Assignment 2

Haskell project

2.1 Submission instructions

1. Unzip the `Haskell-Project.zip` folder. You should find 1 folder and 3 files:

   - `src` folder - your workspace
   - `snippets.ben` - sample snippet database file
   - `.gitignore` - if you want to use version control

2. Edit the `src/Main.hs` and `src/Bencode/Parser.hs` files with your solutions.

3. When done, submit the `src/Main.hs` and `src/Bencode/Parser.hs` files on moodle.

Note: Your solutions must be in these two files (i.e. `src/Main.hs` and `src/Bencode/Parser.hs`). Please don't modify other files or create new files to add helper functions.

2.2 Project resources

Table 2.1: Project Resources

Resource	Link
The `Data.Functor` module	`https://hackage.haskell.org/package/base-4.14.0.0/docs/Data-Functor.html`
The `Control.Applicative` module	`https://hackage.haskell.org/package/base-4.14.0.0/docs/Control-Applicative.html`
The `Control.Monad` module	`https://hackage.haskell.org/package/base-4.14.0.0/docs/Control-Monad.html`
The `System.IO` module	`https://hackage.haskell.org/package/base-4.14.0.0/docs/System-IO.html`
Understanding parser combinators	`https://fsharpforfunandprofit.com/posts/understanding-parser-combinators/`
Understanding parser combinators: a deep dive - Scott Wlaschin	`https://www.youtube.com/watch?v=RDalzi7mhdY`

2.3 Project description, goals and non-goals

In this project you will develop basic command line app to manage code snippets. The main use case for such a program is to organize pieces of code that are hard to remember, but need to be used on a semi-regular basis. An example could be checking if a file exists using bash. One would save the snippet using this app and each time it is needed it can be looked up without searching on the internet and testing if the found solution works.

The main goal of the project is to get hands-on experience for developing close to real-world applications in Haskell, using the main features of the language and advantages of functional programming.

There are also non-goals for this project, the main one being very robust error handling, flexibility and the offered user experience - while these are important for real apps, here we focus on understanding the basic concepts that are needed to build a real application.

2.4 Grading

This project is worth 30% of your final lab grade.

You can obtain in total 30 points:

- 60% (18 points) come from public tests (i.e. that you can run to check your implementation)
- 20% (6 points) come from hidden tests (i.e that are not available to you, but will be run when grading your project)
- 20% (6 points) come from coding style

The tests will cover all functional requirements, but you can implement as much as little as you consider adequate. The grade for functional requirements will be calculated from the

number of tests that pass (failing tests most likely mean that a requirement is missing or is not implemented correctly).

2.5 Getting started with the development

Starting code

You will only have to work in two files:

- `Main.hs` - Here you will implement the handlers for the various subcommands of the application.
- `Bencode/Parser.hs` - Here you will implement the bencode parser.

Of the other files, the following are of interest for your implementation:

- `Parsec.hs` - Parser combinator library. You will use it to implement the bencode parser.
- `Entry/DB.hs` - Data definitions and functions the snippet database.
- `Entry/Entry.hs` - Data definitions and functions the snippet database entries.
- `Args.hs` - Data definitions for parsed command line arguments.
- `Test/Tests.hs` - The file containing the tests for your implementation.

It is highly recommended that you spend some time to understand the existing code and the tests before starting to write your solutions. Specifically, pay attention to record fields of data definitions in the `Args.hs` file and the imports in the `Main.hs` file.

Development process

First, you should run `runhaskell.exe .\Test\Tests.hs` to confirm that the tests fail.

Then you should choose a test group, because groups contain related tests for a given aspect of the application and try to implement a solution such that (some of) the tests pass. Once you are satisfied, you can move on to the next test group, repeating this procedure.

If your Haskell extension for VSCode works, you might also find evaluating the examples placed above function helpful.

2.6 Project tasks (functional requirements)

2.6.1 IO and monads (`Main.hs` file) (12p + 4p)

Exercise 2.6.1 — 1.5p + 0.5p

Implement the `handleInit` function, which will handle the init command. It should create an empty database. If a database file already exists, it should be overwritten.

Hints:
To create an empty database, you can use the `empty` function from the `Entry.DB` module.

Exercise 2.6.2 1.5p + 0.5p

Implement the `handleGet` function, which will handle the get command. It should find the first entry with the given id, or display an error if there is no such entry.

Hints:
To find the first entry that matches a predicate, you can use the `findFirst` function from the `Entry.DB` module.

Exercise 2.6.3 3p + 1p

Implement the `handleSearch` function, which will handle the search command. It should display all entries that match *all* search queries, using the `FmtEntry` newtype (i.e. use `show (FmtEntry entry)`). If no entries match the query, you should print `"No entries found"`.

Hints:
To check if an entry is matched by a list of query item, you can use the `matchedByAllQueries` function from the `Entry.Entry` module. To get all entries that satisfy a predicate, you can use the `findAll` function from the `Entry.DB` module.

Exercise 2.6.4 6p + 2p

Implement the `handleAdd` function, which will handle the add command. It should read the contents of a given file and add a new entry to the database. If an entry with the snippet to be added already exists, you should print the error message `"Entry with this content already exists: "`.

Hints:
To add a new entry, you can use the `insertWith` function from the `Entry.DB` module. To load the database, modify it and then save the modified version, you can use the `modify` function from the `Entry.DB` module.

2.6.2 Parser combinators (`Bencode/Parser.hs` file) (6p + 2p)

Exercise 2.6.5 1.5p + 0.5p

Implement the `int` function which parses bencode integers. A bencode integer starts with the character `i`, followed by digits representing the number in decimal, and ends with the character `e`.

Examples:
The number 10, would be represented in bencode as `i10e`.

Exercise 2.6.6 1.5p + 0.5p

Implement the `string` function which parses bencode strings. A bencode string starts with a number that represents the number of characters that will follow for the string, the character `:` and the characters for the string.

Examples:
The string "Haskell" would be represented in bencode as `7:Haskell`.

Hints:
You should consider using (some of) the following functions from the `Parser` module: `with`, `take`.

Exercise 2.6.7 1.5p + 0.5p

Implement the `list` function which parses bencode lists. A bencode list starts with the character `l`, followed by an arbitrary number of bencode values and ends with the character `e`.

Examples:
The Haskell list `[1, 2]` would be represented in bencode as `li1ei2ee`. We can also represent the heterogeneous list (i.e. which contains elements of different types) that would be invalid in Haskell and Elm, but would be valid in Python or JavaScript `["a", 1, "hello"]` as `l1:ai1e5:helloe`.

Exercise 2.6.8 1.5p + 0.5p

Implement the `dict` function which parses for bencode dictionaries. A bencode dictionary (dict) starts with the character `d`, followed by an arbitrary number of (string, bencode value) pairs and ends with the character `e`.

Examples:
The JSON dict `{"a": 1}` would be represented in bencode as `d1:ai1ee` and the JSON dict `{"name": "John", "age": 35}` would be represented in bencode as `d4:name4:John3:agei35ee`.

2.7 Coding style (non-functional requirements)

Exercise 2.7.1 3p

Properly use Haskell language features and library functions. Examples include:

1p Using unique language features:
- Destructuring in function definitions
- Pattern guards
- Function composition using `(.)`
- Using `where` and `let ... in`

1p Using do notation

1p Using features of standard type classes (`Monoid`, `Functor`, `Applicative`, `Monad`)

Exercise 2.7.2 3p

Use a proper coding style:
1.5p Descriptive names for data definitions and functions
1.5p Readable code structure (proper use of indentation)

2.8 Testing your implementation

To run all test, use:

```
powershell session
PS > runhaskell.exe .\Test\Tests.hs
```

To see detailed output for failed tests (i.e. why did a test fail), you can use the `-d` or `--detailed` switches:

```
powershell session
PS > runhaskell.exe .\Test\Tests.hs -d
```

To run tests only for the parser or the command handlers you can use:

```
powershell session
PS > runhaskell.exe .\Test\Tests.hs parser
```

```
powershell session
PS > runhaskell.exe .\Test\Tests.hs handlers
```