

BNF.md

```
AssignmentLhs M  BNF.md x MarkdownParser.hs M Main.hs TS main.ts M TS types.ts style.css
Haskell > src > BNF.md
1 Your job is to write the `<Array>` and `<Object>` sections of the following BNF. It should look something like the non-ter
  `<String>`.
2
3 ```BNF
4 <Document> ::= <DocumentElements> | <DocumentElements> <Document>
5 <DocumentElements> ::= <BlockElement> | <InlineElement> | <ParagraphElement>
6
7 // DocumentElements
8 <BlockElement> ::= <BlockQuote> | <Header> | <OrderedListWrap> | <Table>
9 <InlineElement> ::= <Image> | <Code> | <FootNoteReference>
10 <ParagraphElement> ::= <Paragraph>
11
12 // ParagraphElement
13 <Paragraph> ::= <TextModifier> <NewLine> | <TextModifier> <Paragraph>
14 <TextModifier> ::= <Italic> | <Bold> | <Strikethrough> | <Link> | <InlineCode> | <FootNote> | <PlainText>
15
16 // Text Modifiers
17 <FootNote> ::= "[^" <Integer> "]"
18 <Italic> ::= "_" <PlainText> "_"
19 <Bold> ::= "***" <PlainText> "***"
20 <Strikethrough> ::= "~" <PlainText> "~"
21 <Link> ::= "[" <PlainText> "]" "(" <PlainText> ")"
22 <InlineCode> ::= "`" <PlainText> "`"
23 <PlainText> ::= <String> | <Space> <PlainText>
24
25 // InlineElement
26 <Image> ::= "![ " <PlainText> "]" "(" <PlainText> <Space> <PlainText> ")" <NewLine>
27 <FootNoteReference> ::= <FootNote> ":" <Space> <PlainText>
28 <Code> ::= "```" <Text> <NewLine> "```"
29 <Text> ::= <PlainText> | <NewLine> <Text>
30
31 // BlockElement
32 <BlockQuote> ::= ">" <Paragraph> | <Paragraph> <BlockQuote>
33
34 <Header> ::= <NormalHeader> | <AltHeader>
35 <NormalHeader> ::= <Hashes> <Space> <TextModifier>
36 <AltHeader> ::= <AltHeader1> | <AltHeader2>
```

Assignment.hs

```
Assignment.hs M X BNF.md MarkdownParser.hs M Parser.hs blockquotes.html M heading.html M
Haskell > src > Assignment.hs > Assignment > convertDocumentHTML

4
5 import Data.Time.Clock (getCurrentTime)
6 import Data.Time.Format (defaultTimeLocale, formatTime)
7 import Instances (Parser (..))

8 import Parser
9 import MarkdownParser
10 import Control.Applicative (Alternative(some))
11 import Control.Applicative
12
13 writeContents :: String -> IO ()
14 writeContents contents = do
15     currentTime <- getTime
16     writeFile (currentTime ++ ".html") contents
17
18 getTime :: IO String
19 getTime = formatTime defaultTimeLocale "%Y-%m-%d %H:%M:%S" <$> getCurrentTime
20
21 parseDocument :: Parser Documents
22 parseDocument = Document . trimNewline <$> some parseDocumentElement <" eof"
23
24 -- Convert TextModifier to HTML
25 convertTextModifierHTML :: TextModifier -> String
26 convertTextModifierHTML (PlainText s) = s
27 convertTextModifierHTML (ItalicText s) = "<em>" ++ s ++ "</em>"
28 convertTextModifierHTML (BoldText s) = "<strong>" ++ s ++ "</strong>"
29 convertTextModifierHTML (StrikeThroughText s) = "<del>" ++ s ++ "</del>"
30 convertTextModifierHTML (Link text url) = "<a href=\"" ++ url ++ "\">" ++ text ++ "</a>"
31 convertTextModifierHTML (InlineCode s) = "<code>" ++ s ++ "</code>"
32 convertTextModifierHTML (Footnote n) = "<sup><a id=\"" ++ n ++ "\" href \"" ++ n ++ "\">" ++ n ++ "</a></sup>"
33
34 -- Convert Paragraphs to HTML
35 convertParagraphHTML :: Int -> Paragraphs -> String
36 convertParagraphHTML indentLevel (Paragraph elements) = indent indentLevel ("<p>" ++ concatMap convertTextModifierHTML elements ++ "</p>")
37
38 -- Convert InlineElements to HTML
39 convertInlineElementsHTML :: Int -> InlineElements -> String
40 convertInlineElementsHTML indentLevel (Image alt url title) = indent indentLevel ("<img src=\"" ++ url ++ "\" alt=\"" ++ alt ++ "\" title=\"" ++ title ++ "\">")
41 convertInlineElementsHTML indentLevel (Code lang content) = indentCode indentLevel ("<pre><code class=\"" ++ lang ++ "\">" ++ content ++ "</code></pre>")
42 convertInlineElementsHTML indentLevel (FootnoteReference n content) = indent indentLevel ("<p id=\"" ++ n ++ "\">" ++ content ++ "</p>")
43
44 -- Convert BlockElements to HTML
45 convertBlockElementsHTML :: Int -> BlockElements -> String
46 convertBlockElementsHTML indentLevel (BlockQuote paragraphs) = indent indentLevel "<blockquote>\n" ++ concatMap (convertParagraphHTML (indentLevel + 4)) paragraphs ++ "\n"
47 convertBlockElementsHTML indentLevel (Header level elements) = indent indentLevel ("<h" ++ level ++ ">" ++ concatMap convertTextModifierHTML elements ++ "</h" ++ level ++ ">")
48 convertBlockElementsHTML indentLevel (OrderedList items) = convertOrderedListHTML indentLevel items
49 convertBlockElementsHTML indentLevel (Table (header:rows)) = convertTableHTML indentLevel (header, rows)
50
51 convertTableHTML :: Int -> (TableHeader, TableBody) -> String
52 convertTableHTML indentLevel (header, rows) = indent indentLevel "<table>\n" ++
53     indent (indentLevel + 4) "<thead>\n" ++
54     convertTableRowHTML (indentLevel + 8) header ++
55     indent (indentLevel + 4) "</thead>\n" ++
56     indent (indentLevel + 4) "<tbody>\n" ++
57     concatMap (convertTableRowHTML (indentLevel + 8)) rows ++
58     indent (indentLevel + 4) "</tbody>\n" ++
59     indent indentLevel "</table>\n"
60
61 convertTableRowHTML :: Int -> TableRow -> String
62 convertTableRowHTML indentLevel (TableRow cells) = indent indentLevel "<tr>\n" ++ concatMap (convertTableCellHTML (indentLevel + 4)) cells ++ "\n"
63
64 convertTableCellHTML :: Int -> TableCell -> String
65 convertTableCellHTML indentLevel (TableCell elements) = indent indentLevel "<td>" ++ concatMap convertTextModifierHTML elements ++ "</td>\n"
66 convertTableCellHTML indentLevel (TableHeader elements) = indent indentLevel "<th>" ++ concatMap convertTextModifierHTML elements ++ "</th>\n"
67
68 -- Convert DocumentElement to HTML
69 convertDocumentElementHTML :: Int -> DocumentElement -> String
70 convertDocumentElementHTML indentLevel (BlockElement blockElem) = convertBlockElementsHTML indentLevel blockElem
71 convertDocumentElementHTML indentLevel (InlineElement inlineElem) = convertInlineElementsHTML indentLevel inlineElem
72
73 -- Convert Document to HTML
74 convertDocumentHTML :: Document -> String
75 convertDocumentHTML (Document elements) = concatMap (convertDocumentElementHTML 0) elements
```



```

Assignment.hs M X BNFAnd MarkdownParser.hs M Main.hs TS main.ts M TS types.ts # style.css
Haskell > src > Assignment.hs > Assignment > convertDocumentHTML
64 convertOrderedListitemHTML indentLevel (OrderedListitem elements sublist) =
65   if null sublist
66   then indent indentLevel ("<li>" ++ concatMap convertTextModifierHTML elements ++ "</li>\n")
67   else indent indentLevel ("<li>" ++ concatMap convertTextModifierHTML elements ++ "\n" ++ concatMap (convertOrderedListHTML
68
69 -- Convert TableRow and TableCell to HTML
70 convertTableRowHTML :: Int -> TableRow -> String
71 convertTableRowHTML indentLevel (TableRow cells) = indent indentLevel "<tr>\n" ++ concatMap (convertTableCellHTML (indentLevel
72
73 convertTableCellHTML :: Int -> TableCell -> String
74 convertTableCellHTML indentLevel (TableCell elements) = indent indentLevel ("<td>" ++ concatMap convertTextModifierHTML element
75 convertTableCellHTML indentLevel (TableHeader elements) = indent indentLevel ("<th>" ++ concatMap convertTextModifierHTML element
76
77 -- Convert DocumentElement to HTML
78 convertDocumentElementHTML :: Int -> DocumentElements -> String
79 convertDocumentElementHTML indentLevel (BlockElement blockElem) = convertBlockElementsHTML indentLevel blockElem
80 convertDocumentElementHTML indentLevel (InlineElement inlineElem) = convertInlineElementsHTML indentLevel inlineElem
81 convertDocumentElementHTML indentLevel (ParagraphElement paragraph) = convertParagraphHTML indentLevel paragraph
82
83 -- Convert entire Document to HTML
84 convertDocumentHTML :: Documents -> String
85 convertDocumentHTML (Document elements) =
86   "<!DOCTYPE html>\n" ++
87   "<html lang='en'>\n" ++
88   "\n<head>\n    <meta charset='UTF 8'>\n    <title>Test</title>\n</head>\n" ++
89   "\n<body>\n" ++ concatMap (convertDocumentElementHTML 4) elements ++ "</body>\n" ++
90   "\n</html>\n"
91
92 -- Helper function to indent lines manually by adding spaces
93 indent :: Int -> String -> String
94 indent n = concatMap (\line -> replicate n ' ' ++ line ++ "\n") . lines
95
96 indentCode :: Int -> String -> String
97 indentCode n = concatMap (\line -> replicate n ' ' ++ line) . lines
98

```

MarkdownParser.hs

```
Assignment.hs M  BNF.md  MarkdownParser.hs M X
Haskell > src > MarkdownParser.hs > MarkdownParser > BlockElem
15 | BoldText String
16 | StrikeThroughText String
17 | Link String String
18 | InLineCode String
19 | FootNote String
20 deriving (Show, Eq)
21
22 data Paragraphs
23   = Paragraph [TextModifier]
24   deriving (Show, Eq)
25
26 data BlockElements
27   = BlockQuote [Paragraphs]
28   | Header String [TextModifier]
29   | OrderedListWrap OrderedLists
30   | Table [TableRow]
31   deriving (Show, Eq)
32
33 data InlineElements
34   = Image String String String
35   | Code String String
36   | FootNoteReference String String
37   deriving (Show, Eq)
38
39 data DocumentElements
40   = BlockElement BlockElements
41   | InlineElement InlineElements
42   | ParagraphElement Paragraphs
43   deriving (Show, Eq)
44
45 data Documents
46   = Document [DocumentElements]
47   deriving (Show, Eq)
48
49 data OrderedLists
50   = OrderedList [OrderedListItem]
51   deriving (Show, Eq)
```

```
49 data OrderedLists
50   = OrderedList [OrderedListItem]
51   deriving (Show, Eq)
52
53 data OrderedListItem
54   = OrderedListItem [TextModifier] [OrderedLists]
55   deriving (Show, Eq)
56
57 data TableRow
58   = TableRow [TableCell]
59   deriving (Show, Eq)
```

```
Assignment.hs M  BNF.md  MarkdownParser.hs M X  blockquote
Haskell > src > MarkdownParser.hs > MarkdownParser > parseTextModifier
60
61 data TableCell
62   = TableCell [TextModifier]
63   | TableHeader [TextModifier]
64   deriving (Show, Eq)
65
```

```

Assignment.hs M  BNF.md  MarkdownParser.hs M X  blockquotes.html M  heading.html M
Haskell > src > MarkdownParser.hs > MarkdownParser > parseDocumentElement
62  = [TableHeader [TextModifier]]
63  | TableHeader [TextModifier]
64  deriving (Show, Eq)
65
66  parseTextModifier :: Parser TextModifier
67  parseTextModifier = parseItalic
68  <|> parseBold
69  <|> parseStrikethrough
70  <|> parseLink
71  <|> parseInlineCode
72  <|> parseFootnote
73  <|> parsePlainText
74
75
76  parseHeader :: Parser BlockElements
77  parseHeader = parseAltHeader
78  <|> parseNormalHeader
79
80  parseBlockElements :: Parser BlockElements
81  parseBlockElements = parseHeader
82  <|> parseOrderedListWrap
83  <|> parseTable
84  <|> parseBlockQuote
85
86  parseInlineElements :: Parser InlineElements
87  parseInlineElements = parseImage
88  <|> parseCode
89  <|> parseFootnoteReference
90
91  parseDocumentElement :: Parser DocumentElements
92  parseDocumentElement = BlockElement <$> parseBlockElements
93  <|> InlineElement <$> parseInlineElements
94  <|> ParagraphElement <$> parseParagraph
95
96  parseParagraph :: Parser Paragraphs
97  parseParagraph = Paragraph <$> many (parseTextModifier <|> parsePlainTextDelimiter) <*> is '\n'
98
99  -- ===== Text Modifiers =====
100
101  parsePlainText :: Parser TextModifier
102  parsePlainText = plainText <$> some (noneof "\n_~[>()'!"")
103
104  parsePlainTextDelimiter :: Parser TextModifier
105  parsePlainTextDelimiter = do
106  c <- oneof "_~[>()'!"
107  return (PlainText [c])
108
109  parseTextModifierDelimiter :: Parser TextModifier
110  parseTextModifierDelimiter = do
111  c <- oneof "_~[>()'"
112  return (PlainText [c])
113
114  parseItalic :: Parser TextModifier
115  parseItalic = do
116  _ <- string "_"
117  content <- some (noneof "\n")
118  _ <- string "_"
119  return $ ItalicText content
120

```



```

121 parseBold :: Parser TextModifier
122 parseBold = do |
123     _ <- string "==="
124     content <- some (noneof "\n*" <|> notPrefixOf "===")
125     _ <- string "==="
126     return $ BoldText content
127
128
129 parseStrikethrough :: Parser TextModifier
130 parseStrikethrough = do
131     _ <- string "~~~"
132     content <- some (noneof "\n~" <|> notPrefixOf "~~~")
133     _ <- string "~~~"
134     return $ StrikethroughText content
135
136 parseLink :: Parser TextModifier
137 parseLink = Link <$> linkText <*> linkUrl
138 where

```

```

Assignment.hs M  BNF.md  MarkdownParser.hs M
Haskell > src > MarkdownParser.hs > MarkdownParser > parsePlainText
263     _ <- parseTableHeaderSeparator expectedColumns
264     rows <- many (parseTableRow expectedColumns)
265     return $ Table (header : rows)
266
267 parseTableHeaderRow :: Parser TableRow
268 parseTableHeaderRow = do
269     _ <- is '|'
270     cells <- some parseTableHeaderCell
271     _ <- is '\n'
272     return $ TableRow cells
273
274 parseTableHeaderCell :: Parser TableCell
275 parseTableHeaderCell = do
276     _ <- inlineSpace
277     content <- some (parseTextModifier <|> parseTextModifierDelimiter)
278     let trimmedContent = trimFirstItemAfterReverse $ reverse content
279     _ <- is '|'
280     return (TableHeader trimmedContent)
281
282 parseTableHeaderSeparator :: Int -> Parser TextModifier
283 parseTableHeaderSeparator len = do
284     _ <- is '|'
285     cells <- some parseTableHeaderSeparatorCell
286     _ <- is '\n'
287     if length cells /= len
288     then empty
289     else return $ PlainText ""
290
291 parseTableHeaderSeparatorCell :: Parser TextModifier
292 parseTableHeaderSeparatorCell = do
293     _ <- inlineSpace
294     _ <- string "..."
295     _ <- many (is '.')
296     _ <- inlineSpace
297     _ <- is '|'
298     return $ PlainText ""

```

```

Assignment.hs M  BNF.md  MarkdownParser.hs M X  blockquotes.html M  head
Haskell > src > MarkdownParser.hs > MarkdownParser > parsePlainText
189 level <- length <$> some (is '#')
190 if level > 6
191   then empty
192   else do
193     _ <- inlineSpace1
194     content <- some (parseTextModifier <|> parsePlainTextDelimiter) <# is '\n'
195     return (Header (show level) content)
196
197 parseAltHeader :: Parser BlockElements
198 parseAltHeader = do
199   _ <- inlineSpace
200   content <- some (parseTextModifier <|> parsePlainTextDelimiter) <# is '\n'
201   syntax <- some (is '=') <|> some (is '-')
202   _ <- is '\n'
203   if all (== '=') syntax && length syntax >= 2      I
204     then return (Header "1" content)
205   else if all (== '-') syntax && length syntax >= 2
206     then return (Header "2" content)
207   else empty
208
209 parseOrderedListWrap :: Parser BlockElements
210 parseOrderedListWrap = OrderedListWrap <$> parseOrderedList
211
212
213 parseOrderedList :: Parser OrderedList
214 parseOrderedList = do
215   first <- parseOrderedListFirstItem 0
216   rest <- many (parseOrderedListItem 0)
217   return $ OrderedList (first : rest)
218
219 parseOrderedListFirstItem :: Int -> Parser OrderedListItem
220 parseOrderedListFirstItem indentLevel = do
221   n <- some digit
222   _ <- is '.'
223   _ <- inlineSpace1
224   content <- some (parseTextModifier <|> parsePlainTextDelimiter)
225   _ <- is '\n'

```

```

AssignmentLhs M  BNF.md  MarkdownParser.hs M X  blockquotes.html M  heading.html M
Haskell > src > MarkdownParser.hs > MarkdownParser > parsePlainText
315     return (TableCell trimmedContent)
316
317     trimSpaces :: TextModifier -> TextModifier
318     trimSpaces (PlainText s) = PlainText (trimLeading (trimTrailing s))
319     trimSpaces other = other -- No trimming needed for other ADT types
320
321     trimLeadingSpaces :: TextModifier -> TextModifier
322     trimLeadingSpaces (PlainText s) = PlainText (trimLeading s)
323     trimLeadingSpaces other = other -- No trimming needed for other ADT types
324
325     trimTrailingSpaces :: TextModifier -> TextModifier
326     trimTrailingSpaces (PlainText s) = PlainText (trimTrailing s)
327     trimTrailingSpaces other = other -- No trimming needed for other ADT types
328
329     trimTrailing :: String -> String
330     trimTrailing [] = []
331     trimTrailing xs = reverse (trimLeading (reverse xs))
332
333     trimLeading :: String -> String
334     trimLeading [] = []
335     trimLeading (x:xs)
336     | x == ' ' = trimLeading xs
337     | otherwise = x:xs
338
339     trimNewLine :: [DocumentElements] -> [DocumentElements]
340     trimNewLine = trimLeadingNewLine . reverse . trimTrailingNewLine . reverse
341
342     trimLeadingNewLine :: [DocumentElements] -> [DocumentElements]
343     trimLeadingNewLine [] = []
344     trimLeadingNewLine (ParagraphElement (Paragraph []) : rest) = trimLeadingNewLine rest
345     trimLeadingNewLine (ParagraphElement (Paragraph (first : content)) : rest) = ParagraphElement (Paragraph (trimLeadingSpaces first : trimLeadingSpaces content)) : trimLeadingNewLine rest
346     trimLeadingNewLine (other : rest) = other : rest -- For non-paragraph elements, stop trimming
347
348     trimTrailingNewLine :: [DocumentElements] -> [DocumentElements]
349     trimTrailingNewLine [] = []
350     trimTrailingNewLine (ParagraphElement (Paragraph []) : rest) = trimTrailingNewLine rest
351     trimTrailingNewLine (ParagraphElement (Paragraph content) : rest) = ParagraphElement (Paragraph (trimFirstItemAfterReverse content)) : trimTrailingNewLine rest
352     trimTrailingNewLine (other : rest) = other : rest -- For non-paragraph elements, stop trimming
353
354     trimFirstItemAfterReverse :: [TextModifier] -> [TextModifier]

```


Main.ts

```
Assignment.hs M  BNF.md  MarkdownParser.hs M  Parser.hs  blockquotes.html M
JS > src > TS main.ts > main > subscription > map() callback
18  const markdownInput = document.getElementById(
20  ) as HTMLTextAreaElement;
21  const checkbox = document.querySelector('input[name="checkbox"]')!;
22  const saveButton = document.getElementById("save-button")! as HTMLButtonElement;
23  const titleInput = document.getElementById("title-input")! as HTMLInputElement;
24
25  type Action = (_: State) => State;
26
27  const resetState: Action = (s) => {
28    return { ...s, save: false };
29  };
30
31  const compose =
32    <T, U>(g: (_: T) => U) =>
33    <V>(f: (_: U) => V) =>
34    (t: T): V =>
35      f(g(t));
36
37  // Create an Observable for keyboard input events
38  const input$: Observable<Action> = fromEvent<KeyboardEvent>(
39    markdownInput,
40    "input",
41  ).pipe(
42    map((event) => (event.target as HTMLInputElement).value),
43    map((value) => (s) => ({ ...s, markdown: value })),
44  );
45
46  const checkboxStream$: Observable<Action> = fromEvent(checkbox, "change").pipe(
47    map((event) => (event.target as HTMLInputElement).checked),
48    map((value) => (s) => ({ ...s, renderHTML: value })),
49  );
50
51  const saveButton$: Observable<Action> = fromEvent(saveButton, "click").pipe(
52    map(() => (s) => ({ ...s, save: true })),
53  );
54
55  const titleInput$: Observable<Action> = fromEvent(titleInput, "input").pipe(
```

Assignment.hs M BNF.md MarkdownParser.hs M Parser.hs blockquotes.html M heading.html M

```

JS > src > TS main.ts > ...
37 // Create an Observable for keyboard input events
38 const input$: Observable<Action> = fromEvent<KeyboardEvent>({
39   markdownInput,
40   "input",
41 }).pipe(
42   map((event) => (event.target as HTMLInputElement).value),
43   map((value) => (s) => ({ ...s, markdown: value })),
44 );
45
46 const checkboxStream$: Observable<Action> = fromEvent(checkbox, "change").pipe(
47   map((event) => (event.target as HTMLInputElement).checked),
48   map((value) => (s) => ({ ...s, renderHTML: value })),
49 );
50
51 const saveButton$: Observable<Action> = fromEvent(saveButton, "click").pipe(
52   map(() => (s) => ({ ...s, save: true })),
53 );
54
55 const titleInput$: Observable<Action> = fromEvent(titleInput, "input").pipe(
56   map((event) => (event.target as HTMLInputElement).value),
57   map((newTitle) => (s) => {
58     // Update the HTML title
59     const updatedTitle = newTitle.trim() || "Converted HTML";
60     const updatedHTML = s.HTML.replace(/<title>.*<\title>/, `<title>${updatedTitle}<\title>`);
61     document.title = updatedTitle;
62     return ({ ...s, title: updatedTitle, HTML: updatedHTML });
63   })
64 );
65
66 function getHTML(s: State): Observable<State> {
67   // Get the HTML as a stream
68   return ajax<{ html: string }>({
69     url: "/api/convertMD",
70     method: "POST",
71     headers: {
72       "Content-Type": "application/x-www-form-urlencoded",
73     },

```

```

74     body: s.markdown,
75   }).pipe(
76     map((response) => response.response), // Extracting the response data
77     map((data) => {
78       const updatedHTML = data.html.replace(/<title>.*<\title>/, `<title>${s.title}<\title>`);
79       return {
80         ...s,
81         HTML: updatedHTML,
82       };
83     }),
84     first(),
85   );
86 }
87
88 function saveHTML(s: State): Observable<State> {
89   const updatedHTML = s.HTML.replace(/<title>.*<\title>/, `<title>${s.title}<\title>`);
90   // Fetch the current time from the backend
91   return ajax({
92     url: "/api/saveHTML",
93     method: "POST",
94     headers: {
95       "Content-Type": "application/x-www-form-urlencoded",
96     },
97     body: updatedHTML,
98   }).pipe(
99     map((response) => response.response), // Extract the response data
100     map((data) => {
101       // console.log(data.status);
102       return {
103         ...s,
104         save: true,
105       };
106     }),
107     first(),

```



```

assignment.hs M  BNF.md  MarkdownParser.hs M  Parser.hs  blockquotes.html M
> src > TS main.ts > main > subscription > mergeScan() callback
111 const initialState: State = {
112   markdown: "",
113   HTML: "",
114   renderHTML: true,
115   save: false,
116   title: "Converted HTML",
117 };
118

```

```

119 function main() {
120   // Subscribe to the input Observable to listen for changes
121   const subscription = merge(input$, checkboxStream$, saveButton$, titleInput$)
122     .pipe(
123       map((reducer: Action) => {
124         // Reset some variables in the state in every tick
125         const newReducer = compose(resetState)(reducer);
126         return newReducer;
127       }),
128       mergeScan((acc: State, reducer: Action) => {
129         const newState = reducer(acc);
130         // getHTML returns an observable of length one
131         // so we 'scan' and merge the result of getHTML in to our stream
132         return newState.save ? saveHTML(newState) : getHTML(newState)
133       }, initialState),
134     )
135     .subscribe((value) => {
136       const htmlOutput = document.getElementById("html output");
137       if (htmlOutput) {
138         htmlOutput.innerHTML = "";
139         htmlOutput.textContent = "";
140         if (value.renderHTML) {
141           const highlight =
142             '<link rel="stylesheet" href="https://unpkg.com/@highlightjs/cdn-assets@11.3.1/styles/default.min.css">';
143           htmlOutput.innerHTML = highlight + value.HTML;
144           // Magic code to add code highlighting
145           const blocks = htmlOutput.querySelectorAll("pre code");
146           blocks.forEach((block) => {
147             hljs.highlightElement(block as HTMLInputElement),
148           });
149           htmlOutput.style.whiteSpace = "normal";
150         } else {
151           htmlOutput.textContent = value.HTML;
152           htmlOutput.style.whiteSpace = "pre-wrap";
153         }
154       }
155     });
156 }
157

```

Ln 150, Col 21 (38 selected) Spaces: 4 UTF-8 CR/LF

Main.hs

```
Assignment.hs M  ENF.md  MarkdownParser.hs M  Parser.hs  blockquotes.html M  heading.html M

Haskell > app > Main.hs > Main > main

18  getResult _ _
19
20  -- Magic code to convert key, value pairs to JSON to send back to the server
21  jsonResponse :: [(String, String)] -> ActionM ()
22  jsonResponse pairs =
23    json $ object [fromString key .<= (pack value) :: Text] | (key, value) <- pairs]
24
25
26  main :: IO ()
27  main = scotty 3000 $ do
28    post "/api/convertMD" $ do
29      requestBody <- body
30      -- Convert the raw request body from ByteString to Text
31      let requestBodyText = decodeUtf8 requestBody
32      -- Convert the Text to String
33      str = unpack requestBodyText
34      -- Parse the Markdown string using 'markdownParser' and apply 'convertAllHTML'
35      converted_html = getResult (parse parseDocument str) convertDocumentHTML
36
37      -- Respond with the converted HTML as JSON
38      jsonResponse [("html", converted_html)]
39
40    post "/api/saveHTML" $ do
41      requestBody <- body
42      -- Convert the raw request body from ByteString to Text
43      let requestBodyText = decodeUtf8 requestBody
44      -- Convert the Text to String
45      str = unpack requestBodyText
46      -- Parse the Markdown string using 'markdownParser' and apply 'convertAllHTML'
47      -- converted html = getResult (parse parseDocument str) convertDocumentHTML
48      <- liftIO $ writeContents str
49      -- Respond with the converted HTML as JSON
50      jsonResponse [("status", "HTML saved")]
```