FIT2102 Programming Paradigms 2024

Assignment 2: Markdown to HTML

Due Date: Friday, 18th October, 11:55 pm
Weighting: 30% of your final mark for the unit
Interview: SWOTVAC + Week 13
Overview: Students will work **independently** to create a parser for a subset of the **Markdown specification** using functional programming techniques. Programs will be implemented in Haskell. **The goal is to demonstrate a good understanding of functional programming techniques as explored throughout the unit**, including written documentation of the design decisions and features.

Submission Instructions

Submit a zipped file named <studentNo>_<name>.zip which extracts to a folder named <studentNo>_<name>

- **It must contain all the code that will be marked** including the **report** and ***all code files***
- You also need to include a report describing your design decisions. The report must be named **<studentNo>_<name>.pdf**.
- No additional Haskell libraries should be used
    - You may use additional libraries only for testing purposes.
- **Before zipping, run `stack clean --full` (to ensure a small bundle)**
- **Do not submit node_modules or the .git folder**
- **Make sure the code you submit executes properly.**

The marking process will look something like this:

1. Extract **<studentNo>_<name>.zip**
2. Copy the **submission** folder contents into the assignment code bundle submission folder
3. Execute `stack build`, `stack test` (for automated testing) and `stack exec main/npm run dev` for front end

Please ensure that you test this process before submitting. Any issues during this process will make your marker unhappy and may result in a deduction in marks.

Late submissions will be penalised at 5% per calendar day, rounded up. Late submissions more than seven days will receive zero marks and no feedback.

Table of Contents

Assignment 2: Markdown to HTML **1**
Submission Instructions **1**
Table of Contents **2**
Introduction **3**
Goals / Learning Outcomes 3
Scope of assignment 3
Exercises (24 marks) **4**
Part A: (12 marks): Parsing Markdown **5**
Aside - Text Modifiers (2 marks) 5
Images (0.5 marks) 6
Footnote References (0.5 marks) 6
Free Text (1 mark) 6
Headings (1 mark) 6
Blockquotes (1 mark) 7
Code (1 mark) 8
Ordered Lists (2 marks) 8
Tables (3 marks) 8
Part B: (6 marks): HTML Conversion **9**
Text Modifiers (1 mark) 9
Images (0.5 marks) 10
Footnote References (0.5 marks) 10
Free Text (0.5 marks) 10
Headings (0.5 marks) 10
Blockquotes (0.5 marks) 10
Code (0.5 marks) 11
Ordered Lists (1 marks) 11
Tables (1 mark) 11
Part C (6 marks): Adding extra functionality to the webpage **13**
Part D (up to 6 bonus marks): Extension **14**
Report (2 marks) **15**
Code Quality (4 marks) **16**
Marking breakdown **17**
Correctness 17
Minimum Requirements: **18**
Changelog **18**

Introduction

In this assignment, we will use Haskell to develop a transpiler that converts Markdown strings into HyperText Markup Language (HTML). This task involves parsing Markdown syntax and generating corresponding HTML output. A web page is provided, in which Markdown will be sent through an HTML-based websocket connection to a Haskell backend server, the Haskell server will need to convert this Markdown into the corresponding HTML and return it back to the website. A skeleton code was provided which will handle the basic communication between the web page and your assignment code.

You are encouraged to utilise materials covered in previous weeks, including solutions for tutorial questions, to aid in the development of your transpiler. **You must reference or cite ideas and code constructs obtained from external sources**, as well as anything else you might find in your independent research, for this assignment.

The assignment is split up into Part A (parsing), Part B (pretty printing) and Part C (extras). However, we do recommend completing Part A/Part B in tandem.

The language you will parse will be based on the Markdown specification, however with additional restrictions to reduce ambiguity. It is important that you read the requirements of each exercise carefully to avoid unnecessary work.

Goals / Learning Outcomes

The purpose of this assignment is to highlight and apply the skills you have learned to a practical exercise (parsing):

- Use functional programming and parsing effectively
- Understand and be able to use key functional programming principles (higher order functions, pure functions, immutable data structures, abstractions)
- Apply Haskell and FP techniques to parse non-trivial Markdown text

Scope of assignment

You are only required to **parse** an expression into the necessary data types and convert the result to an HTML string such that it can be rendered using an existing interpreter. You will **not** be required to render the Markdown or HTML strings.

Exercises (24 marks)

These exercises provide a structured approach for creating the beginnings of a transpiler.

- **Part A (12 marks):** Parsing Markdown strings
- **Part B (6 marks):** Conversion between Markdown and HTML
- **Part C (6 marks):** Adding extra functionality to the webpage.
- **(Extension) ~~Part D~~ Part E:** extensions for bonus marks!

You must parse the input into an intermediary representation (ADT) such as an Abstract Syntax Tree to receive marks. This will allow easy conversion between your ADT and HTML.

You must add `deriving Show` to your ADT and all custom types your ADT contains. (Note that the skeleton code already has `deriving Show` on the `ADT` type for you, which you must not remove.) **You must not override this default `Show` instance as this will help us test your code.**

Your `Assignment.hs` file must export the following functions:

- `markdownParser :: Parser ADT`
- `convertADTHTML :: ADT -> String`

Example Scripts

For each of these exercises, there will be a series of provided Markdown files. By running `stack test`, it will try to parse the Markdown and save the output to a folder. This will generate HTML which you can manually view for correctness in a browser. During marking, we will be running your transpiler on more complex examples than the provided example scripts, therefore, it is important you devise your own test cases to ensure your parser is valid on more complex Markdown. It will also aim to produce a *git diff*, which is the difference between your output and the expected output. However, this requires installing the git command line tool. So, ensure that it is installed.

Furthermore, the more recommended way to test your code will be to use `npm run dev` in combination with `stack run main` can be used to run the webpage with a live editor, running your code in real-time.

Part A: (12 marks): Parsing Markdown

The first part of this task, requires you to parse a markdown string into an Abstract Data Type (ADT). This requires you to define your own Abstract Data Type and define a series of functions that parse everything in the requirements. Consider that you will need to convert the result to HTML and therefore, your ADT should have enough information to assist you in converting to HTML.

Aside - Text Modifiers (2 marks)

There are six different `modifiers` for inline text, which can change the way a markdown string will be rendered. You do **not** have to worry about any escape characters. All text modifiers will need to be **strictly** non-empty.

- Italic Text: Specified by a single underscore character, _. For example, `_italics_`
- Bold Text: Specified by a set of two asterisks, `**`, around a word. For example, `**bold**`
- Strikethrough: Specified by two tilde characters, `~~`, around a word. For example, `~~strikethrough~~`
- Link: Users can include a link to an external page using `[link text](URL)`. For example, `[click here](www.google.com)`. You **do not** need to consider links inside links.
- Inline Code: Users can include code in the middle of sentences, using a backtick character, `` ` ``. For example, `there is `code` here`
- Footnotes: Users can indicate a footnote with [^$\mathbb{Z}^+$], where $\mathbb{Z}^+ = \{1,2,3,\ldots\}$, i.e., any **positive** integer. For example, `[^1]`, `[^2]` and so forth. Note that you do **not** need to validate any sort of ordering on these numbers, e.g., the markdown may only contain one footnote `[^10]`. You also do **not** need to validate that the footnote comes with an appropriate reference (see Footnote References).
    - Note that there must not be any whitespace inside the `[` and `]`. For example, `[^ 1]`, `[^2 ]`, and `[ ^3]` are all not valid footnotes.

You do **not** need to consider text with nested modifiers, such as `**_bold and italics_**`.

Unless specified otherwise, the text inside the modifiers can include any amount of whitespace (**excluding** new lines). For example, `_ italics _`, `**bold **`, `~~ strikethrough~~`, `` ` inline code  ` `` and `[ link text](example.com)` are all valid.

Images (0.5 marks)

An image is specified with three parts:

1. The `Alt Text` is the alternative text for the image, which is displayed if the image fails to load or for accessibility purposes.
2. The `URL` is the URL or path to the image file. This can be a web URL or a local file path. The URL cannot contain any whitespace.
3. The `Caption Text` is the caption for the image.

```
        ![Alt Text](URL "Caption Text")
```

The alternative text and caption text should **not** consider the text modifiers.

There must be **at least one** non-newline whitespace character after the URL and the caption text. For example, `![Alt Text](URL"Caption Text")` is not a valid image.

Footnote References (0.5 marks)

Similarly to footnotes, footnote references consist of `[^`$\mathbb{Z}^+$`]`, where $\mathbb{Z}^+$ = {1,2,3,…}, i.e., any **positive** integer, followed by a colon (`:`), and then some text. Note that this text **will not** include the text modifiers. Leading whitespace before the text should be ignored.

```
        [^1]: My reference.
        [^2]:Another reference.
        [^3]:  The 2 spaces after the colon should be ignored
```

Free Text (1 mark)

There can be any amount of text which does not follow any of the following other types. This text may contain the modifiers. For example:

```
        Here is some **markdown**
        More lines here
        Text
```

Headings (1 mark)

Markdown headings are denoted by one or more hash symbols (`#`), followed by **at least one** whitespace character (excluding new lines). There can be up to 6 `#`'s, producing a heading up to level 6.

```
# Heading 1
## Heading 2
### Heading 3
#### Heading 4
##### Heading 5
###### Heading 6
```

Note that because at least one non-newline whitespace character is required, this is **not** a valid heading: `#Heading 1`

Alternatively, Heading 1 and Heading 2 can be specified with an alternative syntax (shown below). On the line **below** the text, add at least 2 equals sign (=) characters for heading level 1 or at least 2 dash (-) characters for heading level 2. There is no alternative syntax for any other heading levels.

```
Alternative Heading 1

======

Heading level 2

---------------
```

Importantly, headings may include any of the previously mentioned text modifiers, for example, a heading can be bolded, by surrounding it with a double asterisk.

```
# **Bolded Heading 1**
```

Blockquotes (1 mark)

To create a block quote in Markdown, you use the greater than symbol (>) at the beginning of a line followed by the text you want to quote. You can also include multiple lines of text within the **same** block quote by starting each consecutive line with the greater than symbol (>). Leading whitespace after the greater than symbol (>) and before the text should be ignored. The text inside the block quote **may** have text modifiers. You do **not** need to consider nested block quotes. For example:

```
> This is a block quote.
> It can **span** multiple lines.
```

Code (1 mark)

A code block in Markdown starts with three backticks (```) on a line by themselves, followed by an optional language identifier. The code block ends with another three backticks on a line by themselves. The code block should **not** consider the [text modifiers](). An example code block is:

````
```haskell
main :: IO ()
main = do
    putStrLn "Never gonna give you up"
    putStrLn "Never gonna let you down"
    putStrLn "Never gonna run around and desert you"
```
````

Ordered Lists (2 marks)

An ordered list consists of at least one *ordered list item* separated by **exactly** 1 new line character. An *ordered list item* starts with a positive number, a `.` (full stop) character, and at least one whitespace character (excluding new lines). An ordered list **must** start with the number 1, and any number after that can appear. You do not have to consider any other numbering system or an unordered list.

Ordered lists may contain sublists, where there will be **exactly** 4 spaces before each ordered list item. Each sublist **must** also start with the number 1. Similar to previous sections, list items **may** also contain [text modifiers]().

```
1. Item 1
    1. Sub Item 1
    2. Sub Item 2
    3. Sub Item 3
2. **Bolded Item 2**
6. Item 3
7. Item 4
```

You **do not** have to handle unordered lists.

Tables (3 marks)

To create a table in Markdown, you use pipes (`|`) to separate columns and **at least** three dashes (`-`) between each column to separate the header row from the content rows. Each column may contain varying amounts of dashes. Each row is written on a separate line. The beginning and ending pipes (`|`) are compulsory. Each row must have

the same amount of columns. Each cell **may** also contain text with the text modifiers. Leading and trailing whitespace before and after the text in each cell should be ignored.

```
| Tables        | Are           | Cool  |
| ------------- | ------------- | ----- |
| here          | is            | data  |
| here          | is            | data  |
| here | is also | **bolded data** |
```

Part B: (6 marks): HTML Conversion

The second part of this task requires you to convert your ADT into a HTML representation. The resulting HTML file must be formatted such that it is indented with **4 spaces** at the correct level to reflect the tree structure of HTML, ensuring that the HTML is valid and correctly renders the provided markdown. You do not need to indent the text modifiers, but other nested objects should be indentented correctly.

All HTML generated must be a self-contained webpage, i.e., including the following information, placing all generated HTML within the `<body>` tags.

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <title>Test</title>
</head>


<body>
    GENERATED CONTENT GOES HERE
</body>

</html>
```

As a reference for the conversion between markdown and HTML, here will be listed the conversion of all of the examples from above.

Text Modifiers (1 mark)

- Italics: `<em>italics</em>`
- Bold: `<strong>bold</strong>`
- Strikethrough: `<del>strikethrough</del>`

- Link: `<a href="URL">link text</a>`
- Inline Code: `<code>code</code>`
- Footnotes: `<sup><a id="fn1ref" href="#fn1">1</a></sup>`. It is important that you follow this convention precisely, where `1` is the number specified with the footnote, to ensure the footnotes work.

Images (0.5 marks)

The image must be in an image tag, with the appropriate attributes filled.

```
<img src="URL" alt="Alt Text" title="Caption Text">
```

Footnote References (0.5 marks)

A footnote reference must be encased in a `<p>` tag, and have the appropriately numbered `id`.

```
<p id="fn1">My reference.</p>
<p id="fn2">Every reference should be prefixed with 2
spaces.</p>
```

Free Text (0.5 marks)

Every line of free text must be encased in `<p>` tags. You do not need to consider how to handle newlines.

```
<p>Here is some <strong>markdown</strong></p>
<p>More lines here</p>
<p>Text</p>
```

Headings (0.5 marks)

Where, the number after the `h`, contains the level of the heading, for example, in heading level 1:

```
<h1>Heading 1</h1>
```

Blockquotes (0.5 marks)

Each blockquote must be encased by `<blockquote>`, while each line within the blockquote must be encased with a `<p>` tag.

```
<blockquote>
    <p>This is a block quote.</p>
    <p>It can <strong>span</strong> multiple lines.</p>
</blockquote>
```

Code (0.5 marks)

The code block must be encased in both the `<pre>` and the `<code>` tags, and the language (e.g., `haskell`) must be included within the class attribute, prefixed by `language-`. The newlines and code indentation **must** remain.

```
<pre><code class="language-haskell">main :: IO ()
main = do
    putStrLn "Never gonna give you up"
    putStrLn "Never gonna let you down"
    putStrLn "Never gonna run around and desert you"
</code></pre>
```

Ordered Lists (1 marks)

Ordered lists must begin and end with the `<ol>` tag, and each list item must begin and end with the opening/closing `<li>` tag.

```
<ol>
    <li>Item 1
      <ol>
          <li>Sub Item 1</li>
          <li>Sub Item 2</li>
        <li>Sub Item 3</li>
      </ol>
    </li>
    <li><strong>Bolded Item 2</strong></li>
    <li>Item 3</li>
    <li>Item 4</li>
</ol>
```

Tables (1 mark)

The HTML convention for representing tables involves using the `<table>`, `<tr>`, `<th>`, and `<td>` elements. `<table>` represents the entire table, `<tr>` represents a row within the table, `<th>` represents a header cell within a table row, used for the header row, and `<td>` represents a data cell within a table row, used for the content rows.

```
        <table>
            <tr>
                <th>Tables</th>
                <th>Are</th>
                <th>Cool</th>
            </tr>
            <tr>
                <td>here</td>
                <td>is</td>
                <td>data</td>
            </tr>
            <tr>
                <td>here</td>
                <td>is</td>
                <td>data</td>
            </tr>
            <tr>
                <td>here</td>
                <td>is also</td>
                <td><strong>bolded data</strong></td>
            </tr>
        </table>
```

Part C (6 marks): Adding extra functionality to the webpage

This task involves changing the webpage to include extra capabilities allowing a more feature-full UI. You **will not** be marked on the layout, or ease of use of features, as long as they are clearly visible to your marker, e.g., a button should be clearly visible on the screen. This task will involve some light additions to both the HTML page and TypeScript code. This will likely involve creating an observable stream for the data, merging it into the subscription stream, and sending the information to the Haskell backend. The communicated information between the Haskell backend and the webpage will need to be updated to include additional information that the user wants the engine to achieve.

- A button must be added to the webpage for **saving,** where the converted HTML is saved **using** Haskell. The user does not need to be prompted for a file name, and the HTML should be saved according to the current time, formatted in ISO 8601 format for the current date and time: `YYYY-MM-DDTHH:MM:SS`. The function `getTime` is provided which will provide you this time in an `IO String` format.
- A separate input box, to allow the user to change the title of the page, instead of the default `Converted HTML`.

Part D (up to 6 bonus marks): Extension

Implement anything that is interesting, impressive, or otherwise “shows off” your understanding of Haskell, Functional Programming, and/or Parsing.

To achieve the maximum amount of bonus marks, the feature should be similar in complexity to Part C (6 marks):

The bonus marks only apply to **this assignment,** and the final mark for this assignment is **capped** at 30 marks (100%). This means you cannot score more than 30 marks or 100%.

Some suggestions for extensions of varying complexity and difficulty:

- Markdown validation
    - E.g., enforce all table columns have the same width
- **Correct** BNF for the Markdown you are parsing in report (worth 2 marks)
    - For any part of the parser which is not context-free, you may simplify the parsing rules to be context-free.
- Further extensions to the webpage for extra features, using RxJS
- Parse nested text modifiers, such as `**_bold and italics_**` and `[click **here**](https://example.com)`
- Parse further parts of the markdown specification which make use of interesting parsers, which you have not used in other parts of the assignment.
- **Comprehensive** test cases over the parser and pretty printing
    - Warning: It is super hard to be comprehensive, stay away unless you love testing.

(Choosing one of the simpler suggestions to implement may not receive the maximum available marks).

Report (2 marks)

You are required to provide a report in PDF format of max. 600 words (markers will not mark beyond this word limit). Descriptions of extensions can use up to 200 words per extension feature.

Make sure to summarise the intention of the code, and highlight the interesting parts and difficulties you encountered. Focus on the **"why" not** the **"how".**

Additionally, just posting screenshots of code is **heavily discouraged**, unless it contains something of particular importance. Remember, markers will be looking at your code alongside your report, so we do not need to see your code twice.

Importantly, this report must include a description of why and how parser combinators helped you complete the parsing. In summary, your report should include the following sections:

- Design of the code (including data structures)
    - High-level description of approach
    - High-level structure of code
    - Code architecture choices
- Parsing
    - Usage of parser combinators
    - Choices made in creating parsers and parser combinators
    - How parsers and parser combinators were constructed using the Functor, Applicative, and Monad typeclasses
- Functional Programming (focusing on the **why**)
    - Small modular functions
    - Composing small functions together
    - Declarative style (including point free style)
- Haskell Language Features Used (focusing on the **why**)
    - Typeclasses and Custom Types
    - Higher order functions, fmap, apply, bind
    - Function composition
- Description of Extensions (if applicable)
    - What you intended to implement
    - What you did implement
    - What is cool/interesting/complex about it
    - This may include using Haskell features that are not covered in course content

There is some overlap between the sections. You should **avoid** repeating descriptions or ideas in the report.

Code Quality (4 marks)

Code quality will relate more to how understandable your code is. You must have readable and **functional** code, commented when necessary. Readable code means that you keep your lines at a reasonable length (< 80 characters), that you provide comments above non-trivial functions, and that you comment sections of your code whose function may not be clear.

Your functions should all be small and modular, building up in complexity, and taking advantage of built-in functions or self-defined utility functions when possible. It should be easy to read and understand what each piece of your code is doing, and why it is useful. Do **not** reimplement library functions, such as map, and use the appropriate library function when possible.

Your code should aim to re-use previous functions as much as possible, and not repeat work when possible.

Code quality includes your ADT and if it is well structured, i.e., does not have a bunch of repeated data types and follows a logical manner (the JSON example from the applied session is a good example of what an ADT should look like).

Marking breakdown

The main marking criteria for each parsing and pretty printing exercise consists of two parts: **correctness** and **FP style.** Both correctness and FP style will be worth 50% of the marks for each of the exercises, i.e., if your code passes all tests, you will get at least half marks for Exercise A, and Exercise B.

You will be provided with some sample input and tests for determining the validity of the outputted HTML files. The sample inputs provided will **not** be exhaustive, you are **heavily** encouraged to add your own, perhaps covering edge cases.

Correctness

We will be running a series of tests which test each exercise, and depending on how many of the tests you pass, a proportion of marks will be awarded

FP Style

FP style relates to if the code is done in a way that aligns with the unit content and functional programming.

You must apply concepts from the course. The important thing here is that you need to use what we have taught you effectively. For example, defining a new type and its `Monad` instance, but then never actually needing to use it will not give you marks. Note: using bind `(>>=)` for the sake of **using the** `Monad` when it is not needed will not count as "effective usage."

Most importantly, code that does not utilise Haskell's language features, and that attempts to code in a more imperative style, will not be awarded high marks.

Minimum Requirements:

An estimate of a passing grade will be parsing up to and including code blocks, but not lists or tables, where the difficulty and the marks step up. However, this will need to be accompanied by high code quality and a good report.

A higher mark will require parsing of the more difficult data structures, and modifications of the HTML page.

Changelog

- Add note that text modifiers must be non-empty
- Add note about BNF can simplify parser, if and only if the parser is not context free.
- 18 Sep: Remove the requirement to parse nested text modifiers and instead make that an extensio