Crawling the web with Haskell

Project report for the course Program Design and Data Structures (1DL201)
at Uppsala University on the spring of 2015.

Group 11

Kristofer Sundequist Blomdahl, KandDv 2014
Sahand Shamal Taher, KandDv 2014
Sebastian Rasimus, KandDv 2014

Mars 5, 2015

Table of contents

1 Introduction and purpose

The world wide web is essentially a graph of HTML-documents linking to one another, a web graph. For different reason there is often an interest in extracting large amounts of data from parts of the web. A typical way to do this is by using a web crawler.

The aim of this project is to create a program that will crawl the web starting from a specific web page and allow the user to analyze gathered data to produce statistics of different types. The program will allow for crawling internal links exclusively or including external ones.

1.1 Summary

This program crawls the web starting from a specific web page. The user is able to define and input their own functions to create statistics from the data gathered by the program. It is also possible for the user to choose whether to only crawl internal links or to include external ones as well.

The program uses an algorithm that traverses the web graph (the web) using Breadth First Search (BFS). Each website is fetched and parsed with the help of external libraries. User defined functions are then applied to the parsed websites which creates the output statistics. These are returned to the user and can be summarized through a summarization function if desired.

2 Use cases

There are several functions that allow users to crawl pages:

- **basicCrawl** function is an easy way to start crawling the web without providing too many arguments
- **basicCrawlSum** is a similar function that runs basicCrawl and immediately extracts the stats from the results.
- **crawlGen** allows a customizable way to crawl.

2.1 Basic web crawl: basicCrawl

This function allows an easy way to crawl the web without providing a lot of arguments. The function calls a more generic function, *crawlGen*, and simply passes input arguments along with some initial value to it. Read the section on *crawlGen* for more info. The specifications for basicCrawl are as follows:

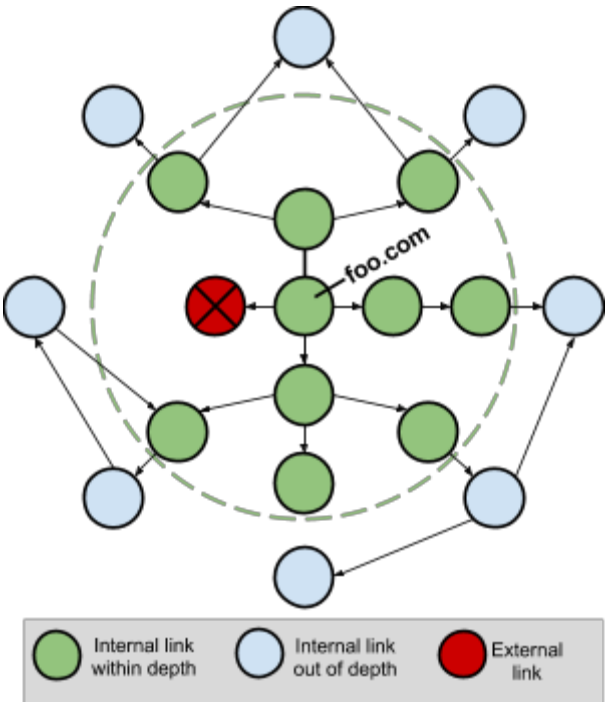
```
basicCrawl :: Url -> Int -> Bool -> IO (G.Graph URI P.Key (String, Int), S.Searched)
```

- Url is the URL to search as a string
- Arg 2 (Int): maximum recursive depth during search
- Arg 3 (Bool): True means external pages will be included in the crawl, False means only internal pages will be crawled.

Example:

```
basicCrawl “http://www.foo.com/” 2 False
```

This returns the result after crawling http://www.foo.com/ along with adjacent sites of two levels as well as a list of the URLs searched.



2.2 Crawling and retrieving a summary: basicCrawlSum

This function is another easy way to crawling the web. It's like basicCrawl with the difference that it only returns the sum of the statistics gathered from the crawl.

```
basicCrawlSum :: Url -> Int  -> Bool -> IO [(String, Int)].
```

Example:

```
basicCrawlSum "http://www.lul.se/" 1 False
```

might return something like:

```
[("Link count:",694)]
```

2.3 Custom crawling: crawlGen

The main feature of our program is that you can give the crawler your own functions to apply to the websites along with choosing how to fetch the websites (one can fetch data from other protocols than the www). This is the specification for crawlGen:

```
crawlGen :: (Show statType) => Url -> Int-> Bool -> HtmlFetcher -> [St.StatFn
statType] -> IO (G.Graph URI P.Key statType, S.Searched)
```

Where:

- Url: the URL to search
- Arg 2 (Int): maximum recursive depth during search
- Arg 3 (Bool): True means external pages will be included in the crawl, False means only internal pages will be crawled.
- *HtmlFetcher*: A function which fetches the HTML data. Use the premade *getHtml* to crawl the internet or provide a different IO function to fetch HTML. The fact that a custom HtmlFetcher can be supplied means crawling can be done over any protocol so long as the result is returned as a string. Read more in the section about HtmlFetcher.
- [*St.StatFn* statType]: List of statistical functions to run on data. Read more in the section about StatFn.

Examples:

```
crawlGen "http://www.uu.se/" 2 False getHtml [statFn1]
```

This will crawl "http://www.uu.se/" up to depth 2 internally and apply statFn1 on those sites.

HtmlFetcher

HtmlFetcher represents a function that given a URL, returns the data from the corresponding node (a website for example) in a graph (the internet for example).

```
type HtmlFetcher = (Url -> IO String)
```

An example of a supplied HtmlFetcher which works against the web (relies on *Network.HTTP* and *Network*, read more about this in the section on external libraries):

```
getHtml url = withSocketsDo
$ simpleHTTP (getRequest url) >>= getResponseBody
```

An example of a HtmlFetcher that does not work against the web (testData is a simple list of data):

```
getTestHtml testData url = do
return $ snd $ head (filter (\(a, b) -> a == url) testData)
```

Statistics.StatFn

The module Statistics contains StatFn and sample functions. The statistical functions consist of a tuple of two subfunctions, a "statCreator" and a "combiner". The statCreators are constrained to take a tagList (TagSoup's format after parsing HTML) and all have to return the same type. The "combiners" must combines two statistics created by their associated statCreator. This is the specification for StatFn in the Statistics module:

```
data StatFn statType = StatFn ( (P.TagList -> statType), (statType -> statType ->
statType)).
```

Example of the link counting function in our sample set:

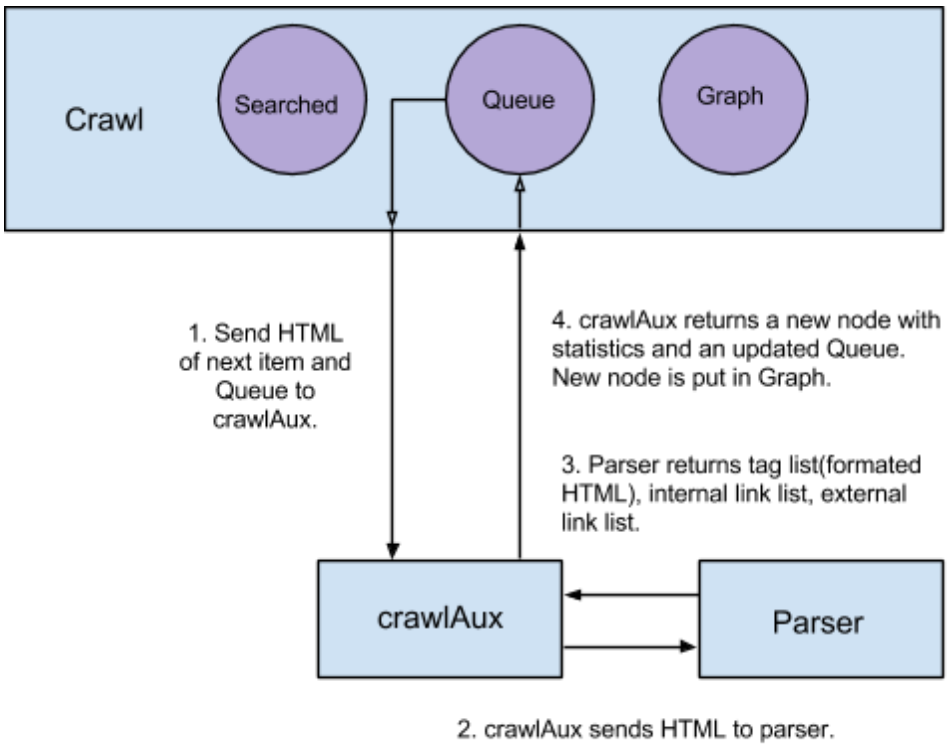
```
statFn1 :: StatFn (String, Int)
statFn1 = StatFn  (fnA, fnB)
      where
            fnA tagList = ("Link count:", length $ P.getUrls tagList)
            fnB a b = (fst a, snd a + snd b)
```

3 Documentation

In this section we will talk about how our program works. We will talk about our data structures, functions and the general control flow of our program.

3.1 Control flow

Crawl is the main function of the program. While crawling, a Searched data structure is used to keep track of visited URLs. Crawl uses a pure auxiliary function (crawlAux) to process the HTML of each URL, this includes applying user defined statistical functions to it. The result after the processing is added to a Graph data structure. The crawlAux function takes help from the Parser module to deal with parsing-related tasks. Here is a rough control flow illustration:

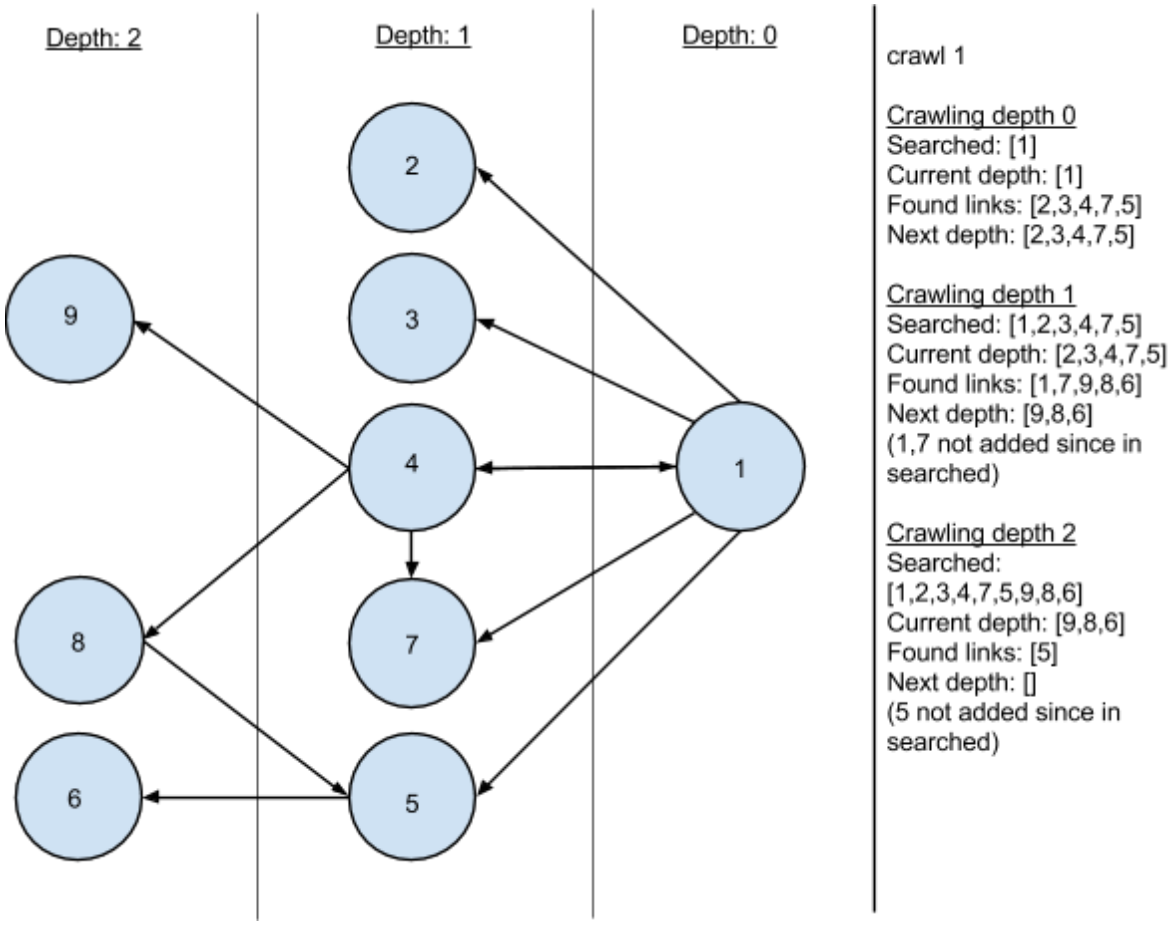


3.2 Algorithms

This section covers the main algorithms of the program. Many of the choices we made regarding algorithms were motivated by efficiency in the pure part of the program. Note that the main bottleneck in the program still lies in the impure part of the program, specifically when a page is downloaded from the web. Another possible bottleneck is in the statistical summarization, this depends on the user-defined functions.

3.2.1 Crawling

The crawling algorithm is basically a Breadth First Search (BFS) of a graph. The crawl starts at a node and creates a queue based on the links found in the HTML of each node (adjacency lists). The program then recursively searches all nodes in the queue. While doing this a separate list of visited nodes is maintained to prevent revisiting the same node. Since the same link can be found at many different depths, BFS guarantees that each node is visited as early as possible (at the highest depth it exists in the traversal tree).



3.2.2 Enqueue/Dequeue

This is a queue with two separate lists to maintain the items. When things are added (enqueued) to the queue they are inserted into the enqueue list in reversed order. The second list (the dequeue list) holds the next items in the queue.

The motivation behind this system is that it reduces the time complexity of queueing. Instead of concatenating new items to the queue each time, they are simply added to the beginning of the enqueue list. Then, once the dequeue list is empty, the enqueue list is reversed and inserted into the dequeue list.

3.3 Data Structures

This section covers how our data structures work and what they represent. There are three main data structures, Queue, Graph and Searched, which are used throughout the crawling process.

3.3.1 Queue

This is an abstract data structure implementing a queue with two list enqueue/dequeue system. Using a queue makes it easy to implement a Breadth First Search using it since it's Last in First Out by nature.

The motivation for using a two list queue (enqueue/dequeue) is mainly that it allows the program to avoid an excessive amount of concatenations when crawling a large amount of nodes. As a bonus the system makes it easy to keep track of which depth the crawler is currently at (see dequeue, dequeueIsEmpty and enqueueIsEmpty in the interface section).

The following functions represent the interface for Queue:

`Queue :: Q ([a], [a])`
The first list (enqueue) is a list of items to be added to the second list (dequeue). The second list (dequeue) is a list of items that are next in line. If both lists are empty, the queue is empty.

`empty :: Queue a`
Returns an empty queue

`getHead :: Queue a -> a`
Returns the head of the dequeue list.

`dequeue :: Queue a -> Queue a`
Returns the same queue but with the head of the dequeue list removed. If the dequeue list is empty, it returns the dequeue of a normalized version of the (using *normalize*), in the crawler this means that a new depth of links are now in the dequeue list. A precondition for using this function is that the queue isn't empty (see *queueIsEmpty*).

Examples:
```
dequeue (Q ([1], [2,3])) == Q ([1],[3])
dequeue (Q ([3,4], []))  == dequeue (Q ([], [4,3]))
dequeue (Q ([], [4,3]))  == Q ([],[3])
```

`normalize :: Queue a -> Queue a`
If the dequeue list is empty, this function replaces the dequeue list with the reversal of the enqueue list, empties the enqueue list and returns the result. If the dequeue list is not empty the queue is returned as it came in.

Examples:
```
normalize (Q ([3,2], [1])) == (Q ([3,2], [1]))
normalize (Q ([3,2], []))  == (Q ([], [2,3]))
```

`queueIsEmpty :: Queue a -> Bool`
Returns true if the given enqueue and dequeue list of the given queue is empty, otherwise false.

Examples:
```
queueIsEmpty (Q ([],[]))  == True
queueIsEmpty (Q ([1],[])) == False
```

`dequeueIsEmpty :: Queue a -> Bool`
Returns true if the dequeue list of given queue is empty, otherwise false.
The crawler stores all items of the current depth in the dequeue list. If the list is empty it means all links at this depth have been searched (the next item in the queue is thus for a deeper depth).

Examples:
```
dequeueIsEmpty (Q ([1],[])) == True
dequeueIsEmpty (Q ([],[1])) == False
```

`enqueueIsEmpty :: Queue a -> Bool`
Returns true if the enqueue list of given queue is empty, otherwise false. The crawler stores all items of the next (and only next) depth in this list.

Examples:
```
enqueueIsEmpty (Q ([1],[])) == False
enqueueIsEmpty (Q ([],[1])) == True
```

3.3.2 Searched

The purpose of this data structure is to keep track of which links we have already crawled and how many times they have been seen. Searched is an association list created with the Data.Map module from the Haskell prelude. The program uses a string as its key, this means that insertion/lookup/deletion is O(n*log n) complexity. The following functions make up the interface for Searched:

`Searched :: Data.Map Parser.Key Int`
The key represents the URL and and value represents the count.

`Key :: String`
A key representing a URL.

`empty :: Data.Map Parser.Key Int`
Returns an empty Searched list.

`insert :: URI -> Searched -> Searched`
Takes an URI and a Searched and returns Searched with URI inserted. If the key already exists, the corresponding value is overwritten.

`insertIncrement :: URI -> Searched -> Searched`
Same as insert except it increments the value (count) of a key with 1 instead of overwriting the value when you insert a value that is already in the map.

`insertMany :: [URI] -> Searched -> Searched`
Returns Searched with the list of URI inserted (using **insert**).

`member :: URI -> Searched -> Bool`

Returns True if URI is in Searched, otherwise False.

```
getCount :: URI -> Searched -> Maybe Count
```
Returns Just count of the corresponding URI in the Searched list, if the URI doesn't exist in the list Nothing is returned.

3.3.3 Graph

Graph is a data structure used to store the information gathered from a crawl. It is, just like Searched, an association list of key value pairs, where keys are strings, created with the Data.Map module. This, again, means that insertion/lookup/deletion is O(n*log n) complexity.

A **GraphWrapper** is a wrapper type used during manipulation of the information. The URI of a website is used to generate a key (string) during manipulation of the data, the key generation is done through a function which is a parameter in the GraphWrapper.

The value of a Graph is where the nodes with all the stored statistics and HTML (or alike) are.

Interface for Graph:

```
Node :: label -> [label] -> P.TagList -> [statType] -> G.Node label statType
```
Arg 1 is the label of the node (a URI in web contexts).
Arg 2 represents the adjacency list of the node (URIs/links on the page)
Arg 3 is the TagList found on the page, this is HTML parsed through the *TagSoup* library.
(Read about this in in the external library section).
Arg 4 is the list of statistics for the node.

```
insert :: Node -> Graph -> Graph
```
Inserts a node into the given graph.

```
newNode :: label -> [label] -> P.TagList -> [statType] -> Node label statType
```
Returns a node with the values given to it.

```
member :: (Ord key) => label -> GraphWrapper label key statType -> Bool
```
Returns True if the label exists within the graph, otherwise else False.

```
getNode :: (Ord key) => label -> GraphWrapper label key statType -> Maybe (Node
label statType)
```
Returns Just node if the associated node if the label exists, otherwise it returns Nothing.

```
getNodes :: Graph label key statType -> [Node label statType]
```
Gets all nodes in a graph.

3.4 Functions

This section covers how the main functions of the program work. It is assumed that the data structures are understood in the function descriptions.

3.4.1 crawl

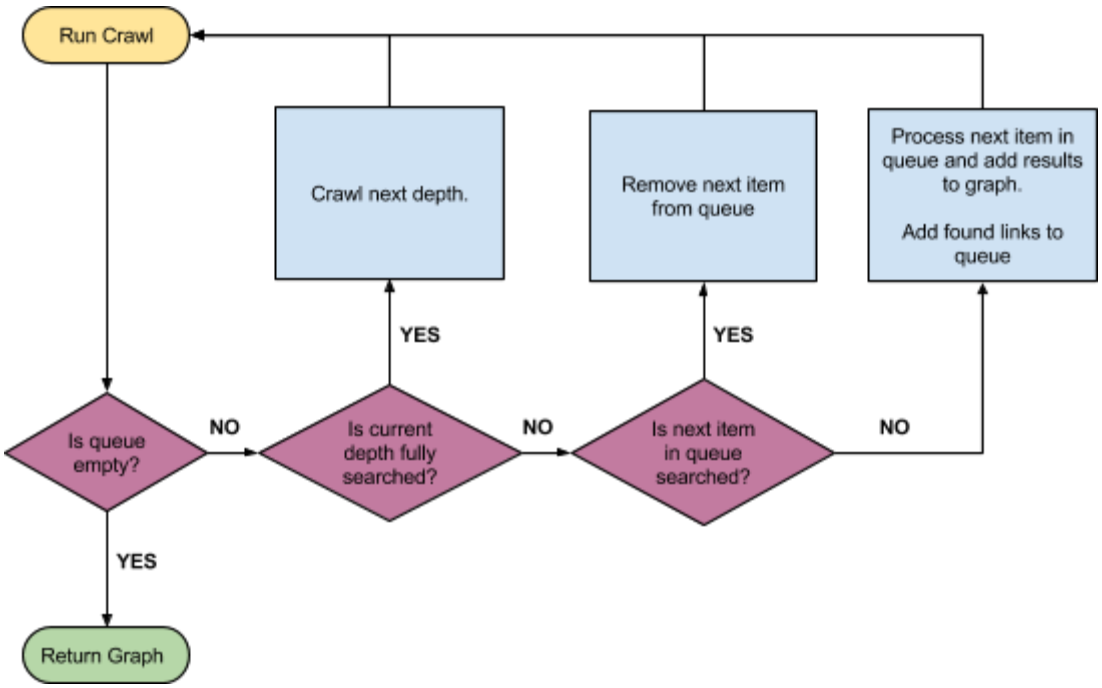
The main function, crawl, is defined as follows:

```
crawl :: (Show statType) => Q.Queue URI -> S.Searched -> Int -> Bool -> HtmlFetcher
-> [St.StatFn statType] -> IO (G.GraphWrapper URI P.Key statType, S.Searched)
```

- Queue is the data structure that controls which site to crawl next. Contains two lists, one for current depth and one for next depth.
- Searched is the data structure that stores which URLs are already searched.
- Depth is the maximum recursive depth to search.
- Bool determines whether to crawl internally or externally. False for internal, True for external.
- HtmlFetcher is a function which takes a string and returns an IO String. It is the function that gets the Data from the provided URL. This is as an argument to be able to run tests on self created graphs (controlled environment) instead of the internet. This is the IO part of the program.
- List of functions to run on every HTML-document gathered.

crawl has four different cases:

- Queue is empty -> Finished. All links on the current depth have been visited and no new links have been found.
- Dequeue is empty -> End of current depth, begin visiting links of the next depth (the queue will thus normalize during the next dequeue).
- Current node is in Searched list -> Skip crawling this node. Instead dequeue and crawl the next item in the queue.
- Current node not in Searched list -> Fetch the HTML and run it through crawlAux. Retrieve the node representing the parsed site and the new queue (the current queue with all found links added to it) from crawlAux. Add the node to the Searched list and crawl recursively using the new queue.



3.4.2 crawlAux

This is a pure helper function to *crawl* which fully processes a single web page. This function operates on a HTML document (represented as a string), parses it, runs statistical functions, and finally returns a node containing results. It also updates the queue for the main function (*crawl*) with the new links found in the document.

```
crawlAux :: (Show statType) => URI -> String -> Q.Queue URI -> Bool -> Bool ->
[St.StatFn statType] -> (G.Node URI statType, Q.Queue URI)
```

- URI: URL of site in URI form
- HTML of site
- Queue: The queue from crawl
- Bool that indicates whether to continue crawling or not (if False new links are not enqueued).
- Bool that indicates whether to crawl external links or not.
- List of functions to run on the data.

The crawlAux function returns a node representing the crawled HTML and an updated queue with new links added to it, the main function (*crawl*) later uses this queue.

The node contains the URI of the target site, all links found on it, the HTML in a TagList format, and a list of statistical data which are a result of the functions provided in [TagListStatFn]. The queue is updated by adding all the internal links found on the site to the queue. Getting the links and separating internal ones takes a lot of parsing. This parsing is done in the Parser module.

3.4.3 Parser module

This module contains all the parsing solutions our program uses. This module relies heavily upon external libraries (see the section on external libraries for more info). The following functions make up the interface of the Parser module:

```
HTMLToTagList :: String -> TagList
```
This function takes HTML as a single string, and with the help of TagSoup parses it into a TagList.

```
getUrls :: TagList -> [String]
```
This function returns all links in a HTML document.

```
normalizeRelativeUrls :: String -> [String] -> [String]
```
This function takes a link, and a list of links, and returns a list of all valid links.

Example: normalizeRelativeUrls "www.uu.se" ["www.uu.se/b", "/c", "d"]
== ["http://www.uu.se/a", "http://www.uu.se/b", "http://www.uu.se/c"]

```
urlsToUris :: [String] -> [Maybe URI]
```

This function takes a list of URLs and converts them to URIs. It also makes sure that the links are really http links and also normalizes them to our chosen standard (this includes converting parts of URLs to lower case which makes equivalent URLs equatable).

`normalizeUri :: URI -> URI`
Removes fragment and makes sure the URI is correctly cased.

Example: `normalizeUri "http://www.Uu.Se/Path/?pId=1#top"`
`== "http://www.uu.se/path/?pId=1"`

`getInternalUris :: [URI] -> String -> [URI]`
Given a list of URIs and a string(link) it returns all URIs that has the same authority as the link.

Example:
```
getInternalUris [(URI "a" (Just(URIAuth "b" "foo" "d")) "e" "f" "g"),
                 (URI "a" (Just(URIAuth "b" "bar" "d")) "e" "f" "g")]
            "foo" == [(URI "a" (Just(URIAuth "b" "foo" "d")) "e" "f" "g")]
```

`myRegName :: URI -> String`
Returns the authority of the URI

`urlToUri :: String -> URI`
Converts a string to an URI

`uriToUrl :: URI -> String`
Converts a URI to a string.

4 External libraries

These are libraries that were used by the program and have to be installed on the client computer for it to run.

4.1 TagSoup (Text.HTML.tagSoup v. 0.13.3)

TagSoup parses the HTML for us. It makes it easy to retrieve the contents of a chosen tag. We use it to convert the raw HTML to tags, represented by a data type called Tag containing the tag type and other attributes. We get all the links from the website by filtering lists of Tags.

Documentation: https://hackage.haskell.org/package/tagsoup

4.2 Network.HTTP and Network (withSocketsDo)

The network package, specifically the HTTP module, enables using HTTP requests in haskell. In short; this is what makes it possible for the program to download the HTML-documents.

Documentation: http://hackage.haskell.org/package/HTTP-4000.0.5/docs/Network-HTTP.HTML

4.3 Network.URI

Network.URI automatically breaks the links(strings) into its subpart. Given a link "http://www.foo.com/path/?query=1#fragment" it conveniently parses the links into five parts:

Scheme: "http:"
Authority: "www.foo.com"
Path: "/path/"
Query: "?query=1"
Fragment: "#fragment"

Our biggest uses of this package are:

- Checking if a link is internal. The link's is compared to the authority of the URL we are crawling.
- Relative links. Most sites mix relative with ordinary links, the link "/subsite" is a relative link for example, and the URI-package makes it easy for us to check whether or not a link is relative and then just add the correct authority to make it a valid link.
- Normalizing links: "http://www.foo.com/path" and "http://www.foo.com/path#top" are equivalent and can be normalized by removing fragments and more (#top in the case of the second link). The URI module makes this easier.

Documentation: http://hackage.haskell.org/package/network-2.1.0.0/docs/Network-URI.HTML

4.4 Data.Map

Data.Map is a module for creating efficient map-structures (dictionaries).
We use this map-structure as a backbone for our Searched and Graph data structures.
This module uses a Binary Search Tree-model and has a time complexity of O(log n) for both insertion and lookup.

Documentation: https://hackage.haskell.org/package/containers-0.4.0.0/docs/Data-Map.HTML

5 Discussion

This sections covers thoughts we have of the project.

5.1 Shortcomings

These are some of shortcomings of the program and some of the challenges we faced during the project:

- The program doesn't check the robots.txt when deciding whether to crawl a website.
- Parsing of links may not always be correctly done, some valid links may get incorrectly parsed leading to an invalid link, leading to an exception.
- No support of https links.
- The statistical functions that the users supply to the crawler are restricted to returning the same type. Writing the functions also require knowledge about the Tag String type (TagSoups parsed HTML), making the crawler more difficult to use, we are not entirely satisfied with this outcome.

On top of these shortcomings, we had some challenges implementing many of the functions due to the guideline to avoid imperative code. The program also became too complex for us to manage properly which resulted in unwanted coupling and low cohesion regarding modules.

5.2 Future work

We have already written our code with future implementations in mind and this section covers what some of those implementations could be.

5.2.1 Generic crawler

Although this crawler has generic elements, one could generalize it even more to allow crawling on any kind of protocol. This would allow users to add their own functions and datatypes for crawling data of their choice. Some of the things that users would have to provide are functions for parsing data and creating unique keys from links as well as link data types.

5.2.2 Rule based crawling

Users could choose which kinds of links to follow or which kinds of pages to continue crawling depending on rules that they supply. As an example users could supply a search query that would filter which links to follow or skip.

Users could also supply functions that analyze the contents of pages and then decide whether to follow links on that page or not. A real world example of this is language based web crawling, where the crawler runs language identification on pages and crawls them if they match the target language. *An Crúbadán* (http://crubadan.org/) is one such crawler.

5.3.3 Downloading media

The crawler is limited to fetching HTML at the moment. Its functionality could be extended to allow downloading of files and media while crawling.

6 Conclusion

The aim of this project was to create a program that will crawl the web starting from a specific web page and allow the user to analyze gathered data to produce statistics of their choosing.

We achieved this by using a Breadth First Search (BFS) algorithm, taking help from external libraries to fetch and parse the HTML. We separated much of our logic into different modules to make our program easy to manage. A queue allowed us to keep track of depth during crawls, and a list of visited links prevented our program from revisiting pages. Users can supply the crawler with custom made functions which operate on the HTML and return data of the visited pages.

Functions provided by users require a higher knowledge cap than we found satisfactory, and there are limits to what data can be collected on a single run of the program. The project goals are met with only minor omissions.