Assignment 2: An HTML Table Generator

Introduction

Our train schedule page produces html reports with nice colors and fonts, but the columns do not line up, and the spacing is too dense. We need a new tool to present schedule data in clear, attractive tables.

We *could* write a special-purpose tool just for train schedules, but a better plan is to write a tool we can use for many similar applications.

In this assignment you will write a software tool to convert text tables to html tables. The assignment will give you experience working with three levels of web programming: HTML, shell scripts, and C.

In the process, you will see how output of one program can be seen as input to be processed by another program. An important part of Unix programming is seeing report output as data.

The first part of this assignment introduces you to the ideas of Unix text tables, the syntax for HTML tables, and Cascading Style Sheets. Work through this first part at the computer so you see the connection between text tables and html tables.

The second part presents the software project of writing a C program to convert text tables into HTML tables. The project is described in three stages, each a more sophisticated version of the one before.

Unix Text Tables

It is best to work through this part and the next part at the computer.

Many Unix commands produce tabular data. Many Unix data and configuration files contain tabular data. All these tables are plain text. Consider, for example, the `who` command.

```
% who
lib113   pts/0        Sep 21 13:48 (pool-141-154-12-129.bos.east.verizon.net)
farger   pts/1        Sep 21 13:03 (24.10.76.242)
tagratniji pts/5        Sep 21 09:33 (pfp11.potrzebie.com)
dkenney  pts/9        Sep 20 18:15 (166.96.197.209)
info113  pts/20       Sep 21 10:07 (pool-141-154-12-129.bos.east.verizon.net)
allin2   pts/24       Sep 21 10:25 (pool-74-104-99-234.bstnma.east.verizon.net)
%
```

The `who` command lists users currently logged into the system. Each row of output represents one login. The columns in this table contain the logname, the terminal name, the month, day, and time when that user logged in, and the address or name of the machine from which the user is connected. Various command line options allow you to request other information (type the command `man who` for details.)

Consider another example, the `ls` command. Used with the `-l` option, ls produces output of the form:

```
% ls -l
total 22828
-rwxr-xr-x   2 bin      mail      212992 Nov 13  1998 Mail
-rwxr-xr-x   2 bin      bin       131072 Jan 27  1999 Rsh
drwxr-xr-x   2 root     system      8192 Jun 16 16:55 X11
-rwxr-xr-x   2 bin      bin        24576 Nov 13  1998 [
-rwxr-xr-x   1 bin      bin        40960 Jan 26  1999 acctcom
-rwxr-xr-x   1 bin      bin       417792 Nov 13  1998 admin
-rwxr-xr-x   1 bin      bin         1518 Nov 13  1998 alias
%
```

Apart from that pesky top line, the output of the command `ls -l` is a text table. Each row represents one file, and each row contains specific columns: the permission attributes, the number of links, the owner, the group, the file size, the modification date, and the name of the file.

The file `/etc/passwd` list information about users. This text table contains colon-delimited data. This colon-delimited format is somewhat different from the space-separated format produced by ls and who.

Another text table is the file `/etc/fstab` Use the `cat` command to look at that file. It lists all the various hard disks that are made available to the current system. Under MS-Windows, different hard disks are assigned drive letters like C:, D:, etc. Under the MacOS, different hard disks appear as different icons each with its own name. Under Unix, different hard disks appear as separate directories, all branches off the base tree. This `fstab` file tells Unix which drives to connect to and where to place them on the tree. This text table is space-delimited.

Part One of the Assignment

[1] For the first part of the assignment, find three more Unix commands or configuration files that produce (or are) text tables. List those three commands on the paper you submit.

These are easy to find. You can `cd /bin` and simply try commands. You can `cd /etc` and look at files. Other good places to find commands are the directories `/sbin` and `/usr/sbin` . When in doubt, use the command `man`, as in `man fstab` , to learn more about the command and its output. If you type a command and it just sits there, it is probably waiting for input. Press Ctrl-D or Ctrl-C and check the manual page.

Sending Text Tables to a Web Browser

There are two main user interfaces to Unix systems: text-based terminals and web browsers. You have been working with a text-based terminal. In this section, you will send these text tables to a web-browser.

Login to your Unix account and cd into your public_html directory. Make a subdirectory called hw2 and change into that. Use an editor to create a file called tt1.cgi containing:

```
#!/bin/sh
#
# cgi program to generate text tables from local Unix commands
#
     echo "Content-type: text/plain"
     echo ""

     echo "Here are the current internet connections:"
     netstat -A inet
     echo "Here are the current files:"
     ls -l
     echo "Here is fstab:"
     cat /etc/fstab
```

Now type `chmod 755 . tt1.cgi` , and then browse your page at:

`http://sites.harvard.edu/~yourname/hw2/tt1.cgi`

It is easy to put a web interface onto Unix commands. The fun part is getting them to look nicer than just plain text. To make them look nicer, we use HTML tables.

HTML Tables: Getting Started

Text tables are perfect for the text-processing/software tools/pipelined world of Unix programming. They look plain, though, on a web page. Web pages have a language for creating tables; it is part of HTML.

For a summary, look at:

`http://www.htmlcodetutorial.com/`

and click on the Table section. Another good introduction is at

`http://www.w3schools.com/html/default.asp`

which has a section on tables. To get started, try the following example.

a. Login to your account and cd into the public_html/hw2 directory. Copy two files into that directory by typing:

```
cp ~lib113/hw/tt2ht/*.css .
```

b. Now, use an editor to create a file called table1.html that contains:

```
<html>
<head><title>A Table Example</title>
<link rel='stylesheet' type='text/css' href='style1.css'/>
</head>
<body style='background-color: white'>
<center> A Table Example </center>

<!-- this table has three rows, each row contained in <tr> and </tr> -->
<table border='1' cellpadding='3'>
   <tr>
        <td>Boston</td>  <td>MA</td>  <td>617</td>
   </tr>
   <tr>
        <td>New York</td>  <td>NY</td>  <td>212</td>
   </tr>
   <tr>
        <td>Washington</td>  <td>DC</td>  <td>202</td>
   </tr>
</table>
</body></html>
```

b. Now use a web browser to look at

`http://sites.harvard.edu/~yourname/hw2/table1.html`

If you typed your html file correctly, you should see a nice-looking table with lines around all the cells, text in a proportional font, and left-justified text within each cell.

c. Compare the display on the screen to the html 'source code' for that page. The table is defined by the stuff contained between the `<table>` tag and the `</table>` tag. Within that table you have defined three rows, each row is defined by the stuff contained between the `<tr>` tag and its matching `</tr>` tag. The tag "tr" stands for "table row". Each of those rows, in turn, consists of a sequence of table data items. Each table datum is defined by the stuff contained between a `<td>` tag and a matching `</td>` tag.

The browser looks at all the data, and it figures out column widths that fit all the data. If you change data or add new rows to the source file, you can reload the page and the browser will adapt the column widths to fit the new data.

The extra space between the tags and the extra newlines after the tags do not affect the output. Like indenting and vertical arrangement of C source code, indenting and vertical arrangement of html tags helps the writer and maintainer of the code see that things are correct.

d. In the previous section you saw the basic rules for creating an HTML table. The rows and cells of the table are just pieces of text marked with pairs of tags. If you want to change the appearance of the table, you can add attributes to the tags. For example, make the following changes to your html file:

1. Replace `<table>` with `<table class='redbox'>`
2. Replace the `<td>` tags before the city names with `<td class='city'>`
3. Replace the `<td>` tags before the area codes with `<td class='area'>`

Now reload your page. You should see how the additional information in the tags has changed the appearance of the table. The first cell on each row has a border and contains right-justified text. The last cell on each row has a light blue background, a dark blue line on top, and contains bold, centered text. How did that happen?

The styles for these cells are defined in the file called style1.css . That file is mentioned in the top

part of the html file. Look at that file now. Notice the section that looks like:

```
.city {
        text-align: right;
        padding-top: 1em;
        border-style: solid;
        border-width: 1px;
        vertical-align: middle;
        padding: 4px;
}
```

This block of code defines a style called "city". The properties of the style are a set of tagged values. (Like the tagged data in the sched file.) Compare the properties described in this style definition to the visual effect in the browser. We apply this list of attributes to the city items by putting the attribute `class='city'` in the html file.

Now examine the definition of the style called "area" and compare those properties to the appearance of the table in the browser.

There are lots of style tags. These tags allow you to describe in detail the appearance of each item on a web page. A short, useful list to start with is at:

`http://www.htmlhelp.com/reference/css/properties.html`

e. Experiment with the descriptions in the style sheet file and see how your changes affect the way the page looks. You must use the correct tags and you must end each tag:value pair with a semicolon.

Part Two of the Assignment: Manual Conversion

The ultimate purpose of this assignment is to automate the process of converting a text table into an html table. Before you write an automated process, it is useful to perform the conversion manually. That way, you know what steps you are automating.

[2] Use a text editor to convert the output of `who` to HTML

To do so, follow these simple steps:

a. Create a file that contains the output of who:

```
who | head -5 > who.html
```

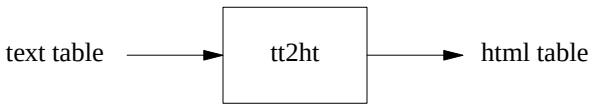
b. Edit the file who.html, adding `HTML` table tags so each row of the who output is a table row, and each item in each row is a table datum. Feel free to use the classes in style1.css or whostyle.css in your table, and feel free to experiment with the styles in those sheets to produce stunning effects.

For Part [2] of your homework submission, include the html file and the stylesheet you create. Make sure you browse the file to make sure it looks the way you want it to.

Part Three of the Assignment: Automated Conversion

It is tedious to convert text tables into HTML tables manually. It would be handy to have a software tool that takes as input a text table and produces as output an HTML table ready to send to a web page or directly, via CGI, to a browser.

Here's a picture:



This is a picture of a typical Unix software tool. Text streams into the tool, the tool does something to the input, and the result streams out of the tool. In this case, the input is a text table (like the output of who)

and the output is an html table.

You will write this tool so that it will read any text table, such as the output of many Unix tools, the contents of many Unix databases, and the results of data processing (such as your work with the sched file) and send them to a web page in a nice-looking html table.

A full version of this program, one that handles various sorts of input and can apply various attributes to table data, is pretty sophisticated. We shall build up to it in three stages.

Version 1: Fixed Format Output

First, write a program that automates the steps you performed in Part [2] of this assignment. Specifically:

Part [3a]: Write a C program called tt2ht1.c that does the following:

The program accepts as input rows of data. Each row contains a sequence of strings separated by one or more spaces or tabs. The program writes as output a table starting tag, a sequence of table rows, and then a table closing tag.

Your program must generate clearly indented and formatted html, like the format in the area codes example shown above. If you just run the output tags and data into one long line or do not indent, you will lose points. The browser does not care about the layout; people who debug your code do.

COMPILING: To compile your code for the server, you must use compile the program using the `-fno-stack-protector` option:

```
cc -fno-stack-protector tt2ht1.c -o tt2ht1
```

Part [3b]: Test Your Program on a Web Page:

In your hw2 directory on your Harvard web site, create a cgi program called part3b.cgi containing the following code. Note: the `#!/bin/sh` line must be the first line in the file.

```
#!/bin/sh
#
# cgi program to generate html tables from local Unix commands
#
     echo "Content-type: text/html"
     echo ""

     echo "<html><body style='background-color: white;'>"
     echo "<p>Here are routes to other computers:"
     /sbin/route | ./tt2ht1
     echo "<p>Here are the current files:</p>"
     ls -l | ./tt2ht1
     echo "<p>Here is fstab:</p>"
     ./tt2ht1 < /etc/fstab
     echo "<p>Here is train 1205:</p>"
     (cd /home/l/i/lib113/lectures/lect04/5_Code/cgi; ./trainsched 1205) | ./tt2ht1
     echo "</body></html>"
```

This program does not have to assign classes or attributes to the cells. You can set the border, cellpadding, and cellspacing in the <table> tag, but you don't have to do anything special about the cells. That's later.

Version 2: Flexible Formatting

This first version is pretty useful. It will take any text table with white-space separated items and make it into an HTML table ready to deliver to a web browser. The format is much nicer than plain text.

But all the tables come out looking pretty much alike. If you want to change the appearance, you need to edit and recompile the code. This is not a flexible program. Different text tables deserve different widths, alignments, and bgcolors. Different tables also deserve different border widths, cellpadding, and cellspacing. The first version of this program allows for none of these.

The solution is to send into tt2ht input consisting of two parts, a preface describing the class to assign to

each column and then the actual text table. The program (tt2ht) will then use the specified attributes to generate the table.

For example, here is an input stream suitable for tt2ht2:

```
<noprocess>
<h4>Current Users</h4>
<table border=1 cellpadding=2 width='80%'>
</noprocess>
<attributes>
class='name'
class='tty'

class='number'

</attributes>
rserved     ttyp4       Oct  6 09:54
hsieh       ttyp5       Sep 25 23:57
msimonet    ttyp6       Oct  1 02:27
barrer      ttyp7       Oct  6 15:41
 ...
<noprocess>
</table>
</noprocess>
```

The output from tt2ht2, given this input, is

```
<h4>Current Users</h4>
<table border=1 cellpadding=2 width='80%'>
<tr>
   <td class='name'>rserved</td>
   <td class='tty'>ttyp4</td>
   <td>Oct</td>
   <td class='number'>6</td>
   <td>9:54</td>
</tr>
<tr>
   <td class='name'>hsieh></td>
   <td class='tty'>ttyp5</td>
   <td>Sep</td>
   <td class='number'>25</td>
   ...
</tr>
   ...
</table>
```

This new program is different in two main ways. First, the new program does not automatically print the opening <html>, <head>, <body>, etc tags. It is more specialized -- it only converts lines of text table into lines of html table.

Second, the new program can handle three sorts of input data: plain text, column attributes, and text table data. Plain text is identified as any input between the tags <noprocess> and </noprocess>. These lines are not converted to html table rows; they are simply printed to standard output. Within a noprocess block, all tags except for </noprocess> are copied to standard output.

Attributes are lines of input between between the tags <attributes> and </attributes> . These lines define the attributes of each column in the table. The first line in the attributes section applies to the first column, the second line in the attributes section applies to the second column, etc. Note that blank lines in the attributes section are significant. Within an attributes block, all tags except for </attributes> are treated as plain text. Furthermore, if another <attributes> block appears in input, that block overwrites the attributes stored from a previous block.

Finally, any lines not in the plain text or column attributes sections are treated as text table data and are made into rows of an html table.

Notice the flexibility this new system offers. If you want to change the alignment, color, width, etc of the cells, you no longer need to edit and recompile the code. The program reads settings that it plugs into the tags as it generates the html table.

The program should accept data lines that contain a number of items different from the number of attributes. If there are more items than attributes, the extra items get tags with no attributes. Your program should accept these section tags even if preceded by spaces or tabs.

Part [4a]: Write a C program called tt2ht2.c that allows the flexible format control described in the above paragraphs.

Part [4b]: Test Your Program on a Web Page:

In your hw2 directory on your Harvard web site, create a file called part4b.top containing:

```
<noprocess>
 <html><head>
        <link rel='stylesheet' type='text/css' href='whostyle.css'/>
       </head>
 <body bgcolor=white>
 <table class='redbox'>
</noprocess>
<attributes>
  class='name'
  class='tty'

  class='number'

</attributes>
```

and also create a cgi program called part4b.cgi containing:

```
#!/bin/sh
#
# cgi program to test tt2ht2
#
     echo "Content-type: text/html"
     echo ""

     ( cat part4b.top ; cat ~lib113/hw/tt2ht/who.output ) | ./tt2ht2
     echo "</table></body></html>"
```

Make part4b.cgi executable, and try it from a web browser. If you don't like the spacing or cellpadding or border width, now you can just modify the stylesheet or the shell script rather than the C code. No more recompiling, just change it and save it.

Remarks the parenthesized command

This shell script shows a way to group a sequence of unix commands into a single component in a pipeline. The three commands in the parentheses are run one after the other and the output of all three flows into tt2ht2. The cat command sends the start of the page, the table header, and the set of attributes. The who command generates the data, and the echo command at the end closes the page.

Implementation Hints

If you are unsure how to proceed with this part, here are some ideas. You need an array of strings, one for each set of attributes. Pick reasonable (to you) sizes for the number of and length of the strings and write code that prevents array overflows.

Read input line by line using fgets(). If you encounter the <noprocess> section, just print those lines out to stdout until you see the closing </noprocess> tag. If you encounter an <attributes> section, read in lines and store them in your array of strings.

When you encounter actual data, enclose the table data items with tags containing the corresponding attributes.

Version 3: Flexible Input Format

Not all text tables consist of strings separated by whitespace. The passwd file contains colon-delimited fields. Some datasets are semicolon-delimited. In particular, the train sched data, after processing by semi2tab2 is tab-delimited but has spaces in some of the station names (e.g. "north station").

A final enhancement to your program is to allow the user to specify the delimiter, just as the `cut` command provides the `-d` option to specify the delimiter.

Later in the course, we shall see how to add commmand line options. For now, we shall add a new control tag: `<delim value=;/>`

This new feature allows you to process a semicolon-delimited file with:

```
#!/bin/sh
#
# cgi program to test tt2ht3
#
     echo "Content-type: text/html"
     echo ""

     echo "<html><body><table>"

     ( echo "<delim value=;/>" ; grep "TR=1205;" sched ) | ./tt2ht3

     echo "</table>"
```

This script does not provide any <noprocess> or <attributes> sections. The program will treat missing <noprocess> and <attributes> tags the same way tt2ht2 did. That is, it will not generate a <table> tag, but it will include any default attributes in the individual cells. The difference is that the program will split the fields on the **;** character rather than on white space.

Part [5]: Modify your solution to part 4 so it supports the <delim/> tag

The details are: The <delim> will appear on a line on its own. The delim tag has an attribute of the form `value=X` delim tag ends with `/>` . Since this tag does not enclose any data, it is a stand-alone tag. The XML syntax specifies that stand-alone tags end with a slash.

Technical Details about the delim Tag

The delim tag tells your program to split the input on each instance of the specified character. The input line

abc;def;;pqr;xyz

turns into five table items.

This behavior differs from the how the program handles white space in its usual mode. In the usual mode

abc def pqr xyz

even, with its many spaces, is still only four items.

Helpful Hints

As you work on this assignment, you may find the following C functions useful: strtok(), sscanf(), strstr(), strchr(). These functions allow you to split apart strings and locate substrings and characters in strings.

The project is broken into a sequence of steps to help you focus on one new idea and skill at a time. You are free to jump ahead and go for the complete solution, but if you prefer to solve smaller problems, follow this outline.

Warning: Nested Tags What will your program do if it encounters a <noprocess> tag in the middle of an <attributes> section? What will it do if it encounters a </noprocess> tag without having seen a

<noprocess> tag? In general, how will your program respond to ill-formed input? On the other hand, an <attributes> tag may be useful inside a <noprocess> block. Document how your program responds to overlapping sections and explain your choices.

Summary of the Assignment

The parts of the assignment are:

[1] List three more Unix tools or files that are text tables. (9 points)

[2] Use a text editor to convert the output of `who` to an HTML table. (10 points)

[3a] Write a C program called tt2ht1.c that converts a space-separated text table to an HTML table. (30 points)

[3b] Show that your version of tt2ht1 works by using it with a cgi program. Run the cgi program from the command line and capture its output by using script. (5 points)

[4a] Write a C program called tt2ht2.c that converts a space-separated text table to an HTML table AND accepts sections containing plain text and attributes sections. These sections do not have to appear at the top of the input, but may appear anywhere in the input. (30 points)

[4b] Show that your version of tt2ht2 works by using it with a cgi program. (5 points)

[5] Write a C program called tt2ht3.c that converts a space-separated *or* delimited text table to an HTML table AND accepts plain text and attributes sections. (11 points)

Turning in Your Work

a) Copy your C code, your cgi programs, README, and typescript of sample runs, into a single directory.

b) In that directory, type:

```
~lib113/handin tt2ht
```