Cameron Himes and Chase Faine

Dr. Andrew Polonsky

C S 3490

December 4, 2021

Final Report: Markdown to HTML Converter

Description

This program converts Markdown files to HTML webpage. The user makes a standard Markdown file and then calls the converter. First, the code is compiled using `ghc converter.hs -o converter` . Then the user invokes the program using a terminal. An example would be `./converter myFile.md myFile.html` where `myFile.md` is the user's Markdown document and `myFile.html` is where to store the generated HTML code.

Organization

Overview

This code is designed to be ran in a sequence. First, the arguments are read and the file paths are set. The input file is then read into the lexer which converts the file to tokens. Next, the parser generates a document object providing an HTML-like representation of the document. The object is fed to our generateHTML function which outputs an HTML file as a single string. Lastly, this string is written to the output file and the program terminates.

Main function

Note: The "from source X" comments will be explained in the research section.

```haskell
main :: IO ()

main = do

  -- get file paths

  args <- getArgs -- get system args [from source 1]

  let (infile, outfile) = getFiles args -- get file names

  putStrLn ("Input: " ++ infile) -- show input file name

  putStrLn ("Output: " ++ outfile) -- show output file name

  -- read the file

  inHandle <- openFile infile ReadMode -- [from source 3]

  mdText <- hGetContents inHandle -- [from source 3]

  putStrLn "=== INPUT CONTENTS ==="

  print mdText

  -- run the lexer

  let lexed = lexer mdText

  putStrLn "=== LEXED TOKENS ==="

  print lexed

  -- run the parser

  let parsed = parser lexed

  putStrLn "=== PARSED TOKENS ==="

  print parsed
```

```
  -- run the converter
  let html = generateHTML parsed
  putStrLn "=== HTML CODE ==="
  print html
  -- write the html file
  outHandle <- openFile outfile WriteMode -- [inferred from source 3]
  hPutStrLn outHandle html -- [inferred from source 3]
  -- close handles
  hClose inHandle -- [from source 3]
  hClose outHandle -- [from source 3]
  -- end the program
  print "OK. :)" -- inform user that program is done
```

Other important functions

```
getFiles :: [String] -> (String, String)
```

This gets the names of the input and output files and will generate errors if anything is incorrect.

```
lexer :: String -> [Token]
```

This converts an entire file into a list of tokens.

```
parser :: [Token] -> [Block]
```

This converts the entire page from a list of tokens to a list of block elements.

```
sr :: [Token] -> [Token] -> Block
```

This function takes in a list of tokens for a single block element and an empty stack and outputs a single block element.

```
generateHTML :: [Block] -> String
```

This converts a list of block elements into a string representing an entire HTML webpage.

Research

Our first challenge was getting arguments from the command line. This was not discussed during our class sessions and was not obvious like in other programming languages. A simple Google search brought us to the official Haskell Wiki[1]. We used this source inside our `main` function for pulling the arguments and passing them to a helper function to extract the file names and ensure the correct number of arguments was provided.

When implementing our own version of Haskell's `words` function, we could not figure out how to split words at a space. We decided to search for a solution after several failed attempts at solving it ourselves. We found a Stack Overflow answer[2] which recommended using the `span` function to perform splitting at a specific character.

While most of main was a simple task, we ran across another issue that was never discussed in class. We had no idea how to read and write from a file. We finally stumbled across a Stack Overflow answer[3] which provided example code for opening a file and reading its contents. Using the function suggestions feature in Visual Studio Code, we found

other function in the same library for writing to a file. This example code was used in our import statements and was modified in our `main` function.

We added citations in our code for all of these above sources. Just look for code comments which say `-- [from source X]` to determine which lines were copied from a source. All sources are neatly numbered at the top of our code file and have a direct link to the webpage being referenced.

Of course, Markdown is a complex system and neither one of us knew all the rules which could be added. All of our Markdown rules are taken from GitHub Flavored Markdown (GFM). For this reason, we used the formal specification[4] and the official documentation[5] to model our formatting rules.

Discussion

Most of this project was very easy. The basic steps outlined in our overview were clear from the beginning. Our first issue was implementing Haskell's `words` function in a way that newlines and tabs would be preserved. We found a great Stack Overflow discussion[2] that described the general algorithm and more details are discussed in the research section. The general idea is to split strings at only the space character and do this recursively to generate a list. This code can be seen in our `splitAtWords` function.

Another issue was generating text. We had some issues with generic text not being promoted to normal text. This is caused by other inline blocks such as bold text being closed by special tokens but normal text is detected by the absence of tokens. This was fixed by

adding more parser rules to “merge” extra text elements into a paragraph block as normal text elements. This code can be seen in our `sr` function.

The biggest challenge of all was nesting lists of different types. This challenge was solved using a divide and conquer method. We implemented single ordered lists and then nested ordered lists. Once this was working, we made ists a generic data type with an attribute to define if it was ordered or unordered. This abstraction allowed us to have ordered and unordered lists be nested inside one another without additional parser rules. This code can be seen in our `sr` function.

Comments

The biggest issue with our project is the parser. We used a shift-reduce parser like many of the homework and lecture programs. While it works for most things, validating ambiguous input is not possible. One such example is the use of a dash (`-`). Does a dash mark an unordered list item or is it being used in a hyphenated word? A shift-reduce parser can not tell the difference. This lead to some interesting constraints such as not having any dashes or braces in the input unless they are used for unordered lists or links respectively. If we had more time for research, a GLR parser would be the better approach. A GLR parser would allow us to backtrack on a bad parse and demote these tokens to normal characters if they did not fit a particular syntax rule.

Conclusion

In conclusion, the project itself went well enough but we did run into some serious issues regarding the shift-reduce parser. At first, everything went smoothly adding in the data types, tokens, markdown documents, etc. The first real challenge we approached before anything else was getting arguments from the command line, but after researching a bit we were able to figure it out. After this, many problems, challenges, and hours of debugging came about, which mainly appeared in the parser sections. Some of these issues included writing a function where newlines and tabs would be preserved, generating text, and nesting lists of different types. Again the biggest issue with this project is the parser, and one of the most important things we learned in this project is just how limiting shift reduction is. Any specific character used in a rule in the parser, could not be wrapped into another rule. For example, we first came across this problem when we were building the rules for hyperlinks, and most links include the dash characters in them, but we quickly learned that the parser cannot tell the difference between the dash in the hyperlink, and the dash that starts an unordered list. Including a dash in the link just completely breaks the program. The same is true for any other characters used in parsing rules. No matter what we tried, we could not figure out a way around this issue, so we assume this is an issue with shift reducing itself. If we were to do this program again, we would try to find a new way of parsing.

References

[1] https://wiki.haskell.org/Tutorials/Programming_Haskell/Argument_handling

[2] https://stackoverflow.com/a/20482547

[3] https://stackoverflow.com/a/7867786

[4] https://github.github.com/gfm/

[5] https://docs.github.com/en/github/writing-on-github/getting-started-with-writing-and-formatting-on-github/basic-writing-and-formatting-syntax