Assignment 2 Design Report

Arya Sadeghi

Design Overview

The code converts Markdown to HTML using parser combinators, handling various Markdown components like inline formatting, block-level structures, and nested content. It recursively finds distinct patterns while addressing whitespace and formatting requirements. The code trims the input to remove unnecessary whitespace and maintains desired styles by handling internal formatting and whitespaces.

Abstract Data Types (ADT)

The ADTs model Markdown elements as distinct types, allowing parsed content to be represented structurally. The main ADTs include:

- **Modifier Types:** **`Bold, Italic, Link, Strike, Code, FooteNote, etc.`**, represent inline formatting elements that modify enclosed content.
- **Block-Level Types:** **`Header, List, Table`**, and **`Quote`** represent larger structures within the document, while **`Image, Footnote Ref`** capture external references.
- **Compositional Structure:** **`StartLine, NewLine`**, and **`HTML`** are used to connect various ADTs, forming a tree-like representation of the document, where nested elements are naturally supported. For example, **`Table ([ADT], [[ADT]])`** represents a table with headers and rows, while **`List`** represents ordered lists and supports nested sublists (similar to cons list).

Approach

The code employs a recursive parsing strategy, using parser combinators to handle nested structures and various Markdown syntax elements. Each parser targets a specific pattern, and parsers are composed to manage more complex structures, providing a layered parsing process:

1. **Parser Combinators:** **`Many, some`**, and **`<|> (alternative)`** combine simple parsers into more complicated ones, such as **`nestedModifiers`**, which recursively parses different inline formatting.

2. **Handling various Markdown structures:** The code differentiates between inline and block-level components by employing distinct parsers. For example, **`headerHash`** parses headers with **`#`**, whereas **`codeBlock`** handles multi-line code blocks with backticks **(`)**. The non-modifier markdown structures are checked first, and if none are found, the search for inline modifiers continues. None-modifier parsing removes leading spaces and parses the information as required.

3. **Error Management:** The Parser combinator (**`ParseResult`**) handles custom errors (**`Result`** or **`Error`**) to signal parsing success or failure, allowing for gentle recovery from unexpected inputs.

Code Structure and Architecture

The code is modular and applies functional programming principles to convert Markdown into HTML, with distinct layers for parsing, HTML conversion, and auxiliary utilities.

1. **Parsing Functions:** This function deal with specific Markdown components like **`boldCheck`** for bold text and **`listCheck`** for ordered lists with nested sublists. Each parser focusses on certain patterns, employing combinators to handle variants and layered structures like headers and multi-line code blocks.

2. **HTML conversion:** The **`convertADTHTMLHelper`** function converts parsed ADTs to HTML. It recursively analyses the ADTs, adding appropriate HTML tags such as for bold or for italic. It also supports nested items through recursive traversal of the ADT.

3. **Auxiliary Functions:** Utilities like **`indent`**, **`wrapWithTag`**, and **`trimFirstEnd`** manage formatting, while parsers like **`manyTill`** (manually created) and **`detectStartEndMod`** help recognise patterns like text modifiers and delimiters.

Parser Combinators, Type-class Usage, and FP Style

Parser combinators such as `<|>`, `many`, and `sepBy1` create complicated parsers from smaller ones, allowing for smooth processing of various Markdown components. They handle nested structures and different syntaxes well. Combining tableHeaderCheck and tableContentRow retains the row-based structure of tables, while `<|>` enables versatile Markdown syntax (e.g., hash-based or underlined headers) with little repetition.

Conditional parsing in Markdown allows the parser to adapt based on content, making it crucial for handling optional structures like nested lists. The optionalSubList parser dynamically checks for indentation, providing a robust way to manage nested elements.

Haskell's typeclasses (Functor, Applicative, and Monad) manage parsing complexity by enabling transformations and sequencing of operations. The Functor typeclass (**`fmap`**) transforms parsed results into structured types (ADTs) like Text, making it easy to work with parsed data. Applicative (**`<*>`**) combines parsers sequentially, ensuring cohesive parsing of multiple components. Monad (**`>>=`**) enables dependent parsing, where subsequent actions are influenced by previous results, which is essential for handling nested structures like lists and sublists.

```haskell
import Control.Applicative (Alternative (many, some, (<|>)), Applicative (liftA2), optional)
import Control.Monad (void, when)
import Data.Maybe (maybeToList)
import Instances (ParseError (..), ParseResult (..), Parser (..), isErrorResult)
```

```haskell
boldCheck
<|> italicCheck
<|> strikeCheck
<|> inlineCode
<|> footNote
<|> linkCheck
<|> normalText
<|> parseAnyChar
```

```haskell
footNote :: Parser ADT
footNote =
  is '[' >>= \_ ->
    is '^' >>= \_ ->
      notFollowedBySpace >>= \_ ->
        notFollowedByString "\n" >>= \_ ->
          int >>= \n ->
            is ']' >>= \_ ->
              (notFollowedByString ":" <|> eof) >>= \_ ->
                return $ FootNote n
```

Extensions

Functional Programming Aspect	Description (Why, How)	Example
Small Modular Functions	The code employs small, focused functions for specific tasks, promoting readability, reusability, and maintainability.	**`boldCheck`** parses bold text, **`italicCheck`** parses italic, **`listCheck`** handles parsing of lists, and **`convertADTHTMLHelper`** converts ADTs to HTML recursively. Every function is designed for specific purpose without overcomplicating the logics. Keeping it modular and small
Function Composition	Uses function composition to build complex behaviors from simpler functions, combining them using operators like `(.)` for sequential transformations.	**`trimFirstEnd = reverse . dropWhile isSpace . reverse . dropWhile isSpace`** composes functions to trim leading and trailing whitespace.
Higher-Order Functions	Functions operate on other functions, enabling flexible and abstract parsing behavior, and allowing code reuse and composability.	**`fmap, <*>, *>, <*, <<=, >>=,`** transforms parsed content to an ADT in **`convertADTHTMLHelper`**, **`many`** and **`some`** repeatedly apply parsers, and **`optional`** allows for conditional parsing in **`nestedModifiers`**. Demonstrating higher ordered functions used in **`convertADTHTMLHelper`**
Typeclasses Usage	Utilizes Haskell's `Functor`, `Applicative`, and `Monad` typeclasses to create modular and composable parsers, enabling powerful abstractions.	**`fmap`** maps results to an ADT, **`<*>`** is used in parser combinations, such as **`tableHeaderCheck`**, and **`optional`** works with applicatives in **`nestedModifiers`**
Algebric Data Types (ADTs)	ADTs are defined to model different Markdown elements, ensuring type safety and clear representation of parsed structures, aiding easy transformation to HTML.	**`data ADT = Bold ADT | Italic ADT | List [(ADT, ADT)] | Table ([ADT], [[ADT]]) | ...`** represents various Markdown elements.
Recursive Data Structures	Employs recursion in ADTs and parsing functions to handle nested Markdown elements effectively, supporting hierarchical document structures.	**`convertADTHTMLHelper`** recursively traverses ADTs for HTML conversion, and **`nestedModifiers`** parses nested elements like inline formatting or lists.
Point-Free Style	Avoids specifying arguments explicitly by focusing on function composition, leading to concise and expressive code.	For example: **`trimFirstEnd = reverse . dropWhile isSpace . reverse . dropWhile isSpace`** is written in a point-free style.
Conditional Parsing	Parsing behavior adapts based on the input content using combinators like `optional` and `when`, allowing flexibility in handling optional elements.	**`optionalSubList`** checks indentation before parsing sublists, and **`optional`** is used in **`nestedModifiers`** to account for possible Markdown structures.
Alternative Handling with **<|>**	The <|> combinator attempts multiple parsers in sequence, trying different parsing strategies to handle varied Markdown syntaxes.	**`parseHeaderLine`** tries parsing with **`headerHash`** and **`parseHeaderLineMatch`**, while **`nestedModifiers`** uses multiple parsers like **`boldCheck`** and **`italicCheck`** to support different Markdown elements.
Error Handling	Implements custom error handling for parsers to provide meaningful feedback when parsing fails, enhancing robustness and debuggability.	**`failParser`** generates **`ParseError`** messages for specific parsing errors, and **`manyTill`** handles parsing until a delimiter or error occurs in functions like **codeBlock.**
Side Effects Management	Isolates side effects using the `IO monad`, ensuring that the pure functional core remains unaffected by external operations like file `I/O`, maintaining purity and referential transparency.	**`saveHtmlFile`** performs file writing within the **`IO`** monad to save the parsed HTML output.

Functional Programming Aspect	Description (Why, How)	Example
Immutable Data	Utilizes immutable data structures throughout, ensuring consistency and preventing unintended side effects, which aligns with functional programming's emphasis on immutability and predictability.	All ADTs (e.g., **`Bold, Italic, List,`** etc.) are immutable, and functions return new ADT instances without modifying existing data.
Declarative Style	Specifies **what** to parse using combinators and not how to parse, making the code expressive and closer to the syntax of Markdown, avoiding imperative instructions.	Combinators like **`<|>, many`**, and **`sepBy1`** are used in **`parseHeaderLine`**, **`nestedModifiers`**, and **`tableHeaderCheck`**. They all define what to parse instead of how to parse.

- **Markdown validation**

Code Block Language: When users specify a language for a code block (e.g., ```` ```haskell ````), the parser checks this language against a predetermined list of the top 100+ known programming languages (`constList`). This validation happens in the codeBlock function. If the language is not recognised, the parser treats the material as plain text rather than a code block. This guarantees that only valid languages are highlighted correctly.

Table's Columns length: Our parser ensures that each row in a table has the same amount of columns as the header. This validation is done by the `tableHeaderCheck` method. If any row

```
when (length lang > 0 && lang `notElem` constList) (failParser "Not valid language")
```

has a different number of columns, the parser flags the table as invalid, and the remaining information is processed as free text. For example:

```
|a|b|
|"---"|"---"|
|c|d|
```

Is known to be valid, where as

```
|a|b|
|"---"|"---"|
|c|d|e|
```

is invalid due to the inconsistent number of columns.

- **BNF Grammar for markdown parser**

```haskell
if length rowContent == columns
  then
    return $
      (\a -> (parseInnerContent (trimFirstEnd $ a) nestingModifiers))
        <$> rowContent
  else failParser "Invalid column number"
```

We created a thorough BNF grammar that closely matches our parser's implementation. The grammar contains precise restrictions that represent the parser's reasoning, such as:

- **Ordered List Constraints**: The BNF specifies that ordered lists must start with the number `1.`, and subsequent items can be any positive integer. This mirrors the parser's requirement that the

```
<ordered_list> ::= <ordered_list_item_1> (<newline> <ordered_list_item>)*
<ordered_list_item_1> ::= "1" "." <whitespace> <text> (<sublist>)?
<ordered_list_item> ::= <number> "." <whitespace> <text> (<sublist>)?
<sublist> ::= <newline> <indentation> <ordered_list_item_1> (<newline> <indentation> <ordered_list_item>)*
<indentation> ::= "    " <indentation>?  /* Allows nested sublists with increasing indentation */
```

  first list item begins with `1.`.

- **Code Block Language Specification**: The BNF permits any letter sequence to be used as the language identification in a code block. However, our parser confines this to a preset set of languages. To align with the parser, the BNF is considered to be context-free in this scenario.

This thorough BNF guarantees that the parsing rules are precisely stated, allowing for easier comprehension and maintenance of the parser.

- **Extra RxJs Features**

We have integrated additional RxJS features to enhance user interaction:

```
const initialState: State = {
    markdown: "",
    HTML: "",
    renderHTML: true,
    style: false,
    title: "Converted HTML",
    save: false,
};
```

```
return ajax<{ html: string }>({
    url: "/api/convertMD",
    method: "POST",
    headers: {
        "Content-Type": "application/x-www-form-urlencoded",
        "Save-State": s.save,
        "Title-Page": s.title,
        "Style-Css": s.style
    },
```

- **Dynamic CSS Inclusion**: A checkbox allows users to incorporate specific CSS style in the stored HTML output. When chosen, RxJS detects the change in the stream and changes the application's state. This creates a new data stream that sends a POST request to the Haskell server containing

the changed preference. The server then incorporates the CSS style into the resulting HTML. This functionality illustrates seamless integration of frontend reactive programming with backend Haskell parsing.

• Nested Modifiers

Our parser supports comprehensive nesting of text modifiers, handled recursively in the `nestingModifiers` function:

- **Modifier Nesting Logic**: Modifiers like bold (`**`), italic (`_`), strike-through (`~~`), and inline code (`` ` ``) can be nested within each other but not within the same type. For instance, nesting bold within bold is not allowed; additional bold markers inside a bold section are treated as plain text.

- **Example**: The input `**~~_Code, Bold, Italic and Strike_~~**` is parsed into: `<p><strong><del><em><code>Code, Bold, Italic and Strike</code></em></del></strong></p>`

These methods are recursively applied to all modifiers for a comprehensive and complete nesting logic.

```haskell
nestingModifiers :: Parser [ADT]
nestingModifiers =
  many
    ( boldCheck
      <|> italicCheck
      <|> strikeCheck
      <|> inlineCode
      <|> footNote
      <|> linkCheck
      <|> normalText
      <|> parseAnyChar
    )
```

```haskell
italicCheck :: Parser ADT
italicCheck =
  Italic
    <$> detectStartEndMod
      "_"
      '_'
      1
      True
      ( many
          ( boldCheck
              <|> strikeCheck
              <|> inlineCode
              <|> footNote
              <|> linkCheck
              <|> normalText
              <|> parseAnyChar
          )
      )
```

```haskell
strikeCheck :: Parser ADT
strikeCheck =
  Strike
    <$> detectStartEndMod
      "~~"
      '~'
      2
      True
      ( many
          ( boldCheck
              <|> italicCheck
              <|> inlineCode
              <|> footNote
              <|> linkCheck
              <|> normalText
              <|> parseAnyChar
          )
      )
```

This demonstrates the parser's ability to handle complex nesting while adhering to markdown rules.

Edge Cases: An input like `**** ****` is parsed as `<p><strong>** **</strong></p>`, treating the inner `** **` as plain text since same-type nesting is not permitted.

• Parsing Further Parts of The Markdown

We have extended the parser to include additional markdown features:

- **Unordered Lists**: Implemented in the `unlistCheck` and `unParseListItem` functions, unordered lists start with a `-` followed by at least one whitespace and the list item text. The parser supports nested unordered lists using indentation (4 spaces per level). For example:

```
- Item 1
    - Sub Item 1
    - Sub Item 2
    - Sub Item 3
- **Bolded Item 2**
- Item 3
- Item 4
```

This input is correctly parsed into nested `<ul>` and `<li>` HTML elements, preserving the markdown structure.

- **Enhanced List Handling**: The parser correctly distinguishes between sorted and unordered lists, applying appropriate syntax rules to each and enabling modifiers within list elements.

• Test Cases

Our project includes a comprehensive collection of test cases spread across three files, each of which is intended to properly check the markdown parser's functioning. The tests span from simple input-output checks to more advanced property-based tests.

- **ComprehensiveTests.hs:** The largest test suite, using **`QuickCheck`** for property-based testing. It creates markdown inputs dynamically and ensures that the parser outputs correct HTML for components such as bold, lists, tables, and hierarchical structures. This approach ensures wide coverage, including edge cases, and is ideal for detecting subtle issues that fixed tests might miss. It may be performed with other tests using stack test. This test case address the comprehensive testing.

- **TestCases.hs:** This file provides fixed input-output test cases that represent real-world markdown settings. It examines common markdown features like headers, links, and lists, as well as more complex combinations like footnotes and photos with captions. Each test compares the parser's HTML output to preset anticipated outputs to ensure that common markdown constructions are appropriately handled. While these cases are simpler than the QuickCheck-based tests, they guarantee that the parser operates correctly in predictable, real-world settings, giving confidence in its overall operation.

- **ParserTestCases.hs:** This file delves deeply into the parser's core processes. It employs fixed input-output tests to evaluate particular parsing components (such as `manyTill`, `nestedModifiers`, and `footNote`), with an emphasis on edge situations and error handling. These tests are designed to ensure that the parser can gracefully accept faulty or uncommon markdown. This file ensures that particular parser functions work properly and deliver consistent outputs under a variety of conditions, including improper markdown syntax.

Meeting the Rubric

These test files match the rubric's requirement for complete test cases that cover both the parser and pretty-printing. **`ComprehensiveTests.hs`**, with **`QuickCheck`** and property-based testing, is the most advanced, since it tests the parser against dynamically produced markdown inputs, assuring reliability across a wide range of possibly unexpected cases.

By combining property testing, predefined test cases, and edge case handling throughout various files, the test suite provides comprehensive coverage of the parser's capabilities, from the most basic

to the most complex. All tests, including **`ComprehensiveTests.hs`** and **`Spec.hs`**, may be run with stack test, guaranteeing seamless integration into the development process and comprehensive validation of the parser's accuracy and resilience.

Empty (Edge Cases - Optional Part)

In this part, we look at how the parser handles empty material, including modifiers and non-modifiers. According to the requirements, all empty modifiers are regarded as plain text, whereas non-modifiers react based on their structure.

Modifiers

- **Bold (`****`)**
    - **Result:** `<p>****</p>`
    - Treated as plain text, no bold formatting is applied.
- **Italic (`__`)**
    - **Result:** `<p>__</p>`
    - Underscores are displayed as plain text.
- **Strike-through (`~~~~`)**
    - **Result:** `<p>~~~~</p>`
    - Tildes are shown as plain text.
- **Inline Code (`` `` ``)**
    - **Result:** `<p>``</p>`
    - Backticks are displayed without code formatting.
- **Footnote (`[^0]`)**
    - **Result:** `<p>[^0]</p>`
    - Footnote reference remains as plain text.
- **Links**
    - **Empty Link Text (`[](a)`)**: `<p>[](a)</p>`
    - **Empty URL (`[a]()`)**: `<p>[a]()</p>`
    - **Both Empty (`[]()`)**: `<p>[]()</p>`
    - Links without text or URLs are treated as plain text.

Non-Modifiers

- **Block Code (`` ``` ``` ``)**
    - **Result:** `<pre><code></code></pre>`
    - Generates an empty code block.

- **Ordered List**
  ```
  1.
  2.
  3.
  ```
  - **Result:**
    ```
    <ol>
        <li></li>
        <li></li>
        <li></li>
    </ol>
    ```
  - Creates list items with no content.

- **Footnote Reference (`[^1]:`)**
  - **Result:** `<p id="fn1"></p>`
  - Creates an empty footnote reference.

- **Header (`#` or `==`)**
  - **Result:** <h1></h1>
  - **Creates an empty header without content.**

- **Block Quote (`>`)**
  - Result: `<blockquote></blockquote>`
  - **Generates an empty blockquote.**

- **Table**
  - Result: `<blockquote></blockquote>`
  - **Generates an empty blockquote.**

- **Image (![a](aweg ""))**
  - **Valid Image:** `<img src="aweg" alt="a" title="">`
  - **Invalid Image (`![ ](a "")`)**: Generates an image with an space alt tag, reducing accessibility.
  - **No URL (`![](  "")`)**: Plain Text