Assignment 2

Haskell project

Deadline: Sunday, 2022 January 15, 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
   - `scenes` folder - scene and image configuration files
   - `scripts` folder - utility scripts
   - `.gitignore` - if you want to use version control
2. Edit the following files with your solutions: `src/Scene/Loader.hs` , `src/Args.hs` , `src/Image.hs` , `src/Scene.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/Scene/Loader.hs` , `src/Args.hs` , `src/Image.hs` , `src/Scene.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 a Haskell raytracer. Specifically, you will complete the argument parsing utility functions, configuration loading functions and some parsers used by the configuration loaders.

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 following files:

- `src/Scene/Loader.hs`
- `src/Args.hs`
- `src/Image.hs`
- `src/Scene.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/Scene/Loader.hs` and `src/Image.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/Scene/Loader.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, 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.

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. In order to do this, you should run the `openall` script before you start editing the files (this is needed due to how the Haskell plugin works).

2.6 Project tasks (functional requirements)

2.6.1 Argument parsing (`Args.hs` file) (6p + 2p)

Exercise 2.6.1 — `2.75p + 0.5p`

Implement the `toArgMap` function. It takes a list of arguments of the form `["-key1", "value1", "-key2", "value2", ...]` and converts it to a list of pairs: `[("key1", "value1"), ("key2", "value2")]`.
If one of the keys doesn't start with `-` or a value is missing (i.e. the number of arguments is odd), and error with `InvalidArgs` should be returned.

Exercise 2.6.2 — 2p + 0.5p

Implement the `getArg` and `readArg` functions. Both functions take a key and the argument map returned by `toArgMap` and return the value of the given key, if it exists in the argument map.
`getArg` is used for `String` arguments and `readArg` is used for other arguments that are instances of the `Read` typeclass.

Hints:
The `read` function will throw an exception if the argument can't be parsed (i.e. converted to the desired value). You should use the `readMaybe` function that returns `Maybe a` instead of throwing exceptions.

Exercise 2.6.3 — 1.25p + 1p

Implement the `procArgs` function. It takes a list of arguments of the form `["-key1", "value1", "-key2", "value2", ...]`, parses them using `toArgMap` and populates the `Args` record using `getArg` or `readArg`.

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

Exercise 2.6.4 — 2p + 0.5p

Implement the `vecParser` function, which parses a 3D vector, represented as `<double> , <double> , <double>`.
See the code and tests for examples.

Hints:
Use the `vec` function to create a `Vec3` instance from 3 `Double`s.
Note that there might be arbitrary whitespace between the commas and doubles that form the vector.

Exercise 2.6.5 — 1.5p + 0.5p

Implement the `colorParser` function, which parses an RGB color represented as 3 8bit hexadecimal numbers, represented as `#hhhhhh` where each `h` represents a hexadecimal digit, and each color channel (reg, green, blue) is represented by 2 hex digits (i.e. the format above can also be written as `#rrggbb`, where `r`, `g`, `b` each represent a hex digit).
See the code and tests for examples.

Hints:
To parse a hexadecimal character, digits 0 through 9 (inclusive) and characters 'a' though 'f' (inclusive, both lower and uppercase) are considered valid.
To parse a fixed number of characters, you can use the `pRepeat` function (or hardcode the repetition with `andThen`).
Use the `readHex :: String -> [(Int, String)]` function to parse a hexadecimal number. Note that it returns `[(Int, String)]`, with the meaning that if the number is successfully parsed, a list containing the parsed number and the remaining input is returned, and if the parsing fails an empty list is returned. Since we control the input to the function

by parsing the hexadecimal digits, we know that the input will always be valid, we can assume the function will always succeed and return a list with the parsed number and the empty string.
Finally, use the `color` function to create a `Color` from 3 doubles, but note that all 3 values should be in the $[0, 1]$ range. Since each color channel is represented on a 8 bit hexadecimal integer, each parsed hexadecimal value will be in the range $[0, 255]$, so all you have to do is divide the parsed value by 255.

Exercise 2.6.6 `0.5p + 0.5p`

Implement the `imageParser` function, which parses an image configuration. The configuration for the final image is represented as a dictionary with 4 fields.
Note that the fields will always be provided in the same order: `width`, `height`, `nr_samples`, `max_depth`.
See the code and tests for examples.

Hints:
You might want to use (some of) the following functions: `dict4Parser`.

Exercise 2.6.7 `2p + 0.5p`

Implement the `materialParser` function, which parses a material definition. A material can either be diffuse, metallic or dielectric. To distinguish between these variants, they are placed in a dictionary with one field, representing the type of the material (i.e. `diffuse` for diffuse materials). The value associated with this key will then contain the data of each material (a color for diffuse materials, a color and optional fuzz for metallic materials and index of refraction for dielectric materials).
See the code and tests for examples.

Hints:
You might want to use (some of) the following functions: `oneOf`, `dict1Parser`, `dict2Parser`, `orElse`.

2.6.3 IO and monads (`src/Image.hs` and `src/Scene.hs`) (6p + 2p)

Exercise 2.6.8 `1.5p + 0.5p`

Implement the `loadImageConfig` function. It takes a path to the configuration file and returns the parsed configuration if everything succeeds.
First it should check if the file exists using `doesFileExist` and return an error containing `FileNotFound` if the file is not found. Then it should read the file and parse it using `imageParser`, returning an error containing `ParseFailed` if the imageParser fails (i.e. the format is invalid).

Exercise 2.6.9 `1.5p + 0.5p`

Implement the `getImageConfig` function. It takes an optional path (i.e. `Maybe String`) to the configuration file and returns the parsed configuration if the file path was provided

and was successfully parsed, or a default configuration (`defaultImage`) if the file path was not provided.

Exercise 2.6.10 — 1.5p + 0.5p

Implement the `loadSceneConfig` function. It takes a path to the configuration file and returns the parsed configuration if everything succeeds.
First it should check if the file exists using `doesFileExist` and return an error containing `FileNotFound` if the file is not found. Then it should read the file and parse it using `sceneParser`, returning an error containing `ParseFailed` if the sceneParser fails (i.e. the format is invalid).

Exercise 2.6.11 — 1.5p + 0.5p

Implement the `getSceneConfig` function. It takes an optional path (i.e. `Maybe String`) to the configuration file and returns the parsed configuration if the file path was provided and was successfully parsed, or a default configuration (`defaultSceneConfig`) if the file path was not provided.

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`)

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 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 certain test group can use:

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

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

```
                                                                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
[ 1 of 33] Compiling Bmp.Bmp          ( Bmp\Bmp.hs, interpreted )
...
[33 of 33] Compiling Test.Tests       ( Test\Tests.hs, interpreted )
Ok, 33 modules loaded.
*Test.Tests> :main
```