Spidr

COMS 4115 Project Proposal

Katherine Haas (kah2190)
Akshata Ramesh (ar3120)
Matthew Meisinger (mrm2205)
Alexander Dong (aqd2000)

Description/Purpose of the Language

In today's world, enormous amounts of data are freely available over the internet in HTML format. This provides a pleasant user experience if a user uses a browser to surf through 4-10 pages and finds the information they are seeking. If the journey increases to a length of 50, 500, or 5000 pages, however, the experience becomes less pleasant.

To make it more convenient for users who need to sift through a large amount of HTML data, we have decided to implement a programming language that is tailored to these types of problems. Applications for this language would range from creating Craigslist scrapers such as PadMapper.com to building language processing search engines like WolframAlpha.

Spider is focused on the making the following operations easier than other languages:

- **Retrieving/parsing HTML pages** – The Spidr language only requires two lines of code to retrieve a page and parse out all links, words, and image references on the page.
- **Traversing webs of pages** – Using the same two lines of code required to retrieve and parse an HTML page, the user can also follow all links on that page, up to a specified link depth.
- **List operations** – All data types are automatically lists in Spidr, and concise list manipulation operations are built in to the language. These make integrating more sophisticated logic into the application easy (e.g. only following links that point to a specific domain).

This language will be tailored toward the following use cases:

- Follow all links on a page and retrieve words from the child pages
- Compile all links from a page into a list
- Get list of URLs of all images on a page
- Get list of all dead links contained within a domain (based on a starting page)

Data Types

All datatypes are interpreted as lists, even if they are assigned only one value.

- Basic
  - int – a list of standard non-floating point number.
  - string – a list of standard sequence of characters.
  - boolean – a list of true and false values (in practice will almost always be a single-valued list)
- Data Structures
  - Web - the user directed query of some html target. The web data structure will hold the raw html data, the url link, as well as, if specified, any 'children URL'.
  - Spider - the data structure that will contain the parsed information from the specified Web object. Our current plan is to hold the parsed information inside of an adjacency-list hash table.
  - class – user-defined data structure that can contain any number of basic data types and complex data types

Language Syntax

In normal usage of this language, a user will create a Web object (a page) and execute Spiders (queries) against it.

When creating a new web object, the notation '`-> [String URL]`' will specify how many 'levels deep' the user wants the program to analyze, and which URL to operate on. The language will check before each query whether the query has already been done. (So that circular calls don't happen.)

`->`	process the specified target URL
`-->`	process the target URL as well as any other URL's found on the target page
`-(int)->`	process the target URL and children up to a maximum of (int) depth. Ex. `-(5)->`

When creating a new spider object, the notation ‘`[:param] [Web object]`‘ will determine the parsing method and which Web to run the query on. Some example query parameters include:

`:html`

Parses the raw html data by the html tags

`:text`

Parses the raw html data, only taking into consideration text seen on the loaded page

`:image`

Parses the raw html data, only retrieving any image files (the path to the image)

`:html*`

Does the same as the non-* equivalent except that it limits its parsing to the target URL.

`:text*`

Does the same as the non-* equivalent except that it limits its parsing to the target URL.

`:image*`

Does the same as the non-* equivalent except that it limits its parsing to the target URL.

There will be a number of functions to retrieve information from the Web and Spider data structures. Some examples include:

`Spider`

`String getParsing()`

Returns the parsing method used

`String getData()`

Returns all the parsed string tokens. (sorted by frequency)

`String getURL()`

Returns all the URL's

`int totalFreq()`

Returns the total number of tokens parsed

`int numKeys()`

Returns the number of unique tokens

`int numURL()`

Returns the number of unique URL's

`int getFrequency(String target)`

Returns the frequency of a target String

`boolean hasKeyword(String target)`

Returns whether the target word was found in the query

`String keywordLoc(String target)`

Returns the list of URL's that contained target keyword

`Web`

`String getURL()`

Returns the initial target URL;

`int getDepth()`

Returns the query depth specified

`String getRaw()`

Returns the raw HTML data

`Web linksTo()`

Returns any Web children that were also queried. Web will essentially be a tree data structure.

`bool isValid()`

Returns whether the server returned a valid, non-error response

List functionality include:

- Concatenating two lists with the ++ operator.
- Functions can be executed over an entire list by using `myFunction(<listName>)` syntax. This executes the function over each of the values in the list, and concatenates the returned variables as a single list.
- Filtering a list using the `:>` operator, and including an expression that evaluates to true or false on the right side of the operator. The keyword `item` represents the value of each item as it is iterated over in the list. E.g. to filter a list of integers to only include those greater than 10:

```
int overTen = listOfNumbers :> value > 10
```

Comments will be denoted as follows (subject to change):

`<<`	Single Line Comments
`<* *>`	Multi-Line Comments
`<*  <* *>  *>`	Nested Comments will be supported

Sample Code

```
Web query = Web -> "www.columbia.edu"
Spider textONLY = Spider :text query

int foodFreq = textONLY.getFrequency("food")
int totalFreq = textONLY.totalFreq()

print(foodFreq/totalFreq)
<* Prints to output the frequency of the word 'food' found at www.columbia.edu *>


<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

Web query = Web --> "www.columbia.edu"
Spider textONLY :text query
print(textONLY.getData()[0]))
<* Prints the most frequent word found on all queried pages *>


<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

class WebImage:
      String images
      String url

function void storeImages(Web query, WebImage storage)
      Spider image = Spider :image* query
      storage.add(new WebImages(image.getData(), query.getURL())
      for(Web w : query.linksTo())
            storeImages(w, storage)

Web query = Web -(20)-> "www.columbia.edu"
WebImage images = new WebImage
storeImages(query, images)
<* Creates an array that holds a data structure that holds all the images at some URL *>


<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

function void main()
      <* get starting web page *>
      Web query = Web -> "www.microsoft.com"
      Spider thisPage = Spider -> :text query

      <* get child pages *>
      Spider children = getChildPages(thisPage, 5)

      <* filter results to only URLs with ‘microsoft.com’ in them *>
      Spider allPages = children :> strin[g.contains(item.URL, "microsoft.com")

      <* Find all Microsoft pages in which the term ‘Google’ appears. (This uses nested
      filters. The inner filter returns any words on the current page that equal
      ‘Google’. The outer filter .) *>
      Spider microsoftPagesMentioningGoogle = microsoftPages :> (length(item.Words :>
      (item == 'Google')) > 0)

      <* Print a list of all the urls of the pages, each on it's own line. *>
      printLine(<microsoftPagesMentioningGoogle>.URL)


function Spider getChildPages(Spider thisPage, int depth)

      if (depth = -1)
            return [];
      else
            string childUrls = thisPage.childUrls

      <* Declare a list of child pages. The constructor of 'Spider' is used as a list-
            valued function by putting carats on either side of child urls.  The
            constructor is executed once for each child url and all of the resulting page
            objects are concatonated into a single list. *>
      Spider childPages = Spider :text (Web -> <childUrls>)

      <* Get all children for each of this page’s children. *>
      Spider pageDescendants = getPages(<childPages>, depth - 1)

      return thisPage ++ pageDescendants
```