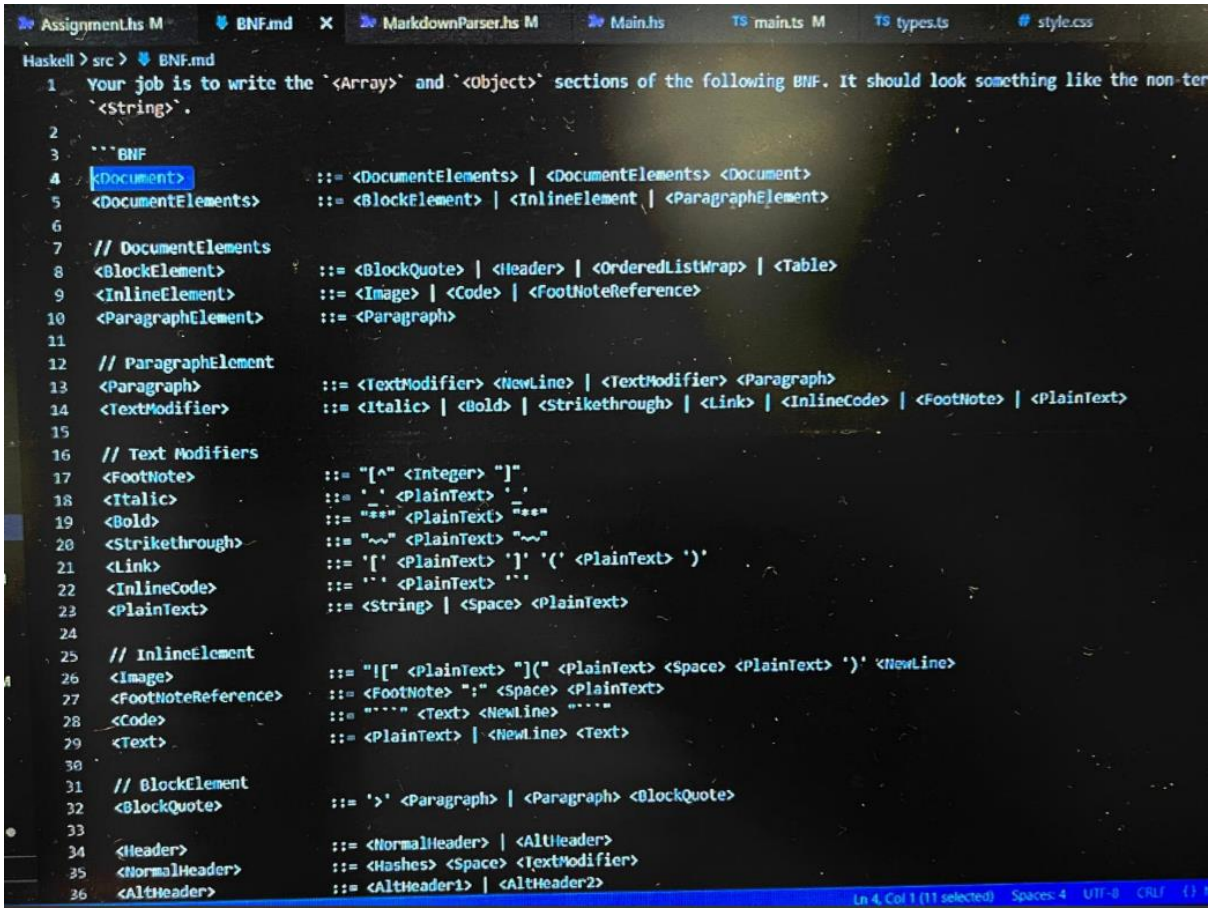
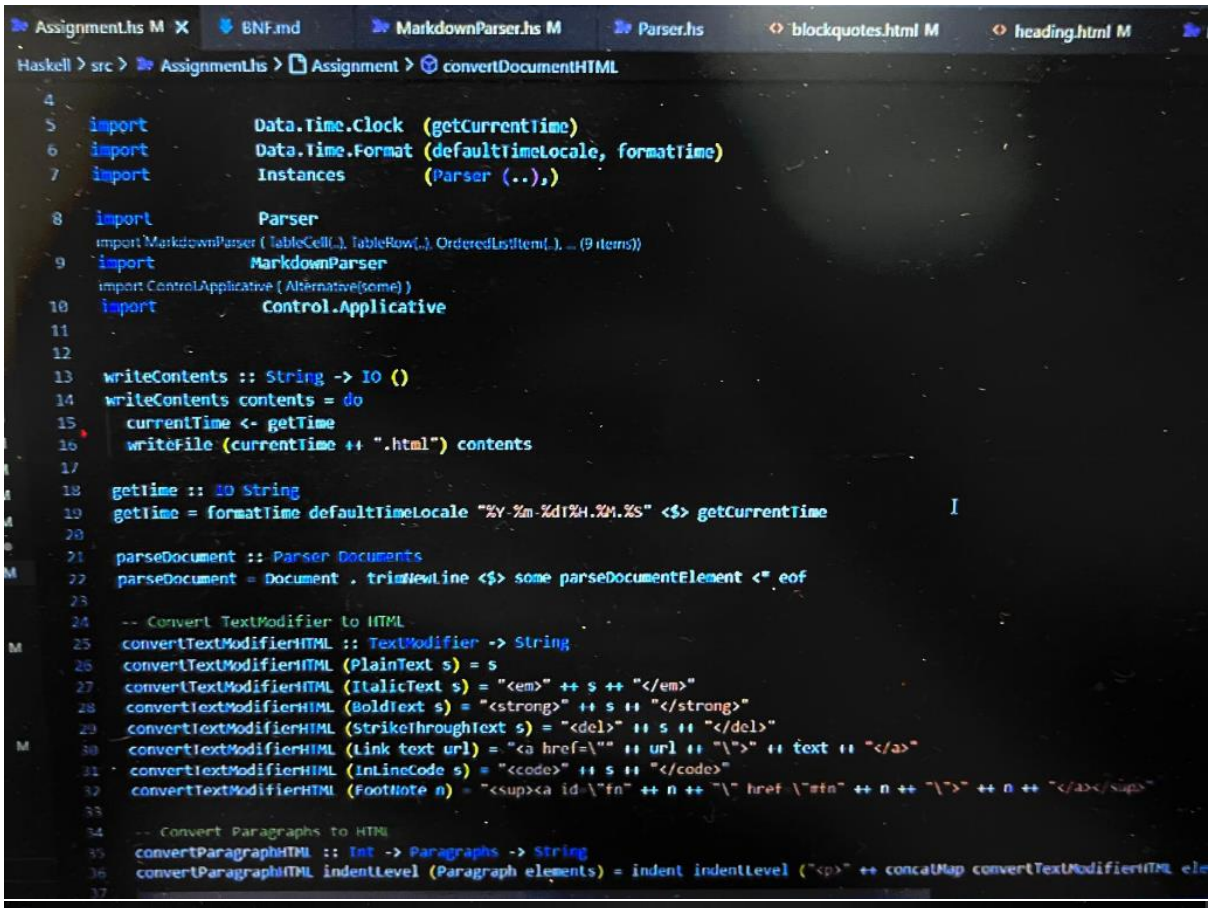
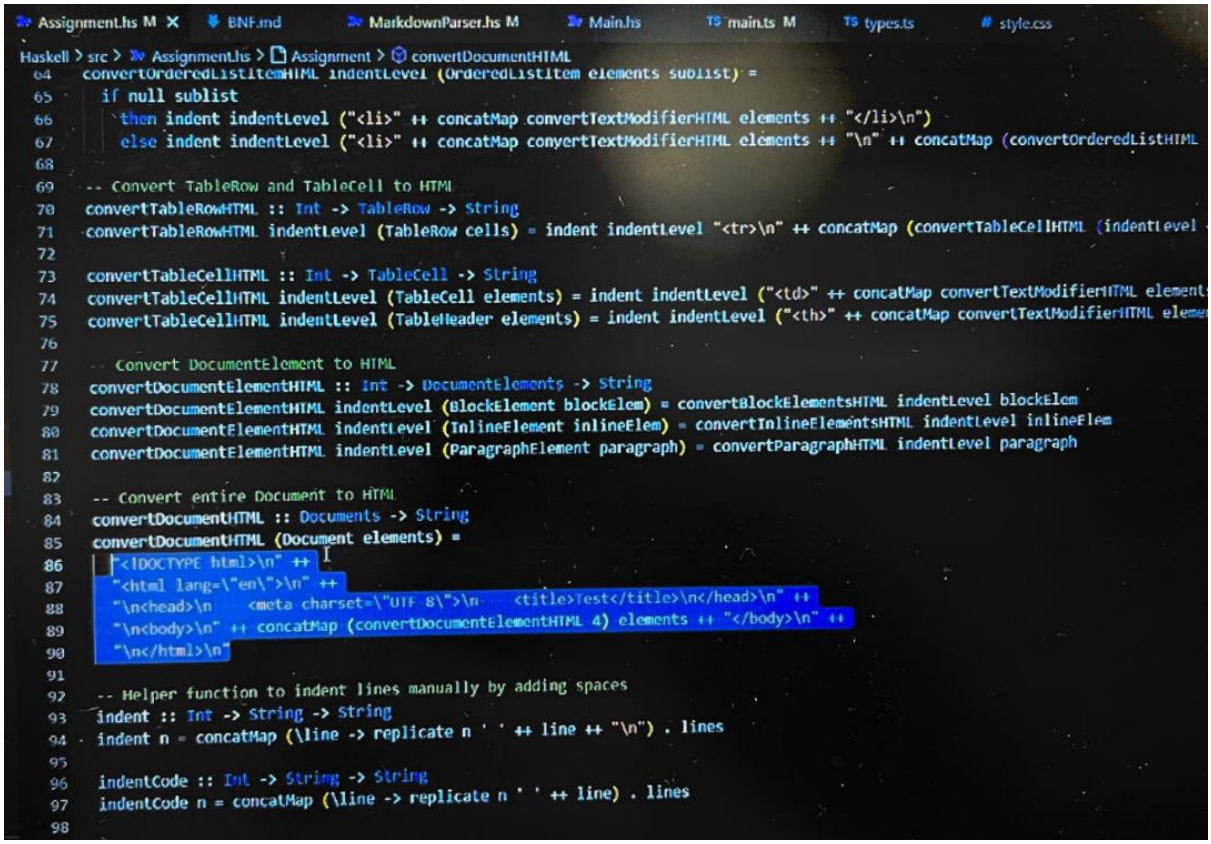
BNF.md

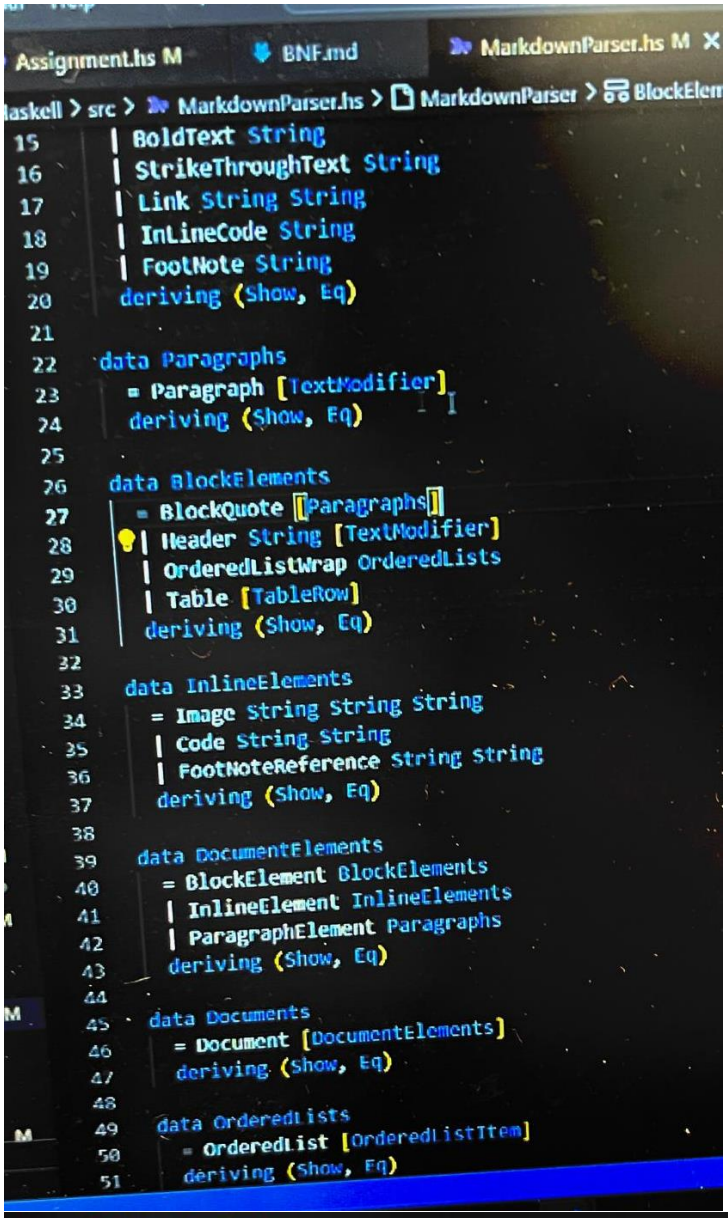

Assignment.hs M
BNF.md
MarkdownParser.hs M
Main.hs
main.ts M
types.ts
style.css
Haskell > src > BNF.md
Your job is to write the `<Array>` and `<Object>` sections of the following BNF. It should look something like the non-ter
`<String>`.
```BNF
<Document> ::= <DocumentElements> | <DocumentElements> <Document>
<DocumentElements> ::= <BlockElement> | <InlineElement> | <ParagraphElement>
// DocumentElements
<BlockElement> ::= <BlockQuote> | <Header> | <OrderedListWrap> | <Table>
<InlineElement> ::= <Image> | <Code> | <FootNoteReference>
<ParagraphElement> ::= <Paragraph>
// ParagraphElement
<Paragraph> ::= <TextModifier> <NewLine> | <TextModifier> <Paragraph>
<TextModifier> ::= <Italic> | <Bold> | <Strikethrough> | <Link> | <InlineCode> | <FootNote> | <PlainText>
// Text Modifiers
<FootNote> ::= "[^" <Integer> "]"
<Italic> ::= '_' <PlainText> '_'
<Bold> ::= "**" <PlainText> "**"
<Strikethrough> ::= "~~" <PlainText> "~~"
<Link> ::= '[' <PlainText> ']' '(' <PlainText> ')'
<InlineCode> ::= '`' <PlainText> '`'
<PlainText> ::= <String> | <Space> <PlainText>
// InlineElement
<Image> ::= "![" <PlainText> "](" <PlainText> <Space> <PlainText> ')' <NewLine>
<FootNoteReference> ::= <FootNote> ":" <Space> <PlainText>
<Code> ::= "```" <Text> <NewLine> "```"
<Text> ::= <PlainText> | <NewLine> <Text>
// BlockElement
<BlockQuote> ::= '>' <Paragraph> | <Paragraph> <BlockQuote>
<Header> ::= <NormalHeader> | <AltHeader>
<NormalHeader> ::= <Hashes> <Space> <TextModifier>
<AltHeader> ::= <AltHeader1> | <AltHeader2>
Ln 4, Col 1 (11 selected)
Spaces: 4
UTF-8
CRLF
```

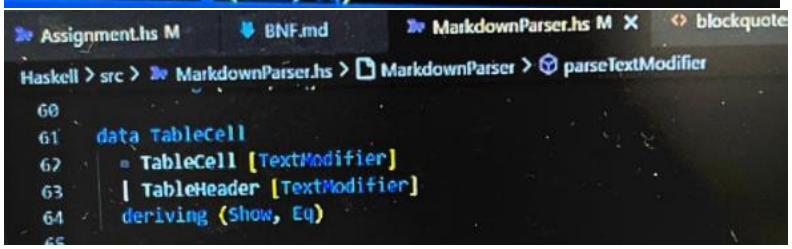
Assignment.hs

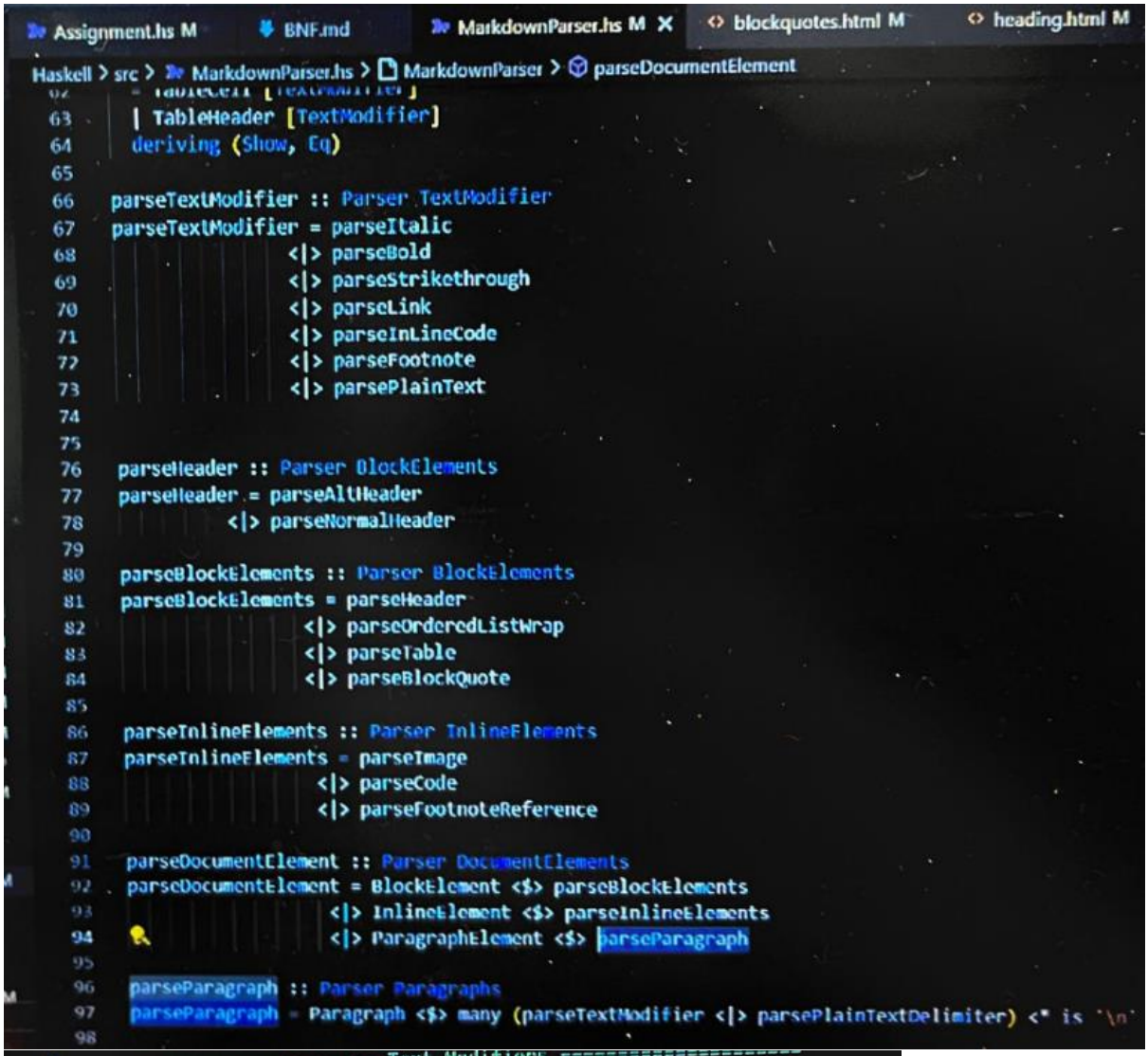


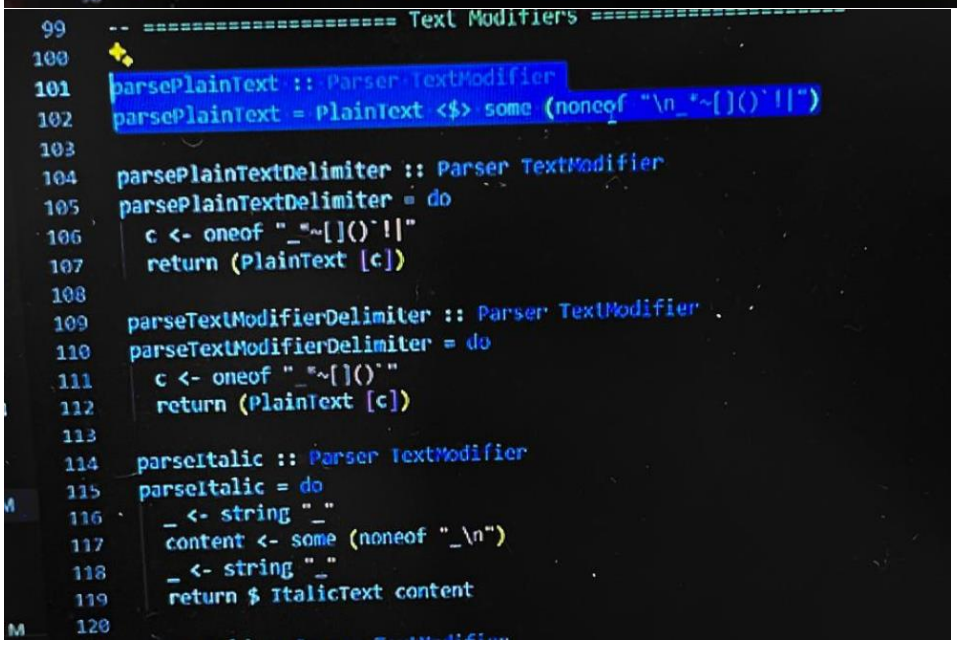

Assignment.hs M
BNF.md
MarkdownParser.hs M
Main.hs
main.ts M
types.ts
style.css
Haskell > src > Assignment.hs > Assignment > convertDocumentHTML
convertOrderedListItemHTML indentLevel (OrderedListItem elements sublist) =
if null sublist
then indent indentLevel ("
" ++ concatMap convertTextModifierHTML elements ++ "
\n")
else indent indentLevel ("<li>" ++ concatMap convertTextModifierHTML elements ++ "\n" ++ concatMap (convertOrderedListHTML
-- Convert TableRow and TableCell to HTML
convertTableRowHTML :: Int -> TableRow -> String
convertTableRowHTML indentLevel (TableRow cells) = indent indentLevel "<tr>\n" ++ concatMap (convertTableCellHTML (indentLevel
convertTableCellHTML :: Int -> TableCell -> String
convertTableCellHTML indentLevel (TableCell elements) = indent indentLevel ("<td>" ++ concatMap convertTextModifierHTML element
convertTableCellHTML indentLevel (TableHeader elements) = indent indentLevel ("<th>" ++ concatMap convertTextModifierHTML eleme
-- Convert DocumentElement to HTML
convertDocumentElementHTML :: Int -> DocumentElements -> String
convertDocumentElementHTML indentLevel (BlockElement blockElem) = convertBlockElementsHTML indentLevel blockElem
convertDocumentElementHTML indentLevel (InlineElement inlineElem) = convertInlineElementsHTML indentLevel inlineElem
convertDocumentElementHTML indentLevel (ParagraphElement paragraph) = convertParagraphHTML indentLevel paragraph
-- Convert entire Document to HTML
convertDocumentHTML :: Documents -> String
convertDocumentHTML (Document elements) =
"<!DOCTYPE html>\n" ++
"<html lang=\"en\">\n" ++
"\n<head>\n <meta charset=\"UTF-8\">\n <title>Test</title>\n</head>\n" ++
"\n<body>\n" ++ concatMap (convertDocumentElementHTML 4) elements ++ "</body>\n" ++
"\n</html>\n"
-- Helper function to indent lines manually by adding spaces
indent :: Int -> String -> String
indent n = concatMap (\line -> replicate n ' ' ++ line ++ "\n") . lines
indentCode :: Int -> String -> String
indentCode n = concatMap (\line -> replicate n ' ' ++ line) . lines

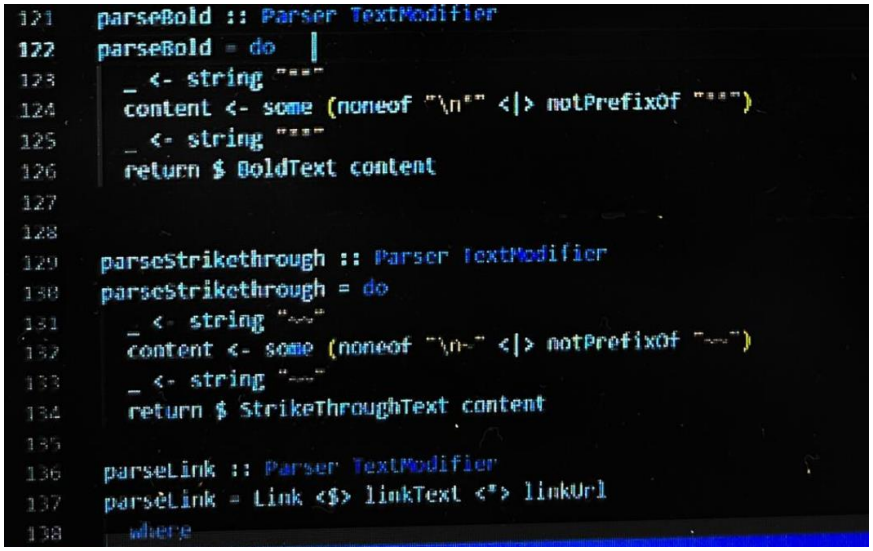
MarkdownParser.hs



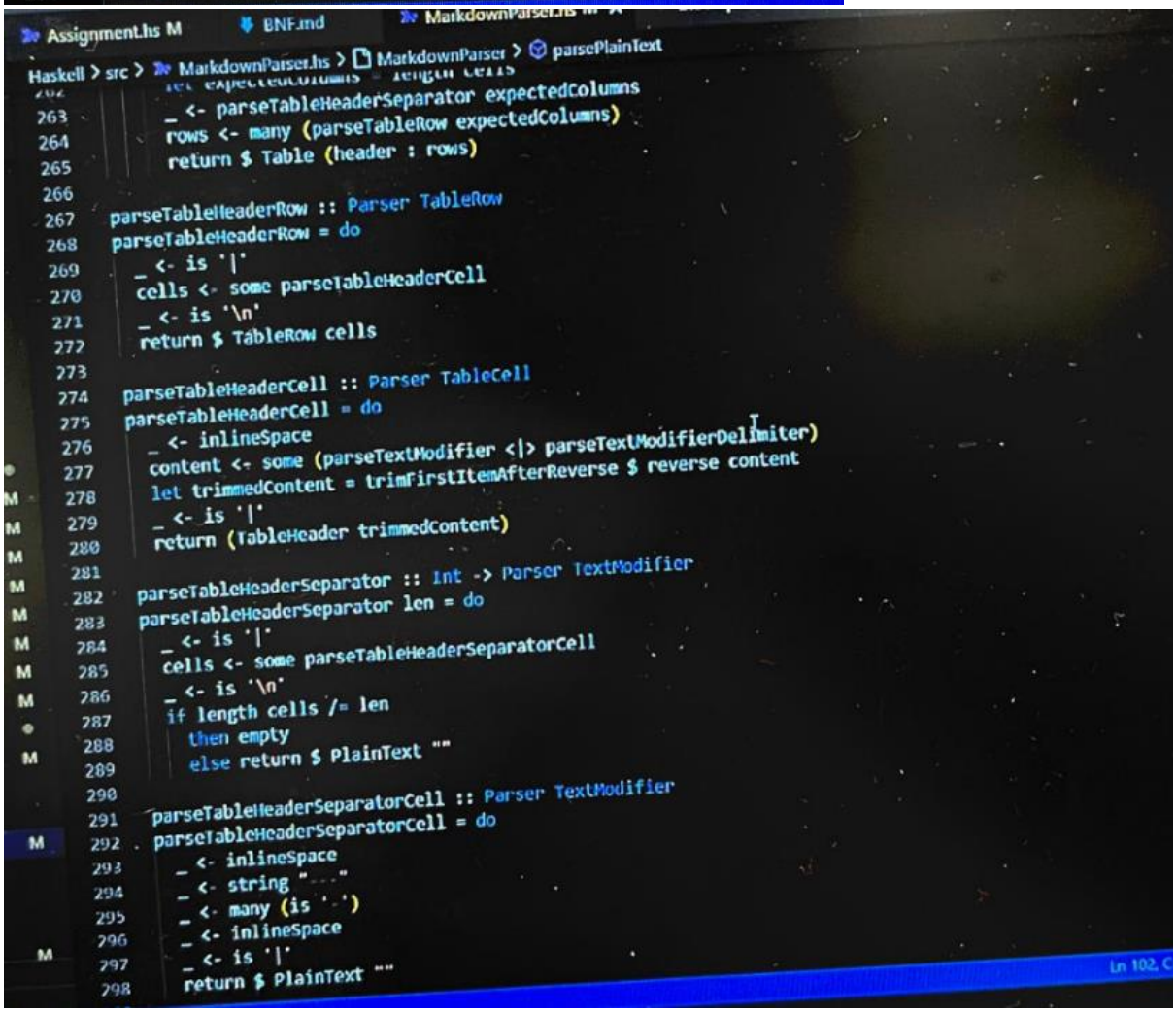



Assignment.hs M
BNF.md
MarkdownParser.hs M
blockquotes.html M
heading.html M
Haskell > src > MarkdownParser.hs > MarkdownParser > parseDocumentElement
| TableHeader [TextModifier]
deriving (Show, Eq)
parseTextModifier :: Parser TextModifier
parseTextModifier = parseItalic
<|> parseBold
<|> parseStrikethrough
<|> parseLink
<|> parseInLineCode
<|> parseFootnote
<|> parsePlainText
parseHeader :: Parser BlockElements
parseHeader = parseAltHeader
<|> parseNormalHeader
parseBlockElements :: Parser BlockElements
parseBlockElements = parseHeader
<|> parseOrderedListWrap
<|> parseTable
<|> parseBlockQuote
parseInlineElements :: Parser InlineElements
parseInlineElements = parseImage
<|> parseCode
<|> parseFootnoteReference
parseDocumentElement :: Parser DocumentElements
parseDocumentElement = BlockElement <$> parseBlockElements
<|> InlineElement <$> parseInlineElements
<|> ParagraphElement <$> parseParagraph
parseParagraph :: Parser Paragraphs
parseParagraph = Paragraph <$> many (parseTextModifier <|> parsePlainTextDelimiter) <* is '\n'

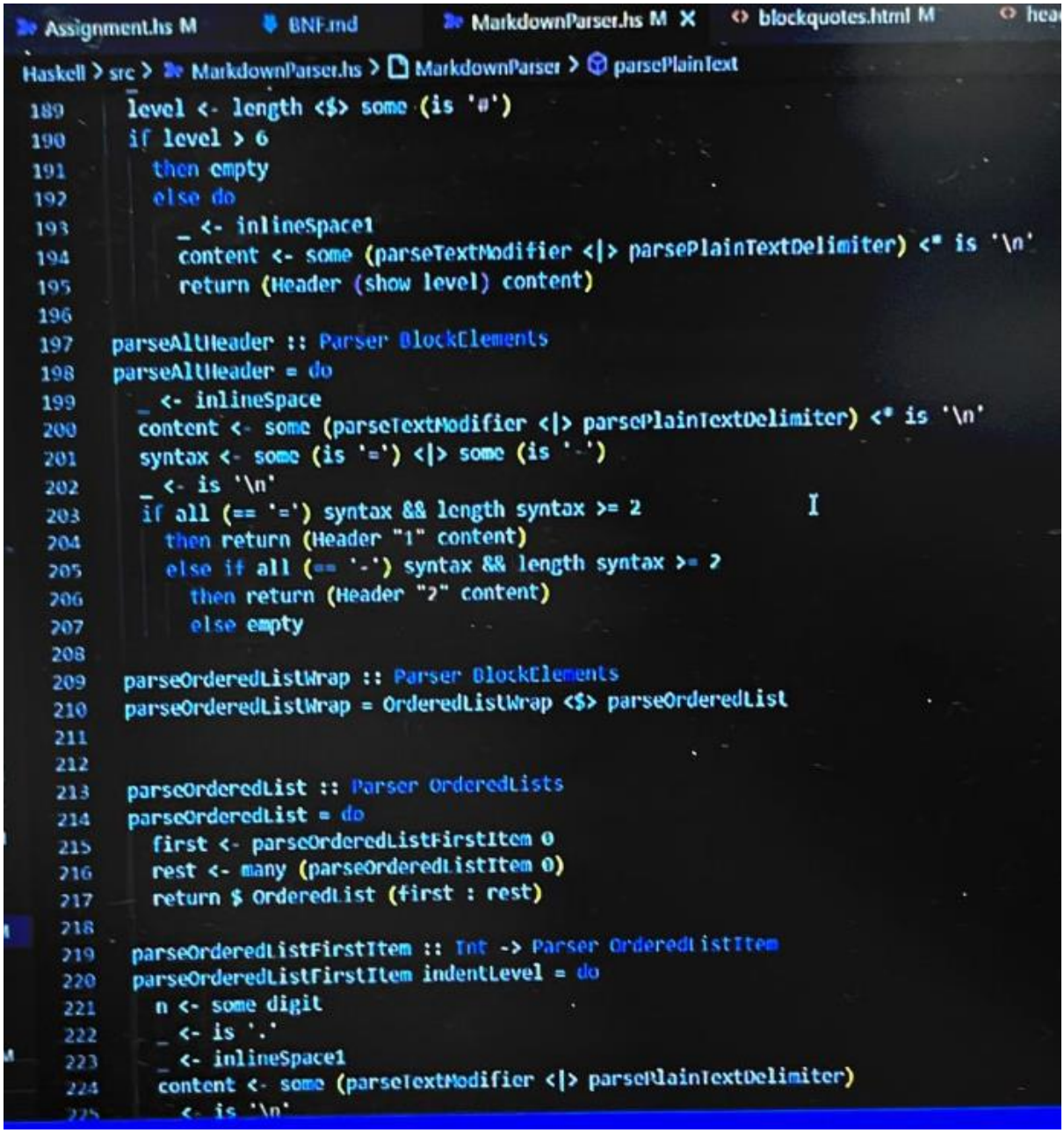

-- ========== Text Modifiers ==========
parsePlainText :: Parser TextModifier
parsePlainText = PlainText <$> some (noneof "\n_*~[]()`!|")
parsePlainTextDelimiter :: Parser TextModifier
parsePlainTextDelimiter = do
c <- oneof "_*~[]()`!|"
return (PlainText [c])
parseTextModifierDelimiter :: Parser TextModifier
parseTextModifierDelimiter = do
c <- oneof "_*~[]()`"
return (PlainText [c])
parseItalic :: Parser TextModifier
parseItalic = do
_ <- string "_"
content <- some (noneof "_\n")
_ <- string "_"
return $ ItalicText content

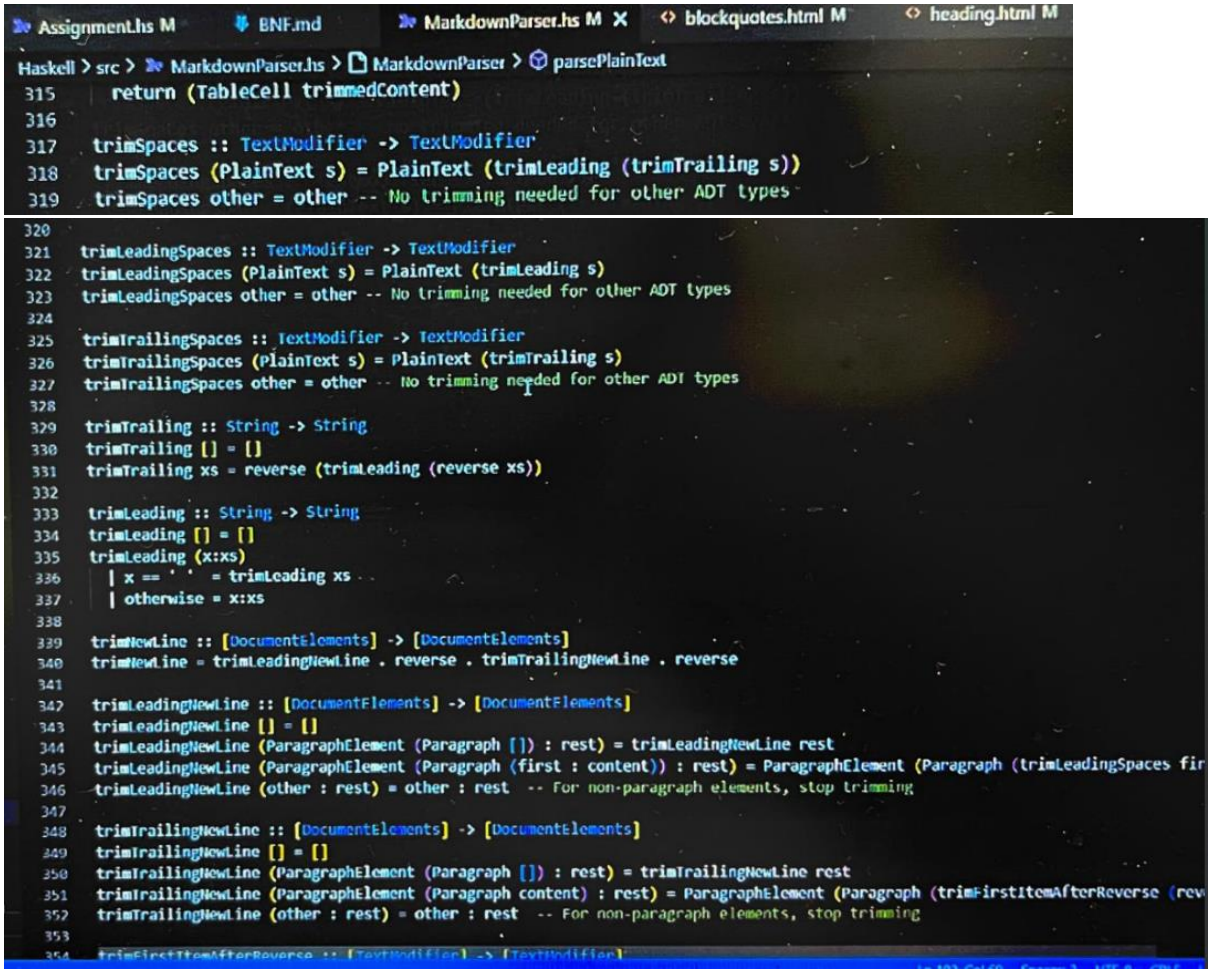


```haskell
parseBold :: Parser TextModifier
parseBold = do
  _ <- string "***"
  content <- some (noneOf "\n*" <|> notPrefixOf "***")
  _ <- string "***"
  return $ BoldText content


parseStrikethrough :: Parser TextModifier
parseStrikethrough = do
  _ <- string "~~"
  content <- some (noneOf "\n~" <|> notPrefixOf "~~")
  _ <- string "~~"
  return $ StrikeThroughText content

parseLink :: Parser TextModifier
parseLink = Link <$> linkText <*> linkUrl
  where
```

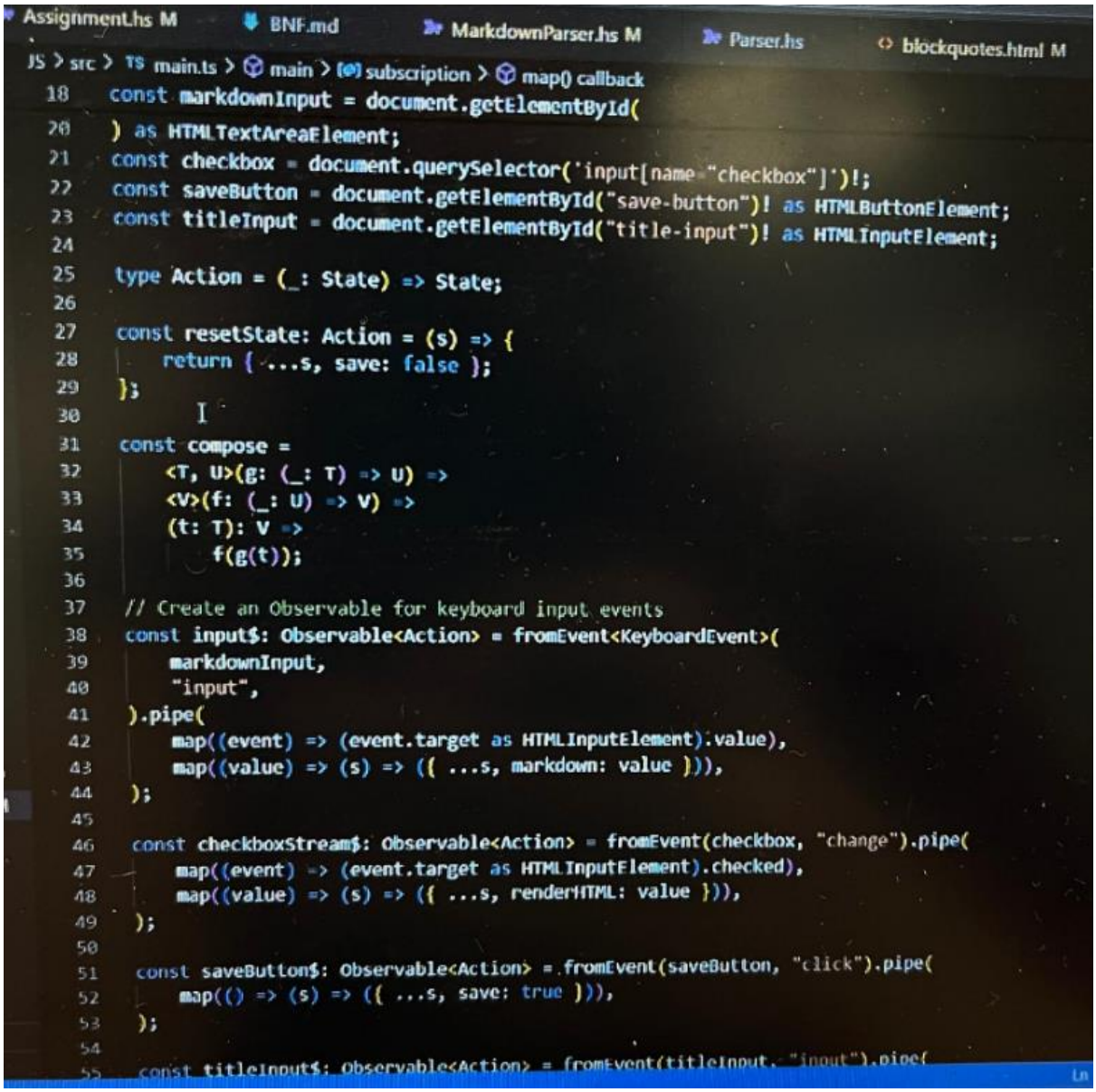


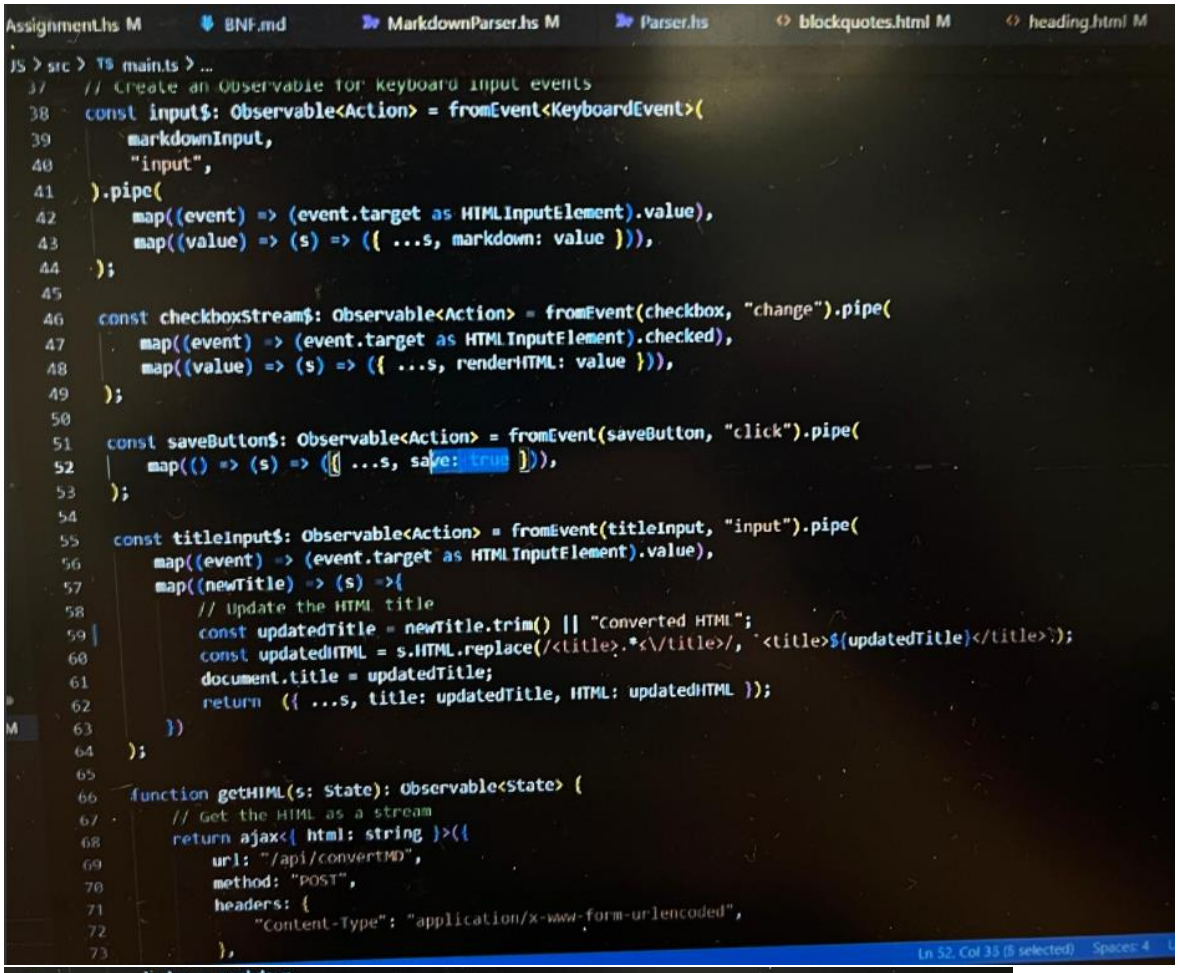
```haskell
        _ <- parseTableHeaderSeparator expectedColumns
        rows <- many (parseTableRow expectedColumns)
        return $ Table (header : rows)

parseTableHeaderRow :: Parser TableRow
parseTableHeaderRow = do
  _ <- is '|'
  cells <- some parseTableHeaderCell
  _ <- is '\n'
  return $ TableRow cells

parseTableHeaderCell :: Parser TableCell
parseTableHeaderCell = do
  _ <- inlineSpace
  content <- some (parseTextModifier <|> parseTextModifierDelimiter)
  let trimmedContent = trimFirstItemAfterReverse $ reverse content
  _ <- is '|'
  return (TableHeader trimmedContent)

parseTableHeaderSeparator :: Int -> Parser TextModifier
parseTableHeaderSeparator len = do
  _ <- is '|'
  cells <- some parseTableHeaderSeparatorCell
  _ <- is '\n'
  if length cells /= len
    then empty
    else return $ PlainText ""

parseTableHeaderSeparatorCell :: Parser TextModifier
parseTableHeaderSeparatorCell = do
  _ <- inlineSpace
  _ <- string "---"
  _ <- many (is '-')
  _ <- inlineSpace
  _ <- is '|'
  return $ PlainText ""
```

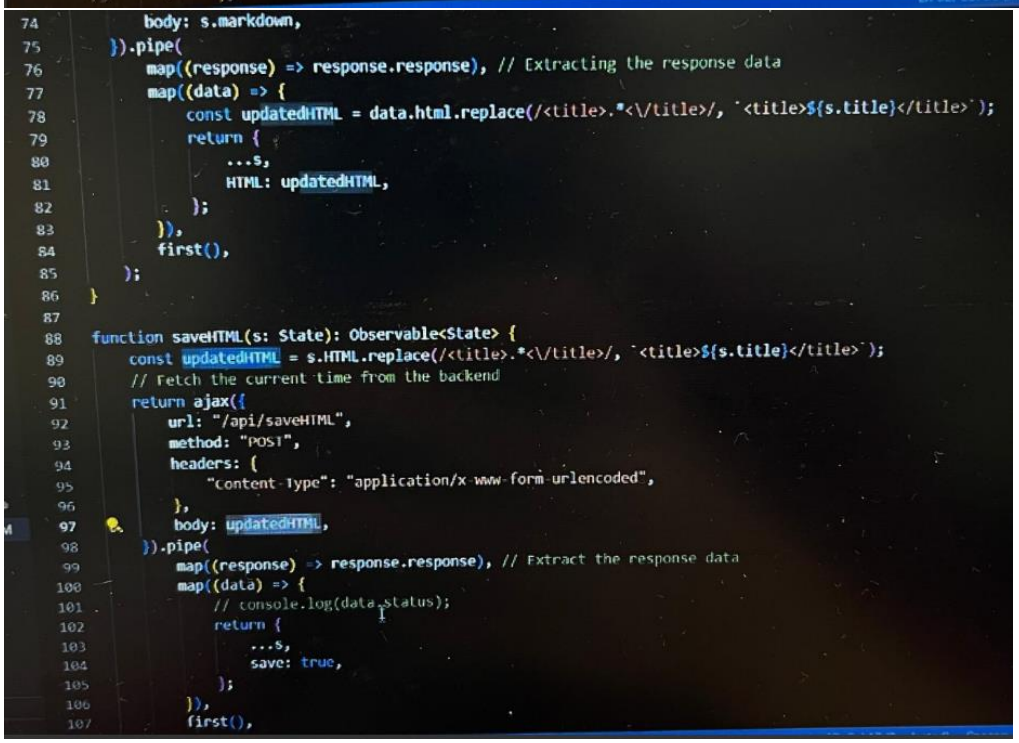

Assignment.hs M
BNF.md
MarkdownParser.hs M
blockquotes.html M
Haskell > src > MarkdownParser.hs > MarkdownParser > parsePlainText
level <- length <$> some (is '#')
if level > 6
then empty
else do
_ <- inlineSpace1
content <- some (parseTextModifier <|> parsePlainTextDelimiter) <* is '\n'
return (Header (show level) content)
parseAltHeader :: Parser BlockElements
parseAltHeader = do
_ <- inlineSpace
content <- some (parseTextModifier <|> parsePlainTextDelimiter) <* is '\n'
syntax <- some (is '=') <|> some (is '-')
_ <- is '\n'
if all (== '=') syntax && length syntax >= 2
then return (Header "1" content)
else if all (== '-') syntax && length syntax >= 2
then return (Header "2" content)
else empty
parseOrderedListWrap :: Parser BlockElements
parseOrderedListWrap = OrderedListWrap <$> parseOrderedList
parseOrderedList :: Parser OrderedLists
parseOrderedList = do
first <- parseOrderedListFirstItem 0
rest <- many (parseOrderedListItem 0)
return $ OrderedList (first : rest)
parseOrderedListFirstItem :: Int -> Parser OrderedListItem
parseOrderedListFirstItem indentLevel = do
n <- some digit
_ <- is '.'
_ <- inlineSpace1
content <- some (parseTextModifier <|> parsePlainTextDelimiter)
<- is '\n'

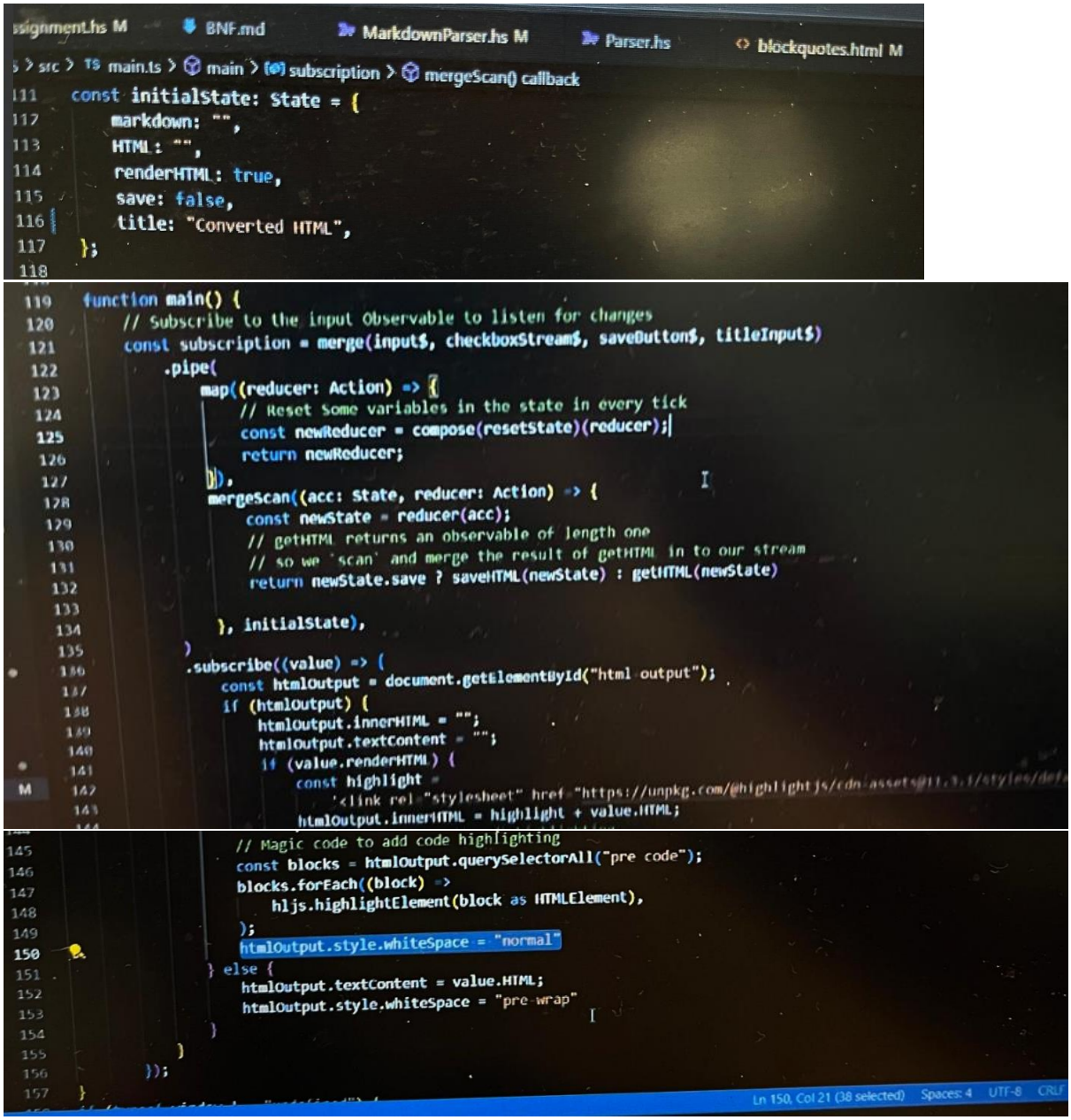

AssignmentLhs M
BNF.md
MarkdownParser.hs M
blockquotes.html M
heading.html M
Haskell > src > MarkdownParser.hs > MarkdownParser > parsePlainText
315 return (TableCell trimmedContent)
316
317 trimSpaces :: TextModifier -> TextModifier
318 trimSpaces (PlainText s) = PlainText (trimLeading (trimTrailing 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 fir
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 (rev
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
const markdownInput = document.getElementById(
) as HTMLTextAreaElement;
const checkbox = document.querySelector('input[name="checkbox"]')!;
const saveButton = document.getElementById("save-button")! as HTMLButtonElement;
const titleInput = document.getElementById("title-input")! as HTMLInputElement;
type Action = (_: State) => State;
const resetState: Action = (s) => {
return { ...s, save: false };
};
const compose =
<T, U>(g: (_: T) => U) =>
<V>(f: (_: U) => V) =>
(t: T): V =>
f(g(t));
// Create an Observable for keyboard input events
const input$: Observable<Action> = fromEvent<KeyboardEvent>(
markdownInput,
"input",
).pipe(
map((event) => (event.target as HTMLInputElement).value),
map((value) => (s) => ({ ...s, markdown: value })),
);
const checkboxStream$: Observable<Action> = fromEvent(checkbox, "change").pipe(
map((event) => (event.target as HTMLInputElement).checked),
map((value) => (s) => ({ ...s, renderHTML: value })),
);
const saveButton$: Observable<Action> = fromEvent(saveButton, "click").pipe(
map(() => (s) => ({ ...s, save: true })),
);
const titleInput$: Observable<Action> = fromEvent(titleInput, "input").pipe(

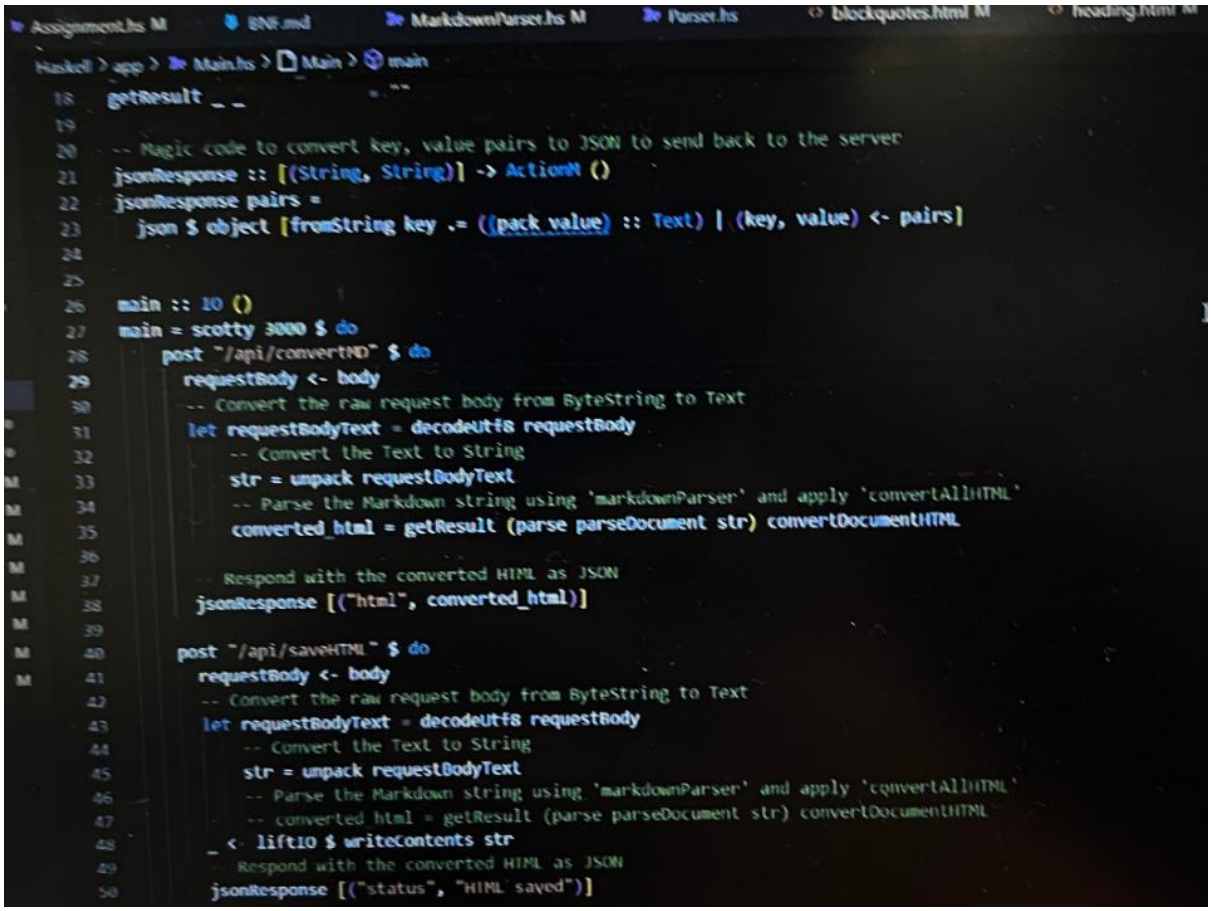

AssignmentLhs M
BNF.md
MarkdownParser.hs M
Parser.hs
blockquotes.html M
heading.html M
JS > src > TS main.ts > ...
// Create an Observable for keyboard input events
const input$: Observable<Action> = fromEvent<KeyboardEvent>(
markdownInput,
"input",
).pipe(
map((event) => (event.target as HTMLInputElement).value),
map((value) => (s) => ({ ...s, markdown: value })),
);
const checkboxStream$: Observable<Action> = fromEvent(checkbox, "change").pipe(
map((event) => (event.target as HTMLInputElement).checked),
map((value) => (s) => ({ ...s, renderHTML: value })),
);
const saveButton$: Observable<Action> = fromEvent(saveButton, "click").pipe(
map(() => (s) => ({ ...s, save: true })),
);
const titleInput$: Observable<Action> = fromEvent(titleInput, "input").pipe(
map((event) => (event.target as HTMLInputElement).value),
map((newTitle) => (s) =>{
// Update the HTML title
const updatedTitle = newTitle.trim() || "Converted HTML";
const updatedHTML = s.HTML.replace(/<title>.*<\/title>/, `<title>${updatedTitle}</title>`);
document.title = updatedTitle;
return ({ ...s, title: updatedTitle, HTML: updatedHTML });
})
);
function getHTML(s: State): Observable<State> {
// Get the HTML as a stream
return ajax<{ html: string }>({
url: "/api/convertMD",
method: "POST",
headers: {
"Content-Type": "application/x-www-form-urlencoded",
},
Ln 52, Col 33 (5 selected) Spaces: 4


body: s.markdown,
}).pipe(
map((response) => response.response), // Extracting the response data
map((data) => {
const updatedHTML = data.html.replace(/<title>.*<\/title>/, `<title>${s.title}</title>`);
return {
...s,
HTML: updatedHTML,
};
}),
first(),
);
}
function saveHTML(s: State): Observable<State> {
const updatedHTML = s.HTML.replace(/<title>.*<\/title>/, `<title>${s.title}</title>`);
// Fetch the current time from the backend
return ajax({
url: "/api/saveHTML",
method: "POST",
headers: {
"Content-Type": "application/x-www-form-urlencoded",
},
body: updatedHTML,
}).pipe(
map((response) => response.response), // Extract the response data
map((data) => {
// console.log(data.status);
return {
...s,
save: true,
};
}),
first(),



```
ssignment.hs M    BNF.md    MarkdownParser.hs M    Parser.hs    blockquotes.html M
s > src > TS main.ts > main > subscription > mergeScan() callback
111  const initialState: State = {
117      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
134              }, initialState),
135          )
136          .subscribe((value) => {
137              const htmlOutput = document.getElementById("html-output");
138              if (htmlOutput) {
139                  htmlOutput.innerHTML = "";
140                  htmlOutput.textContent = "";
141                  if (value.renderHTML) {
M 142                     const highlight =
143                         '<link rel="stylesheet" href="https://unpkg.com/@highlightjs/cdn-assets@11.9.1/styles/default.min.css">';
144                     htmlOutput.innerHTML = highlight + value.HTML;
145                     // Magic code to add code highlighting
146                     const blocks = htmlOutput.querySelectorAll("pre code");
147                     blocks.forEach((block) =>
148                         hljs.highlightElement(block as HTMLElement),
149                     );
150                     htmlOutput.style.whiteSpace = "normal"
151                 } else {
152                     htmlOutput.textContent = value.HTML;
153                     htmlOutput.style.whiteSpace = "pre-wrap"
154                 }
155             }
156         });
157 }
```

Main.hs


Assignment.hs M
BNF.md
MarkdownParser.hs M
Parser.hs
blockquotes.html M
Haskell > app > Main.hs > Main > main
getResult _ _
-- Magic code to convert key, value pairs to JSON to send back to the server
jsonResponse :: [(String, String)] -> ActionM ()
jsonResponse pairs =
json $ object [fromString key .= ((pack value) :: Text) | (key, value) <- pairs]
main :: IO ()
main = scotty 3000 $ do
post "/api/convertMD" $ do
requestBody <- body
-- Convert the raw request body from ByteString to Text
let requestBodyText = decodeUtf8 requestBody
-- Convert the Text to String
str = unpack requestBodyText
-- Parse the Markdown string using 'markdownParser' and apply 'convertAllHTML'
converted_html = getResult (parse parseDocument str) convertDocumentHTML
-- Respond with the converted HTML as JSON
jsonResponse [("html", converted_html)]
post "/api/saveHTML" $ do
requestBody <- body
-- Convert the raw request body from ByteString to Text
let requestBodyText = decodeUtf8 requestBody
-- Convert the Text to String
str = unpack requestBodyText
-- Parse the Markdown string using 'markdownParser' and apply 'convertAllHTML'
-- converted_html = getResult (parse parseDocument str) convertDocumentHTML
_ <- liftIO $ writeContents str
-- Respond with the converted HTML as JSON
jsonResponse [("status", "HTML saved")]