т.н. Марченко О. І. _____
(підпис)

Консультант з нормоконтролю, доц.каф. СП і СКС, к.т.н. Клятченко Я. М. _____
(підпис)

Рецензент_____
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище та ініціали) _____
(підпис)

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2021 року

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ**
**«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ**
**імені ІГОРЯ СІКОРСЬКОГО»**

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

Кафедра системного програмування і спеціалізованих комп'ютерних систем

Рівень вищої освіти – перший (бакалаврський)

Спеціальність 123 «Комп'ютерна інженерія»

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Віталій РОМАНКЕВИЧ
(підпис) (ініціали, прізвище)
«___» _____20___ р.

**ЗАВДАННЯ**
**на дипломний проєкт студента**
Хомутника Дмитра Юрійовича

1. Тема проєкту "Система підтримки роботи з JSON-файлами для програмістів на мові Haskell", керівник проєкту Марченко О.І., кандидат технічних наук, доцент кафедри системного програмування і спеціалізованих комп'ютерних систем, затверджені наказом по університету від «__» _____2021р. №_____

2. Термін подання студентом проєкту _____

3. Вихідні дані проєкту: див. Технічне завдання

4. Зміст пояснювальної записки
   - аналіз існуючих рішень;
   - розробка системи підтримки роботи з JSON-файлами;
   - тестування системи;
   - керівництво програміста.

5. Перелік графічного матеріалу (із зазначенням обов'язкових креслеників, плакатів, презентацій, тощо)
   - структурна схема системи підтримки роботи з JSON-файлами;
   - структурна схема взаємовикликів основних функцій системи;
   - алгоритм роботи головної функції системи main;
   - алгоритм виконання запиту фільтрації даних;

- презентація за темою проєкту.

6. Консультанти розділів проєкту

| Розділ | Прізвище, ініціали та посада консультанта | Підпис, дата | |
|---|---|---|---|
| | | завдання видав | завдання прийняв |
| Нормоконтроль | Клятченко Я.М., к.т.н., доцент каф. СП і СКС | | |
| | | | |
| | | | |

7. Дата видачі завдання: 8 жовтня 2020р.

Календарний план

| № з/п | Назва етапів виконання дипломного проєкту | Терміни виконання етапів проєкту | Примітка |
|---|---|---|---|
| 1. | Вивчення літератури за тематикою проєкту | 09.11.2020 | |
| 2. | Розроблення та узгодження технічного завдання | 16.11.2020 | |
| 3. | Аналіз існуючих рішень | 19.03.2021 | |
| 4. | Розроблення структури системи | 25.03.2021 | |
| 5. | Програмна реалізація системи | 18.04.2021 | |
| 6. | Підготовка матеріалів першого розділу дипломного проєкту | 20.04.2021 | |
| 7. | Підготовка матеріалів другого розділу дипломного проєкту | 04.05.2021 | |
| 8. | Підготовка матеріалів третього розділу дипломного проєкту | 06.05.2021 | |
| 9. | Підготовка матеріалів четвертого розділу дипломного проєкту | 07.05.2021 | |
| 10. | Підготовка графічної частини дипломного проєкту | 11.05.2021 | |
| 11. | Оформлення документації дипломного проєкту | 13.05.2021 | |
| 12. | Попередній огляд матеріалів диплому на кафедрі | 25.05.2021 | |

Студент _____ Дмитро ХОМУТНИК
(підпис)

Керівник проєкту _____ Олександр МАРЧЕНКО
(підпис)

# АНОТАЦІЯ

Кваліфікаційна робота включає пояснювальну записку (54 с., 45 рис., 4 додатки).

Об’єкт розробки — створення системи для підтримки роботи з JSON-файлами для програмістів мовою Haskell, з можливістю використання у вигляді бібліотеки або застосунку для командного рядку.

Розроблена система дозволяє:

- перевіряти JSON-файли на синтаксичні помилки;
- отримувати корисні дані з JSON-файлів за допомогою виразів фільтрації;
- виконувати пошук JSON-об’єктів за критеріями;
- створювати власні синтаксичні аналізатори за допомогою інструментів, реалізованих в бібліотеці;
- використовувати окремі її модулі для виконання конкретних задач.

В процесі розробки була використана мова *Haskell*, без жодної сторонньої бібліотеки.

В ході виконання дипломного проєкту:

- проведено аналіз існуючих рішень;
- розроблено модуль комбінаторів СА;
- розроблено предметно-орієнтовану мову для виразів фільтрації та пошуку;

Використання цієї системи полегшить роботу програмістів з даними у форматі JSON.

Ключові слова: СИСТЕМА, СИНТАКСИЧНИЙ АНАЛІЗАТОР, КОМБІНАТОР СИНТАКСИЧНИХ АНАЛІЗАТОРІВ, МОНАДИЧНИЙ КОМБІНАТОР, JSON, HASKELL, ІНТЕРФЕЙС КОМАНДНОГО РЯДКУ.

ABSTRACT

Qualifying work includes an explanatory note (54 p., 45 fig., 4 applications).

The object of development is to create a system that supports work with JSON-files for developers, written in Haskell, that can be used as a library or command-line tool.

Developed system allows:

- validation of JSON-files;
- retrieving useful data from JSON-files using filtration queries;
- searching of JSON-objects by criteria;
- creating own parsers with tools that are implemented in library;
- using separate modules for specific tasks.

*Haskell* programming language was used in the development process, without any external library.

During the implementation of the diploma project:

- analysis of existing solutions was made;
- parser-combinators module developed;
- DSL for filtration and search queries was developed.

Usage of this system will ease developers' work with data in JSON format.

Keywords: SYSTEM, PARSER, PARSER-COMBINATOR, MONADIC COMBINATOR, JSON, HASKELL, COMMAND LINE INTERFACE.

| Поз. | Формат | ПОЗНАЧЕННЯ | НАЙМЕНУВАННЯ | Кількість аркушів | № прим. | Примітки |
|---|---|---|---|---|---|---|
| | А4 | ІАЛЦ.045490.002 ТЗ | Система підтримки роботи з JSON-файлами для програмістів на мові Haskell Технічне завдання | 4 | | |
| | А4 | ІАЛЦ.045490.003 ТП | Система підтримки роботи з JSON-файлами для програмістів на мові Haskell Відомість технічного проекту | 2 | | |
| | А4 | ІАЛЦ.045490.004 ПЗ | Система підтримки роботи з JSON-файлами для програмістів на мові Haskell Пояснювальна записка | 54 | | |

| Змін. | Арк. | № докум. | Підпис | Дата | ІАЛЦ.045490.001 ОА | | |
|---|---|---|---|---|---|---|---|
| Розробив | | Хомутник Д.Ю. | | | Система підтримки роботи з JSON-файлами для програмістів на мові Haskell **Опис альбому** | Літ. | Аркуш 1 / Аркушів 2 |
| Перевірив | | Марченко О.І. | | | | | |
| Консульт. | | | | | | КПІ ім. Ігоря Сікорського, ФПМ, КВ-71 | |
| Н. контроль | | Клятченко Я.М. | | | | | |
| Зав. каф. | | Романкевич В.О. | | | | | |

| Поз. | Формат | ПОЗНАЧЕННЯ | НАЙМЕНУВАННЯ | Кількість аркушів | № прим. | Примітки |
|---|---|---|---|---|---|---|
| | А4 | ІАЛЦ.045490.005 Д1 | Модулі системи. | 1 | | |
| | | | Схема структурна | | | |
| | | | | | | |
| | | | | | | |
| | А4 | ІАЛЦ.045490.006 Д2 | Взаємовиклики основних | 1 | | |
| | | | функцій системи. | | | |
| | | | Схема структурна | | | |
| | | | | | | |
| | | | | | | |
| | А4 | ІАЛЦ.045490.007 Д3 | Робота головної функції | 1 | | |
| | | | main. | | | |
| | | | Схема алгоритму | | | |
| | | | | | | |
| | | | | | | |
| | А4 | ІАЛЦ.045490.008 Д4 | Виконання запиту | 1 | | |
| | | | фільтрації даних. | | | |
| | | | Схема алгоритму | | | |
| | | | | | | |
| | | | | | | |
| | | Диск CD-ROM | Текст пояснювальної | | | |
| | | | записки. | | | |
| | | | Графічний матеріал | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |

ЗМІСТ

## 1. НАЙМЕНУВАННЯ І ОБЛАСТЬ ЗАСТОСУВАННЯ

Найменування проєкту — “Система підтримки роботи з JSON-файлами для програмастів на мові Haskell”.

Область застосування: інформаційні технології.

## 2. ПІДСТАВА ДЛЯ РОЗРОБКИ

Підставою для розробки є завдання на виконання першого (бакалаврського) рівня вищої освіти, затверджене кафедрою системного програмування та спеціалізованих комп’ютерних систем Національного технічного університету України “Київський Політехнічний Інститут імені Ігоря Сікорського”.

## 3. ЦІЛЬ І ПРИЗНАЧЕННЯ РОБОТИ

Метою даного проєкту є розробка системи  підтримки роботи з JSON-файлами для програмастів на мові Haskell, що є комплексною, гнучкою та зручною у використанні як у вихідному коді мовою Haskell, так і у інтерфейсі командного рядку.

## 4. ДЖЕРЕЛА РОБОТИ

Джерелами інформації для розроблення є технічна література, публікації у періодичних виданнях та Інтернет ресурси.

# 5. ТЕХНІЧНІ ВИМОГИ

## 5.1 Вимоги до системи, що розробляється

Система повинна забезпечувати такі функції:

- надання засобів для створення комбінаторів синтаксичних аналізаторів;
- синтаксичний аналіз даних у форматі JSON;
- форматування вихідних даних, виходячи з потреб користувача;
- використання запитів, написаних предметно-орієнтованою мовою програмування, для фільтрування та пошуку даних;
- можливість застосування у складних bash-скриптах з використанням конвеєрів, перенаправленням вхідних та/або вихідних потоків тощо.

## 5.2 Рекомендовані вимоги до апаратного забезпечення

- Процесор: Intel Core i5.
- Оперативна пам’ять: 2 Гб.
- Простір на диску: 1 Гб.

## 5.3 Вимоги до мінімального програмного забезпечення

- Операційна система Windows, Linux або Mac OS.
- Встановлений компілятор GHC для мови програмування Haskell.

## 6. ЕТАПИ РОЗРОБКИ

| № з/п | Назва етапів виконання дипломного проєкту | Терміни виконання етапів проєкту |
|---|---|---|
| 1. | Вивчення літератури за тематикою проєкту | 09.11.2020 |
| 2. | Розроблення та узгодження технічного завдання | 16.11.2020 |
| 3. | Аналіз існуючих рішень | 19.03.2021 |
| 4. | Розроблення структури системи | 25.03.2021 |
| 5. | Програмна реалізація системи | 18.04.2021 |
| 6. | Підготовка матеріалів першого розділу дипломного проєкту | 20.04.2021 |
| 7. | Підготовка матеріалів другого розділу дипломного проєкту | 04.05.2021 |
| 8. | Підготовка матеріалів третього розділу дипломного проєкту | 06.05.2021 |
| 9. | Підготовка матеріалів четвертого розділу дипломного проєкту | 07.05.2021 |
| 10. | Підготовка графічної частини дипломного проєкту | 11.05.2021 |
| 11. | Оформлення документації дипломного проєкту | 13.05.2021 |
| 12. | Попередній огляд матеріалів диплому на кафедрі | 25.05.2021 |

# ВІДОМІСТЬ ТЕХНІЧНОГО ПРОЄКТУ

| № з/п | Формат | Позначення | Найменування | Кількість листів | Примітка |
|---|---|---|---|---|---|
| 1 | А4 | ІАЛЦ.045490.000 | Завдання на дипломний проєкт | 2 | |
| 2 | А4 | ІАЛЦ.045490.001 ОА | Опис альбому | 2 | |
| 3 | А4 | ІАЛЦ.045490.002 ТЗ | Технічне завдання | 4 | |
| 4 | А4 | ІАЛЦ.045490.003 ТП | Відомість технічного проєкту | 1 | |
| 5 | А4 | ІАЛЦ.045490.004 ПЗ | Пояснювальна записка | 54 | |
| 6 | А4 | ІАЛЦ.045490.005 Д1 | Модулі системи. Схема структурна | 1 | |
| 7 | А4 | ІАЛЦ.045490.006 Д2 | Взаємовиклики основних функцій системи. Схема структурна | 1 | |
| 8 | А4 | ІАЛЦ.045490.007 Д3 | Робота головної функції main. Схема алгоритму | 1 | |
| 9 | А4 | ІАЛЦ.045490.008 Д4 | Виконання запиту фільтрації даних. Схема алгоритму | 1 | |

| | | | | ІАЛЦ.045490.003 ТП | | |
|---|---|---|---|---|---|---|
| | ПІБ | Підп. | Дата | | | |
| Розробн. | Хомутник Д.Ю. | | | Відомість технічного проєкту | Лист | Листів |
| Керівн. | Марченко О.І. | | | | 1 | 1 |
| Консульт. | | | | | КПІ ім. Ігоря Сікорського Каф. СП і СКС Гр. КВ-71 | |
| Н/контр. | Клятченко Я.М. | | | | | |
| Зав.каф. | Романкевич В.О. | | | | | |

# Пояснювальна записка
## до дипломного проєкту

на тему: Система підтримки роботи з JSON-файлами для програмістів на мові Haskell

Київ – 2021 року

ЗМІСТ

**ДОДАТКИ**

**Додаток 1. Копії графічних матеріалів**

- ІАЛЦ.045490.005 Д1. Модулі системи. Схема структурна
- ІАЛЦ.045490.006 Д2. Взаємовиклики основних функцій системи. Схема структурна
- ІАЛЦ.045490.007 Д3. Робота головної функції main. Схема алгоритму
- ІАЛЦ.045490.008 Д4. Виконання запиту фільтрації даних. Схема алгоритму

**Додаток 2. Фрагменти програмного коду**

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ

МП — Мова Програмування

СА — Синтаксичний Аналізатор

DSL — Domain-Specific Language

JSON — JavaScript Object Notation

GHC — Glasgow Haskell Compiler

EOF — End Of File

# ВСТУП

Основним призначенням мережі Інтернет є передача даних між вузлами мережі. Для форматування цих даних було розроблено багато різноманітних форматів та мов розмітки, таких як, наприклад, XML. Однак, через зростаючу популярність МП *JavaScript* у практично всіх веб-сервісах, а також через прагнення до полегшення візуального сприйняття даних людиною, було розроблено формат JSON.

На сьогодні складно знайти веб-сервіс, що не використовує формат JSON для обміну даними, адже ці дані дуже легко зчитувати та оброблювати у вихідному коді мовою *JavaScript* завдяки вбудованій функціональності у саму МП, та походження формату JSON від цієї МП. Але зі збільшенням популярності цього формату виникла необхідність використовувати обробки даних JSON у інших проєктах, написаних іншими МП. Через це було розроблено багато бібліотек для різних популярних МП, та застосунків для використання у командному рядку UNIX-систем.

Однак для менш популярних МП знайти бібліотеки для зручної обробки даних у форматі JSON може бути досить складно. У випадках, коли користувачу необхідні інструменти для комплексної обробки таких даних, кількість зовнішніх залежностей його проєкту значно виростає.

Завданням дипломного проєкту є розробка системи для програмістів мовою Haskell, модулі якої дозволяють виконувати різні комплексні задачі з використанням даних у форматі JSON. Така система може зменшити кількість явних та неявних зовнішніх залежностей проєктів користувачів та покращити процес розробки ПЗ, що оброблює дані у форматі JSON.

# 1. АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ

## 1.1 Аналіз бібліотек СА JSON файлів

На теперішній час існує багато різних бібліотек СА даних у форматі JSON для різних мов програмування. Так як система розроблювалась мовою Haskell, то буде доцільно оглянути та проаналізувати бібліотеки лише для цієї мови.

Усі розглянуті нижче бібліотеки синтаксичного аналізу побудовані за технологією комбінаторів СА. Технологія комбінаторів СА (яку також називають “монадні комбінатори СА”) вперше була описана у 1989 році у контексті створення інтерпретаторів природніх мов [1]. Практичну користь ж несе документ, написаний у 2008 році (автори Frost, Hafiz, Callaghan) [2], у якому була описана множина реалізованих комбінаторів мовою Haskell, що вирішували проблеми не поліноміального часу аналізу, неможливості обробки ліворекурсивних граматик, експоненціальної складності часу та/або простору для неоднозначних вхідних даних, які були недоліками попередніх реалізацій комбінаторів СА.

Комбінатори СА — це функції вищого порядку, що використовуються для створення базових СА і конструювання комплексних СА заданої граматики з комбінаторів підграматик цієї граматики. Комбінатори СА дозволяють визначати аналізатори у вбудованому стилі у вихідному коді, що робить їх визначення структурно схожими до правил у граматиках мов. Як такі, реалізації граматик за допомогою комбінаторів можуть розглядатися як виконувані специфікації з усіма відповідними перевагами. [2]

Основна ідея комбінаторів CA полягає у наступному:

- Існує набір базових аналізаторів, які можуть аналізувати вхідні дані (наприклад, зчитування символів в діапазоні від “0” до “9”, символ пробілу, один символ з певного набору символів тощо)
- Існує набір простих комбінаторів, що дозволяють на основі одних CA створювати інші (наприклад, аналізатор A 1 або більше разів підряд, аналізатор A, що перемежовується аналізатором B, вибір одного з аналізаторів в залежності від вхідного тексту тощо)

Такий підхід може чимось нагадувати регулярні вирази, але комбінатори CA є зручнішими у використанні через більшу легкість читання, а у деяких випадках і ефективнішими [5].

Розглянемо існуючі бібліотеки для мови Haskell.

**Бібліотека *aeson*** — розроблена для синтаксичного аналізу даних у форматі JSON. Основна властивість - представлення JSON-об’єктів за допомогою структур даних мови Haskell. На рис. 1 зображено приклад репрезентації об’єкта JSON через структуру даних мови *Haskell*.

```
{ "name": "Joe", "age": 12 }
```

We once again create a matching data type, without bothering to add a `Generic` instance this time:

```
data Person = Person {
      name :: Text
    , age  :: Int
    } deriving Show
```

Рисунок 1 – Структура даних *Person*, що співпадає з JSON-об’єктом

Даний об’єкт JSON можна представити у вигляді структури з двома полями. Для правильної роботи аналізу необхідно визначити екземпляри типів *ToJson* та *FromJson* для створеного типу *Person.* На рис. 2 зображено приклад перетворення структури типу *Person* у рядок формату JSON.

```
instance ToJSON Person where
    -- No need to provide a toJSON implementation.

    -- For efficiency, we write a simple toEncoding implementation, as
    -- the default version uses toJSON.
    toEncoding = genericToEncoding defaultOptions

instance FromJSON Person
    -- No need to provide a parseJSON implementation.
```

We can now encode a value like so:

```
>>> encode (Person {name = "Joe", age = 12})
"{\"name\":\"Joe\",\"age\":12}"
```

Рисунок 2 – Перетворення структури даних у JSON-формат

Такий підхід часто використовується у програмах, де дані отримуються напряму з HTTP-запитів для зручної роботи з ними.

Звісно, ця бібліотека також дає можливість отримати певну форму представлення синтаксичного дерева розбору і надалі вже працювати з ним, але можливості синтаксичного аналізу досить обмежені — не видається повідомлення про помилку при аналізі, сам СА JSON написаний з використанням сторонньої бібліотеки ***attoparsec***.

**Бібліотека *json*** — бібліотека, що за принципом схожа до попередньої. Для синтаксичного аналізу використовується стороння бібліотека ***parsec***, в наслідок чого синтаксичний аналіз виконується довше, ніж у бібліотеці ***aeson***. Код у репозиторії бібліотеки не оновлювався більше року. Існує невирішена проблема з синтаксичним аналізом від'ємного нуля у файлах формату JSON.

**Бібліотека *parsec*** — наразі, найбільш популярна бібліотека, що надає користувачам прості комбінатори СА для створення власних аналізаторів під вимоги конкретних задач. Визначає такі загальні комбінатори:

- Зчитування одного конкретного символу.
- Зчитування символу, що входить/не входить у список символів.

- Зчитування символу кінця файла.
- Зчитування послідовності символів.
- Використання комбінатору один і більше разів.
- Вибір першого комбінатора, з двох або більше, який завершився успіхом.
- Використання комбінатору A, розмежованого комбінатором B.

Таким чином, користувач може використовувати цю бібліотеку для створення власних комбінаторів СА, які необхідні для вирішення його задачі. Наприклад, для синтаксичного аналізу JSON-файлів, XML-файлів, CSV-файлів, виразів зі створеної власноруч граматики тощо.

**Бібліотека atto*parsec*** — бібліотека, що певним чином базується на бібліотеці ***parsec***. Основна відмінність полягає у тому, що бібліотека ***attoparsec*** працює швидше майже у 10 разів, а при правильному використанні може досягати майже однакової швидкості з аналізаторами, написаними мовою *C*. Але це є причиною і певних недоліків:

- Більша частина виграшу у продуктивності досягається використанням високоефективних комбінаторів. Якщо ж користувачеві необхідно створювати комплексні комбінатори, що повертають списки байтів або символів — продуктивність може майже не відрізнятись від такої ж реалізації на бібліотеці ***parsec***.
- Відсутні інформативні повідомлення про помилки. Відслідковування та коректне повернення усіх помилок зменшує продуктивність комбінаторів, тому бібліотека ***attoparsec*** вирішила відмовитись від інформативних помилок.
- Ще одне обмеження полягає у типі вхідних даних СА — комбінатори бібліотеки ***attoparsec*** працюють виключно з рядками типу *ByteString*, які не є рядками за замовчуванням для мови Haskell.

В цілому, ця бібліотека не така універсальна, як ***parsec***, але у більшості випадків може значно пришвидшити роботу програми клієнта, при правильному використанні.

1.2 Аналіз застосунків для роботи з JSON

Більшість веб-застосунків на сьогодні відправляють та отримують дані саме у форматі JSON (JavaScript Object Notation), який був розроблений саме з цією метою. Його легко читати людям та аналізувати комп'ютерам за допомогою СА. JSON базується на стандарті мови програмування *JavaScript* 1999 року, але, незважаючи на це, є повністю незалежним від конкретних мов. Хоча конвенції цього формату є знайомими усім програмістам, що працюють з мовами сім'ї *C*.

JSON побудований на двох структурах даних:

- Послідовність пар ім'я/значення. В різноманітних мовах програмування ця структура реалізована як об'єкт, запис, словник, хеш-таблиця, асоціативний список тощо.
- Список значень. В більшості мов програмування ця структура реалізована як масив, вектор або список.

Ці структури даних є універсальними, що полегшує роботу з ними у майже всіх сучасних мовах програмування. [6]

Усі описані нижче застосунки зчитують дані з файлів або стандартного потоку вводу, що дозволяє використовувати перенаправлення потоків систем UNIX та комбінувати виклики застосунків у конвеєрах.

**Застосунок bunz** (рис. 3)— застосунок для роботи з JSON даними у командному рядку, що написаний на мові Haskell. Призначення — форматований вивід даних у форматі JSON. Не виконує синтаксичний аналіз тексту.

```
// Beautify plain json string
bunz "{\"foo\":\"bar\"}"
{
  "foo": "bar"
}

// Beautify the input
cat test.json | bunz
{
  "name": "sendy",
  "popular": false,
  "friend": {
    "name": "\"The rock\n\n\t?",
    "points": [
      1,
      2,
      34,
      5
    ]
  }
}
```

Рисунок 3 – Застосунок ***bunz***

**Застосунок *jsawk*** — застосунок для роботи з JSON-даними у командному рядку, написаний за допомогою скриптової мови *bash*. За принципом роботи схожий до застосунку UNIX-систем ***awk***, що фільтрує та оброблює текст за допомогою регулярних виразів.

Для фільтрації даних та маніпуляцій з ним використовує мову програмування *JavaScript*, що дозволяє виконувати скрипти напряму перед або після обробки вхідних даних у форматі JSON. Звідси і недолік — для коректної роботи застосунку на комп'ютері повинен бути встановлений інтерпретатор *JavaScript*'у.

На рис. 4 зображено приклад використання застосунку у bash-скрипті.

```
resty http://example.com:8080/data*.json
GET /people/47 | jsawk 'this.favoriteColor = "blue"' | PUT /people/47
```

Рисунок 4 – Приклад використання застосунку ***jsawk***

Такий скрипт отримає JSON-файл з серверу, змінить значення поля *favoriteColor* головного об'єкту на **"blue"**, та передасть результат у запит типу *PUT,* що обновить даний ресурс на сервері.

Репозиторій застосунку не оновлювався уже більше шести років.

**Застосунок *jq*** — застосунок для роботи з JSON-даними у командному рядку, написаний мовою програмування *C* з нуля (без використання зовнішніх залежностей). За принципом роботи схожий з застосунком UNIX-систем ***sed***, що виконує різноманітні дії над текстом, використовуючи регулярні вирази.

***jq*** використовує власну предметно-орієнтовану мову програмування для опису виразів фільтрації даних. Крім можливості створення власних фільтрів, користувачу надається набір "базових" — отримання конкретного поля об'єкту, конкретного елемента масиву за індексом, та фільтрів для виконання інших різноманітних стандартних задач.

Приклади деяких фільтрів:

Фільтр на рис. 5 повертає значення, ідентичне отриманому. Це є найпростіший фільтр *jq*.

```
jq '.'
Input "Hello, world!"
Output "Hello, world!"
```

Рисунок 5 – Фільтр тотожності

Фільтр на рис. 6 отримує значення поля об'єкту з вказаним ключем, або повертає *null* у разі відсутності такого ключа.

```
       jq '.foo'
 Input {"foo": 42, "bar": "less interesting data"}
Output 42
```

Рисунок 6 – Фільтр отримання значення поля об’єкту

Фільтр на рис. 7 повертає підмножину даного масиву, що складається з елементів між індексами, вказаними у виразі.

```
       jq '.[2:4]'
 Input ["a","b","c","d","e"]
Output ["c", "d"]
```

Рисунок 7 – Фільтр отримання “зрізу” масива

Застосунок ***jq*** складається з великої кодової бази, що постійно розширюється та покращується користувачами, адже ***jq*** являється відкритим програмним забезпеченням. Останній офіційний випуск був майже 3 роки тому.

**Застосунок *jtc*** — застосунок для роботи з JSON даними у командному рядку, що був розроблений за прикладом застосунку ***jq***. Написаний мовою програмування C++.

***jtc*** надає потужний метод фільтрування та виконання дій над даними у форматі JSON. Для написання виразів використовується проста, але потужна граматика, що може бути легшою для вивчення та використання у порівнянні з граматикою виразів ***jq***.

Застосунок може бути використаний у всіх випадках, де може бути використаний ***jq***, але з деякими перевагами:

- Завдяки використанню концепта “шляху” у виразах фільтрації вони можуть бути написаними більш універсально, що допоможе уникнути помилок при деякій зміні структури даних формату JSON;
- Завдяки відсутності використання вказівників та адрес у вихідному коді напряму, застосунок гарантовано не має витоків пам’яті. Це є можливим через використання стандарту *C++14* та виключно стандартної бібліотеки шаблонів, на відміну від ***jq***;

- Підтримує багатопотоковість для обробки декількох файлів одночасно;
- В середньому, виконує однакові дії з ***jq*** за вдвічі менший час.

Застосунок ***jtc*** не є таким популярним, як ***jq***, можливо тому, що він молодший на 5 років. Його кодова база менша за кодову базу ***jq***, та підтримується не так активно. Але, не дивлячись на це, досі здійснюється пошук та виправлення помилок, а нові випуски виходять на регулярній основі.

### 1.3 Обґрунтування теми дипломного проєкту та постановка задачі

Виходячи з проведеного аналізу існуючих бібліотек та застосунків для роботи з даними у форматі JSON можна підкреслити, що кількість засобів для вирішення задач синтаксичного аналізу та обробки даних є великою та постійно зростає. Це пояснюється великою популярністю формату JSON, особливо для передачі даних при взаємодії веб-сервісів.

Однак універсальний засіб знайти важко — бібліотеки надають засоби лише для синтаксичного аналізу та представлення даних у форматі JSON за допомогою структур даних МП, застосунки надають можливості для обробки даних за допомогою командного рядку, але вони структурно не спроєктовані для використання частин їхнього вихідного коду для власних потреб клієнта.

Також варто зазначити, що переважна більшість таких засобів використовує сторонні залежності, що може не відповідати потребам користувача.

Можливі два підходи:

1) використання декількох окремих утиліт, що виконують різні дії над даними формату JSON, та якось намагатись комбінувати їх у власних проєктах;
2) розробка спеціальної системи для роботи з такими даними, використовуючи ті її частини, що необхідні для вирішення в конкретних ситуаціях.

Виходячи з того, що такої системи, принаймні написаної на МП Haskell, поки не існує, то розробка такої системи для екосистеми бібліотек МП Haskell є доцільною.

Сформулюємо функціональні властивості системи:

- Власні засоби для створення комбінаторів СА;
- Синтаксичний аналіз даних у форматі JSON;
- Форматування вихідних даних, виходячи з потреб користувача;
- Використання DSL для зручного фільтрування та пошуку даних;

Можливість застосування у складних bash-скриптах з використанням конвеєрів, перенаправлення вхідних та/або вихідних потоків тощо.

# 2. РОЗРОБКА СИСТЕМИ ПДТРИМКИ РОБОТИ З JSON-ФАЙЛАМИ

Система побудована з деяких окремих модулів, що взаємодіють певним чином. Основний модуль системи — модуль комбінаторів СА, на основі якого побудовані модулі синтаксичного аналізу тексту у форматі JSON та виразів для фільтрації даних. Базуючись на модулях СА, побудовані модуль для виконання запитів фільтрації даних та модуль генерації тексту у форматі JSON. Також окремо винесено модуль для вводу/виводу даних. Взаємодія модулів між собою та обробка запитів користувача реалізовані у головному модулі.

На рис. 8 зображено структуру схему зв’язку модулів системи.

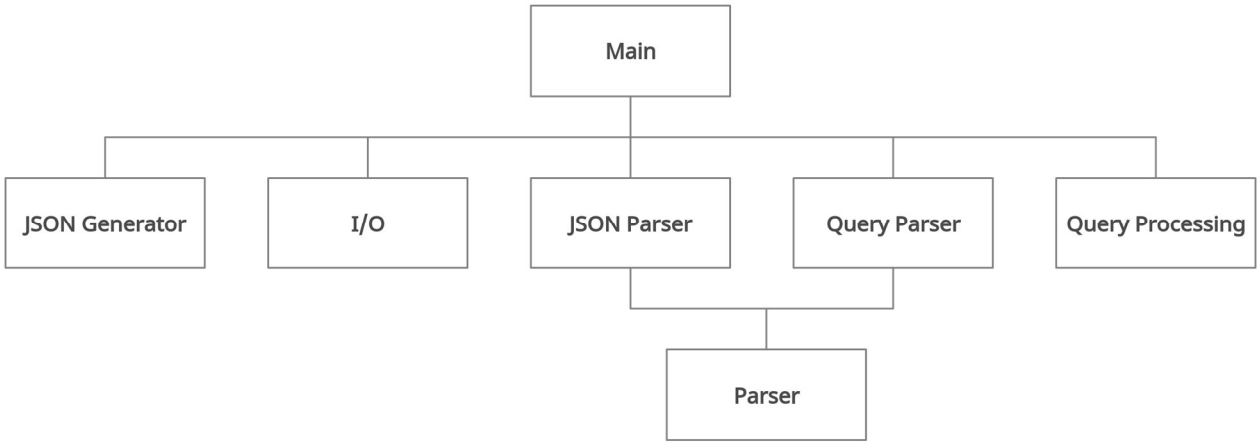


Рисунок 8 – Структурна схема системи підтримки роботи з JSON-файлами

Для кращого розуміння роботи системи, на рис. 9 наведено структурну схему взаємовикликів основних функцій модулів системи.

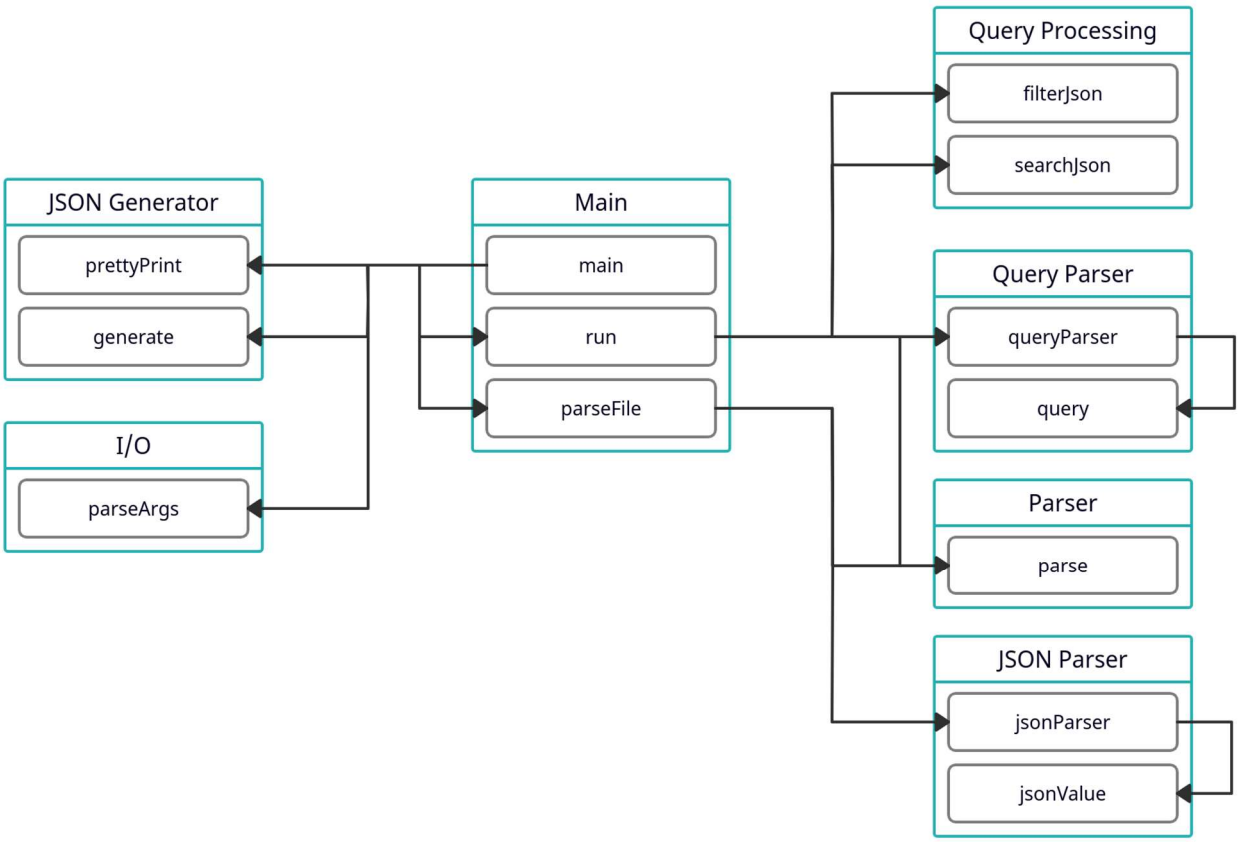


Рисунок 9 – Структурна схема взаємовикликів основних функцій системи

## 2.1 Модуль комбінаторів CA

Розглянемо модуль системи, що реалізує функції для роботи з комбінаторами CA. Основні типи даних, що оголошені у модулі:

```
newtype Parser a = Parser { parse :: Input -> Either ParseError (a, Input) }

type Position = (Int, Int)

data Input = Input { pos :: Position, input :: String } deriving (Show)

data ParseError = ParseError Position String String

data Number = Number {
  sign     :: Integer,
  integral :: Integer,
  decimal  :: Maybe Double,
  expnt    :: Maybe Integer,
  num      :: Double
  }
```

Тип *Parser* містить у собі функцію *parse*, що приймає значення типу *Input* та повертає тип *Either*, що може містити в собі помилку синтаксичного аналізу або результат у вигляді пари: значення, що було зчитане з вхідного рядку, та залишок вводу після аналізу.

Тип *Input* являється обгорткою над базовим типом *String*, та створений з метою відслідковування позиції зчитування у конкретний момент.

Тип *ParseError* допоможе відслідкувати місце помилки при аналізі та зберегти повідомлення про саму помилку.

Тип *Number* являється структурою для збереження інформації про зчитані числа, яка потім використовується для генерації вихідних даних.

Для того, щоб працювати з комбінаторами СА, необхідно створити функцію, що буде повертати наступний символ з тексту, що аналізується.

```
nextChar :: Input -> Maybe (Char, Input)
```

Дана функція приймає вхідний рядок та повертає значення типу *Maybe — Nothing,* якщо вхідний рядок порожній, або наступний символ та новий рядок, вже без цього символу. Це допоможе зручно отримати та обробити кінець вводу завдяки потужній системі типів мови Haskell.

```
char :: Char -> Parser Char
```

Ця функція створює самий базовий СА, що зчитує потрібний користувачу символ. Вона отримує символ, який необхідно зчитати, та повертає значення типу *Parser*, що зчитує цей символ з вхідного рядка.

```
string :: String -> Parser String
```

Дана функція приймає рядок (послідовність символів), та повертає значення типу *Parser*, що зчитує цю послідовність з вхідного рядка.

Цих функцій достатньо для створення СА, що можуть зчитати будь-яку послідовність символів. Але для повноцінної роботи з ними необхідно реалізувати комбінування цих СА. На мові Haskell, завдяки потужній системі типів, це робиться досить просто.

```
instance Functor Parser where
  fmap f (Parser p) =
    Parser $ \input -> do
      (v, input') <- p input
      Right (f v, input')
```

Дана реалізація класу *Functor* для значень типу *Parser* дозволяє комбінувати функції, що виконують деякі операції над результатом синтаксичного аналізу, та самі СА. Це робиться для того, щоб мати можливість виконувати маніпуляції над результатом аналізу, ще не маючи самого результату.

```
instance Applicative Parser where
  pure v = Parser $ \input -> Right (v, input)

  (Parser pf) <*> (Parser p) =
    Parser $ \input -> do
      (f, rest) <- pf input
      (v, rest') <- p rest
      pure (f v, rest')
```

Дана реалізація класу *Applicative* для значень типу *Parser* дозволяє комбінувати декілька СА у один за допомогою спеціальних функцій та типів, що надає стандартна бібліотека GHC. Це дозволяє зручно та швидко створювати нові СА комбінуючи вже створені. Наприклад, тепер можна створити СА, що зчитує ім’я змінної у дужках, скомбінувавши три аналізатори — СА, що зчитує відкриту дужку, СА, що зчитує ім’я змінної, та СА, що зчитує закриту дужку. Про помилки під час аналізу думати не потрібно — потужна система типів та реалізації потрібних класів генерують помилки, виходячи з даних, що необхідно зчитати користувачу. І якщо хоча б один з “внутрішніх” СА поверне помилку під час обробки — ця помилка буде автоматично передана на рівень вище.

```
instance Alternative Parser where
  empty = Parser $ \input -> Left $ ParseError (pos input) "" ""
  (Parser p1) <|> (Parser p2) = Parser $ \input ->
    let originalPos = pos input
        res1        = p1 input
    in
```

```
      case res1 of
        Left (ParseError position _ _) ->
          if position /= originalPos
          then res1
          else p2 input
        Right _ -> res1
```

Реалізація класу *Alternative* для значень типу *Parser* дозволяє використовувати логічні функції вибору для створення нових СА. Тобто, можна створити аналізатор, що може зчитати один з двох (або більше) символів, скомбінувавши СА, що окремо зчитують кожен з них. Це розширює можливості комбінації СА та дозволяє створювати гнучкі комбінатори СА. Необхідний СА обирається по наступному принципу: якщо аналізатор завершився не помилкою — результат аналізу повертається, не враховуючи наступні аналізатори у ланцюжку. Якщо аналізатор завершився з помилкою, то повертається зчитане значення наступного СА. Якщо ж аналізатор завершився з помилкою і він зміг зчитати хоча б один символ з вхідного рядку — увесь ланцюжок вибору провалюється з поверненням цієї помилки.

```
instance Monad Parser where
  return a = Parser $ \input -> pure (a, input)
  p >>= f = Parser $ \input ->
    case parse p input of
      Right (p', cs) -> parse (f p') cs
      Left error -> Left error
```

Остання реалізація для значень типу *Parser* — реалізація класу *Monad*. Це дає змогу зручніше комбінувати послідовність зчитувань, як у “імперативному стилі”, за допомогою вбудованих конструкцій мови Haskell.

Описаних реалізацій та функцій вистачить для того, щоб створити будь-які комбінатори СА.

Також, для більшої зручності, у цьому модулі знаходяться функції, що часто використовуються при побудові комбінаторів СА.

```
try :: Parser a -> Parser a
```

Ця функція створює з існуючого СА новий аналізатор, що при помилці поводиться так, ніби не зчитував жодного символу з вхідного рядку. Це необхідно для створення передбачаючих СА у тих випадках, де це справді потрібно. Така поведінка не обрана поведінкою за замовчуванням тому, що це може створити велике навантаження на систему через постійне повернення курсора після невдалих спроб аналізу.

```
failing :: Parser a -> Parser a
```

Ця функція містить логіку, обернену до функції *try* — вона формує новий СА з існуючого, який при помилці буде завжди імітувати зчитування хоча б одного символу з вхідного рядку. Це потрібно у випадках, коли по граматиці певний токен повинен строго знаходитись у певному місці, яке повинен проаналізувати СА.

```
oneOf :: [Parser a] -> Parser a
oneOf = foldl (<|>) empty
```

Ця функція приймає список СА та повертає новий аналізатор, що здатен обрати необхідний СА при зчитуванні. Це можливо легко зробити за допомогою саме реалізації класу *Applicative*. Якщо СА у ланцюжку вибору завершився з помилкою та зчитав хоча б один символ — увесь ланцюжок перерветься з цією помилкою.

```
tryOneOf :: [Parser a] -> Parser a
tryOneOf = foldl ((<|>) `on` try) empty
```

Ця функція схожа з функцією *oneOf*, але кожний СА зі списку перетворюється на передбачаючий СА.

```
manySepBy :: Parser a -> Parser b -> Parser [a]
```

Ця функція приймає два СА — перший зчитує певний токен, а другий зчитує роздільник — та повертає новий аналізатор, що зчитує послідовність таких токенів, розділених роздільником. Це дозволяє зручно створювати СА, що зчитують, наприклад, масиви, у яких елементи розділені комою, або інші послідовності рядків будь-якої складності.

```haskell
whitespace :: Parser Char
whitespace = oneOf [char '\n', char '\r', char '\t', char ' ']

ws :: Parser String
ws = many whitespace

ws1 :: Parser String
ws1 = some whitespace
```

СА *whitespace* зчитує символ, що рахується пробілом у вихідних файлах JSON. СА *ws* зчитує нуль або більше пробілів підряд, а *ws1* — один або більше.

```haskell
between :: Parser a -> Parser b -> Parser c -> Parser a
```

Дана функція приймає СА певного типу, та два інших СА, і повертає новий аналізатор, що застосовує спочатку другий, потім перший, а потім третій СА — тобто токен, що зчитує перший аналізатор, повинен знаходитись у вхідному рядку між токенами другого і третього СА відповідно. Це дозволяє швидко створити СА, що зчитують, наприклад, вираз, що знаходиться у дужках, або елементи масиву, що також знаходяться між дужками.

```haskell
parseWhen :: (Char -> Bool) -> Parser Char
```

Дана функція приймає предикат, який перевіряє символ, та повертає СА, що зчитує даний символ лише у випадку, якщо він задовольняє цьому предикату. Це дозволяє створити аналізатори, що зчитують, наприклад, лише цифри, або маленькі літери латинського алфавіту тощо.

Можливості для створення СА обмежені лише цілями користувача, що їх створює. Він може створити власні комбінатори або допоміжні функції для роботи з ними, використовуючи вже існуючі, і таким чином інкапсулювати логіку аналізу складних токенів та створення специфічних СА, необхідних для виконання конкретних задач його проєкту. Це є однією з головних переваг комбінаторів СА, а їхня реалізація на мові Haskell є лаконічною та структурованою, та дозволяє використовувати зручний та стислий синтаксис мови для побудови необхідних аналізаторів.

## 2.2 Модуль СА даних у форматі JSON

У цьому модулі реалізовано СА даних у форматі JSON. Основний тип даних, що оголошений у модулі:

```
data Json
  = JsonNull
  | JsonBool Bool
  | JsonString String
  | JsonNumber Double
  | JsonArray [Json]
  | JsonObject [(String, Json)]
```

Фактично, тип *Json* містить у собі перелічення всіх лексем, з яких може складатися JSON-файл.

```
jsonNull :: Parser Json
jsonNull = JsonNull <$ string "null"

jsonTrue :: Parser Json
jsonTrue = (JsonBool True) <$ string "true"

jsonFalse :: Parser Json
jsonFalse = (JsonBool False) <$ string "false"
```

Дані СА зчитують такі базові токени, як *null*, та булеві значення — *true* і *false*.

```
jsonString :: Parser Json
jsonString = JsonString <$> between (many character) (char '"') (char '"')
```

СА *jsonString* зчитує послідовність будь-яких символів між подвійними лапками, крім самих лапок та оберненої скісної риски (через те, що вона використовується при екрануванні символів у рядках).

```
jsonNumber :: Parser Json
```

Даний СА зчитує числа у JSON файлі, відповідно до стандарту (вставити посилання).

```
jsonArray :: Parser Json
```

Даний СА зчитує масив у форматі JSON — будь-які токени, що розділені комами, та знаходяться між квадратними дужками. Даний СА дуже легко створити, використовуючи комбінації створених функцій з модуля *Parser*.

```
jsonArray = JsonArray <$> (try (between (const [] <$> ws) (char '[') (char ']'))
                      <|>  between elements (char '[') (char ']'))
  where
    elements = manySepBy (failing element) (char ',')
```

Даний СА аналізує однин з двох варіантів — порожні квадратні дужки, або послідовність елементів, розділених комою. Причому, СА елемента масиву створено з використанням функції *failing*, що означає, що елемент обов'язково повинен бути присутнім на першому місці масиву або після зчитування коми — стандарт JSON не дозволяє залишати коми в кінці масивів або об'єктів [7].

```
jsonKey :: Parser Json
```

Даний СА дозволяє зчитати коректний ключ (або ім'я поля) у парах ключ-значення об'єктів JSON. Він перевіряє, що ключ починається з літери або нижнього підкреслювання, за яким слідують 0 або більше можливих символів у ключі.

```
fieldNameChar :: Parser Char
fieldNameChar = parseWhen (\c -> isDigit c || isLetter c || c == '_')
```

Даний СА дозволяє зчитати усі можливі значення символу, що може знаходитись у імені поля об'єктів JSON.

```
jsonObject :: Parser Json
```

Даний СА зчитує JSON-об'єкт — перелік пар ключ-значення, що розділені комами та знаходяться між фігурними дужками, або порожні фігурні дужки. Реалізація подібна до реалізації зчитування масиву, відрізняються лише структурні елементи.

```
jsonValue :: Parser Json
```

Це основний СА — комбінує усі вище зазначені аналізатори у один, що дозволяє повністю зчитати файл у форматі JSON.

2.3 Модуль СА виразів фільтрації та пошуку даних

У цьому модулі реалізовано СА для зчитування запитів фільтрації та пошуку даних у JSON-форматі. Розглянемо граматику таких виразів.

```
query := dot        |
         pipe       |
         comma      |
         array      |
         field      |
         comparison |
         number     |
         string     |
         bool
         ;

dot   := '.' ;
pipe  := '|' ;
comma := ',' ;

array := '[' index? ']' ;

index := number | number ':' number ;

field := letter (letter | digit)* ;

comparison := '<=' | '>=' | '<' | '>' | '==' | '!=' ;

number := digit digits* ('.' digits*)? ;

string := '"' letter* '"' ;

bool   := 'true' | 'false' ;

digit  := '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | '0' ;
letter := 'a' | 'b' | ... | 'z' | 'A' | 'B' | ... | 'Z' ;
```

Визначимо типи даних, що будуть відповідати лексемам граматики виразів:

```
data Comparison
  = LT
  | LE
  | GT
  | GE
  | EQ
  | NEQ
  deriving (Show, Eq)

data ArrayIndex
  = EmptyArray
  | Index Int
  | IndexRange (Int, Int)
  deriving (Show, Eq)

data Query
  = Dot
  | Array ArrayIndex
  | Pipe
  | Field String
  | QueryNumber Double
  | Comma
  | Compare Comparison
  | QueryString String
```

Ці типи даних визначають лексеми граматики предметно-орієнтованої мови виразів фільтрації та пошуку. Для фільтрації даних користувачу необхідно передати вираз як аргумент у командному рядку (детальний опис у розділі 3).

```
dot :: Parser Query
dot = Dot <$ char '.'

pipe :: Parser Query
pipe = Pipe <$ (ws *> char '|' <* ws)
```

```
comma :: Parser Query
comma = Comma <$ (ws *> char ',' <* ws)
```

Дані СА зчитують односимвольні лексеми запитів — крапку, вертикальну риску та кому відповідно. Слід зауважити, що кома може знаходитись між будь-якою кількістю пробілів.

```
index :: Parser ArrayIndex
index = Index <$> number


indexRange :: Parser ArrayIndex
indexRange = do
  left <- number
  char ':'
  right <- number
  return $ IndexRange (left, right)

array :: Parser Query
```

Комбінатори *index* та *indexRange* зчитують індекси для масиву — число, або два числа, розділених двокрапкою, відповідно. Останній аналізатор зчитує індекси, що знаходяться всередині квадратних дужок, або порожні дужки.

```
field :: Parser Query
field = Field <$> do
  c1   <- parseWhen isLetter
  rest <- many $ parseWhen (\c -> isDigit c || isLetter c)
  return (c1:rest)
```

Даний СА зчитує ім’я поля, яке необхідно знайти у об’єкті типу JSON. Перший символ не може бути числом, тому логіку для зчитування першого символу та інших символів необхідно розділити, а результат об’єднати.

```
comparison :: Parser Query
comparison = Compare <$> (ws *> (le <|> ge <|> lt <|> gt <|> eq <|> neq) <* ws)
  where lt  = const QueryParser.LT  <$> string "<"
        le  = const QueryParser.LE  <$> string "<="
        gt  = const QueryParser.GT  <$> string ">"
        ge  = const QueryParser.GE  <$> string ">="
```

```
    eq  = const QueryParser.EQ  <$> string "=="
    neq = const QueryParser.NEQ <$> string "!="
```

Даний СА зчитує усі можливі оператори порівняння, строго у вказаному порядку.

```
query :: Parser [Query]
```

Основний СА, що зчитує послідовність (список) лексем типу *Query*, використовуючи аналізатори, описані вище.

### 2.4 Модуль виконання дій над JSON-файлом

Даний модуль містить у собі функції для виконання запитів користувача — фільтрації або пошуку.

#### 2.4.1 Фільтрація даних

Фільтрація даних відбувається за наступним принципом: запит користувача розділюється по символам конвеєра (прямої риски) на декілька окремих запитів, після чого кожен запит виконується на результатах попередніх запитів зі списку.

Виконання запиту фільтрації над одним об’єктом типу *Json* повертає список таких об’єктів, що задовольнили запит. Після чого над кожним об’єктом у цьому списку виконується фільтрація з використанням уже наступного запиту у конвеєрі. Усі результати об’єднуються в один список та передаються далі до закінчення конвеєра.

Також запити можна розділити комами. Це дозволяє виконувати декілька запитів над об’єктом типу *Json*, або над результатами виконання попереднього запиту у конвеєрі. Це може бути корисно для запобігання виконання однакових дій над даними — виконання подібних запитів фільтрації, у яких відрізняється лише певна невелика частина.

Основна функція для фільтрації:

```
filterJson :: [Query] -> Json -> Either String [Json]
```

Вона отримує список значень типу *Query* та сам JSON, над яким необхідно ці запити виконати. Значення результату функції типу *Either* дозволяє повертати повідомлення з помилками, якщо такі були, які потім будуть оброблюватися у головному модулі.

```
filterSeparated :: [Query] -> Json -> Either String [Json]
```

Допоміжна функція, що розділяє запити по комам та виконує семантичну перевірку над кожним з них. Після чого виконує кожен запит над об'єктом типу *Json* та об'єднує результати в один список. Значення типу *Either* дозволяє уникнути пряме повернення та обробку помилок — завдяки системі типів мови Haskell усі помилки автоматично зупинять подальше виконання функцій та повернуться на рівень виклику вище автоматично.

### 2.4.2 Пошук даних

Запити пошуку даних синтаксично являють собою частину запитів фільтрації. Пошук відбувається лише по JSON-об'єктах. Можливі варіанти пошуку: пошук за присутністю поля з певною назвою у об'єкті, або пошук об'єктів, у яких поля відповідають певному значенню. Тому підтримується два можливих види запитів: "*Поле*", або "*Поле *порівняння* значення*".

Слід зауважити, що пошук даних більше корисний для ручного перегляду кінцевим користувачем, ніж для виконання операцій над результатом цього пошуку. Якщо користувачеві необхідно відфільтрувати дані після пошуку об'єктів, то для цього слід скористатися функцією фільтрування .

```
searchJson :: [Query] -> Int -> Json -> Either String [Json]
```

Основна функція, що виконує пошук у об'єкті типу *Json* відповідно до запиту пошуку. Функція також приймає порядковий номер предка, який необхідно повернути у разі успішного виконання. Так, якщо цей номер дорівнює 0, то повернеться сам об'єкт, якщо 1 — повернеться прямий батько об'єкта, 2 — батько батька і т.д.

Завдяки механізму співставлення з шаблоном у мові Haskell при створенні функцій, є можливість не виконувати напряму семантичну перевірку запитів. Для цього необхідно визначити функції лише для підтримуваних комбінацій значень типу *Query*, а для всіх інших випадків повертати помилку.

## 2.5 Модуль вводу/виводу

Функції цього модуля призначені для зчитування аргументів командного рядку та роботи з ними.

Для розбору аргументів командного рядку використовується стандартна бібліотека системи **GNU getopt**, що була написана для використання у мові *C*. У стандартній бібліотеці мови *Haskell* **System.Console.GetOpt** знаходиться портований вихідний код цієї бібліотеки.

Основний тип даних, що оголошений у модулі:

```
data Flag
  = Filter String       -- -f
  | Duplicates          -- -d
  | Search String       -- -s
  | Parent Int          -- -p
  | One                 -- -o
  | Minimize            -- -m
  | Indentation Int     -- -i
  | Help                -- --help
  deriving (Show, Eq, Ord, Typeable, Data)
```

Кожен конструктор данного типу відповідає за одну опцію командного рядку. Наприклад, конструктор Duplicates відповідає за опцію **-d**, яка видаляє усі дублікати з результату роботи програми. Для прикладу використання див. розділ 3.5.

```
optionIsPresent :: Flag -> [Flag] -> Bool
```

Функція, що перевіряє, чи міститься деяке конкретне значення типу *Flag* у списку таких значень. Зазвичай, цей список являє собою список усіх аргументів, що були передані користувачем програмі. Окрема функція необхідна через те, що

порівняти між собою два значення типу *Filter String*, наприклад, не так просто. Адже значення цього типу містять у собі рядок, а при пошуку аргументів порівняння рядків між собою не є доцільним. Для цього використовуються вбудовані можливості мови Haskell для порівняння конструкторів даних між собою, таких як *Filter*, не враховуючи самих даних, що у них знаходяться.

```
parseArgs :: [String] -> IO ([Flag], String)
```

Основна функція модуля, що приймає список рядків (аргументи, що були передані користувачем програмі) та повертає пару - список значень типу *Flag*, що вдалося синтаксично проаналізувати з цих рядків, та файл, що був переданий до програми окремо від опцій.

### 2.6 Модуль генерації JSON-даних

Даний модуль містить у собі дві основні функції для виводу даних формату JSON — у форматованому та не форматованому вигляді.

```
type Indent = Int
```

Допоміжний тип для збільшення легкості читання коду. Являється синонімом цілочисленного типу *Int*.

```
prettify :: Indent -> Json -> String
prettify = prettyPrint 0
```

Дана функція отримує кількість пробілів у відступах та значення типу *Json*, яке потрібно перетворити у текст. Викликає допоміжну функцію *prettyPrint*, що відслідковує рівні вкладеності об’єктів при перетворенні.

```
prettyPrint :: Indent -> Indent -> Json -> String
```

Дана функція отримує два значення типу *Indent*: перше значення — це відступ у конкретний момент, друге — базове значення відступу, яке вказав користувач. Початкове значення першого відступу — 0, яке вказується у тілі функції *prettify*. Функція *prettyPrint* перетворює будь-яке значення типу *Json* у рядок, який потім виводиться у термінал. При перетворенні масивів або JSON-об’єктів, функція генерує відповідні відкриваючі та закриваючі дужки з певним

відступом (значенням першого аргументу), та рекурсивно викликається на усіх елементах масива або об’єкту, передавши першим аргументом збільшене значення відступу. Таким чином, усі елементи масивів та об’єктів будуть згенеровані з більшим відступом, що покращує зручність читання даних.

```
generate :: Json -> String
```

Дана функція отримує значення типу *Json* та повертає рядок з даними у форматі JSON, що відповідають переданому значенню. Основна відмінність цієї функції від функції *prettify* у тому, що результат *generate* не містить відступів та символів переходу на новий рядок. Це може бути корисно, коли результат виконання програми необхідно відправити на веб-сервер, адже розмір файлу значно скорочується, не втрачаючи інформаційної користі — пробіли та відступи не потрібні програмам, що працюють з JSON-файлами.

### 2.7 Основний модуль

Даний модуль є основним, тому що він об’єднує між собою роботу усіх інших модулів розробленої системи.

```
main :: IO ()
```

Основна точка входу у програму. Спочатку аналізує усі аргументи, що були передані у програму з командного рядку, після чого передає їх у функцію *run*, разом з файлом формату JSON. Отримавши результат виконання функції *run*, прибирає дублікати з нього, якщо був переданий відповідний аргумент командного рядку, та виводить результат у термінал. За замовчуванням результат виводиться у відформатованому вигляді — з відступами та символами переходу на новий рядок. Але у користувача є можливість отримати невідформатований текст, передавши відповідний аргумент командного рядку.

```
run :: [Flag] -> String -> IO [Json]
```

Функція, що виконує основні дії над JSON-файлом — фільтрація або пошук. Отримує список значень типу *Flag* (аргументи командного рядку) та шлях до файлу формату JSON, що був переданий користувачем. Якщо користувач не передавав інших аргументів крім самого файлу, то функція повертає

проаналізований файл без будь-яких змін. В іншому випадку аналізується запит користувача, якщо він був переданий, та результат аналізу передається у функції фільтрації або пошуку відповідно до переданих аргументів.

```
parseFile :: String -> Parser Json -> IO Json
```

Функція, що аналізує файл отриманим СА, та повертає результат аналізу. Отримує шлях до файлу формату *JSON*, що був переданий користувачем, та об’єкт типу *Parser Json*, який буде використовуватися для синтаксичного аналізу вмісту файла.

Файли зчитуються не повністю у оперативну пам’ять комп’ютера. Натомість програма використовує файлові потоки, що дозволяє зчитувати файл поступово, за необхідністю.

У випадку, коли користувач не вказав файл для аналізу, або замість шляху передав символ дефісу `'-'` - дані зчитуються з потоку вводу. Ця поведінка дозволяє використовувати програму у ланцюжках конвеєрів у системах Unix, та використовувати такі функції системи, як перенаправлення потоків вводу та виводу.

# 3. ТЕСТУВАННЯ СИСТЕМИ

Тестування розробленої системи буде виконуватись поетапно, для перевірки кожної окремої функціональної властивості.

Для виклику програми, назва якої ***hson***, вона була розміщена у спеціально налаштованій директорії системи, що дає змогу виконувати її, не вказуючи повного шляху до неї.

Для початку протестуємо запуск програми з аргументом **-h**. На рис. 10 зображено результат виконання програми з виводом допоміжного повідомлення.

```
xomute@xomute-G3-3779:~/Desktop/Diploma$ hson -h
Usage: hson [OPTIONS] [FILES]
  -f EXPR  --filter=EXPR  Filter json by given expression.
  -d       --dupl         Remove all duplicates from output after filtering or searching.
  -s EXPR  --search=EXPR  Find object with given field and value.
  -p NUM   --parent=NUM   Works along with '-s' flag. Get parent of found object at given level, where level 0 is the object itself.
  -o       --one          Return only first result of filter or search.
  -m       --minimize     Minimize json output.
  -i NUM   --indent=NUM   Number of spaces in indentation.
  -h       --help         Display this help and exit.
```

Рисунок 10 – Вивід допоміжного повідомлення

Програма успішно виводить допоміжне повідомлення для користувача з поясненням використання аргументів даної утиліти.

## 3.1 Синтаксичний аналіз JSON

За замовчуванням, програма зчитує дані з файлу або потоку вводу, та виводить зчитані дані у форматованому вигляді, не виконуючи ніяких дій над самими даними.

Для початку спробуємо зчитати та проаналізувати простий об’єкт формату JSON з двома полями, що збережений у файлі *simpleTest.json* (на рис. 11) у не форматованому вигляді.

```
1 {"name": "Joe","age": 25  }
```

Рисунок 11 – Файл *simpleTest.json*

На рис. 12 зображено результат форматування файлу *simpleTest.json*.

```
xomute@xomute-G3-3779:~/Desktop/Diploma$ hson simpleTest.json
{
    "name": "Joe",
    "age": 25
}
```

Рисунок 12 – Форматування даних

Як бачимо, файл було успішно зчитано та проаналізовано, а його вміст — відформатовано. Якщо ми хочемо зберегти результат у файл — необхідно перенаправити стандартний потік виводу програми. На рис. 13 зображено приклад використання перенаправлення вихідного потоку програми у окремий файл.

```
xomute@xomute-G3-3779:~/Desktop/Diploma$ hson simpleTest.json > formatted.json
xomute@xomute-G3-3779:~/Desktop/Diploma$ cat formatted.json
{
    "name": "Joe",
    "age": 25
}
```

Рисунок 13 – Запис результату виконання у файл

Спробуємо проаналізувати більш складний файл формату JSON, який згенеруємо за допомогою веб-утиліти JSON Generator [9]. Даний файл (рис. 14) імітує можливий результат запиту до деякого веб-сервера, для обробки якого частіше всього використовують утиліти, схожі за принципом дії до розробленої.

```
{
  "_id": "60927dffe20174c57b49d163",
  "index": 0,
  "guid": "e6d98b18-b59d-4f63-8e3c-e732207a261c",
  "isActive": true,
  "balance": "$3,836.13",
  "picture": "http://placehold.it/32x32",
  "age": 34,
  "eyeColor": "brown",
  "name": "Le Irwin",
  "gender": "male",
  "company": "FURNAFIX",
  "email": "leirwin@furnafix.com",
  "phone": "+1 (984) 556-3447",
  "address": "118 Ditmas Avenue, Goldfield, Oregon, 3454",
  "registered": "2015-09-30T07:02:53 -03:00",
  "latitude": 77.759611,
  "longitude": 179.639219,
  "tags": [
    "cillum",
    "ea",
    "irure",
    "cillum",
    "sunt",
    "qui",
    "dolor"
  ],
  "friends": [
    {
      "id": 0,
      "name": "Clare Gillespie"
    },
    {
      "id": 1,
      "name": "Diane Leonard"
    },
    {
      "id": 2,
      "name": "Autumn Navarro"
    }
  ],
  "greeting": "Hello, Le Irwin! You have 1 unread messages."
}
```


Рисунок 14 – Файл *generated.json*

На рис. 15 зображено частковий результат успішного виконання програми з файлом *generated.json*.

```
xomute@xomute-G3-3779:~/Desktop/Diploma$ hson generated.json
{
    "_id": "60927dffe20174c57b49d163",
    "index": 0,
    "guid": "e6d98b18-b59d-4f63-8e3c-e732207a261c",
    "isActive": true,
    "balance": "$3,836.13",
    "picture": "http://placehold.it/32x32",
    "age": 34,
    "eyeColor": "brown",
```

Рисунок 15 – Частина результату виконання програми з файлом *generated.json*

Система успішно виконала синтаксичний аналіз та вивела вміст файлу на екран. Тепер перевіримо коректність повідомлень про помилки. Видалимо з файлу *generated.json* останню фігурну дужку та поглянемо на результат виконання (рис. 16).

```
xomute@xomute-G3-3779:~/Desktop/Diploma$ hson generated.json
Line: 44, column: 1
Unexpected end of input
Expecting '}'
```

Рисунок 16 – Повідомлення про помилку EOF

Повідомлення про помилку на рис. 16 вказує на те, що СА досягнув кінця вхідних даних, хоча відповідно до граматики йому необхідно було зчитати закриваючу фігурну дужку.

Тепер видалимо з файлу *generated.json* кому між третім та четвертим полем об’єкта (рис. 17).

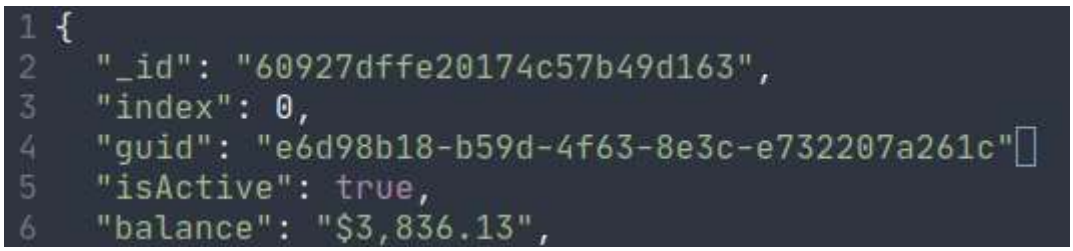


Рисунок 17 – Файл *generated.json* з синтаксичною помилкою

Повідомлення про помилку на рис. 18 вказує на те, що CA зчитав символ подвійних лапок у п’ятому рядку, хоча очікував закриваючу фігурну дужку, що вказує на кінець об’єкта JSON, так як в кінці 4 рядку відсутня кома.

```
xomute@xomute-G3-3779:~/Desktop/Diploma$ hson generated.json
Line: 5, column: 3
Unexpected '"'
Expecting '}'
```

Рисунок 18 – Повідомлення про помилку при зчитуванні подвійних лапок

3.2 Фільтрація даних

Тепер розглянемо фільтрацію даних за допомогою розробленої програми. Спочатку виконаємо найпростіший запит фільтрації на файлі simpleTest.json. На рис. 19 зображено результат виконання такого запиту.

```
xomute@xomute-G3-3779:~/Desktop/Diploma$ hson simpleTest.json -f "."
{
    "name": "Joe",
    "age": 25
}
```

Рисунок 19 – Фільтрація

Фільтр “.” повертає сам об’єкт без будь-яких змін, як видно з результатів його виконання. Тепер виконаємо фільтри для отримання значень полів об’єкта JSON. На рис. 20 зображено результати виконання таких запитів.

```
xomute@xomute-G3-3779:~/Desktop/Diploma$ hson simpleTest.json -f ".name"
"Joe"
xomute@xomute-G3-3779:~/Desktop/Diploma$ hson simpleTest.json -f ".age"
25
```

Рисунок 20 – Фільтрація

Фільтр виду “.*ключ*” шукає у заданому об’єкті JSON запис з таким ключем, та повертає відповідне значення. Фільтри такого типу можна виконати декілька разів, записуючи їх один за одним. Це дозволяє легко дістати поля вкладених об’єктів.

На рис. 21 зображено результат отримання значень з вкладених об’єктів у файлі *nestedTest.json* за допомогою декількох фільтрів доступу до значень полів об’єктів, записаних підряд.

```
xomute@xomute-G3-3779:~/Desktop/Diploma$ cat nestedTest.json
{
    "foo": {
        "bar": {
            "buz": {
                "qux": [
                    {
                        "rnd": 2
                    },
                    {
                        "ok": true
                    }
                ]
            }
        }
    }
}
xomute@xomute-G3-3779:~/Desktop/Diploma$ hson nestedTest.json -f ".foo.bar.buz.qux"
[
    {
        "rnd": 2
    },
    {
        "ok": true
    }
]
```

Рисунок 21 – Запит з вкладеними ключами

Тепер проведемо тестування роботи з масивами. Для отримання кожного елемента масиву необхідно виконати запит “.[]” - після крапки написати порожні квадратні дужки. Спробуємо отримати значення поля **“friends”** у файлі *generated.json*. На рис 22 зображено результат виконання такого запиту.

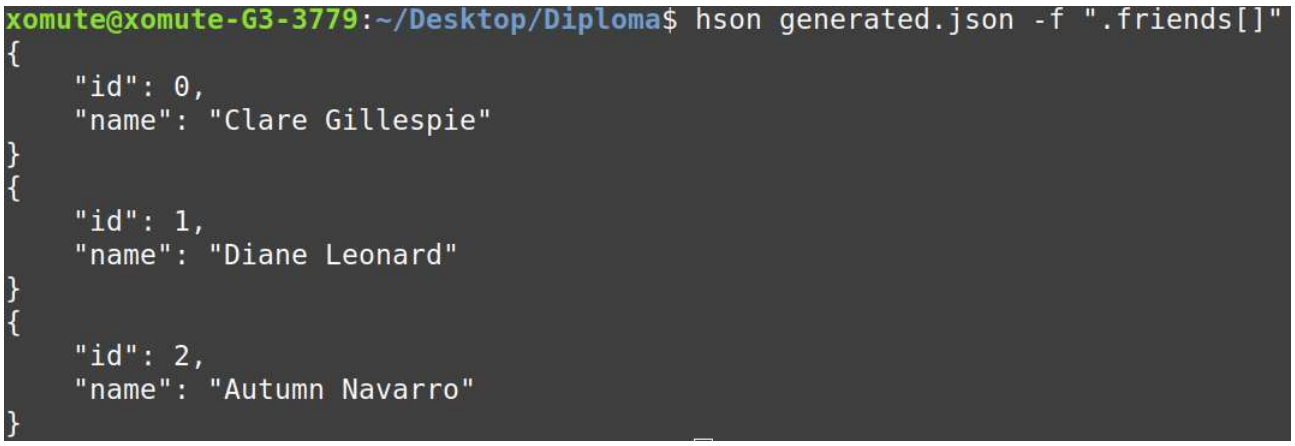


Рисунок 22 – Запит з отриманням кожного елемента масиву

Для доступу до конкретного елемента масиву необхідно вказати індекс у квадратних дужках. Принцип дії схожий до доступу за індексом до елементів масивів у більшості сучасних МП. Індексація елементів масивів починається з нуля.

Спробуємо дістати значення поля **“id”** другого елемента масиву **“friends”**. Результат зображено на рис. 23.

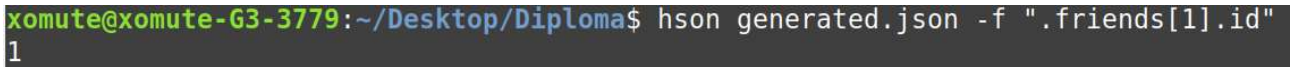


Рисунок 23 – Доступ до елементів масива за індексом

При спробі доступу до неіснуючого елемента масиву — помилковий індекс — виведеться помилка (рис. 24).


Рисунок 24 – Помилка індексації

При доступі до елементів масива, можна вказати два індекси, розділених двокрапкою. Результат такого запиту (рис. 25) поверне декілька елементів масиву між даними індексами, не включаючи елемент на позиції правого індексу.


Рисунок 25 – Запит для отримання частини масива

Результатом виконання запиту фільтрації, як видно на рис. 25, є два об’єкти. Корисною буде можливість виконати певний запит на кожному елементі отриманого масиву. Для цього у DSL запитів фільтрації доданий символ конвеєра (вертикальна риска), який працює за схожим принципом до конвеєрів мови *bash*. Над кожним результатом фільтрації зліва від символу вертикальної риски виконується запит, що записаний праворуч від нього.

Спробуємо за допомогою конвеєра дістати значення поля **“name”** у кожного елемента масиву **“friends”**. На рис. 26 зображено результат виконання такого запиту.

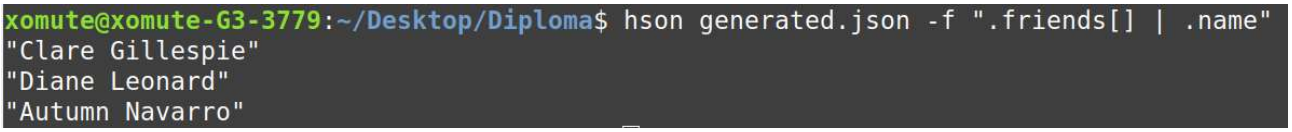


Рисунок 26 – Використання конвеєра

Конвеєри працюють з будь-якими запитами, не лише з запитами доступу до елементів масивів. Наприклад, їх можна використовувати для доступу до вкладених об’єктів, хоча це може бути зайвим, враховуючи можливість прямого доступу до таких об’єктів. Приклад такого запиту зображено на рис. 27

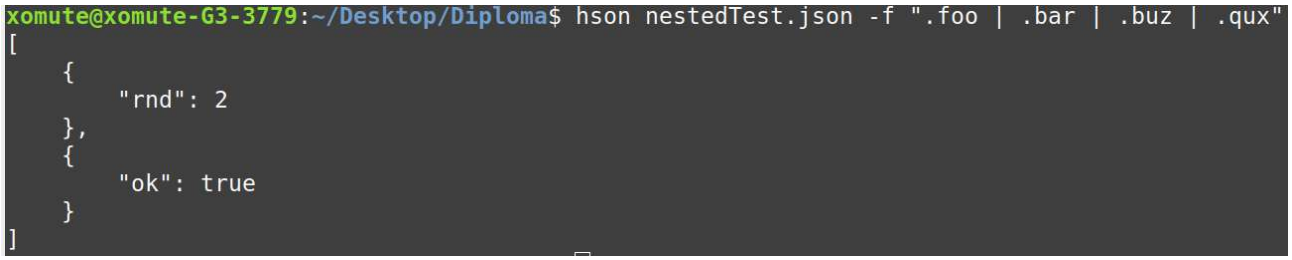


Рисунок 27 – Доступ до вкладених об’єктів за допомогою конвеєрів

У DSL запитів також присутній символ коми — він дозволяє виконати незалежні запити на одному і тому ж результаті фільтрації.

Наприклад, ми хочемо отримати значення декількох полів одного об’єкту JSON у файлі *generated.json* - **“_id”, “name”** та **“email”**. Приклад такого запиту зображено на рис. 28.

```
xomute@xomute-G3-3779:~/Desktop/Diploma$ hson generated.json -f "._id, .name, .email"
"60927dffe20174c57b49d163"
"Le Irwin"
"leirwin@furnafix.com"
```

Рисунок 28 – Приклад використання коми

Також, у одному запиті можна використовувати коми разом з конвеєрами (рис. 29). У конвеєрів більший пріоритет при виконанні, тому спершу оброблюються символи прямої риски, а потім коми. Спробуємо отримати значення полів **“id”** та **“name”** останніх двох елементів масиву **“friends”**.

```
xomute@xomute-G3-3779:~/Desktop/Diploma$ hson generated.json -f ".friends[1:3] | .id, .name"
1
"Diane Leonard"
2
"Autumn Navarro"
```

Рисунок 29 – Запит з конвеєром та комою

Як бачимо на рис. 29, кожен результат запиту “.friends[1:3]” передається у запит праворуч від символу конвеєра, який повертає значення конкретних полів об’єкта.

Для користувачів може бути корисною фільтрація елементів за певними значеннями, тому у DSL запитів присутні оператори порівнянь. Спробуємо знайти усі елементи масиву **“friends”**, значення поля **“id”** яких більше або дорівнює одиниці. Для ускладнення запиту (рис. 30), спробуємо дістати поля **“name”** тих об’єктів, що підпадають під такий критерій.

```
xomute@xomute-G3-3779:~/Desktop/Diploma$ hson generated.json -f ".friends[] | .id >= 1 | .name"
"Diane Leonard"
"Autumn Navarro"
```

Рисунок 30 – Фільтрація за критеріями

Порівнявши результат виконання програми на рис. 30 з оригінальним вмістом файлу *generated.json*, бачимо, що запит виконаний коректно.

### 3.3 Пошук даних

Тепер розглянемо пошук даних за допомогою розробленої програми. Пошук виконується за наступними принципами:

- Пошук об’єктів JSON, у яких присутні записи з певними ключами;
- Пошук об’єктів JSON, записи яких підпадають під критерії пошуку.

Розглянемо пошук по ключам. Спробуємо знайти об’єкти, у яких присутнє поле **“id”** у файлі *generated.json*. Результат виконання зображено на рис. 31.

```
xomute@xomute-G3-3779:~/Desktop/Diploma$ hson generated.json -s "id"
{
    "id": 0,
    "name": "Clare Gillespie"
}
{
    "id": 1,
    "name": "Diane Leonard"
}
{
    "id": 2,
    "name": "Autumn Navarro"
}
```

Рисунок 31 – Пошук за наявністю поля у об’єкті

Як бачимо, результат виконання програми вивів на екран усі об’єкти, у яких є поле **“id”**. Тепер спробуємо знайти усі об’єкти, значення поля **“id”** яких менше двох. Результат пошуку зображено на рис. 32.

Як бачимо, у результаті на екран вивелись два перших об’єкти.

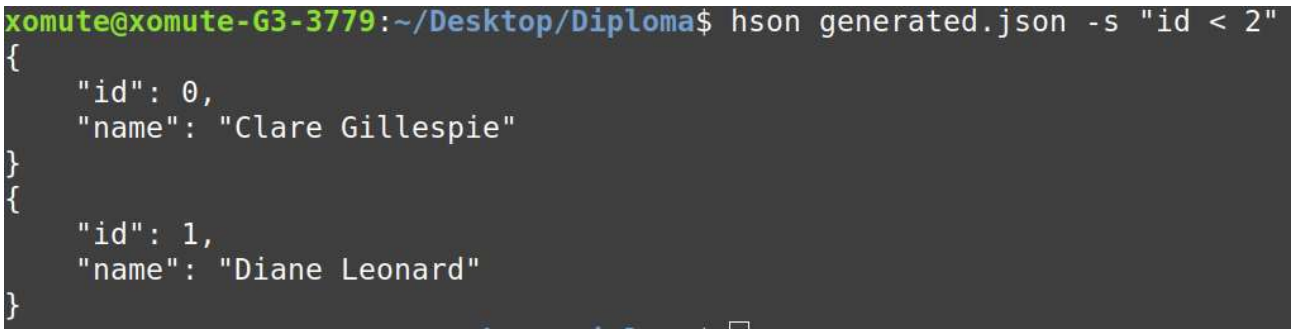


Рисунок 32 – Пошук об’єктів за критерієм

Оскільки за структурою системи, функції пошуку та фільтрації працюють окремо одна від одної, користувач не зможе обробити результати пошуку. Якщо він хоче, наприклад, отримати усі значення полів **“name”** отриманих об’єктів — йому необхідно буде виконати фільтрацію, що поверне ці об’єкти.

Якщо користувач хоче дізнатись, у якій структурі знаходяться ці об’єкти, він може скористатись аргументом командного рядку ***-p***, що у результаті пошуку поверне не сам об’єкт, а структури вище нього на вказану кількість рівнів. Наприклад, щоб дізнатись, якій структурі належать повернуті об’єкти (і чи належать взагалі), необхідно передати аргумент ***-p 1***, де одиниця вказує на структуру рівнем вище. Приклад використання зображено на рис. 33.

У результаті виконання такого запиту було повернуто два однакових масиви, адже об’єкти, що підпадають під критерій пошуку **“id < 2”** знаходяться у одній і тій ж структурі даних. У таких випадках, користувач може передати аргумент командного рядку ***-o***, що поверне перший успішний результат пошуку. Приклад використання зображено на рис. 34

```
xomute@xomute-G3-3779:~/Desktop/Diploma$ hson generated.json -s "id < 2" -p 1
[
    {
        "id": 0,
        "name": "Clare Gillespie"
    },
    {
        "id": 1,
        "name": "Diane Leonard"
    },
    {
        "id": 2,
        "name": "Autumn Navarro"
    }
]
[
    {
        "id": 0,
        "name": "Clare Gillespie"
    },
    {
        "id": 1,
        "name": "Diane Leonard"
    },
    {
        "id": 2,
        "name": "Autumn Navarro"
    }
]
```

Рисунок 33 – Результат пошуку з отриманням батьківських об’єктів

```
xomute@xomute-G3-3779:~/Desktop/Diploma$ hson generated.json -s "id < 2" -p 1 -o
[
    {
        "id": 0,
        "name": "Clare Gillespie"
    },
    {
        "id": 1,
        "name": "Diane Leonard"
    },
    {
        "id": 2,
        "name": "Autumn Navarro"
    }
]
```

Рисунок 34 – Повернення тільки першого результату пошуку

Тепер результатом пошуку є структура, що містить у собі перший об’єкт, який підпадає під критерій пошуку — у нашому випадку, структура зі значенням поля **“id”** 0. Якщо користувач хоче дізнатись, у якій структурі знаходиться цей масив, він може збільшити аргумент ***-p*** на одиницю. Приклад такого запиту зображено на рис. 35.

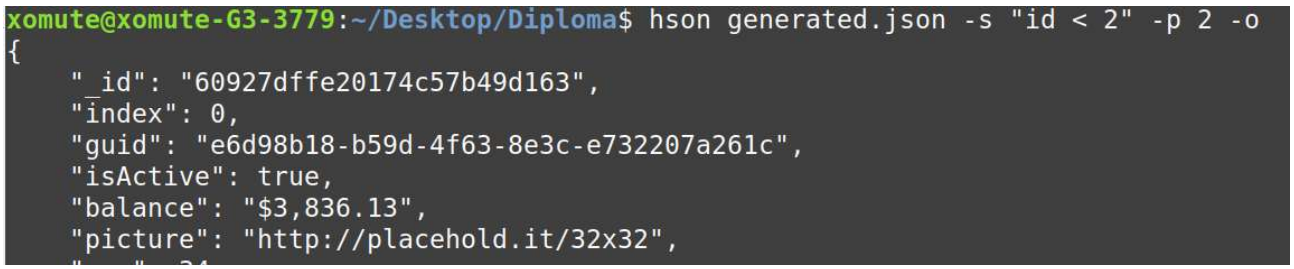


Рисунок 35 – Частковий результат пошуку предка об’єкта

Так як масив, що є результатом виконання програми на рис. 34, є значенням поля **“friends”** головного об’єкту у файлі *generated.json* — він повністю виводиться у термінал, щоб користувач міг переглянути структуру цього об’єкту, та зрозуміти, де саме знаходиться цей масив.

Якщо б цей об’єкт був вкладеним значенням у інший об’єкт або масив — користувач міг би дізнатись батька цього об’єкту, збільшивши аргумент ***-p*** ще на одиницю. Якщо структура є найвищою у ієрархії файлу формату JSON, збільшення цього аргументу не змінить результат виконання такого запиту.

Для тестування більших рівнів вкладеності виконаємо запити пошуку на файлі *nestedTest.json*.

Виконання запиту пошуку на рис. 36 повернуло об’єкт на найнижчому рівні у даному файлі. Дізнаємося прямих та непрямих предків цього об’єкту. Результати зображено на рис. 37 та рис. 38.

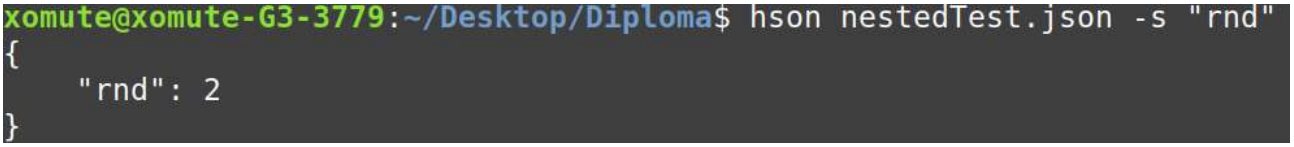


Рисунок 36 – Результат пошуку вкладеного об’єкта

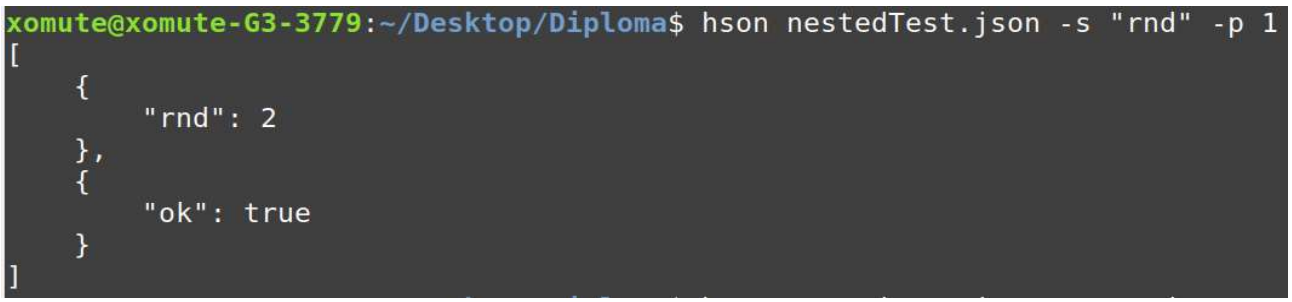


Рисунок 37 – Прямий предок

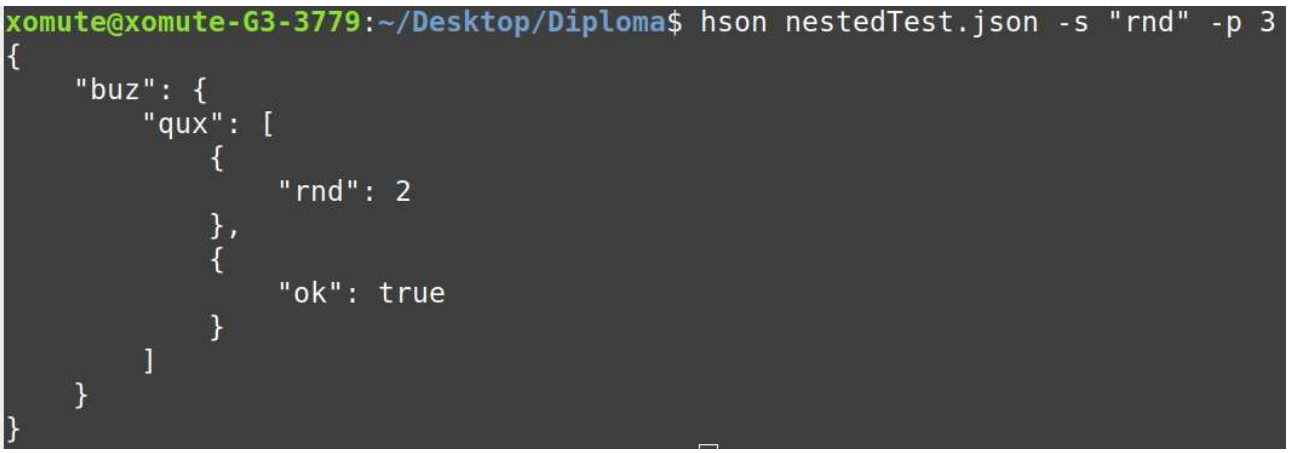


Рисунок 38 – Непрямий предок

Як бачимо з результатів, пошук здатен знайти об’єкт на будь-якому рівні вкладеності, завдяки рекурсивному алгоритму.

3.4 Форматування виводу

Для покращення швидкодії передачі даних в мережі Інтернет доцільно використовувати відформатовані дані без символів, які ігнорують СА, такі як пробіли або переходи на новий рядок. Для цього у розробленій програмі додана можливість виводу даних JSON без зайвих символів, що досягається передачею аргумента командного рядку ***-m***.

На рис. 39 зображено результат виконання програми на файлі *nestedTest.json* з використанням прапорця ***-m***.


Рисунок 39 – результат виконання програми з мінімізацією розміру вихідних даних

В даному прикладі, розмір мінімізованого файлу JSON майже втричі менший за відформатований варіант. При мінімізації даних більшого розміру, наприклад, у файлі *generated.json*, різниця у розмірі стає ще більшою, що у перспективі дозволяє набагато швидше передавати такі дані до веб-серверів.

На рис. 40 та рис. 41 зображено приклади порівняння розмірів мінімізованих файлів з їх відформатованими аналогами.

```
xomute@xomute-G3-3779:~/Desktop/Diploma$ hson nestedTest.json -m > mNestedTest.json
xomute@xomute-G3-3779:~/Desktop/Diploma$ ls -l nestedTest.json mNestedTest.json
-rw-rw-r-- 1 xomute xomute  56 May  6 12:52 mNestedTest.json
-rw-rw-r-- 1 xomute xomute 131 Apr 26 22:43 nestedTest.json
```

Рисунок 40 – Приклад 1 порівняння розміру файлів

```
xomute@xomute-G3-3779:~/Desktop/Diploma$ hson generated.json -m > mGenerated.json
xomute@xomute-G3-3779:~/Desktop/Diploma$ ls -l generated.json mGenerated.json
-rw-rw-r-- 1 xomute xomute 880 May  5 14:57 generated.json
-rw-rw-r-- 1 xomute xomute 682 May  6 13:06 mGenerated.json
```

Рисунок 41 – Приклад 2 порівняння розміру файлів

Також у користувача є можливість задати розмір відступів при форматуванні даних. Для цього необхідно передати бажане число пробілів у відступах за допомогою прапорця ***-i***.

На рис. 42 показано два різних варіанти форматування файлу *nestedTest.json*.


Рисунок 42 – результати форматування з різними відступами

### 3.5 Видалення дублікатів

У випадках, коли користувач хоче позбутись усіх однакових об’єктів у результаті виконання запитів, він може передати програмі аргумент командного рядку ***-d***, що видаляє усі дублікати перед виводом результату у термінал. На рис. 43 зображено результат виконання запиту пошуку предків усіх об’єктів з полем **“id”** у файлі *generated.json* з використанням прапорця ***-d***.

```
xomute@xomute-G3-3779:~/Desktop/Diploma$ hson generated.json -s "id" -p 1 -d
[
    {
        "id": 0,
        "name": "Clare Gillespie"
    },
    {
        "id": 1,
        "name": "Diane Leonard"
    },
    {
        "id": 2,
        "name": "Autumn Navarro"
    }
]
```

Рисунок 43 – результат роботи програми з прапорцем ***-d***

Як видно на рис. 43, замість трьох однакових масивів програма вивела лише один, так як два інших були видалені як дублікати.

### 3.6 Використання системи у складних скриптах командного рядку

Спробуємо використати розроблений програмний продукт у скриптах мовою *bash*. Скрипт на рис. 44 отримує дані з веб-серверу у вигляді JSON-файлу за допомогою UNIX-утиліти ***curl***, що в даному випадку відправляє запит GET за вказаним шляхом. Після чого, результат виконання ***curl*** передається за допомогою конвеєра у стандартний потік вводу розробленого програмного засобу. Результатом виконання скрипта *test.sh* є список імен астронавтів, що наразі знаходяться у космосі. Дані отримуються з джерел NASA за допомогою відкритого програмного забезпечення [8].

Спробуємо ускладнити скрипт та вивести для кожного астронавта відповідне повідомлення про те, що він знаходиться у космосі. На рис. 45 зображено результат виконання такого скрипта.

```
xomute@xomute-G3-3779:~/Desktop/Diploma$ cat test.sh
#!/bin/bash
curl http://api.open-notify.org/astros.json -s | hson -f ".people[] | .name"
xomute@xomute-G3-3779:~/Desktop/Diploma$ ./test.sh
"Mark Vande Hei"
"Oleg Novitskiy"
"Pyotr Dubrov"
"Thomas Pesquet"
"Megan McArthur"
"Shane Kimbrough"
"Akihiko Hoshide"
```

Рисунок 44 – Результат виконання скрипта *test.sh*

```
xomute@xomute-G3-3779:~/Desktop/Diploma$ cat test.sh
#!/bin/bash
curl http://api.open-notify.org/astros.json -s | \
        hson -f ".people[] | .name"             | \
        sed -E 's#"(.*)"#Astronaut \1 is in space now.#'
xomute@xomute-G3-3779:~/Desktop/Diploma$ ./test.sh
Astronaut Mark Vande Hei is in space now.
Astronaut Oleg Novitskiy is in space now.
Astronaut Pyotr Dubrov is in space now.
Astronaut Thomas Pesquet is in space now.
Astronaut Megan McArthur is in space now.
Astronaut Shane Kimbrough is in space now.
Astronaut Akihiko Hoshide is in space now.
```

Рисунок 45 – Результат виконання ускладненого скрипта *test.sh*

З результатів тестування можна визначити, що розроблена система пристосована до використання у проєктах, які працюють з JSON-файлами.

# 4. КЕРІВНИЦТВО ПРОГРАМІСТА

## 4.1 Опис аргументів командного рядку

Перелік аргументів можна отримати, передавши програмі аргумент командного рядку ***-h*** або --***help.***

Перелік аргументів та їх пояснення:

- Аргумент ***-f*** відповідає за фільтрацію даних. Після аргумента необхідно вказати вираз запиту фільтрації, за яким буде оброблено передані дані;
- Аргумент ***-d*** визначає видалення усіх дублікатів об’єктів з результату виконання роботи програми;
- Аргумент ***-s*** відповідає за пошук об’єктів у файлі JSON. Після аргумента необхідно вказати вираз запиту пошуку, за яким буде знайдено відповідні об’єкти у файлі;
- Аргумент ***-p*** працює разом з аргументом ***-s***. Числове значення, що передається після цього аргументу, зазначає рівень вкладеності, з якого треба вийти після успішного знайдення об’єкта. Рівень 0 вказує на сам об’єкт, рівень 1 — структура (об’єкт або масив), у якому знаходиться знайдений об’єкт і т.д.;
- Аргумент ***-o*** повідомляє програму про необхідність повернення лише одного результату фільтрації або пошуку;
- Аргумент ***-m*** визначає мінімізацію результату виконання програми – результат виводиться без пробілів та переходів на новий рядок;
- Аргумент ***-i*** визначає вид форматованого виводу. Числове значення, що надається після цього аргументу, вказує кількість пробілів у відступах при форматуванні вихідних даних програми;

- Аргумент ***-h*** визначає виведення у термінал допоміжного повідомлення, що описує аргументи програми.

# ВИСНОВКИ

Система підтримки роботи з JSON-файлами для програмістів мовою Haskell, що є результатом дипломного проєкту, була створена для зручності роботи з даними у форматі JSON у вихідному коді МП Haskell та командному рядку UNIX-систем в цілому.

Після аналізу існуючих бібліотек та застосунків для роботи з JSON-файлами, який був проведений в першому розділі дипломного проєкту, не було знайдено подібної системи саме для МП Haskell, тому розробка такої системи була доцільною.

При реалізації системи важливим було створити незалежні один від одного модулі, що дозволить використовувати окремі частини системи, що необхідні програмісту для вирішення його задач, та за необхідністю доповнювати їхню функціональність. Також, використання цієї системи у проєктах допоможе запобігти великій кількості зовнішніх залежностей, що часто трапляється у сучасних розробках.

Загалом система може виконувати такі функції:

- Створення СА за методом монадичних комбінаторів;
- Синтаксичний аналіз JSON у синтаксичне дерево для подальшої його обробки;
- Функції для фільтрування даних;
- Використання у інтерфейсі командного рядку.

В майбутньому розроблена система може бути розширена, наприклад, додаванням нових операторів та можливостей у DSL запитів фільтрації, покращенням швидкодії монадичних комбінаторів, або створенням нових СА для різних форматів даних.

# СПИСОК ВИКОРИСТАНИХ ЛІТЕРАТУРНИХ ДЖЕРЕЛ


1. Parser Combinator : URL : https://en.wikipedia.org/wiki/Parser_combinator (дата звернення: 03.05.2021).
2. Richard A. Frost, Rahmatullah Hafiz, Paul Callaghan. Parser Combinators for Ambiguous Left-Recursive Grammars — 2008. — 15 с.
3. Graham Hutton, Erik Meijer. Monadic Parser Combinators — 1996. — 38 с.
4. Graham Hutton, Erik Meijer. Monadic parsing in Haskell — 1998. — 8 с.
5. Regular expressions vs Parser Combinators in Haskell : URL : https://pl-rants.net/posts/regexes-and-combinators (дата звернення: 04.05.2021).
6. Introducing JSON : URL : https://www.json.org/json-en.html (дата звернення: 14.03.2021).
7. Standard ECMA-404. The JSON Data Interchange Syntax — 2017. — 16 с.
8. Open Notify : URL : http://open-notify.org (дата звернення: 04.05.2021).
9. JSON Generator : URL : http://www.json-generator.com (дата звернення: 04.05.2021).