NAME: ER JUN YET **ID: 33521026**

FIT2102 - Programming Paradigms

Assignment 2: Markdown to HTML

Introduction

This assignment focuses on building a Markdown parser using functional programming principles in Haskell. The objective is to parse a simplified version of Markdown and convert it into a custom Abstract Data Type (ADT) representation, which can then be further converted into HTML or other formats. The main tasks involve parsing various Markdown elements such as headers, paragraphs, lists, block quotes, tables, and text formatting like bold or italic text. This involves utilizing parser combinators, a key concept in functional programming, to break down complex parsing problems into smaller, modular components.

Design of the Code

High-level Description

The intention of this project was to implement a Markdown parser using Haskell, which would convert Markdown documents into structured data using Algebraic Data Types (ADT). The design revolves around recursively parsing elements like paragraphs, headers, blockquotes, lists, and free text (text modifiers like bold and italics). The parser needs to handle both block-level elements (like headers or code blocks) and paragraph elements (like free text and plain text). The end goal is to structure this parsed document in a format that can later be converted to HTML or other formats.

Code Architecture Choices

The architecture relies heavily on defining clear data types to represent the different components of a Markdown document. Each part of a document—whether a header, paragraph, or blockquote—is represented by specific data structures (Document, Element, BlockElement, Paragraph and other more). This approach simplifies the parsing logic by breaking down complex Markdown structures into modular components.

The code is organized in such a way that each component has its own parser, which is composed into more complex parsers for the entire document. This structure promotes reusability and maintains clean separation between block-level elements and paragraph elements.

Usage of Parser Combinators

The project heavily leverages parser combinators, which enable the composition of small parsers to form more complex ones. For example, parsers like some, many, and <|> are fundamental in handling sequences and optional parsing branches. These combinators allow for handling varied Markdown syntax flexibly. For instance, many is used to parse multiple elements in a row, while <|> lets the parser handle alternative Markdown elements (like headers or blockquotes).

Parser combinators provide an elegant solution for breaking down the parsing tasks into smaller, manageable parts. By using combinators, the parser avoids explicitly dealing with complex state management or manual backtracking, as the combinator library does this efficiently.

Construction Using Applicative and Instances Typeclasses

The parser construction also leverages Haskell's typeclasses like Functor, Applicative and Monad. These typeclasses allow for function application within the parser context. For example, fmap (<$>) is used to apply a function to the result of a parser, such as in plainHeaderParser, where the parsed content is wrapped in a Header constructor. Similarly, the Applicative typeclass allows for sequential application, such as parsing the header level and then the content in sequence.

Small Modular Functions

One of the most significant aspects of functional programming is breaking down large problems into small, reusable, and composable functions. In this project, each Markdown element has its own parser function. This modularity ensures that each function is small and focused on a single task, for example boldParser only handles bold text, and paragraphElementParser only handles paragraph-level content. This modular design makes the code easier to maintain, test, and extend.

The beauty of this code lies in how these small modular parsers are composed to build more complex parsers. For instance, the elementParser combines blockElementParser and paragraphElementParser, which in turn combine smaller parsers. This compositional approach keeps the code declarative.

Declarative Style

The code avoids imperative constructs, opting instead for a declarative style. This is particularly evident in the use of combinators, which describe what to parse, rather than specifying low-level details of how to parse. Moreover, the point-free style used in some cases like elementParser further enhances readability by focusing on the function composition itself, rather than unnecessary boilerplate arguments.

Typeclasses and Custom Types

The use of custom types like Document, Element, and BlockElement helps structure the parsed data in a way that is easy to work with. These types, combined with Haskell's strong type system, prevent many classes of errors by ensuring that only valid Markdown structures are produced by the parser. Typeclasses like Functor, Applicative and Monad enable flexible composition and transformation of parsers.

Higher Order Functions

Haskell's higher-order functions, such as fmap ($) and apply (<*>) are used extensively to manage the parsing flow. These functions allow us to apply transformations to the results of parsers, compose them in sequence, and handle optional elements. The use of these functions ensures that the parsing logic remains clean, reusable, and functional in nature.

Function Composition

Function composition is a key feature of the design, allowing parsers to be combined in meaningful ways. For example, blockElementParser is composed of smaller parsers like plainHeaderParser and blockQuoteParser using the <|> combinator, allowing for a clear and concise description of possible Markdown block elements.

Description of Extensions

I aimed to implement a comprehensive Markdown parser that covers most Markdown features like headers, blockquotes, code blocks and text formatting (bold, italic, links). Additionally, I added support for more complex elements like tables and images.

The most interesting aspect of this implementation is how parser combinators abstract away much of the complexity. By breaking down the task into small, composable functions, the final result is both powerful and flexible.

In summary, this project highlights the power of functional programming and parser combinators in building concise, maintainable parsers. Through the careful composition of small functions, the code remains clean and flexible, making it a strong example of declarative programming in Haskell.