Assignment 2

Haskell project

Deadline: Friday, 2024 January 19, 23:45

2.1 Submission instructions

1. Unzip the `Haskell-Project.zip` folder. You should find 3 folders and 1 file:
   - `src` folder - your workspace
   - `scripts` folder - utility scripts
   - `test_files` folder - html files to test the complete implementation
   - `.gitignore` - if you want to use version control
2. Edit the following files with your solutions: `src/Html/Parser.hs` , `src/Args.hs` , `src/App.hs`
3. When done, run the `zip` script from the scripts folder and submit the `src.tar.gz` on moodle.

Note: Your solutions must be only in the files enumerated above (i.e. `src/Html/Parser.hs` , `src/Args.hs` , `src/App.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 complete various parts of an app that can query HTML documents using a subset of CSS selectors. The project was inspired by `https://github.com/ericchiang/pup`.

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:

- 10% (3 points) are awarded by default (i.e. represent the starting grade)
- 50% (15 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 following files:

- `src/Html/Parser.hs`
- `src/Args.hs`
- `src/App.hs`

Of the other files, the `Parser.hs` is of interest for your implementation. It contains a parser combinator library that you will use it to implement the parsers in `src/Html/Parser.hs` .

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 the existing parsers in `Parser.hs` and `src/Html/Parser.hs` .

Development process

First, you should run `runhaskell.exe .\Test\Tests.hs` (in the `src` directory) to confirm that the tests fail.

Then you should choose a test group - 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.

Note that tests that fail because of the `error` or `undefined` function are marked as `TODO` and won't cause the whole testrun to fail.

If your Haskell extension for VSCode works, you might also find evaluating the examples placed above function helpful. You can also load the module in GHCi and run each example manually by copying the part after each `>>>` into the REPL.

2.6 Project tasks (functional requirements)

More detailed requirements and examples are given in the documentation comment of each function.

2.6.1 Main app logic (`src/App.hs` file and `src/Args.hs` file) (3.5p + 2p)

Exercise 2.6.1 — 1.5p + 0.75p

Implement the `parseArgs` function. It takes as parameters the program name and a list of arguments and returns the parsed arguments for the rest of the app.

Exercise 2.6.2 — 1p + 0.5p

Implement the `parseContents` function. Given a list of files, it tries to parse the contents of each file.

Exercise 2.6.3 — 1p + 0.75p

Implement the `searchFiles` function. Given a query and a list of parsed files, it searches for matches in each file.

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

Exercise 2.6.4 — 0.5p

Implement the `text` function, which parses text.

Exercise 2.6.5 — 1p + 0.5p

Implement the `selfClosing` function, which parses a self closing tag.

Exercise 2.6.6 — 1p + 0.5p

Implement the `openTag` function, which parses an opening tag.

Exercise 2.6.7 — 1.5p + 0.5p

Implement the `attributes` function, which parses possibly empty list of attributes.

Exercise 2.6.8 — 1p

Implement `betweenHtmlTags` function, which runs a parser between html tags.

Exercise 2.6.9 — 0.5p

Implement the `htmlNode` function, which parses a html node (a pair of tags, other tags or text between them, or a self closing tag).

Exercise 2.6.10 — 0.5p + 0.5p

Implement the `html` function, which parses a text or html node.

2.6.3 IO and monads (`src/App.hs` file) (5.5p + 2p)

Exercise 2.6.11 — 2.5p + 0.75p

Implement the `readContents` function, which parses a text or html node.

Exercise 2.6.12 — 1.5p + 0.25p

Implement the `maybeReadFile` function, which parses a text or html node.

Exercise 2.6.13 — 1.5p + 1p

Implement the `printMatches` function, which parses a text or html node.

2.7 Coding style (non-functional requirements)

Exercise 2.7.1 — 3p

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

1.5p Using unique language features:
- Destructuring in function definitions
- Pattern guards
- `where` and `let ... in`
- do notation
- list comprehensions

1.5p Using features of standard library
- Function application and composition
- Functions provided by the standard type classes (`Monoid`, `Functor`, `Applicative`, `Monad`)

Note that the goal of the list above is only to give you a general idea of the features that you should consider when writing the code. Your goal is to show that you know when to use and when to not use various features. For example, there are two extremes that you should clearly avoid:

- writing Haskell code that is just like Elm code (not using any Haskell language features)
- using all Haskell features in a way that makes the code harder to understand (obfuscates the intent)

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 indentation
- Functions should have a reasonable size (i.e. should not be too long)

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 certain test group you can use:

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

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

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

Alternatively, you can run the tests using ghci:

```
Shell session
> ghci
Prelude> :l Test\Tests.hs
*Test.Tests> :main
```