Screen-scraping
Informatics 1 – Functional Programming: Tutorial 4

Due: The tutorial of week 6 (22/23 Oct.)

Please attempt the entire worksheet in advance of the tutorial, and bring with you all work, including (if a computer is involved) printouts of code and test results. Tutorials cannot function properly unless you do the work in advance.

You may work with others, but you must understand the work; you can't phone a friend during the exam.

Assessment is formative, meaning that marks from coursework do not contribute to the final mark. But coursework is not optional. If you do not do the coursework you are unlikely to pass the exams.

Attendance at tutorials is obligatory; please let your tutor know if you cannot attend.

Basic Screen Scraper

A "screen scraper" is a tool used to extract data from web sites, by looking at their source. In this exercise, you will write one of the most hated screen scrapers: one that extracts email addresses. Why is it hated? Because people use screen scrapers like that to collect email addresses to send spam to. However, in this exercise we will show you a *useful* purpose of the email screenscraper!

We are going to be extracting names and emails from web pages written in HTML (HyperText Markup Language). For instance, from the following HTML:

```
<html>
  <head>
     <title>FP: Tutorial 4</title>
  </head>
  <body>
     <h1>A Boring test page</h1>
     <h2>for tutorial 4</h2>
     <a href="http://www.inf.ed.ac.uk/teaching/courses/inf1/fp/">FP Website</a><br>
     <b>Lecturer:</b> <a href="mailto:dts@inf.ed.ac.uk">Don Sannella</a><br>
     <b>TA:</b> <a href="mailto:m.k.lehtinen@sms.ed.ac.uk ">Karoliina Lehtinen</a>
   </body>
</html>
```

We are going to extract a list of the "`<a>`" elements, which contain URLs (Uniform Resource Locators). If a URL begins with `http:` it is an address of a web page; if it begins with `mailto:` the rest of it is an email address. For the document above, here is the list of links (each one contains some extra data at the end, which is an artifact of the technique we use):

```
["http://www.inf.ed.ac.uk/teaching/courses/inf1/fp/\">FP Website</a><br><b>Lecturer:</b> "
```

```
,"mailto:dts@inf.ed.ac.uk\">\Don Sannella</a><br><b>TA:</b> "
,"mailto:m.k.lehtinen@sms.ed.ac.uk\">Karoliina Lehtinen</a></body></html>"]
```

From this list, we will in turn extract a list of names and email addresses:

```
[("Don Sannella","dts@inf.ed.ac.uk"),
 ("Karoliina Lehtinen","m.k.lehtinen@ed.ac.uk")]
```

The file `tutorial4.hs` contains the test html-document and the lists above: `testHTML`, `testLinks`, and `testAddrBook`.

Notice that the type of `testLinks` is `[Link]` and the type of `testAddrBook` is `[(Name,Email)]`. In other words: `testLinks` is a list of `Link`s, and `testAddrBook` is a list of tuples containing both a `Name:` and an `Email`. These appear to be new types which we have not encountered before, but if you look in the file `tutorial4.hs` you will find the following type expressions:

```
type Link = String
type Name = String
type Email = String
type HTML = String
type URL = String
```

These type declarations simply define aliases for the very familiar type `String`. Aliases are not strictly necessary, but they make your program more readable.

Note: If you want to know more about HTML, have a look at: `http://www.w3schools.com/html/`.

Exercises

1. Write a function `sameString :: String -> String -> Bool` that returns `True` when two strings are the same, but ignores whether a letter is in upper- or lowercase. For example:

   ```
   *Main> sameString "HeLLo" "HElLo"
   True
   *Main> sameString "Hello" "Hi there"
   False
   ```

   Warning: Unintuitively, the mapping between upper and lower case characters is not one-to-one. For example, the greek letter μ and the micro sign map to the same upper case letter. What does your code do on `sameString "\181" "\956"`? In this case either behaviour is acceptable, as long as the tests don't fail on input containing these characters!

2. Write a function `prefix :: String -> String -> Bool` that checks whether the first string is a prefix of the second, like the library function `isPrefixOf` that you used before, but this time it should be case-insensitive.

   ```
   *Main> prefix "bc" "abCDE"
   False
   *Main> prefix "Bc" "bCDE"
   True
   ```

   Check your function using the predefined test property `prop_prefix`.

3. (a) Write the function `contains` as in tutorial 2, but case-insensitive. For example:

   ```
   *Main> contains "abcde" "bd"
   False
   *Main> contains "abCDe" "Bc"
   True
   ```

(b) Write a test property `prop_contains :: String -> Int -> Int -> Bool` to test your `contains` function. You can take inspiration from `prop_prefix`.

4. (a) Write a case-insensitive function `takeUntil :: String -> String -> String` that returns the contents of the second string *before* the first occurrence of the first string. If the second string does not contain the first as a substring, return the whole string. E.g.:

```
*Main> takeUntil "cd" "abcdef"
"ab"
```

(b) Write a case-insensitive function `dropUntil :: String -> String -> String` that returns the contents of the second string *after* the first occurrence of the first string. If the second string does not contain the first as a substring, return the empty string. E.g.:

```
*Main> dropUntil "cd" "abcdef"
"ef"
```

5. (a) Write a case-insensitive function `split :: String -> String -> [String]` that divides the second argument at every occurrence of the first, returning the results as a list. The result should not include the separator. For example:

```
*Main> split "," "comma,separated,string"
["comma","separated","string"]
*Main> split "the" "to thE WINNER the spoils!"
["to "," WINNER "," spoils!"]
*Main> split "end" "this is not the end"
["this is not the ",""]
```

Your function should return an error if the first argument, the separator string, is an empty list. You will find your functions `takeUntil` and `dropUntil` useful here.

(b) Write a function `reconstruct :: String -> [String] -> String` that reverses the result of `split`. That is, it should take a string and a list of strings, and put the list of strings back together into one string, with the first string everywhere in between (but not at the start or at the end).

(c) Look at the predefined test function `prop_split` and explain what it does. Use it to test your `split` function.

6. Use your function `split` to write a function `linksFromHTML :: HTML -> [Link]`. You can assume that a link begins with the string `<a href="`. Don't include this separator in the results, and don't include the stuff in the HTML that precedes the first link. Example:

```
*Main> linksFromHTML testHTML
["http://www.inf.ed.ac.uk/teaching/courses/inf1/fp/\">FP Website</a><br><b>Lecturer:</b> ",
 "mailto:dts@inf.ed.ac.uk\">\Don Sannella</a><br><b>TA:</b> ",
 "mailto:m.k.lehtinen@sms.ed.ac.uk\">Karoliina Lehtinen</a></body></html>"]
```

Note: to include the character " in a string, precede it with a backslash (\), as \".

Use `testLinksFromHTML` to test your function on the given sample data. Note that this test does not require QuickCheck, since it does not depend on randomly generated input.

7. Write a function `takeEmails :: [Link] -> [Link]` which takes just the email addresses from a list of links given by `linksFromHTML`. Example:

```
*Main> takeEmails testLinks
["mailto:dts@inf.ed.ac.uk\">\Don Sannella</a><br><b>TA:</b> ",
 "mailto:m.k.lehtinen@ed.ac.uk\">Karoliina Lehtinen</a></body></html>"]
```

8. Write a function `link2pair :: Link -> (Name, Email)` which converts a mailto link into a pair consisting of a name and the corresponding email address. The name is the part of the link between the `<a href="...">` and `</a>` tags; the email address is the part in the quotes after `mailto:`. Add an appropriate error message if the link isn't a `mailto:` link. Example:

```
*Main> link2pair "mailto:john@smith.co.uk\">John</a>"
("John","john@smith.co.uk")
```

9. Combine your functions `linksFromHTML`, `takeEmails` and `link2pair` to write a function `emailsFromHTML :: HTML -> [(Name, Email)]` that extracts all mailto links from a webpage, turns them into `(Name, Email)` pairs, and then removes duplicates from that list. Example:

   ```
   *Main> emailsFromHTML testHTML
   [("Don Sannella","dts@inf.ed.ac.uk"),
    ("Karoliina Lehtinen","m.k.lehtinen@sms.ed.ac.uk")]
   ```

   Note: the library function `nub :: [a] -> [a]` removes duplicates from a list.

   You can test your function with `testEmailsFromHTML`.

Pulling in live URLs

In `tutorial4.hs` a test URL is predefined, `testURL`. Since it is just a string, you can ask GHCi to display it. Do this, and copy-paste the link into your web browser to see what page it refers to. To see the HTML of the page right-click and select 'view page source', or a similar option depending on your browser.

```
*Main> testURL
"http://www.inf.ed.ac.uk/teaching/courses/inf1/fp/testpage.html"
```

The function `emailsFromURL`, which is already defined in `tutorial4.hs`, extracts email addresses from a URL using your very own `emailsFromHTML`. Test your function `emailsFromHTML` by testing it on real URLs of your choice.

As you will have seen, `emailsFromURL` sometimes produces a rather long list of names and email addresses. Sometimes you have a vague idea of who it is you are looking for and in that case, you do not want to go through the entire list of names one-by-one. Over the next few exercises you will be implementing a function `emailsByNameFromURL` in order to find the email address of a person whose name you know.

Exercises

10. Write a function `findEmail :: Name -> [(Name,Email)] -> [(Name,Email)]` which given (part of) a name and a list of `(Name,Email)` pairs, returns a list of those pairs which match the name. Example:

    ```
    *Main> findEmail "Karoliina" testAddrBook
    [("Karoliina Lehtinen","m.k.lehtinen@sms.ed.ac.uk")]
    *Main> findEmail "San" testAddrBook
    [("Don Sannella","dts@inf.ed.ac.uk")]
    *Main> findEmail "Fred" testAddrBook
    []
    ```

11. Define the function `emailsByNameFromHTML :: HTML -> Name -> [(Name, Email)]`. This function should take an HTML string and (part of) a name, and return all `(Name,Email)` pairs which match the name.

    ```
    *Main> emailsByNameFromHTML testHTML "Karoliina"
    [("Karoliina Lehtinen","m.k.lehtinen@sms.ed.ac.uk")]
    ```

The function `emailsByNameFromURL`, which is already defined in `tutorial4.hs`, uses your very own `emailsByNameFromHTML` function to extract the email address of a certain person from a live URL. Maybe you can try it on your own webpage, if you have one.

Optional Material

Searching for strings

In the previous section you have written functions to find email addresses which belong to people whose name contains the input string. You will now write code to select names which match more elaborate criteria.

Exercises

12. Write a function `hasInitials :: String -> Name -> Bool` which returns true if the initials of the second argument are exactly the first argument.

    ```
    *Main> hasInitials "DS" "Don Sannella"
    True
    *Main> hasInitials "MKL" "Karoliina Lehtinen"
    False
    ```

13. Write a function `emailsByMatchFromHTML :: (Name -> Bool) -> HTML -> [(Name,Email)]` . It should find all the emails belong to people whose name match the criterion set out by the first argument. Note the type of the first argument of this function (the brackets are important!).

    Then write a funciton `emailsByInitialsFromHTML :: String -> HTML -> [(Name,Email)]` which finds emails of people whose initials match the first argument.

14. Write a function `myCriteria :: Name -> Bool` which tests whether a name matches a criterion of your choice. If you are stuck for ideas, match names of which the initials contain a reference string, in the right order but not necessarily in consecutive positions. For example "Don T. Sannella" matches "DS". You may want your function to take more than one argument, in which case you can adjust its type. Use this function and the previous ones to write `emailsByMyCriteriaFromHTML :: HTML -> [(Name,Email)]` which finds emails belonging to people whose names match your criterion.

Pretty printing

We often want to look at the output of a function (say `emailsFromHTML`) in a slightly nicer way. This is called *pretty printing*. In `emailsFromURL` the output of `emailsFromHTML` is currently being pretty printed by a function called `ppAddrBook`. In this exercise, you will be rewriting that function to make `emailsFromURL` produce a different output.

You will need two pieces of information to complete this exercise. First of all, you may assume that if a name has more than two words, the first name is the first word and the last name is the remaining words[1]. Second, all of the names should line up and all of the email addresses should line up—no matter how long the names are. For example:

```
Lehtinen, Karoliina      m.k.lehtinen@sms.ed.ac.uk
Sannella, Don            dts@inf.ed.ac.uk
```

In order to print a block of text like this to the screen, we can't simply return it from a function, because GHCi will faithfully escape all the funny characters in the string, such as newlines. The function `putStr` takes a string and prints it to the screen, which involves turning newline characters `'\n'` into actual new lines. For example:

```
*Main> putStr "First Line\nSecond Line\nThird Line\n"
First Line
```

[1] Note that this is the way the British classification system works, but that it does not provide a correct classification for many non-English names.

```
Second Line
Third Line
```

Exercises

15. Rewrite the function `ppAddrBook :: [(Name,Email)] -> String` so that it lines up the names and email addresses in two separate columns. For example:

    ```
    *Main> putStr (ppAddrBook testAddrBook)
    Sannella, Don        dts@inf.ed.ac.uk
    Lehtinen, Karoliina  m.k.lehtinen@sms.ed.ac.uk
    ```

    You will find, in general, that some names are listed in "surname, first name" format and some are given in the regular "first name surname" format. Make sure your function can cope with both formats.