FUNCTIONAL PROGRAMMING CS-IS-2010-1 FINAL PROJECT REPORT

Haskell Scraper and Code-Text Separation

Name: Saptarishi Dhanuka

ID: 1020211525 Supervisor:

Partha Pratim Das Date: 13 May

2024 Ashoka University

Contents

1 Problem Statement and Requirements

Assigned Project Statement

Develop a scraper using Haskell to extract text and code snippets separately.

1. **Input**: Scrape the text and code snippets from the given text source
2. **Output**: A Word document containing the text and `.txt` file containing the code.
3. **Method**: Write the algorithm to scrape (you can use the `tagsoup` library) and all the input-output facilities using Haskell. Do not use any other language.

1.1 Requirements

1. The user shall be able to give any text source as input.
2. The scraper shall get all the code snippets of the source and write it into a Plaintext file.
3. The scraper shall get all non-code text of the source and write it into a Word Document.

2 Specifications and Analysis

2.1 Specifications

1. The user will be able to enter a text source as input, whose code and non-code parts they wish to be separated.
2. The scraper will parse the contents of the text and separate the code snippets from the rest of the text.
3. The scraper will output a `.docx` file containing the textual content.
4. The scraper will output a `.txt` file containing the code snippets.

2.2 Analysis

1. There are many ways of solving the problem both by syntactic and semantic approaches. Some semantic approaches are as follows:
   a) Lexical and semantic analysis with the use of regular expressions
   b) Using a large language model to differentiate the code and the rest of the text
   c) Using computer vision to attempt to read text like a human and identify text from the code
2. Analyzing the text source by considering not just it's underlying syntactical structure but also it's semantic meaning will add layers of complexity to the project.
3. Hence, we limit the scope of the project to text sources being HTML formatted web pages that have code snippets demarcated by particular tags. This gives us a more precise approach to the problem based on syntax and tag structure.
4. Consequently, we consider the text source to be a web page, supplied as input through a URL.
5. We also consider that the web page is formatted with HTML and that the code snippets are demarcated with certain tags that distinguish them from the rest of the textual content.
6. The separation of the text from the code will be carried out based on the HTML tags that differentiate them from each other.
7. For readability, the output files will preserve the ordering and formatting of the source content.

3 Design and Architecture

3.1 High-Level Architecture

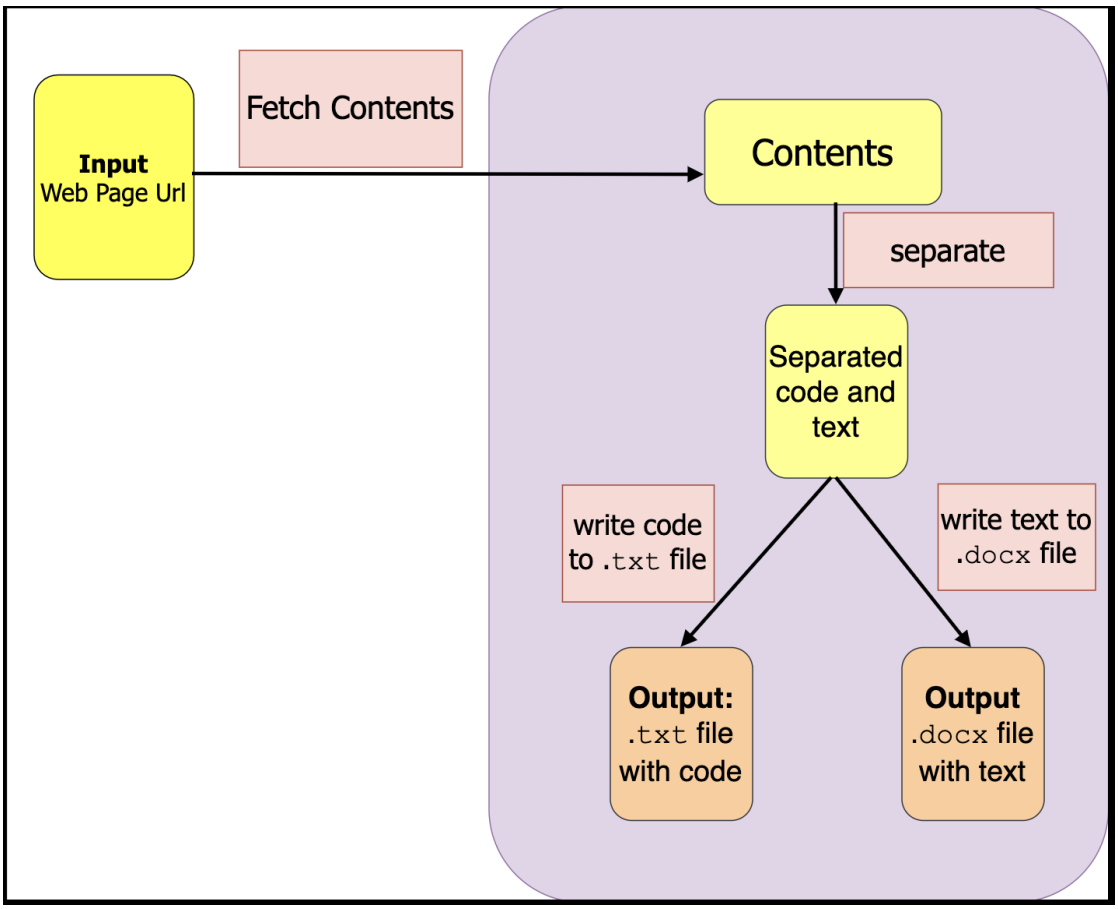

Figure 3.1: High-Level Architecture

The **high-level architecture** consists of the following:

1. Get the contents of the web page specified by the URL.
2. Separate the code snippets from the text.
3. Writing the code snippets into a `.txt` file.
4. Write the non-code textual content into a `.docx` file.

3.2 High-Level Design

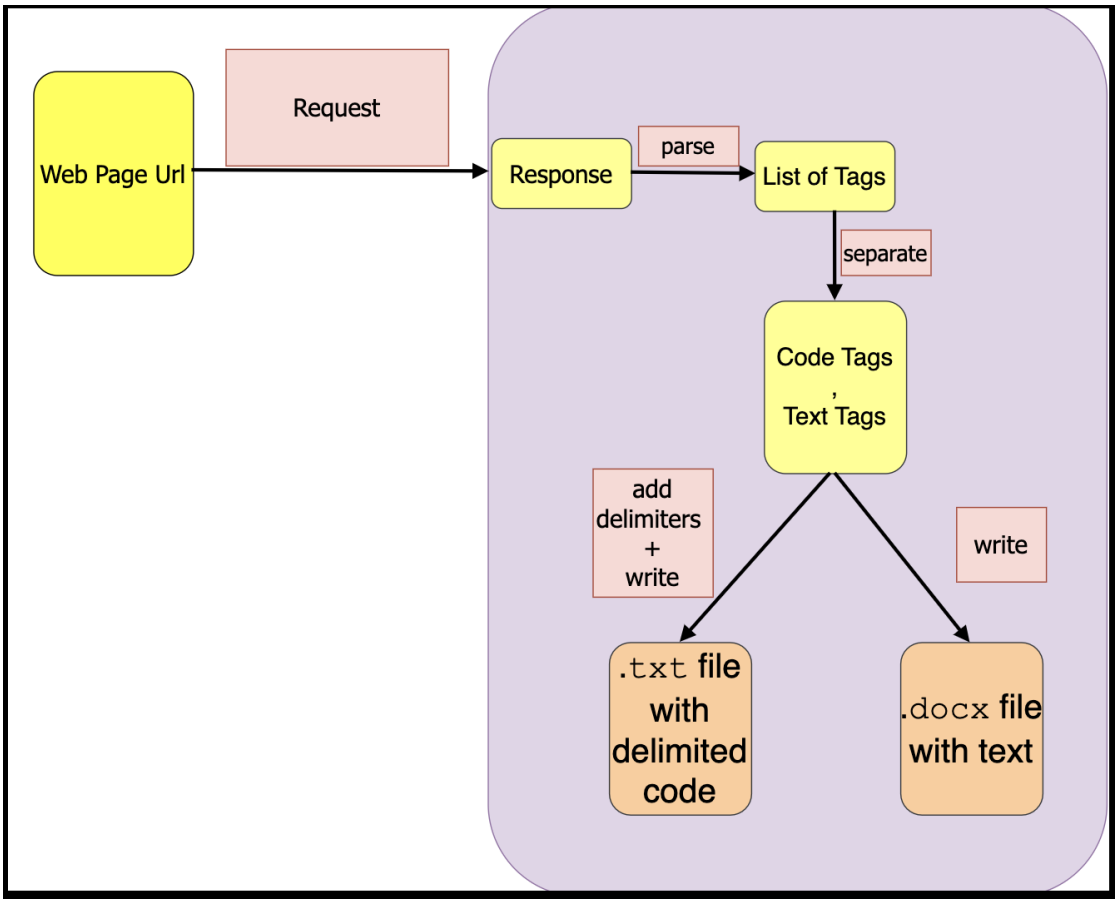

Figure 3.2: High-Level Design

A more detailed description of the **high-level design** which implements the architecture consists of the following:

1. Execute an HTTPS request for fetching the HTML contents of the web page
2. Get the body i.e. the HTML content from the response received after executing the request
3. Parse the HTML into a list of Tags according to the `tagsoup` library
4. Separate the Tags corresponding to the code from the Tags corresponding to the textual content. By inspecting the HTML, we can see that the code snippets are within `<pre>` tags, so we need to separate everything enclosed within these tags from the rest of the HTML content. We also remove images.
5. Insert delimiters between each different code snippet for formatting purposes.

6. Convert the Tags back to HTML
7. Write the code snippets to the `.txt` file
8. Write the non-code textual content to the `.docx` file

The lower level implementation details are mentioned in the prototype implementation section.

4 Tools and Languages

4.1 Languages

Only Haskell was used for the project as mentioned in the problem statement

4.2 Tools

1. The Glasgow Haskell Compiler (GHC) is used for compilation.
2. Stack is used as the build tool. This manages installing project dependencies, building and running the project and testing the project.
3. **Libraries**:
   a) `Network.HTTP.Client.TLS` was chosen for handling HTTPS connections in order to use the `newTlsManager` function
   b) `Network.HTTP.Client` was chosen for parsing the url into a request, executing the request with the manager, and getting the body from the response. The functions used were `parseRequest, httpLbs, responseBody` respectively.
   c) `Tagsoup` was used to parse the HTML into a list of Tag Strings, and then also to convert the separated Tag Strings back into an HTML-formatted string . It was also used in a helper function that inserted a delimiter between the code snippets. The functions used were `parseTags, isTagOpenName, isTagCloseName, renderTags`, along with the `TagText` constructor and `Tag` for `Tag String`.
   d) `Pandoc` was used to read the HTML-formatted strings for both the code and the text into intermediate Pandoc representation. Then it was used to write the content into another intermediate string-like representation, which in turn was written into the final files with a standard library. The functions used were `readHtml, writePlain, writeDocx, runIO`.
   e) `Data.ByteString.Lazy.Char8` was used to convert the ByteString obtained from the response body into a string for further processing. The function used was `unpack`
   f) `Data.ByteString.Lazy` was used to write the ByteString obtained from the `writeDocx` function from `pandoc`, into the final `.docx` file. The function used was `writeFile`

g) `Data.Text.Conversions` was used to convert the separated HTML strings into the `Text` type defined under `Data.Text`, so that it could be converted into intermediate pandoc representation by `readHtml`. The function used was `convertText`.

h) `Data.Text` was used in order to use the `Text` type

i) `Data.Text.IO` was used to write the Text obtained from the `writePlain` function from `pandoc`, into the final `.txt` file. The function used was `writeFile`

j) The standard `Prelude` was used for various operations.

5 Test Plan

5.1 Test Suite Outline

1. **Unit Testing :** The following functions can be tested
   a) `getHTML` :
      i. Test if the function returns the correct HTML for various URLs by using sample inputs and outputs.
      ii. Test if the function gracefully handles invalid URLs with error handling.
      iii. Test if the function handles network errors and HTTPS errors with error handling.
   b) `parseTheTags` :
      i. Test if the function returns the correct list of Tag Strings for various HTML content.
      ii. Test if the function handles malformed HTML and erroneous inputs with error handling.
   c) `fillTrue` :
      i. Test if the function correctly fills in True values for all elements within any two matching pair of True values corresponding to `<pre>` tags.
      ii. Test if the function handles edge cases with empty lists or single element lists.
   d) `insertNewLines` :
      i. Test if the function correctly adds the delimiter before every closing `<pre>` tag.
      ii. Test if the function handles cases where there are no `<pre>` tags, or the Tag String is malformed
   e) `separateTextCode` :
      i. Test if the function correctly returns a tuple of two lists where one list has the Tag Strings corresponding to all the content enclosed within `<pre>` tags, and that the other list has all the other elements, excluding images.
      ii. Test edge cases of no `<pre>` tags, nested `<pre>` tags
   f) `writeToTxt` : While File I/O cannot be unit-tested in the traditional sense, we can still manually test the file outputs.

i. Test if the function correctly writes the content of the input Tag String with delimters to the `.txt` file using sample input output

g) `writeToDocx :` While File I/O cannot be unit-tested in the traditional sense, we can still manually test the file outputs.

i. Test if the function correctly writes the content of the input Tag String to the `.docx` file using sample input output

2. **Performance Testing :**

a) Test the time taken for the full system to execute from start to finish

b) Test the resource consumption of the system

3. **Functional Testing :**
Test whether the program meets the requirements by using sample input URLs and sample output files

6 Prototype Implementation Details

6.1 Prototype File Structure

```
scraper
|-- app
|   | -- Main.hs (main driver code)
|-- src
|   |-- Lib.hs (implementation of functions)
|-- output_files
|   | -- final_code.txt (delimited code snippets)
|   | -- final_text.docx (textual content)
```

6.2 Prototype features and limitations

The prototype correctly identifies the code and text portions of the web page and writes them into the `.txt` and `.docx` files as per the requirements and specifications.

Limitations

1. The prototype has not been robustly tested, verified for correctness or structured to handle errors. However, this can be developed while making the final system since the prototype is already relatively modular.

2. Since the HTML structure of different web pages can vary, it is not necessary that this particular scraper will work for all web pages. It is designed specifically for the given page and may work for some other pages. But no generalisation can be made about the correctness of its text and code extraction for other pages.

3. Moreover, it will not necessarily work for web pages with malformed HTML or different structure.

4. It is not designed to be robust to design changes, which is in line with the rule stated on the `tagsoup` library's documentation example, since the website's HTML can change in unpredictable ways. If the site's HTML structure changes, for instance if the code snippets change from being enclosed in `<pre>` tags to `<code>` tags, then the scraper will not be able to separate out the code from the text and extract them accurately.

5. It clearly does not work with text sources that are not HTML formatted web pages.

6.3 Implementation Details

6.3.1 Main.hs

In-line with the design, architecture and choice of tools as mentioned above, the prototype obtains the HTML content of the url in line 2 and parses it into Tag Strings in line 3. Then we separate it into code tags and non-code textual tags in lines 4-6. Then we write the Tags to respective files in like 7-8.

The core structure of the `Main.hs` file is

```
1. let url = "https://eli.thegreenplace.net/2018/type-erasure-and-reification/"
2. response_html <- getHTML url
3. let parsed_tags = parseTheTags response_html
4. let separated_text_code = separateTextCode parsed_tags
5. let preTags             = fst separated_text_code
6. let nonPreTags          = snd separated_text_code
7. writeToTxt preTags
8. writeToDocx nonPreTags
```

6.3.2 Lib.hs

Imports and pragmas

```
1  {-# LANGUAGE OverloadedStrings #-}
2  import qualified Network.HTTP.Client as Client
3  import qualified Network.HTTP.Client.TLS as ClientTLS
4
5  import qualified Text.HTML.TagSoup as Soup
6  import Text.Pandoc
7
8  import qualified Data.Text.Conversions as TextConv
9  import qualified Data.Text as T
10 import qualified Data.Text.IO as TIO
11 import qualified Data.ByteString.Lazy as LBS
12 import qualified Data.ByteString.Lazy.Char8 as LBSC
13
```

getHTML

`getHTML` takes the url as input and sets up a new TLS manager with `newTlsManager`, parses the url with `parseRequest` and then executes the request with `httpLbs`.
Then it gets the HTML content from the response using `responseBody`.

```
getHTML :: String -> IO LBSC.ByteString
getHTML url = do
    mymanager <- ClientTLS.newTlsManager
    myrequest <- Client.parseRequest url
    response <- Client.httpLbs myrequest mymanager
    let response_html = Client.responseBody response
    return response_html

```

parseTheTags

`parseTheTags` takes the HTML ByteString and parses it into a list of Tag Strings using `parseTags` from Tagsoup.

```
parseTheTags :: LBSC.ByteString -> [Soup.Tag String]
parseTheTags response_html = (Soup.parseTags :: String -> [Soup.Tag
↪  String]) (LBSC.unpack response_html)
```

fillTrue

`fillTrue` is a helper function that fills in True for every element enclosed within a matching pair of True statements corresponding to an opening and closing `<pre>` tags

```
fillTrue :: [Bool] -> [Bool]
fillTrue [] = []
fillTrue [a] = [a]
fillTrue (False:False:xs) = False:fillTrue (False:xs)
fillTrue (False:True:xs) = False:fillTrue (True:xs)

-- fill True for elements after a <pre> tag starts
fillTrue (True:False:xs) = True:fillTrue (True:xs)

-- seeing 2 consecutive Trues denotes end of <pre> tag, so skip them
fillTrue (True:True:xs) = True:True:fillTrue (xs)
```

insertNewlines

`insertNewLines` is a helper function that inserts a TagText for a delimiter before every closing `<pre>` tag for formatting purposes

```haskell
insertNewlines :: [Soup.Tag String] -> [Soup.Tag String]
insertNewlines [] = []
insertNewlines (x:xs) = if (Soup.isTagCloseName "pre" x)
    then Soup.TagText
    ↪  "\n\n======================\n\n":x:insertNewlines (xs)
    else x:insertNewlines (xs)
```

separateTextCode

The idea of `separateTextCode` is as follows. Create a boolean list corresponding to every element of the list of Tag Strings. A boolean list element will be True if a Tag is a `<pre>` tag or within a `<pre>` tag. Then zip the boolean list with the original Tag list and get the Tags which have a correponding True value; this is the list of code tags.
Now mark elements to be true if the corresponding tags are `<img>` tags and create an overall boolean list which has false if the tag is not code and not an image. Extract the tags which have a false value; this is the list of non-code textual tags.
It essentially solves the parenthesization problem with the `<pre>` tags.

```haskell
separateTextCode :: [Soup.Tag String] -> ([Soup.Tag String], [Soup.Tag
↪  String])
separateTextCode parsed_tags =
    let bool_mapping_open = Prelude.map (Soup.isTagOpenName "pre")
    ↪  parsed_tags
        bool_mapping_close = Prelude.map (Soup.isTagCloseName "pre")
        ↪  parsed_tags

        -- do element-vise or of the two lists
        bool_mapping1 =  Prelude.zipWith (||) bool_mapping_open
        ↪  bool_mapping_close

        filled_true_pre = fillTrue bool_mapping1
        combined_pre = Prelude.zip filled_true_pre parsed_tags
        preTags = Prelude.map snd (Prelude.filter (fst) combined_pre)

        bool_mapping_img_open = Prelude.map (Soup.isTagOpenName "img")
        ↪  parsed_tags
        bool_mapping_img_close = Prelude.map (Soup.isTagCloseName
        ↪  "img") parsed_tags
        bool_mapping2 =  Prelude.zipWith (||) bool_mapping_img_open
        ↪  bool_mapping_img_close

        bool_mapping =  Prelude.zipWith (||) bool_mapping1
        ↪  bool_mapping2
```

```
18
19          filled_true = fillTrue bool_mapping
20          -- zip parsed_tags list with the boolean list and
            ↪  Prelude.filter elements of the first list based on the
            ↪  second list to create tuple list
21          combined = Prelude.zip filled_true parsed_tags
22          -- get tags of tuples from combined where the boolean is false
23          nonPreTags = Prelude.map snd (Prelude.filter (\(a,_) -> not a)
            ↪  combined)
24
25      in (preTags, nonPreTags)
26
27
```

Writers to files

`writeToTxt` takes the code Tags and renders them as an HTML formatted string after delimiting the snippets. Then the string is converted into the Text type, which is then fed into `readHtml` which converts it into an intermediate Pandoc format.
`writePlain` converts this pandoc into Text, which is finally written to the `.txt` file.

`writeToDocx` takes the text Tags and renders them as an HTML formatted string. Then the string is converted into the Text type, which is then fed into `readHtml` which converts it into an intermediate Pandoc format.
`writeDocx` converts this pandoc into a ByteString, which is finally written to the `.txt` file.

```
1  writeToTxt :: [Soup.Tag String] -> IO ()
2  writeToTxt preTags = do
3      let htmlPre = Soup.renderTags (insertNewlines preTags)
4      pandocPre <- runIO $ readHtml def ( TextConv.convertText (htmlPre
       ↪  :: String ) :: T.Text )
5
6      case pandocPre of
7          Right x -> do
8              y <- runIO $ writePlain def x
9              case y of
10                 Right direct_pan_pre -> do
11                     TIO.writeFile "output_files/final_code.txt"
                       ↪  direct_pan_pre
12
13                 Left err -> Prelude.putStrLn $ "Error with pandoc
                   ↪  writePlain: " ++ show err
14
```

```
15          Left err -> Prelude.putStrLn $ "Error parsing pandoc for pre
            ↪  tags: " ++ show err
16
17      putStrLn "Completed writing to txt"
18
19
20  writeToDocx :: [Soup.Tag String] -> IO ()
21  writeToDocx nonPreTags = do
22      let htmlNonPre = Soup.renderTags nonPreTags
23      pandocNoPre <- runIO $ readHtml def ( TextConv.convertText
        ↪  (htmlNonPre :: String ) :: T.Text )
24
25      case pandocNoPre of
26          Right x -> do
27              y <- runIO $ writeDocx def x
28              case y of
29                  Right direct_pan -> do
30                      LBS.writeFile "output_files/final_text.docx"
                        ↪  direct_pan
31
32                  Left err -> Prelude.putStrLn $ "Error with pandoc
                    ↪  writeDocx: " ++ show err
33
34          Left err -> Prelude.putStrLn $ "Error parsing pandoc for non
            ↪  pre tags: " ++ show err
35
36      putStrLn "Completed writing to docx"
37
```

7 Plan for Completion

- Implement error handling gracefully wherever possible so that no unhandled errors can occur.
- Implement the test suite
- Attempt to make the scraper more generalized so that it can work with other websites and other HTML structures.
- Document the code well
- Explore methods of separating the code from the non-code textual content that are independent of HTML structure. Such methods were mentioned in the analysis and implementation of these methods will make the scraper more generalizable.