MARKDOWN TO HTML CONVERTER

By: Aaron

1. Design of the Code

High-level Description of Approach:

The Assignment.hs code handles the parsing and conversion of markdown text to Abstract Data Types (ADTs) and into HTML, while the JavaScript code manages the user interface and interactions.

High-level Structure of Code:

- **Haskell Code:**
    - Assignment.hs contains markdownParser, convertADTHTML,saveHTML, which parses the input string, converts it to HTML format, and saves it in a .html file respectively

Code Architecture Choices:

- **Reusability:**

  Small, reusable functions and modules, make them easier to maintain and extend. For example, helper functions such as parseFootnoteNumber could be utilized by both parseFootnote and parseFootnoteReference. Besides that, the helper function parseDelimitedContent has helped simplify the parsing of TextModifiers substantially.

- **ADT and AdtType**
    - The AdtType is defined as a sum type (also known as an algebraic data type) with multiple constructors, each representing a different type of “complex” Markdown element. Sum types enable pattern matching, which simplifies the processing of different Markdown elements. Functions can easily deconstruct AdtType values and handle each case appropriately.
    - The ADT type is defined as a newtype wrapper around a list of AdtType. The newtype wrapper encapsulates the list of AdtType, providing a clear distinction between a single AdtType and a collection of them. Thus, it is

easier to extend the functionality of the ADT type. For example, additional functions or instances can be defined specifically for ADT without affecting the underlying list of AdtTypes.

2. Parsing

Usage of Parser Combinators:

Parser combinators are used extensively to build complex parsers from simpler ones, leveraging Haskell's powerful type system and functional programming features. To illustrate, parseModifier combines the six text modifier parsers using the <|> symbol, which tries the first parser and, if it fails, tries the subsequent one. As a result, simple markdown parsers can be combined to form a more complex parser that can handle a variety of input combinations.

Choices Made in Creating Parsers and Parser Combinators:

- **Basic Parsers:** Defined for fundamental elements like positiveInt, footnoteNumber, and trimming.
- **Combinators:** Used to combine basic parsers into more complex ones, such as parsers for heading, lists, and code blocks.

Construction Using Functor, Applicative, and Monad Typeclasses:

- **Functor:** Allows for mapping functions over parsed results. By using <$>, I can map the ADT constructor over the result of parseAllADTs, transforming a Parser [AdtType] into a Parser ADT.
- **Applicative:** Through the example of the “someTill” helper function, the <*> operator sequences the parsers p and manyTill p end, ensuring that p is parsed first, followed by manyTill p end.
- **Monad:** In parseOrderedListItem, the ability to parse an optional sublist based on the current level of indentation is made straightforward by the use of “do-notation” monad. Without them, handling such dependencies would require more complex and less readable code.

3. Functional Programming (Focusing on the Why)

Small Modular Functions:

The code is composed of small, focused functions that each handle a specific task. Small functions like parseDelimitedContent can be reused in different parts of the codebase, reducing duplication.

Composing Small Functions Together:

Functions like parseOrderedListItem are composed of smaller functions (positiveInt, parseFreeTextLine, parseSubOrderedList), making the code modular and easier to debug, whilst allowing for the creation of complex parsing from simple parser blocks.

Declarative Style:

Declarative style of the 'trim' helper function focuses on what the code should accomplish rather than how, making the code more expressive and closer to natural language.

4. Haskell Language Features Used (Focusing on the Why)

Typeclasses and Custom Types:

Usage of AdtType ensure that only valid data structures are used as part of parseAllADTs, reducing runtime errors.

Higher Order Functions, fmap, apply, bind:

Higher-order functions like manyTill and some allow for reusable parsing logic, making the code more modular and maintainable, while (<$>) and (>>) allows chaining of multiple markdown parsers together.

Function Composition:

Functions like indent are composed using operators like (.), allowing for expressive code.

5. Description of Extensions

Nested *collection* of Text Modifiers

```
**normal bold _italic bold_ ~~strikethrough bold~~**
[`**_~~penta-nested~~_**`](in all 5 way)
```

Intention

The goal was to extend the existing markdown parser to support nested text modifiers within link texts and other markdown elements. This would allow more complex and expressive markdown elements like bold, italic, and strikethrough to be nested within link texts and other modifiers.

Implementation

The implementation involved modifying the TextModifier data type to use FreeText instead of String for link texts. The parseLink function was updated to parse the link text as FreeText, allowing it to contain nested modifiers. Additionally, the convertTextModifier function was updated to handle the conversion of FreeText to HTML, ensuring that nested modifiers are correctly rendered, without imposing any nesting limit through recursion.

Cool/interesting/complex

The complexity of this extension lies in co-recursion and converting various patterns of nested text modifiers. The convertFreeText function has to be modified to be capable of identifying and processing various markdown elements, whether one is simply nested inside another, or even; an outer modifier containing a *collection* of nested modifiers in it. The result is a more powerful and flexible markdown parser that can handle complex markdown documents with nested formatting elements. This extension enhances the expressiveness of the markdown language and demonstrates the power of recursive parsing techniques in functional programming.

Table Column part D

Intention

In my extended Markdown parser, I intended to implemented an additional constraint to ensure that each table cell (before trimming) has the same number of characters across all rows (excluding delimiters). The idea is that each cell, even though it may contain varying content (e.g., text modifiers or whitespace), should have identical pre-trimmed lengths.

Implementation

To achieve this, I introduced a “RawCell” data type, which captures both the raw content of a cell and its trimmed version. The parseTableRow function included rawLengths and was eta-reduced, and will return both the parsed row and the raw lengths of each cell, which are later checked by parseTableRowWithCheck against the header row's cell lengths. The parseTableRowWithCheck :: Int -> [Int] -> Parser TableRow function ensures each row has the same number of cells (expectedLen) and each cell's raw length matches the header's (expectedRawLengths).

Cool/interesting/complex

The complexity lies in enforcing this pre-trim rule, while still allowing flexibility in the content itself, especially when parsing text modifiers. Using Haskell's record syntax for RawCell enables a clear distinction between raw and trimmed content, making the code more readable and modular.

*(Note: The **tables.diff** fails as the expected output had varying lengths per column)*
*(Note: The **heading.diff** also fails as the expected output had 2 empty tags)*

Read Markdown Files

Choose File No file chosen

Intention

I intended to implement a feature that allows users to upload a Markdown file, which would then be read and its content displayed in the Markdown input area. This feature aims to enhance user experience by providing a convenient way to load and add to existing Markdown files directly within the application.

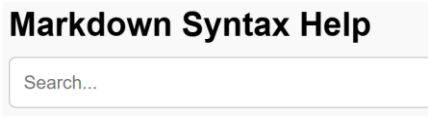
Implementation

An HTML file input element, and an Observable stream to handle file changes were added. After selecting a file, its content is read and updated in the application state, which then populates the Markdown input area. To maintain the original content of the uploaded Markdown file, I added the initialHTML attribute to the State type. This attribute stores the initial HTML content derived from the input Markdown file, allowing for easy reference and potential restoration of the original state if needed.

Cool/interesting/complex

The implementation leverages RxJS to create a reactive stream that listens for file input changes. The file.text() method returns a Promise that resolves with the file's text content. By converting this Promise into an Observable using from(), the implementation seamlessly integrates asynchronous file reading into the reactive stream. This approach ensures that the UI is automatically updated whenever a new file is uploaded, providing seamless user experience.

Markdown Help Library



Intention

I intended to implement a feature that allows users to toggle the visibility of a help section containing Markdown syntax guidelines, for quick access during editing.

Implementation

An Observable stream listens for click events on the help button. When the button is clicked, the stream toggles the visibility of the help section using T Flip-Flop, which is then updated in the subscription block. Additionally, I added a search input feature to allow users to lookup the required syntax for their desired markdown. This was achieved by creating a more complex Observable stream that listens for input events on the search field and filters the help list items based on the user's input value.

Cool/Interesting/Complex

By adding a helpVisible: false in the input$ observable, any typing action in the markdown area, such as a user resuming his essay, will automatically minimize the guideline pop-up. This allows the user to seamlessly continue their current activity without needing to manually toggle the button to hide the guidelines, thereby enhancing user experience. The *method Array.from(items).forEach((item) => { … })* is crucial as it iterates over each <li> and dynamically updates their visibility based on the search input, ensuring real-time filtering of the help content.

Word Count

Intention

I intended to implement a feature that dynamically counts and displays the number of words in the Markdown input area. This feature aims to provide users with real-time feedback on their word count, which can be useful for meeting specific content length requirements.

Implementation

I implemented the word count feature by creating an Observable stream that listens for input events in the Markdown input area. The main pipe processes the input to count the number of words (excluding the title box) and updates the application state with the current word count. This count is then displayed and updated in the UI through the subscription block.

Cool/Interesting/Complex

What defines a 'word'? The process below explains how I formulated 'wordcount' :

- **String Splitting:** The split(/\s+/) method splits the Markdown content into an array of words based on whitespace characters. The regular expression 's+' matches one or more whitespace characters, effectively breaking the content into individual words.
- **Filtering Empty Strings:** The filter((word) => word.length > 0) method removes any empty strings from the array. This is important because splitting a string can sometimes result in empty strings, especially if there are multiple consecutive whitespace characters.

BNF Markdown for Parsers

For alternative view, refer to the raw text in README.md file

```Markdown
<ADT>                ::= "[" <AdtTypeList> "]"

<AdtTypeList>        ::= <AdtType>
                       | <AdtType> "," <AdtTypeList>

<AdtType>            ::= <Image>
                       | <FootnoteReference>
                       | <Heading>
                       | <Blockquote>
                       | <CodeBlock>
                       | <OrderedListADT>
                       | <Table>
                       | <EmptyLineADT>
                       | <FreeTextADT>

<Heading>            ::= <HashHeading>
                       | <AlternativeHeading>
<HashHeading>        ::= <inlineSpace> <Hashes> " " <FreeText>
<AlternativeHeading> ::= <inlineSpace> <FreeText> <HeadingLevel> "\n"
<Hashes>             ::= "#" | "##" | "###" | "####" | "#####" | "######"

<HeadingLevel>       ::= <inlineSpace> <EqualsLevel>
                       | <inlineSpace> <DashLevel>
<EqualsLevel>        ::= "==" <EqualsLevelTail>
<DashLevel>          ::= "--" <DashLevelTail>
<EqualsLevelTail>    ::= "=" <EqualsLevelTail>
                       | ""
<DashLevelTail>      ::= "-" <DashLevelTail>
                       | ""

<OrderedListADT>             ::= <OrderedList>
<OrderedList>                ::= <FirstOrderedListItem> <SubsequentOrderedListItems>
<FirstOrderedListItem>       ::= "1. " <FreeText> <OptionalSubList>

<SubsequentOrderedListItems> ::= ""
                               | <OrderedListItem> <SubsequentOrderedListItems>
<OrderedListItem>            ::= <Digits> ". " <FreeText> <OptionalSubList>
<OptionalSubList>            ::= ""
                               | <SubOrderedList>
<SubOrderedList>             ::= <FirstSubOrderedListItem> <SubsequentSubOrderedListItems>
<FirstSubOrderedListItem>    ::= "    " <FirstOrderedListItem>
<SubsequentSubOrderedListItems> ::= ""
                               | <SubOrderedListItem> <SubsequentSubOrderedListItems>
<SubOrderedListItem>         ::= "    " <OrderedListItem>

<Blockquote>         ::= <inlineSpace> ">" <FreeText>
                       | <inlineSpace> ">" <FreeText> <Blockquote>

<Table>              ::= <TableRow> <SeparatorRow> <TableRows>
<TableRow>           ::= <inlineSpace> "|" <TableCells> "|" "\n"
<TableCells>         ::= <CellContent>
                       | <CellContent> "|" <TableCells>
<CellContent>        ::= <inlineSpace> <FreeText> <inlineSpace>
<SeparatorRow>       ::= <inlineSpace> "|" <SeparatorCells> "|" "\n"
<SeparatorCells>     ::= <SeparatorCell>
                       | <SeparatorCell> "|" <SeparatorCells>
<SeparatorCell>      ::= <inlineSpace> "---" <DashLevelTail> <inlineSpace>
<TableRows>          ::= <TableRow>
                       | <TableRow> <TableRows>

<FootnoteReference>  ::= <FootnoteNumber> ":" <String>

<Image>              ::= "![" <String> "](" <String> " \"" <String> "\")"

<CodeBlock>          ::= <inlineSpace> "```" <String> "\n" <String> "\n```\n"

<FreeTextADT>        ::= <FreeText> "\n"
```

```
<FreeText>           ::= "[" <FreeTextorModifierList> "]"
<FreeTextorModifierList> ::= <FreeTextorModifier>
                           | <FreeTextorModifier> "," <FreeTextorModifierList>
<FreeTextorModifier> ::= <FreeTextChunk>
                       | <TextModifier>
<FreeTextChunk>      ::= <String>
<TextModifier>       ::= <Italic>
                       | <Bold>
                       | <Strikethrough>
                       | <Link>
                       | <InlineCode>
                       | <Footnote>
<Italic>             ::= "_" <FreeText> "_"
<Bold>               ::= "**" <FreeText> "**"
<Strikethrough>      ::= "~~" <FreeText> "~~"
<Link>               ::= "[" <FreeText> "](" <String> ")"
<InlineCode>         ::= "`" <FreeText> "`"
<Footnote>           ::= <FootnoteNumber>
<FootnoteNumber>     ::= <inlineSpace> "[^" <PositiveInt> "]"

<EmptyLineADT>       ::= <inlineSpace> "\n"
<inlineSpace>        ::= ""
                       | " "
                       | " " <inlineSpace>

<String> ::= <Char> | <Char> <String>
<Char> ::= "a" | ... | "z" | "A" | ... | "Z" | "0" | "1" | "2" | ... | "9" | " " | "!" | ... | "~"

<PositiveInt> ::= < PositiveDigit> | < PositiveDigit > <Digits>
<Digits> ::= <Integer> | <Integer> <Digits>
<PositiveDigit> ::= "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
<Integer> ::= "0" | <PositiveDigit>
```