Markdown to HTML Converter

For my project I have created a program that takes a markdown file (.md) as input and outputs an HTML file (.html) with the markdown language converted into HTML language. I wrote this program using Haskell using a bottom-up parsing technique. To use the program you either need to compile the Haskell file and run it or use the Haskell interpreter to run the main function. The program will ask for a file as input without any extensions as it will assume it is a .md extension. The program will then output a .html file containing the corresponding HTML code to the current directory.

My program is organized in a similar fashion to the IMP interpreter that we built in class. Firstly, it goes through the text and preprocesses characters to make sure that the spacing of the text is what is intended. Next the program classifies special characters within the text into their corresponding Exp datatype constructor. Newlines and tabs are included in the special characters since they are important to how markdown language is interpreted. Once the spacing and special characters have been identified the next step is to use the shift-reducer to combine items into the appropriate Exp type constructors and to verify that a legal parse exists for the given input. If a legal parse is not found then the program will show the incomplete parse. Finally, if a complete parse of the input is found the output will be written to the appropriate file and the complete parse will be shown. The main function that directs the program is shown here along with the other important functions in the program:

```
main :: IO ()
main = do
   putStrLn "Enter file name without extension: "
   filename <- getLine
   contents <- readFile (filename ++ ".md")
   let analyzed = lexer contents
   let parsed = parser analyzed
   let html = convert parsed
   writeFile (filename ++ ".html") html
   putStrLn (show parsed)
```

classify :: String -> Exp

If a special character is identified convert it to the appropriate Exp type representation otherwise use the Word constructor to create an Exp type.

sr :: [Exp] -> [Exp] -> [Exp]

Pattern match for special characters and patterns that indicate the need for an HTML tag, create a Tag type and convert it to Exp using the PA constructor.

lexer :: String -> [Exp]

Map the classify function to each word after being preprocessed and return a list of Exp types.

parser :: [Exp] -> HTML

Takes a list of Exp types and verifies a complete parse has occurred then types the output under the HTML type.

preproc :: String -> String

Goes through the initial input and modifies spacing to prepare the input for parsing.

This project had some hidden difficulties which made it far more challenging than I had anticipated. The main obstacle of creating this program was by far accounting for nested objects. For example inside of an ordered list tag you may have several list items and inside of those you may have several other tags. Because of this you can't simply parse one item at a time or else the nesting of tags will be off. The list tags themselves were particularly challenging because of the fact that lists can be nested within lists as well as other items. Delimiting lists and some other tags such as paragraphs is also a tricky because there is no clear indication character of when these tags are supposed to end so you have to depend on line breaks as well as the context of surrounding items.

To overcome the challenge of delimiting tags I classified all of the line breaks and tabs as special characters and left them in the input. Then I used Haskell's pattern matching to look for the specific patterns that exist within the markdown language. The tabs are absolutely necessary and without them it would be impossible to differentiate nested items within lists. Spaces can also play a role in converting markdown for example in line break tags where two spaces followed by a newline signifies a line break tag.

Another technique that I used was creating a Tag datatype as well as an Exp datatype. The Tag datatype is meant to represent the possible tags that can be represented in this markdown language converter. All Tag constructors except for BR (line breaks), and HR (horizontal rules) contain a list of Exp types so that Tags can be combined with other Tags and special characters. It was very useful to be able to represent each possible Tag with the ability to nest other tags, text, and characters inside of the Tag constructor.

I believe that there are definitely some areas to improve on this project such as extending it to more possible HTML tags and improving the format of the code itself. The extent of this project was limited to a certain number of tags but I do believe it would be possible to extend it to other HTML tags such as links and images. As for the code itself after implementing this program I did try out a few different ideas of parsing nested items to see if I could find a simpler solution. One idea that seems promising is to have a shift-reduce function for all the kinds of tags that can be nested. I think if done right this could make delimiting nested tags easier. The way it would work is every time the main shift-reduce function detected a tag it would recursively call itself with the output of the specific tag shift-reducer appended to the stack. For example the following line shows this approach to parsing ordered lists:

```
sr (Word “1.” : stack) input = sr (NewLine : NewLine : head out : stack) (tail out)
        where out = srOL [Word “1.”] input
```

I enjoyed making this program although it was difficult and frustrating at times. I feel like it is something that could possibly be used in the future given it was polished a bit more. The project did prove to be more difficult than I had initially anticipated but I am proud of how much I accomplished with it in the time that I had. I feel like doing this has really improved my understanding of how language can be interpreted by computers. Something that I had not given much thought before was the possibility of using the space and newline characters in a text file to give different meaning to a language. I feel like it is an entire aspect of languages that is often overlooked because we as humans are accustomed to ignoring it.

Bibliography

Cone, Matt. “Basic Syntax.” *Markdown Guide*, 2021, www.markdownguide.org/basic-syntax/#horizontal-rules.