Markdown to HTML

BNF

```
<Char>                ::= (* one or more of any character except newline *)
<Number>              ::= ( '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' )+
<Text>                ::= <Char>+*
<OptText>             ::= { <Char> }
<Whitespace>          ::= ( '\t' | '\r' | '\f' | '\v' | ' ' )+
<OptWhitespace>       ::= { <Whitespace> }
<Indentation>         ::= { '    ' }
<Newline>             ::= '\n'

<Italic>              ::= '_' <FreeText> '_'
<Bold>                ::= '**' <FreeText> '**'
<Strikethrough>       ::= '~~' <FreeText> '~~'
<Link>                ::= '[' <FreeText> ']' '(' <Text> ')'
<InlineCode>          ::= '``' <Text> '``'
<Footnote>            ::= '[^' <Number> ']'
<Modifiers>           ::= ( <Italic> | <Bold> | <Strikethrough> |
                            <Link> | <InlineCode> | <Footnote> )+

<Image>               ::= '![' <Text> '](' <URL> <Whitespace> '"' <Text> '")'
<FootnoteReference>   ::= <Footnote> ':' <OptWhitespace> <Text>
<FreeText>            ::= <Text> | <Modifiers>
<Heading1>            ::= '#' <Whitespace> <FreeText>
<Heading2>            ::= '##' <Whitespace> <FreeText>
<Heading3>            ::= '###' <Whitespace> <FreeText>
<Heading4>            ::= '####' <Whitespace> <FreeText>
<Heading5>            ::= '#####' <Whitespace> <FreeText>
<Heading6>            ::= '######' <Whitespace> <FreeText>
<AltHeading1>         ::= <FreeText> '\n' <OptWhitespace> '==' { '=' } <OptWhitespace> <EOL>
<AltHeading2>         ::= <FreeText> '\n' <OptWhitespace> '--' { '-' } <OptWhitespace> <EOL>
<Heading>             ::= <Heading1> | <Heading2> | <Heading3> | <Heading4> |
                          <Heading5> | <Heading6> | <AltHeading1> | <AltHeading2>
<Blockquote>          ::= '>' <OptWhitespace> <FreeText>
<Code>                ::= '```' <OptText> '\n' <Text> '\n```' <EOL>
<ListItem>            ::= <Indentation> <Number> '.' <Whitespace> <FreeText> <EOL>
<OrderedList>         ::= <ListItem> | <ListItem> <OrderedList>
<TableCell>           ::= <FreeText>
<TableRow>            ::= <TableCell>+ <EOL>
<TableBody>           ::= <TableRow> | <TableRow> <TableBody>
<TableHeader>         ::= <TableRow>
<Table>               ::= <TableHeader> <TableBody>
<Markdown>            ::= <Newline> | <Image> | <FootnoteReference> | <Heading> |
                          <Blockquote> | <Code> | <OrderedList> | <Table> | <FreeText>
```

Design of the Code

The purpose of the code is to parse simplified Markdown and convert it into HTML. It leverages an ADT to represent different Markdown elements. This ADT provides a clear structure, making it easy to handle complex Markdown constructs and convert them into HTML.

The code is divided into two main parts: the parser and the renderer. The parser transforms Markdown input into the ADT, while the renderer converts the ADT into HTML. This separation of concerns keeps the parsing and rendering processes modular and maintainable. The parser uses modular parsers and parser combinators to break down the Markdown syntax into smaller components, with each parser handling a specific Markdown feature.

In addition to the core parsing and rendering functionality, the code also demonstrates Haskell's capability to interact with external services and perform I/O operations by sending POST requests to the Haskell backend to render and save the HTML.

Parsing

The code heavily relies on parser combinators to compose small parsers into more complex ones, providing both flexibility and reusability. For instance, the `lookAhead` parser is used to execute a parser without consuming input, which is particularly helpful in cases like the `table` parser, where trailing whitespace needs to be checked without being consumed.

Parser combinators like `manyTill`, `someTill`, and `sepBy1` are used extensively to reduce redundancy and compose parsers for more complex elements. For example, `sepBy1` is used to handle table rows, while `someTill` allows different parsers to be attempted until a specific one succeeds, resulting in more flexible parsing logic. These combinators simplify the handling of complex Markdown elements like tables and lists, contributing to cleaner and more maintainable code.

Functional Programming

The functional programming paradigm emphasizes small, modular functions that can be composed together, a principle central to this code. Each parser is designed to handle one aspect of Markdown and is combined with other parsers to form a complete solution. This results in a declarative style that makes the code readable and easy to maintain.

Haskell Language Features Used

The code extensively leverages polymorphism within the ADT types, particularly for handling nested text modifiers. The ADT design allows various Markdown elements to be treated uniformly, enabling flexible composition and extension. For example, the `nestedBetween` function uses polymorphism to handle any modifiers. This demonstrates how ADTs and polymorphism make the code more scalable and modular.

Functor, Applicative, and Monad typeclasses are used throughout the parsing logic. For instance, the `fmap` function is extensively used to apply ADT constructors to parsed content. The Applicative typeclass, through functions like `liftA2`, combines results from multiple parsing steps. In the `link` parser, `liftA2` is used to parse both the link text and the URL, and it lifts the `Link` ADT constructor into the combination, ensuring that both the parsed text and URL are wrapped together in a `Link` ADT.

The Monad instance enables sequential parsing, where one parser's result influences the next, as seen in the `table` parser. Using monadic bind, the parsing process can depend on prior results. For example, after parsing the row, it's passed to `checkNCol` using bind, ensuring each subsequent row has the correct number of columns. The `do` notation also simplifies the chaining of operations, keeping complex parsing logic readable and maintainable.

Finally, point-free style is employed where possible, reducing verbosity by eliminating unnecessary parameters. This is evident in parsers like `newline`, which uses function

composition to construct the result without explicitly referring to the input. This approach keeps the code concise and emphasizes the declarative nature of Haskell.

Description of Extension

The extension I implemented are nested text modifiers, which handle cases where Markdown elements like bold, italic, or strikethrough are nested inside each other. The challenge here was to ensure that closing tags are properly identified without incorrectly parsing incomplete closing tags and correctly parsing opening tags which are not immediately followed by a closing tag.

To overcome the opening tag issue, I had to use `lookAhead` to check if the next character is not part of the opening tag after parsing the opening tag. This ensures that we do not parse empty text modifiers. For the closing tag issue, I repeatedly try all the text modifiers to cover nested cases, then if that fails, `nested` is run which parses character by character as long as the closing tag is not found, and this whole process is repeated until the closing tag is found. This approach ensures that nested text modifiers are correctly parsed while handling edge cases related to incomplete tags.