FUNCTIONAL PROGRAMMING CS-IS-2010-1 MIDTERM EVALUATION PROJECT REPORT

Haskell Web Scraper

Saptarishi Dhanuka

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 Problem Description

The given web page is made of text and code snippets, which we need to scrape and extract separately into a `.docx` file containing the text portions and a `.txt` file which has the code snippets.
For this, we need to fetch the given web page, parse and analyze its HTML structure to identify the HTML tags of the code snippets and the tags of the rest of the text, so that we can effectively separate them into different documents.

1.2 Requirements

1. The scraper shall be written entirely in Haskell
2. The scraper shall get all text of the given page and write it into a `.docx` file.
3. The scraper shall get all the code snippets of the given page and write it into a `.txt` file.

2 Specifications

1. The scraper will use Haskell HTTP libraries for fetching the HTML content of the given web page.

2. The scraper will separate the code snippets from the rest of the textual content using an algorithm that utilizes the `tagsoup` library to parse the HTML content, along with other standard libraries for string and text handling.

3. The scraper will write the text into a Word document and the code snippets into a `.txt` file mainly using the `tagsoup` and `pandoc` libraries, along with some standard Haskell libraries.

4. The `.txt` file will be formatted such that the code snippets are visibly delimited for readability purposes.

5. The `.docx` file will be formatted in a manner similar to the original web page in terms of demarcating headings, footnotes and the order of the text.

3 Design and Architecture

3.1 High-Level Architecture

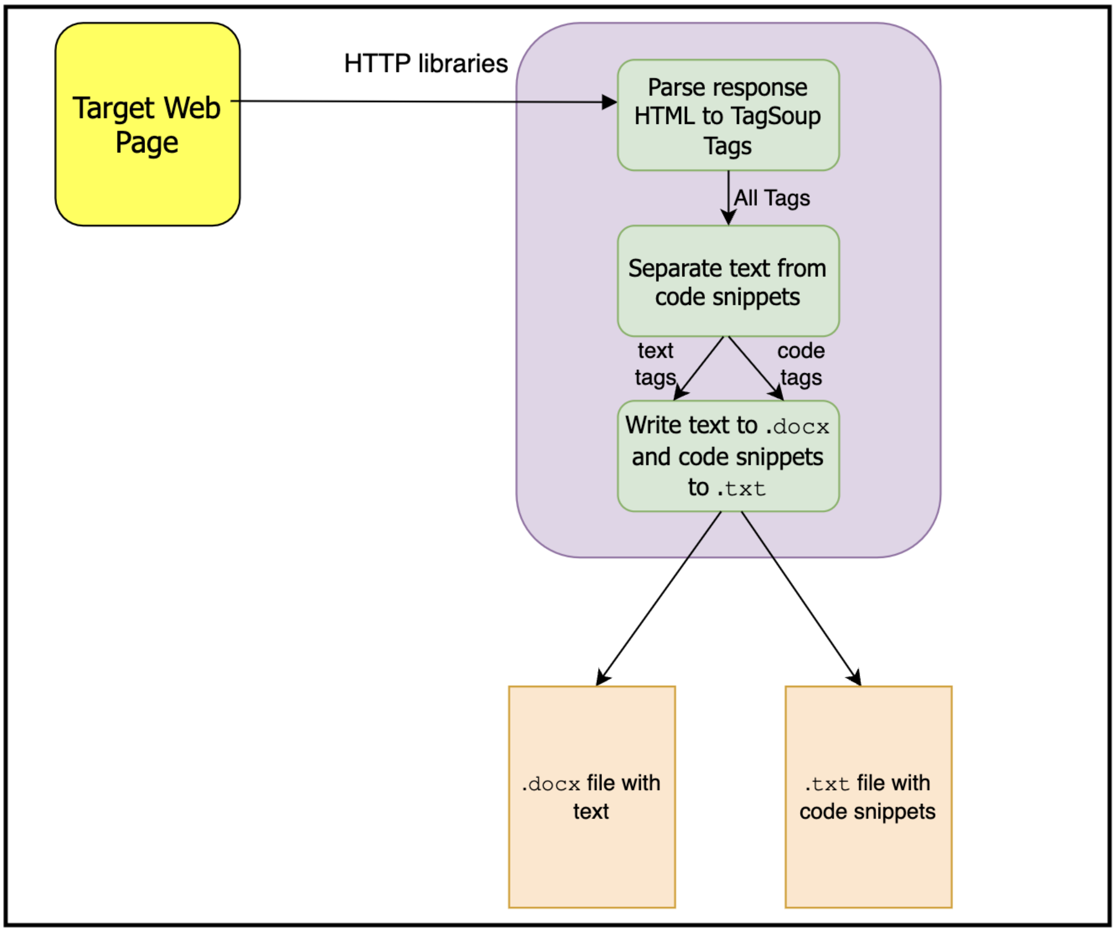

Figure 3.1: High-Level Architecture

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

1. Functionality for getting the target web page using HTTP libraries.
2. Parsing the response obtained from the HTTP libraries into Tags from the `tagsoup` library.

3. Separating the text from the code snippets using the descriptions of each Tag from the above Tags
4. Extracting the visible content from the text tags and writing them into a `.docx` file
5. Extracting the visible content from the code tags and writing them into a `.txt` file

3.2 High-Level Design

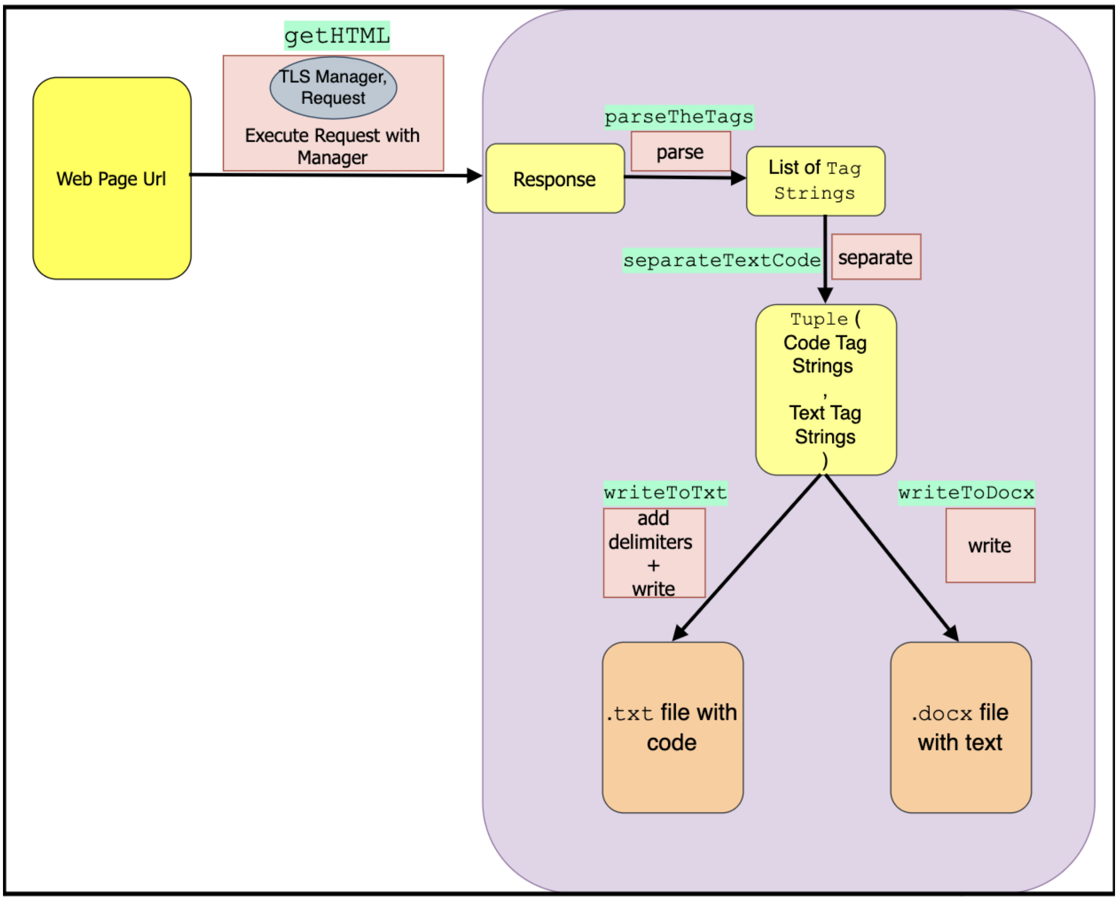

Figure 3.2: High-Level Design with functions in highlighted in cyan

The **high-level design** which implements the above architecture consists of the following:

1. A TLS manager for handling HTTPS requests, since the given URL is prefixed with `https`
2. Parsing the url into a request

3. Executing the request with the TLS manager
4. Get the body i.e. the HTML content from the response received after executing the request
5. Parse the HTML into a list of Tag Strings according to the `tagsoup` library
6. 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.
7. Insert delimiters between each `<pre>` tag for formatting purposes.
8. Convert the list of Tag Strings corresponding to the code and to the text each back into an HTML-formatted string, which we then convert into a `pandoc` document as intermediate representation
9. Convert the pandocs into another intermediate string-like format which can then be written into the respective `.docx` and `.txt` files

3.3 Low-Level design

The core low-level design that has been abstracted away from the above design and architecture is mentioned below

1. **Separation of text from code**

   Idea: 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, or if it's an `<img>` tag. Then zip the boolean list with the original list and divide into two lists based on the boolean value, then extract the Tag elements from each separate list.

   Steps:

```
1. Create list of booleans which have elements to be True if the
corresponding Tags from the Tag String list are <pre> tags
2. Make all those tags in between open and closing <pre> tags also true
3. Create a tuple which combines each Tag with their
corresponding Boolean value, then select those Tags who have a True value. T
4. Create list of booleans which have elements to be True if the
corresponding Tags from the Tag String list are <img> tags
5. Create overall boolean list whose elements are true if the
 corresponding tag is within a <pre> tag or is an <img> tag
6. Create a tuple which combines each Tag with their corresponding
```

```
Boolean value, then select those Tags who have a False value.
This is our list of text Tags
```

2. **Writing to .txt file**
   Steps:

```
1. Insert delimiter after every closing <pre> tag
2. Convert the Tag Strings back to HTML-formatted string
3. Convert into pandoc as intermediate representation, and
then further into Text with writePlain
4. Write this Text into the .txt file
```

3. **Writing to .docx file**
   Steps:

```
2. Convert the Tag Strings back to HTML-formatted string
3. Convert into pandoc as intermediate representation, and
then further into ByteString with writeDocx
4. Write this ByteString into the .docx file
```

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 Tools to be used for Testing

5.2 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.
      iii.
   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.

6 Prototype Implementation Details

7 Plan for Completion