600.466 ASSIGNMENT 4
Web Robots

Due date: Tuesday, April 23, 2013 in class

1 Background

1.1 Link Parsers

Both lwp_parser and lwp_parser_two make use of the Perl libwww package. This package takes care of most of the details of connecting and communicating with a web server.

To understand basically what's going on in these programs try this

```
jhu> telnet www.jhu.edu 80

# returned by server

Trying 128.220.2.80...
Connected to jhuniverse.hcf.jhu.edu.
Escape character is '^]'.

# typed by you

GET / HTTP/1.0 <CR><CR>

# returned by server

        HTTP/1.0 200 OK
        Server: Netscape-Enterprise/2.01
        Date: Fri, 09 Apr 1999 05:22:15 GMT
        Accept-ranges: bytes
        Last-modified: Wed, 07 Apr 1999 16:17:06 GMT
        Content-length: 7007
        Content-type: text/html

        <HTML>
        <HEAD>
        <BODY BGCOLOR="#fffafc" link="#330099" vlink="#006666"
        alink="#6666cc" text="#000000"
        BACKGROUND="/www/images/bk_home.gif">

        <TITLE>JOHNS HOPKINS UNIVERSITY</TITLE>

 ... other nifty stuff
```

What you have done is contacted JHU's web server and asked for the root page using the GET method. The first few lines

```
HTTP/1.0 200 OK
Server: Netscape-Enterprise/2.01
...
```

are the header. They tell what type of content is being presented (text/html), when the page was last modified (Wednesday, April 7), what server is being used (Netscape), etc. The following text is the body or content

```
<HTML>
<HEAD>
<BODY BGCOLOR="#fffafc" link="#330099" vlink="#006666"
...
```

which represents the page we requested ("/").

The tcp_client.pl program shows how to do the preceding with sockets. But the libwww package provides an easier way. The parsers use the class LWP::UserAgent to act as a liason between the server and the program. To contact a server and retrieve a particular page is done as simply as

```
use HTTP::Request;
use HTTP::Response;
use LWP::UserAgent;

my $ua = new LWP::UserAgent;

my $request  = new HTTP::Request 'GET' => "http://www.jhu.edu";
my $response = $ua->request( $request );
```

We first create our liason, $ua. We then create a HTTP::Request object. This object associates the page we want to retrieve with the method we want to use to retrieve it. We then request the page from the UserAgent which contacts the server and retrieves the text and stores it in an HTTP::Response object. To print out this text we code

```
if ($response->is_success) {
    print $response->content;
} else {
    print "Error: " . $response->code . " " . $res->message;
}
```

We first check to see if everything went smoothly (did we get the page) and if so print out the content. Otherwise we report the error code and message that the server passed to our UserAgent.

1.2 Explanation of lwp_parser.pl

The lwp_parser parses through a retrieved text/html file, parses and prints out all the links within the page. It does this by first retrieving the text via the method above and then parsing the text using an HTML::TreeBuilder. A TreeBuilder object simply takes an html page and creates a tree were each HTML tag represents a node in the tree. This structure can get very large and take a tremendous amount of memory but for small pages it's okay. To make the TreeBuilder we code

```
my $html_tree = new HTML::TreeBuilder;
$html_tree->parse( $response->content );
```

We can then easily extract all links via

```
foreach my $item (@{ $html_tree->extract_links( )}) {

    my $link = shift @$item;
    my $furl = (new URI::URL $link)->abs( $response->base );

    print $furl, "\n";
}
```

This simple extracts each link of the form:

```
<a  href="somelink.html"   ... >
<img src="somepicture.jpg" ... >
<body background="somebackground.jpg" ... >
```

The line:

```
(new URI::URL $link)->abs( $response->base );
```

creates a new fully qualified URL by prepending the base URL of the $response object to it. For example if the retrieved link is "../img/webagent.book.gif" and we retrieved the link from the page "http://www.cs.jhu.edu/˜yarowsky/cs466" then the above object would return:

```
http://www.cs.jhu.edu/~yarowsky/img/webagent.book.gif
```

When we are done with the TreeBuilder Object we should delete it since perl's garbage collector has trouble with circular links (which the TreeBuilder creates).

1.3 Explanation of lwp_parser_two.pl

lwp_parser_two does the same thing as lwp_parser only in a different way. lwp_parser_two uses an HTML::LinkExtor to extract links while the page is being transferred to the User-Agent object. To create an HTML::LinkExtor we first must create a parsing function. This function will be passed two elements

```
sub function {
    my ( $tag        ,     # the current HTML tag being parsed
         %attributes ,     # the attributes of that tag
       );
    ....
}
```

For example, if we are currently parsing the line:

```
<a href="../sad_things/my_poetry.ps"> And another sad thing . . .
```

The above function would be passed

```
$tag = "a";
$attributes{ "href" } = "../sad_things/my_poetry.ps";
```

This function is passed by reference to a newly instantiated HTML::LinkExtor object and is called every time we call the HTML::LinkExtor->parse([$text]) method. So to create an HTML::LinkExtor we code

```
sub function {
    my ( $tag    ,
         %attribs,
       );

    ... check to see if it's what we want ...
}

my $extract = new HTML::LinkExtor( \&function );
```

The UserAgent->request( ) method allows you to pass it a function reference which it will call every time it receives a chunk of text from the web server. The received text will be passed to this function. So to let the HTML::LinkExtor parse this text we wrap the object in a function and pass the function to the UserAgent->request( ) method

```
my $ua = new LWP::UserAgent;
my $request  = new HTTP::Request 'GET' => "http://www.jhu.edu";
my $response = $ua->request( $request, sub { $extract->parse($_[0])} );
```

That's it.

Part 1

For Part 1 of Homework 4, modify either lwp_parser.pl or lwp_parser_two.pl so that it extracts only links which are not self-referencing or non-local.

A self-referencing link is of the form

```
<a href="somepage.html#samepage">
```

A non-local link is a link referencing a URL from some other domain. For example if we retrieved a page from `www.cs.jhu.edu` then the following link:

```
<a href="www.ora.com/index.html">Cool book place</a>
```

is non-local. In order to get the base URL of a HTTP::Response object use the method

```
my $base_url = $response->base;
```

Explanation of robot_base.pl and Assignment 4 Part 2

`robot_base.pl` is a very basic robot which walks down an HTML directory. For each page of text/html it reads it extracts two items – links and the text immediately associated with the links. For example, given the following

```
<a href="some_link.html"> Some link you might like </a>
```

It will extract

```
$link = some_link.html
$text = Some link you might like
```

If there is no plain text embedded between the <a href...> ... </a> tag then nothing is retrieved for $text.

The links are pushed onto a stack and traversed in order of their discovery. That is the robot performs a breadth-first search of the HTML directory.

The robot is relatively robust in that it will not trapse down broken or non-existent links. It also will not attempt to parse for links anything other than text/html. It is also relatively nice in that it only will contact a particular server once every minute. There is a way to contact the server's more frequently but I don't recommended it (unless you like your mailbox filled with nasty email from web sys admins).

The robot **will** traverse recursive and non-local links. You should change this function (and will be graded down if you don't). You also must stay within the `cs.jhu.edu` domain.

Your job is to modify the robot so that it (1) seeks out and prints links to postscript/pdf documents that it finds in its search, and (2) extracts some basic contact information from all text/html web pages as described below.

To verify if a page is a certain type, such as postscript, you can use the following method.

```
my $response = $ua->request( $request );
$response->content_type =~ m@application/postscript@;
```

There are three methods which you should write (or modify) in the program

```
#
# wanted_content
#
#    UNIMPLEMENTED
#
#  this function should check to see if the current URL content
#  is something which is either
#
#    a) something we are looking for (e.g. postscript, pdf,
#       plain text, or html). In this case we should save the URL
#       in the @wanted_urls array.
#
#    b) something we can traverse in the search for links
#       (this can be just text/html).
#

sub wanted_content {
    my $content = shift;

    # right now we only accept text/html
    #   and this requires only a *very* simple set of additions
    #

    return $content =~ m@text/html@;
}

#
# extract_information
#
#    UNIMPLEMENTED
#
#  this function should read through the context of all the text/html
#  document retrieved by the web robot and extract the three types of
#  contact information described in the assinment

sub extract_information {
    my $content = shift;
    my $url = shift;
    # parse out information you want
    # print it in tuple form to the CONTENT and LOG files, for example:

    print CONTENT "($url; EMAIL; $email)\n";
    print LOG "[CONTENT] ($url; EMAIL; $email)\n";

    print CONTENT "($url; PHONE; $phone)\n";
    print LOG "[CONTENT] ($url; PHONE; $phone)\n";

    return;
}
```

and

```
#
# grab_urls
#
#     PARTIALLY IMPLEMENTED
#
#   this function parses through the content of a passed HTML page and
#   picks out all links and any immediately related text.
#
#   Relevancy based on both the link itself and the related text should
#   be calculated and stored in the %relevance hash
#
#   Example:
#
#      $relevance{ $link } = &your_relevance_method( $link, $text );
#
#   Currently _no_ relevance calculations are made and each link is
#   given a relevance value of 1.
#

sub grab_urls {
    my $content = shift;
    my @urls    = ();


  skip:
    while ($content =~ s/<\s*[aA] ([^>]*)>\s*(?:<[^>]*>)*(?:([^<]*)(?:<[^aA>]*>)
*<\/\s*[aA]\s*>)?//) {
        my $tag_text = $1;    # ex: <a href="some_link"> ...
        my $reg_text = $2;    # ex: <a href ... > Some plain text </a>
        my $link = "";

        if (defined $reg_text) {
            $reg_text =~ s/[\n\r]/ /;
            $reg_text =~ s/\s{2,}/ /;

            #
            # compute some relevancy function here for related text
            #
        }

        if ($tag_text =~ /href\s*=\s*(?:["'](["']*)["']|([^\s])*)/i) {
            $link = $1 || $2;

    #
    # compute some relevancy function here for the link itself
    #
    # Note: this is also an excellent place to check for non-
    #       local or self-referencing links.
    #

            $relevance{ $link } = 1;    # apply the relevance total here
            push @urls, $link;
        }
    }

    return @urls;
}
```

All links found are then sorted base upon the numeric value of their relevance and trapsed in the sorted order. This order may be modified as more relevant links are discovered. The robot stops when it runs out of links.

Note in the above function (grab_urls) that $content is the *entire content* of a text/html page. It is therefore possible to use surrounding text other than $reg_text and $tag_text to determine relevancy of a link.

A Note on Logging & Debugging

We have added in logging to the web robot to make it easier for you to debug your programs and keep track of where they are visiting. When you invoke the web robot, you will need to specify on the command line the logging file that you want results logged in, the file that you want extracted content to be placed into, and the URL of the web page that you want to begin your web traversal at. The program is currently set up to log each HTTP transaction (HEAD & GET) and all active content that was retrieved. If you want to log additional things, simply print out to the LOG file anywhere in your program. For example:

```
print LOG "Information you want logged.\n";
```

To execute your web robot, we suggest the following method:

```
./robot_base.pl mylogfile.log content.txt http://www.cs.jhu.edu/ &
tail -f mylogfile.log
```

This will start your web robot and then show you your log file as things are added to it. To stop your web robot, type:

```
CTRL-C (to stop the view of the log file)
fg
CTRL-C (to stop the web robot)
```

Your log file and content file will be over-written each time you start the web robot. But until you start it again, you can view your file with any text editor or viewer, i.e.:

```
less mylogfile.txt
```

Summary of What to do for Part 2

1. Modify the robot to only traverse a single local department in the JHU network, and non self-referencing links. For example, if you choose the Computer Science Department, you would limit your web robot to visiting servers within the (`cs.jhu.edu` domain).

2. modify the `sub wanted_content` function to check for postscript content (or whatever document type you choose to search for) and text/html content to determine whether the current URL is either traversable or contains postscript.

3. modify the `sub extract_information` function to parse through the returned content from all html and plain text web page using regular expressions to extract and print out the following 3 types of regular format "contact" information: all US phone numbers, e-mail addresses, and paterns of the format "<CITY>, <STATE> <ZIP>" found on a web page. Be prepared for some mild variation, such as "San Diego, CA 92122" and "Cambridge, Mass. 02138", but you can pretty much match on any word string of the form "Word+, Word 5-digit-number". Output should be printed as a simple tuple: (<URL>;<TYPE>;<DATA>), e.g.

```
(http://www.cs.jhu.edu/~yarowsky; PHONE; 410-516-5372)
(http://www.cs.jhu.edu/~yarowsky; EMAIL; yarowsky@cs.jhu.edu)
(http://www.cs.jhu.edu/~yarowsky; CITY; Baltimore, MD 21218)
```

Clearly a small variant of this could be used to generate "insert into" SQL command for database loading.

4. modify the `sub grab_urls` function to rank all links found by some relevancy function of your own design. All unvisited links will then be sorted based upon this relevancy function to determine where the robot will visit next.

What to Turn in:

Please hand in your assignment both in hard copy and electronically. For hard copy submissions, please provide printouts of your code, commented sufficiently so that we can understand what you did solely from reading the comments.

For electronic submissions, please use the submission protocols descriped in HW3, with the only change that "hw3" is replaced with "hw4".

Submit any code that you have written or changed, but this should minimally include `lwp_parser.pl` and `robot_base.pl`.
Please DO NOT send these as an attachment or MIME encoded.