Wiki

Sean Leather, adapted by Andres Löh

Difficulty: medium

Use the documentation of the following modules:

- Prelude
- Text.Html
- Network.URI
- Network.Shed.Httpd

Note that these are links in the PDF. You can click on anything with a light blue box around it to go to the webpage.

Use the file `Wiki.hs` linked from the wiki.

Writing Your Own Wiki

In this assignment, you will develop your own wiki. A wiki is a "website that allows the easy creation and editing of any number of interlinked web pages via a web browser" (from Wikipedia) and often uses a simplified markup language. First, you implement a collection of functions that combine to build a wiki *parser*, a function that translates a markup string into a collection of datatypes. Next, you implement an HTML generator to display the wiki datatypes. Lastly, you implement a web server that creates a webpage from wiki markup.

Wiki Parser

We use a wiki markup based on Creole, a standard format for many wiki implementations. The format has two forms of markup: *inline* and *block*. Block markup determines the horizontal sections of a page. This includes paragraphs, lists, and headers. Inline markup only works inside a block and includes bold, italics, and links.

The inline markup:

`//italics//`	*italics*
`**bold**`	**bold**
`Link to [[wikipage]]`	Link to wikipage
`[[URI|linkname]]`	linkname

Here is an example with formatted and unformatted text:

`**Today** is a //beautiful// day!`	**Today** is a *beautiful* day!

The block markup:

```
 * item 1
 * item 2
```

- item 1
- item 2

```
 # item 1
 # item 2
```

1. item 1
2. item 2

```
== Header
```

Header

All blocks start with a newline (indicated by the character `'\n'`). Text following a newline with no markup is simply a paragraph. A formatted block such as those above is restricted to a single line, but can be multiple lines.

In the associated `Practicum2.hs` file, we provide a collection of datatypes to represent the syntax of the wiki markup. (These datatypes are typically called an *abstract syntax tree* (AST) to distinguish it from the *concrete syntax* above.) In the following exercises, you will use and extend these datatypes as well as the code provided.

1. Before you get started with the actual exercises, you need to install a package that will be using (except if you have it already, such as on the Linux lab machines). The easiest way to do this is using `cabal`, a command-line tool for installing from Hackage, an online repository of Haskell packages.

If you have installed the Haskell Platform, you should already have `cabal` available just as you do `ghci`. If you do not have `cabal`, you will need to install it. Documentation for `cabal` is available at `http://haskell.org/haskellwiki/Cabal-Install`. Talk to a student assistant if you need help.

We want to install the package `httpd-shed`. To do so, we simply run the following at the command line `>`:

```
> cabal install httpd-shed
```

If it works, you will see a bunch of information about compiling and installing packages fly across your screen. If you can now load the file `Practicum2.hs` into `ghci` without seeing error messages, then your installation worked.

2. We have provided you with some code in `Practicum2.hs` to make your life easier. Familiarize yourself with the datatypes, inlinify, blockify, and mplus. You may wish to use and even modify these in later exercises, so it is important that you understand them. Try them out with different inputs in `ghci`.

3. Familiarize yourself with the functions readInline, readUnformattedInline, and readBold. Add top-level documentation to these functions, describing what they do, what kind of input they expect, etc.

Extend the datatype Inline with a constructor for italic formatting.

```
data Inline
  ...
  | Italic String
```

Define a function readItalic with the same type as readBold that parses the italic markup (see above) and uses the new Italic constructor. (Note that you can modify any functions you need to in order to do this.)

Modify readInline to add support for parsing italic markup.

4. Extend the datatype Inline with constructors for wiki and URI link formatting.

```
data Inline
  ...
  | WikiLink String
  | URILink { uriLinkRef :: String, uriLinkName :: String }
```

Define a function readFormattedInline that *generalizes* the functions readBold, readItalic, and the functionality for parsing wiki links and URI links. The readFormattedInline function should do everything necessary to parse the different markup formats except where they differ.

Modify readInline to use readFormattedInline instead of readBold and readItalic. Also add support for parsing wiki and URI links using readFormattedInline. Delete the functions readBold and readItalic since they are no longer necessary. (The grade of the previous exercise will be determined by parsing italic markup.)

Good programming style: The best definition for readFormattedInline is the one that captures everything except the differences between markup and leaves those for arguments. The best type is the most general, e.g. "as polymorphic as possible."

5. Define the function

```
readText :: String → Text
```

that converts a wiki markup string to a Text. It should support all of the inline markup that we have defined.

6. Familiarize yourself with the functions readBlock, readUnformattedBlock, and readBullet. Add top-level documentation to these functions, describing what they do, what kind of input they expect, etc.

Extend the datatype Block with constructors for header and numbered list item formatting.

```
data Block
  ...
  | Header Text
  | Number Text
```

Define a function readFormattedBlock that generalizes the parsing for all block formatting.

Modify readBlock to use readFormattedBlock for each of the block formats. Delete the function readBullet since it is no longer necessary.

7. Define the function

```
readPage :: String → Page
```

that converts a wiki markup string to a Page. It should support all of the inline and block markup that we have defined.

HTML Generator

Now that we have an AST for wiki markup, we need to convert it to HTML. HTML stands for *hypertext markup language*, and it is used to create webpages. If you have never seen it, open up your favorite web browser to your favorite webpage and choose the menu option resembling "View Source" or "Page Source." Or simply search the web for "HTML." There are thousands of tutorials.

For our work, we're going to use Text.Html, a module in the package `html`, included with the Haskell Platform. It has a large collection of functions for building webpages, many of which are based on the tags of HTML. For example, given the value

```
exampleHtml = thehtml << body << p << bold << "Hello, World!"
```

in `Practicum2.hs`, when type exampleHtml in `ghci`, we would see the output

```
<HTML>
   <BODY>
      <P>
         <B>
            Hello, World!
         </B>
      </P>
   </BODY>
</HTML>
```

This represents a full HTML page with a body containing a paragraph block with a single bolded string. (Note that indentation and nesting are not necessary, but make it easier to read.)

If you are not familiar with HTML, you should read a basic tutorial. Odds are that you will see it again outside this class.

8. Read through that Haddock docs for Text.Html to get familiar with the library. It is not well documented, but many of the functions correspond to entities in HTML. Also, experiment with the various forms of transforming HTML to String.

9. Define the functions

```
inlineToHtml :: Inline → Html
textToHtml   :: Text → Html
```

that convert Inline and Text values, respectively, to Html values.

10. Define the functions

```
blockToHtml :: Block → Html
pageToHtml :: Page → Html
```

that convert Block and Page values, respectively, to Html values.

For this exercise, you will need to solve the mismatch between the wiki representation of lists and the HTML representation. In the wiki markup, we have lines of markup, e.g.

```
* item 1
* item 2
```

but in HTML, this type of list is first wrapped with an *unordered list* tag.

```
<UL>
   <LI>
      item 1
   </LI>
   <LI>
      item 2
   </LI>
</UL>
```

There are numerous ways of solving this problem. One possibility is to extend the Block datatype with constructors for bulleted and numbered (*ordered* in HTML) lists of Text. Then, you can write a function that collects all the consecutive bullet (number) items in a Page and transforms them into single bulleted (numbered) lists. This function can be written using foldr and tuples.

Web Server

We can now read wiki markup and print HTML. Let's put our work online!

In this section, we will define web servers. While a webpage is often written using HTML, the connection between your browser and a server happens over HTTP, the *hypertext transfer protocol*, a very simple text-based protocol that supports requests and responses. A request comes from the browser and has a form like this:

```
GET /index.html HTTP/1.1
User-Agent: Mozilla/5.0
Accept: text/html
Accept-Language: en-us
Accept-Charset: ISO-8859-1,utf-8
```

This says the the browser wants the URI on the reciever called `/index.html` using the protocol `HTTP/1.1` and provides some extra information to help the server know what to send back. This URI may be a file, but it does not have to be.

The server then sends back a response. Here is a successful one:

```
HTTP/1.x 200 OK
Date: Tue, 01 Dec 2009 07:51:29 GMT
Server: Apache
Content-Type: text/html; charset=UTF-8

<html>...
```

The server is telling the browser which protocol it is using and that everything is OK (status code 200). The content follows the header (where the `<html>` starts) and has the type `text/html`.

There are two different methods that we will use. The one above, `GET`, is the method your browser uses when you click on a link or type the URI in the location bar. Another, `POST`, occurs when you click a button on a form. The primary difference is that `GET` only sends a location, e.g. `/index.html`, while `POST` sends that plus some content in the body of the message. The `POST` content is encoded as `application/x-www-form-urlencoded` which is described here.

You do not need to be extremely familiar with HTTP for this exercise, but it is good to know what's happening behind the scenes. If you ever wonder about something, use your favorite search engine to look it up. Plenty of information about HTTP is freely available online.

11. Read through the documention of `httpd-shed`. This package hides all the difficult stuff and lets us write a server as function of type Request → IO Response.

Try out the code included in the file `Practicum2.hs`. There is a simple server that prints the Request value as text. There are some other values provided for your convenience in working with the next few sections. After running initMyServer simpleServer, you can send your browser to the location `http://localhost:7734/` to see the result.

12. Define the function

```
fileServer :: Request → IO Response
```

that can be used with initMyServer to load files in the current directory. Thus, if you browse to the location `http://localhost:7734/myFile.txt` and `myFile.txt` is available in the current working directory, your server will display the contents in your browser.

To be safe and prevent mischievous hackers from seeing all your data, only load text, HTML, and wiki files. If the requested file ends with `.txt`, respond with a content type of `text/plain` and the contents of the file. If the request ends with the suffix `.html`, respond with the content type of `text/html` and the file contents. If the request ends with `.wiki`, respond with the content type of `text/html` and the file contents rendered from wiki markup to HTML. Otherwise, respond with the 404 status code and a message describing the problem.

What happens if you try to read a file that's not there? How do you know it's not there? Look into handling exceptions in Haskell and fix fileServer so that even if a file is not there, you still respond with the 404 status code and message.

13. Define the function

```
wikiFormServer :: Request → IO Response
```

by copying fileServer and adding support for editing and displaying a wiki page.

The URI `/wiki/edit` provides an HTML form (see `Practicum2.hs` for an example and search for tutorials) that accepts wiki markup. On clicking a "Submit" button, the server processes the text and returns a new page `/wiki/show` with the HTML.

The form should have a large text area (not the default). It must also use the `POST` method, so that the text does not show up in the URI. You can extract the fields from the body of the request using queryToArguments. You should handle spaces, but do not worry about reserved or non-alphanumeric characters.

For all other URIs, the wikiFormServer should do the same thing as fileServer.