

HW2 Site Sucker



Webviborous SiteSuckeri

For HW2 you will build a "web robot" that uses the HTTP protocol from the client side. SiteSucker will download HTML text and extract and analyze URLs. The homework should give you a better understanding of the Internet's most important protocol. This homework is more difficult than HW1, so don't start it a couple days before it's due. It's also not really a toy program -- SiteSucker is a real program for solving the problem of tracking down broken links on pages. In order to spread out the load a bit, SiteSucker is due early in the morning: 2:00 am Fri May 11th (i.e. early Fri morning). As before, P/NC students may work in teams of 2, and they are not required to support the part 3 -c option.

Site Sucker is a bit large, but it divides up pretty nicely into several parts. This handout presents those parts in a graduated order -- the best strategy is to get each part working before proceeding to the next ("divide the big thing into several little things that can be built independently" is standard CS wisdom). The handout also includes some code decomposition suggestions. The part-3 functionality can be built almost entirely by re-using the part-1 and part-2 functions. Sketch out your overall functional decomposition before coding so your earlier parts are designed so they can be used by the later parts.

Formatting note

Site sucker operates in a simple way -- it takes command line arguments, and then prints things to standard output in a regular way. It's important that your solution follow the format closely so the grading scripts can make sense of your output.

There's some simple starter code in the `/usr/class/cs193i/hw` to get you started. Do not use the Perl HTTP connection module -- the idea for this assignment is to learn about HTTP and HTML by working with them first hand.

Part 1 — Simple GET Mode

With the "-a" option, SiteSucker takes a fully specified URL, performs an HTTP GET, and prints out the header and body that are sent back...

```
% ss.pl -a 'http://www.stanford.edu/class/cs193i/test/basic.html'
http://www.stanford.edu/class/cs193i/test/basic.html  OK
HTTP/1.1 200 OK
Date: Fri, 21 Apr 2000 08:17:19 GMT
Server: Stronghold/2.4.1 Apache/1.3.3 C2NetEU/2409 (Unix) mod_fastcgi/2.2.2
Last-Modified: Fri, 21 Apr 2000 08:16:54 GMT
ETag: "70741d16-15f-39000e76"
Accept-Ranges: bytes
Content-Length: 351
Connection: close
Content-Type: text/html
```

```
<html>
<head>
<title>Basic</title>
</head>
```

```
<body>
<h1>Basic</h1>
```

A few straightforward links

```
<ul>
<li><A HREF="http://www.yahoo.com/">yahoo</a>
<li><a href=http://www.theonion.com/>The Onion</a>
<li><A HREF=http://www.stanford.edu/class/cs193i/test/a.html>a absolute</A>
<li><a href=a.html>a relative</a>
</ul>

</body>
</html>
```

On the command line, the URL can be put in single quotes (') so the shell is not confused by any funny chars (e.g. ?) in the URL. The first line of the output is the full URL followed by a tab followed by a short message -- "OK" if the URL worked or an error message otherwise. Here's what some of the error cases look like...

```
% ss.pl -a http://ftp.stanford.edu/this/wont/work
http://ftp.stanford.edu/this/wont/work  ERR Bad connection
% ss.pl -a http://www.cnn.com/nosuchfile.html
http://www.cnn.com/nosuchfile.html      ERR HTTP 404 Not found
elainell:~/193i> ss.pl -a http://no.such.host/
http://no.such.host/      ERR Bad host
% ss.pl -a http://www.stanford.edu/class/cs193i
http://www.stanford.edu/class/cs193i    ERR HTTP 301 Moved Permanently  Location:
http://www.stanford.edu/class/cs193i/
## note, the above is a single long line
```

To formally define the output, we'll call the message that follows the URL its "disposition". The possible dispositions fall into three categories...

1. OK

OK The URL is a valid HTTP URL and returns text/html data which is the one MIME type SiteSucker can deal with.

2. STOP

The URL appears valid, but SiteSucker is not pursuing it for any of a number of reasons.

STOP Not http The URL does not use the HTTP scheme. SiteSucker only deals with HTTP URLs. No connection is attempted.

STOP Visited The URL is one we've already checked (this case will make sense in phase 3 when we deal with multiple URLs). No connection is attempted.

STOP Not text/html The URL is valid and we have not already visited it, but the HTTP Content-Type: returned is something other than text/html, so SiteSucker is not going to pursue it. A connection must be attempted to learn that the MIME type is wrong. It's ok if SiteSucker downloads the whole HTTP response and then decides that it doesn't like the MIME type.

3) ERR

Trying to follow the URL runs into a real error...

ERR Bad host The host name lookup failed.

ERR Bad connection Can't open a connection to port 80 on that host

ERR HTTP 404 Not Found The server returned a result code other than 200. Append the exact result code and short status message that the server returned.

ERR HTTP 302 Found<tab>**Location: http://fooby.com/**
As an extension to the above "not-200" case, if the error code is in the range 300-399, and the header includes a "location:url" line, then a tab and the location should be appended to the line after the regular error output.

ERR Bad HTML This is a catch-all case if your parser is looking at HTML, and cannot make sense of it e.g. . This case should never happen on our test pages which will always use syntactically valid HTML, so we will not grade on this error message. However, out on the Internet at large this case may happen, and here's the error to report for it.

ERR Bad HTTP response This case will probably never happen, but you can get a case where the HTTP response cannot even be parsed enough to do anything. e.g. the first line has the wrong syntax, or there is no blank line separating the head from the body. Generate this error for those cases.

Assumptions

The GET request send to the server should look like the following (some servers appear to require that the GET and HTTP be upper case)...

```
GET request HTTP/1.0\015\012\015\012
```

If there are no errors, then the output should be printed starting on the line after the URL. In STOP and ERR cases, don't print anything further for that URL after its disposition.

You may make the following assumptions about URLs...

1. Each URL has the following structure..

There is no whitespace anywhere in the URL.

All absolute URLs have at least have a **<scheme>:<body>** structure.

The **<scheme>** contains only alphabetic characters [a-zA-Z], followed by a colon (:), followed by the **<body>**. The body may contain any sort of non-whitespace character except for quote (") and right bracket (>).

For HTTP URLs, the structure is **http://<host><path><file><suffix>**.

The **<host>** contains only alphabetic letters, dots, dashes, digits, and possibly a colon ([a-zA-Z.\-:]) — note that the dash (-) in the [] is escaped with a backslash (\). The path will begin with a slash (/) and the subparts of the path will be separated from each other by slashes. There are some official constraints on what characters may be used in the URL, but it's best not to rely on them. As a practical matter there's a wide variety of URLs in the world, and most servers deal with them fine. The **<file>** is made up of the characters after the last slash. At the end of the **<file>**, there may be a **<suffix>** section starting with a '?' or '#'. As a practical matter, the suffix section may contain any character except whitespace, quote ("), or right-bracket (>). We'll just lump the block of characters after the first '?' or '#' together and call it the suffix. For '#', omit the suffix (including the '#') when making the GET request. For '?', leave the suffix appended to the **<file>** and let the server will deal with it.

HTTP host names like "www-tour.stanford.edu:1081" refer to an HTTP connection that uses a port other than 80. We're going to ignore this case and just treat the :1081 as part of the host name (leading to a few extra ERR Bad hosts).

Special case parsing: for a full URL, if the path, file, and suffix are all the empty string, then treat the path as a single "/". So the URL "http://www.cnn.com" should print and be treated as "http://www.cnn.com/". Do not add a slash for any other cases such as when the path or file are not empty.

You will find that HTTP headers arrive from the Internet at large with mixed up EOLN combinations. Your code must deal with the two most common endings "\r\n" and "\n". A server may switch between the two conventions during the dialog. When producing text output, SiteSucker should substitute the locally-correct "\n" line ending for the server's line ending. The most reliable way I have seen to do this is `$data =~ s/\015?\012/\n/g;`

Use the pages listed at <http://www.stanford.edu/class/cs193i/test/> to try out your code. You may examine the raw HTML files in file-system space at `/usr/class/cs193i/WWW/test`. We are not telling you what the output is supposed to be for all those files -- that's what the assignment is about. As a newly minted Internet Engineer, you should be able to look at an HTML file and figure out the semantics of the URLs within it. SiteSucker is realistic enough that most pages around the web can be used to test it...www.intel.com, www.3com.com -- see the list on the test page.

Part 1 Options

Here are the three command line options for part 1. Once you have -a working, -h and -b should be trivial...

- a prints the header followed by the body. The blank line counts as part of the header.
- h like -a, but only prints the header
- b like -a, but only prints the body

Part 1 Decomposition Ideas

Dividing a large task up into smaller, testable functions is key for a large program like this. Here are some suggestions...

Write a function which takes a full URL and tries to retrieve the whole HTTP response. The function should open the connection, send the request, and return the entire response. The function should return an appropriate string for the many different possible ERR or STOP cases which can arise at this stage. You can use the return-multiple-things-in-a-list trick from the Perl handout. You cannot just exit the program (die) in error cases since it probably has more processing to do — you need to return an indication of the error to the caller so they can react appropriately.

Write a function that takes the entire HTTP response, and parses it to deal with the HTTP status code, parse the head from the body, and find the content-type.

Write a function which takes an HTTP URL (possibly relative) and parses it into its constituent strings: scheme, host, path, file, suffix. This function is tricky, but it's self contained so you can concentrate on it with the various test.html URLs and work on it until it's right. (also very useful for Part 2.)

Part 1 Milestone

With part 1 working, you should be able to print out HTTP responses for a variety of URLs. You should be able to generate the many different error cases.

Part 2— URL Extraction

With the `-x` option, instead of printing the HTML, you should extract and print all of the `<a ...>` URLs in the HTML. Each URL should be printed in its full form on its own line. (Please use exactly this format — we will use automated test tools on your solution.) Here's the `-x` of the `basic.html`...

```
% ss.pl -x http://www.stanford.edu/class/cs193i/test/basic.html
http://www.stanford.edu/class/cs193i/test/basic.html      OK
http://www.yahoo.com/
http://www.theonion.com/
http://www.stanford.edu/class/cs193i/test/a.html
http://www.stanford.edu/class/cs193i/test/a.html
```

The hardest problem here is converting relative URLs in the HTML source to their full URL form. The two cases which must be solved are...

1. Relative URL like "books.html" (not beginning with a '/'). In that case, the URL picks up the scheme, host, and path of its referring document.
2. "Root Relative" URLs like "/images/red.gif" (beginning with a '/'). They pick up the scheme and host of their referring document, but not the rest of the path.

(See the lecture discussion of relative and root-relative URLs)

This relative -> full URL conversion is exactly what a browser has to do whenever you click on a relative link in a web page. The Google indexing robot also needs to solve this problem as it wanders around. You are in good company. We will not solve the case for URLs beginning with ".." such as "../images/red.gif" -- just treat those like regular relative URLs.

URLs should be printed as they appear in the source HTML. So `mailto:president@whitehouse.gov` can be repeated in the `-x` listing even though we don't really know how to parse `mailto:` URLs.

Part 2 Ideas

When parsing the HTML, you can look for hrefs which look like `<a something something something href=url something >`. The "`<a`" comes at the start (upper or lower case) and is followed by at least one whitespace character. The `<something>` sections may contain anything except a "`>`" or another "`href =`". A good strategy is to find the "`<a`" and its matching "`>`" first. Then dig around for

the "href=url" between the two. There can be arbitrary whitespace separating the various components. The "a" and the "href" may be upper or lower case. You should preserve the case of the URL itself. In rare cases it matters. The URL may or may not be enclosed in double quotes.

In Perl, the dot (.) matches any character except newline. Use the /s option in a regular expression to force (.) to also match newline. This may be necessary in your parsing if a multi-line chunk of HTML is stored in a string like this...

```
<a href
=
http://foo.bar/
>
```

Part 2 Decomposition Ideas

For the relative -> full conversion, a good strategy is: 1) Use the URL parser from part 1. It needs to be beefed up to deal with relative URLs where the scheme, host, path... may be missing. 2) For relative -> full conversion, just parse the relative URL and its referring base URL into their parts, notice which parts of the relative URL are missing, and substitute them in from the base URL. The relative -> full function is tricky, but it can be written and tested extensively by running -x. Use "urls.html" in the test directory to get started with your -x testing.

Part 3 — Checking Mode

The final phase is what makes SiteSucker the most useful for web authoring. Ideally part 3 should not require too much additional code if the first two parts are well decomposed. The -c option is an enhanced version of -x which checks each of the URLs. -c mode is like -x, but every URL in the listing should be followed by a tab followed by its "disposition" as defined in Part 1.

```
elaine40:~/193i> ss.pl -c http://www.3com.com/
http://www.3com.com/      OK
http://search.3com.com/index.html?col=&qp=&qs=&qc=support+public+jobs+pa
lmpilo&pw=600&ws=0&ql=a&nh=10&lk=1&rf=0      OK
http://www.3com.com/news/0320/index.html      OK
http://www.3com.com/news/releases/pr00/apr1800a.html      OK
http://www.3com.com/buydirect/index.html      ERR HTTP 302 Moved Temporarily
Location: http://buydirect.3com.com/iom_dcms/b2c/catalog/index.html
http://www.3com.com/news/0320/charles_schwarb.html      OK
http://www.palm.com/      ERR HTTP 500 Server Error
http://www.3com.com/index.html      OK
http://www.3com.com/buydirect/index.html      STOP Visited
http://www.3com.com/products/index.html      OK
http://support.3com.com/index.htm      OK
http://www.3com.com/util/contact.html      OK
http://search.3com.com/      OK
http://www.3com.com/util/login.html      OK
http://www.3com.com/technology/index.html      OK
http://www.3com.com/online_education/index.html      OK
http://www.3com.com/inside/index.html      OK
http://www.3com.com/news/index.html      OK
http://www.3com.com/partners/index.html      OK
http://w3n.3com.com/hr/extpost.nsf/d6216364fcd114878825683c00557c3e?OpenForm      OK
```

```

http://www.3com.com/legal/privacy.html OK
http://www.3com.com/util/features.html OK
http://www.3com.com/legal/copyright.html OK

```

Don't do all your testing on 3Com or they'll get mad at us. Use the graduated cases in the test directory.

-f Filter Option

The optional `-f target-string` can come after `-x` and `-c`. It restricts the program to only operate on the absolute URLs that contain the given target (case insensitive). We'll use this to keep from generating too much network traffic...

```

% ss.pl -c -f prod http://www.3com.com/
http://www.3com.com/ OK
http://www.3com.com/products/index.html OK

```

Ignore the filter string in `-a`, `-h`, and `-b` modes.

Part 3 Ideas

http: Only

You only need to check HTTP URLs — others can stop with a STOP Not HTTP.

HEAD vs GET

Do not check each HTTP document by doing a full GET. Use the HEAD request instead which works just like GET, except it returns only the small HTTP header and not the whole body. You do not need to make a new socket for every connection — just use `close()` to close the old connection and then `connect()` to connect a new one on the same socket. Some hosts generate a server error on HEAD — that's their problem.

Visited

Before trying a URL, make sure that it is not one which has already been tried — for this purpose it's sufficient to store the tried URLs as all lowercase strings in a single buffer or hash table and just look up each absolute URL before trying it. For URLs with a '#' suffix, use only the part of the URL to the left of the # for detecting VISITED. Keeping track of the already visited URLs seems like one of those rare times where a global variable is appropriate. (The boolean presence of the various command line `-a`, `-x`, etc. options could also be global.

Final Testing

We will test your program with files similar to the ones in the test directory. Look at the raw HTML in the test directory to get ideas for the various cases that can come up. Take your solution for a spin around the net. Use "View Source" from within the browser on different sites to see what their HTML looks like. SiteSucker is not really a toy program— you may find that you can run most real-world cases through it and get interesting results. The practical value of SiteSucker is that you can run it over a site `-c` to check to see if its URLs all work. Just search for the word ERR in the output. See if your `-x` can deal with `www.yahoo.com` or `www.news.com` which are quite messy. A `-c` on these sites

will probably take too long and/or result in a nasty letter being sent to your instructor. Use `-f` to restrict it so a subset of the URLs.

Deliverables

Name your program `ss.pl` and make sure it works on the elaines. Use the submit script in `/usr/class/cs193i`

SiteSucker 2000

SiteSucker doesn't need any extras! If you can get it to work pretty much, that's plenty! But just for fun, I'll just mention some other features that come to mind...

One simple thing which makes it more useful is a `-e` option for `-c` where it only prints a URL line if there's an ERR. That way you can run it over a site, and it'll only print the broken URLs instead of all of them. I defined STOP and ERR in such a way that the ERR cases are exactly the ones you care about from a site authoring point of view. You can simulate this with the unix `grep` command...`"ss.pl -c http://www.foo.com | grep ERR"`.

Another possible extension is to make SiteSucker recursive. Instead of just checking the `-x` links, do a full follow on them and get *their* links, and so on to some pre-defined depth limit from the original URL. The recursive links are just printed indented a little from their parent link. The code is extremely similar to SiteSucker as presented — it's just a little high-level fiddling of what gets passed to the same routines which make up the non-recursive SiteSucker (yay modularity!). In fact, that's the way I wrote SiteSucker originally, but I eventually changed it to the form described for the homework. The problem with the recursive form is that it's a combination of scary and useless. You very quickly end up jumping (slowly) all over the net, and the output is pretty incomprehensible. You also end up creating so much network traffic that it becomes important to think about following the robot exclusion standard. Our non-recursive form has the same core code, but it solves a common problem and produces output that's useful.

Another natural thing to think of for SiteSucker is concurrency - realize that you don't need to process the `-c`'s of links sequentially. It would be much faster to fork off a separate thread to check each URL so they all happen in parallel. (CS193k does something like this.)