Cameron Himes and Chase Faine

Dr. Andrew Polonsky

C S 3490

November 10, 2021

# Project Description

The basis of this project is simple, to convert markdown into HTML. The reason a program like this is important, or why somebody may want to use it, is that markdown is ultimately easier to write and easier to read than HTML. In this case, using such a tool could be beneficial to save time and energy spent by web developers.

The user would interact with this program by invoking the program with `converter <input> <output>`. In the example `converter file.md file.html`, `file.md` represents the input file and`file.html` represents the output file. The program then parses the input file and writes the resulting HTML code in the output file. For example, in the markdown file, we may have the code `# Heading 1`, but once the code is converted into an HTML file, the resulting conversion would look like `<h1>Heading 1</h1>`. Another example would be if we had the markdown code `**Bold Letters**`, the resulting HTML code would appear as `<b>Bold Letters</b>`. After the program has finished executing and successfully converted the markdown into HTML, the new HTML file will be downloaded to the user's computer. If the program's execution is not successful due to any reason, an error message will be displayed instead.

The internal data types are very simple. HTML and Markdown are both text markup languages. This means they both do the same thing: represent a formatted document to the user. Since they are both used for the same purpose, they have many elements in common. The only realistic difference is how that document structure is generated. In HTML, there are two types of elements called `block` level elements and `inline` level elements. All inline elements exist inside block level elements. A webpage is made up of one or more block level elements. For this reason, our main structure called `document` will simply be a list of block types. Our `block` type will contain representations of HTML's typical block elements like ordered lists, unordered lists, paragraphs, and code blocks. Each block element can contain any number of inline elements, such as unformatted text, inline code, bold or italic text, links, and other items typically found in a paragraph. We will use a parser and lexer similar to our homework and in class assignments. Our lexer will detect syntax symbols and convert them into tokens. We will then use a parser to reconstruct the document from the lexer output.

Our actual implementation will be very simple. Starting in `main`, it will call a function called `getArgs` to determine the input and output files and also handle any errors that may occur. Main will then read the file and call `lexer` and `parser` to generate the `document` data type defined in the previous paragraph. Once the internal data representation is obtained, a function `printHTML` will take in a document object and return a `string` with the HTML code. The string is then written to a file inside the main function and the program exits.

# Example Implementation and Psudocode

```
-- data types

type Text = String

data Document = [Block]

data Inline = Normal Text         -- normal text

            | Bold Text           -- bold text

            | Italic Text         -- italic text

            | Preformatted Text -- inline code

data Block = UL [Inline]          -- Unordered List

           | OL [Inline]          -- Ordered List

           | Paragraph [Inline] -- Paragraph

           | Code [Text]          -- Code block

data Token = Star                 -- used for bold/italic

           | Dash                 -- used for unordered lists

           | BackTick             -- used for code/preformatted text

           | NewLine              -- used to shorten lines or end a block

           | B Block              -- preparsed Block type

           | I Inline             -- preparsed Inline type

           | T Text               -- preparsed Text type


-- function definitions

getArgs :: IO [String] -- this is defined in the System.Environment package
```

```haskell
getFiles :: IO [String] -> [String, String]

lexer :: String -> [Token]

parser :: [Tokens] -> [Block]

printHTML :: [Block] -> String]


-- pseudocode for main

main :: IO ()

main = do

  let infile outfile = getFiles getArgs

  let mdText = read infile

  let lexed = lexer mdText

  let parsed = parser lexed

  let html = printHTML parsed

  write html outfile

  print "OK."
```