CS61C Homework 3 - An HTML parser

Due Wednesday, September 22nd, at 11:59pm

September 21, 2004

1 Introduction

Your job is to finish a simple HTML parser whose skeleton is provided to you. An HTML parser receives an HTML file, counts the number of occurrences of each tag name, and then returns an alphabetically ordered list of the tag names, alongside its frequency.

2 HTML Primer (All You Need Know For This Project)

This section discusses the HTML language. You can skip it if you know what's it about.

HTML stands for **HyperText Markup Language.** [RLJ98]. It is a markup language, which for now you can compare to a text file with extra stuff. This extra stuff includes, among others, the ability to specify rich text format (color, italics, etc.), add multimedia elements (songs, videos, etc.), and include links to other files. All this additions are specified alongside the text in the file.

The way to include these extra elements in an HTML file, is by using HTML **tags**. Tags are pieces of text that are not to be interpreted as text, but instead as markers that something is happening. HTML tags are always enclosed in angle-brackets ($<$ and $>$).

For example, the HTML string table represents a chunk of text with the word "table." The HTML string <table>, however, represents an occurrence of an HTML tag, whose **tag name** is table. Such tag is used to mark the beginning of a table.

While there are lots of documents that will help you understand HTML [Mey95], this primer should cover what you would need to finish this homework.

2.1 Tag Qualifiers

Some tags can have qualifiers that refine the semantics of a tag name. For example, you may want to specify some features about the table you are defining, namely whether the table should be printed with a border or not. In that case, HTML permits inserting the name of the qualifier and the value of the qualifier between the tag name and the $>$ character. To mark that a table must have border, the correct HTML tag will be <table border=1>, where the tag name is "table," and there is a "border" qualifier with the value 1. You can add several name=value qualifiers, separating them with blanks.

Note that, with the exception of HTML tag markers, characters $<$ and $>$ are illegal in HTML files, either as text or inside other tags. In case you are curious, if you want to add $<$ or $>$ as text, you have to use escaped sequencies: the HTML strings < or >, respectively.

Therefore, it is pretty easy to know where a tag starts (whenever you see the $<$ character) and when it ends (whenever you see a $>$ character).

2.2 Tag Pairs

Some tags occur in begin-end pairs. For example, when describing a table, we want to mark the end of the table, as well as the beginning. Pair tags are in the form <tag> ... </tag>, where <tag> indicates the beginning of a tag pair, and </tag> indicates the end.

In the table example, you will mark the end of a table by using the HTML string </table>. For this homework's purposes, end tags must be considered as equivalent to its correspondent begin tag (i.e., <table> and </table> must be considered as 2 occurrences of the table tag.)

2.3 HTML Parser Finite State Machine

The HTML parser that we want you to write can be described using the **Finite State Machine** (FSM) in Figure 1.

If you know what an FSM is, you can skip this section.

An FSM [FC00] is "an imaginary machine that is used to study and design systems that recognize and identify patterns." In our case, the patterns are the HTML tags and the angle brackets that surround them.

An FSM is represented as a series of states (represented by circles in our figure) and transitions among the states (represented as arrows between the states).

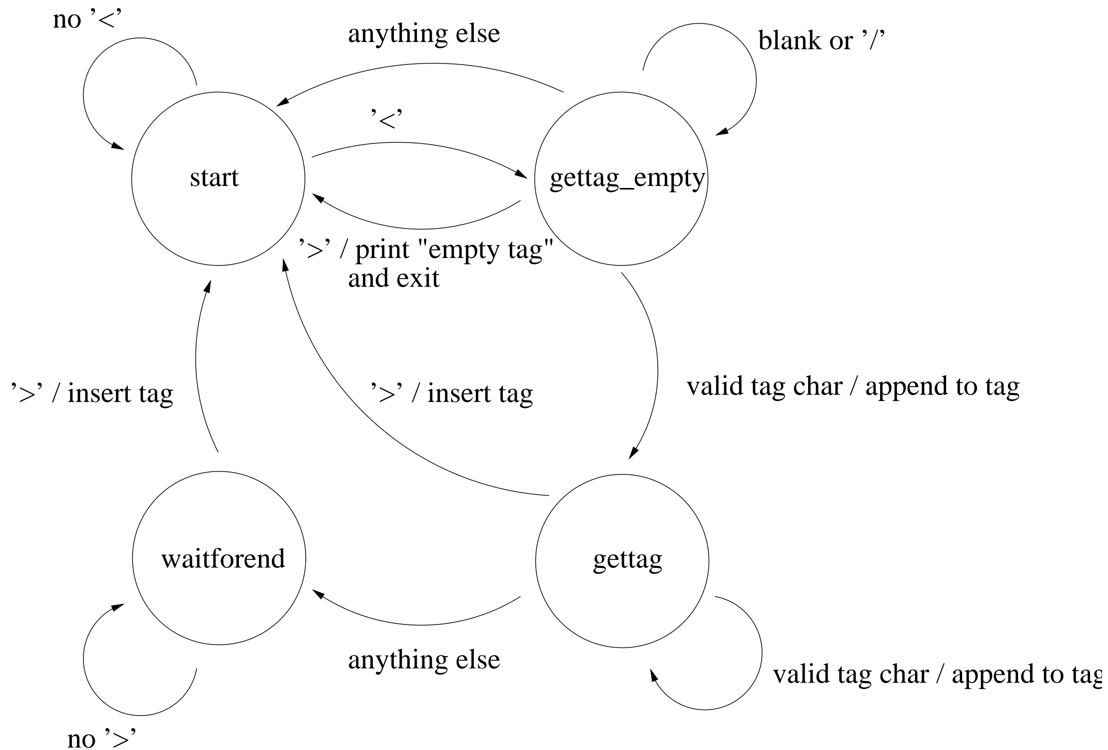

Figure 1: HTML Parser FSM

In our case, we have 4 states, named "start," "gettag_empty," "gettag," and "waitforend," and 11 transitions.

At any moment, the FSM is in one of the states, and only one. For example, our FSM starts in the "start" state. The FSM receives a string of symbols as input. These symbols (which in our case are the ASCII characters that compose the HTML file we are parsing) are read by the FSM one by one, and each one causes a transition between 2 states. Note that an FSM must explicit its behavior for any input and any state.

As an example, let's assume we are at the "start" state. We see that 2 arrows come from this state. The one in the right occurs when the next character in the file is '<'. In that case, our FSM moves from the "start" state to the "gettag" one. The one in the left occurs when the next character is anything but '<'. If our FSM machine is in the "start" state, and the next character is anything but '<', the state will move to… "start." (There is no reason why a transition to the same state is not OK.)

Some transitions actually fire actions alongside state transition. In our case, please take a look at any of the 5 transitions that point to the "start" or "gettag" states. In these transitions, the character that fires them is appended with an action that must be carried out when taking the transition. The three valid actions of our FSM are "insert a tag", "print "empty tag"", and "append (character) to tag."

This FSM defines the behavior of the HTML parser. You start in the "start" state, and keep reading characters until you find a tag delimiter ('<' character). This takes the FSM to the "gettag_empty" state.

Once in "gettag_empty", the FSM keeps reading blanks (remember that blanks are valid characters between a tag delimiter and the tag name), until it finds a valid tag character (a-z, A-Z, '-', and '_'), which moves it to the "gettag" state. The only exception is if it finds '>' first, in which case the FSM detected an empty tag.

In the "gettag" state, the FSM keeps reading valid tag characters until it reads something that is not a valid tag character. Then, if what the FSM gets is a tag end ('>' character), it inserts the tag just read, and moves back to the "start" state. If it's anything else, the FSM moves to the "waitforend" state, where it waits until receiving a '>' character.

The way to implement an FSM is the following: Your program will have an `fsm_state` variable that will store the current state of the FSM. The program will read its input, one character at a time. Every character is to be

interpreted depending on the FSM current state. For example, let's assume your program reads the character 'a'.

- If the current state is "start" or "waitforend," your program won't do anything with it, and will remain in the same state
- If the current state is "gettag_empty", it will append the character to the current tag, and move to state "gettag"
- If the current state is "gettag", it will append the character to the current tag, and remain in the same state

When the FSM action requires your program to insert a tag, it will do so by calling a function called `tag_insert`, and then will clean up the tag, so that the next tag starts from scratch.

`tag_insert` is used to manage a dynamic structure that stores the number of times a tag appears in the HTML file. The function will check whether the tag has already been inserted, in which case it will update the tag counter. Otherwise, it will add a new entry for the tag (this should smell enough like using malloc).

Note that this dynamic structure must be able to grow when you find new tags, so you don't want to use an structure with a bounded number of supported tags. A binary tree is a good choice. You can also allocate a structure with space for a fixed amount of tags, and then make it grow when needed.

3 Your Task

Your task is to complete the file `html_parser.c`, which parses an HTML file, and builds a dynamic structure that stores the number of times a tag appears in the HTML file.

After finishing parsing the input file, your program should dump the contents of the dynamic structure, and report the most popular tag. More concretely, it must print:

1. A list of entries (one per line), composed of tags in the file (in lower case), alongside its frequency. Each entry entry will be of the type "`tag, frequency`." Entries must be printed in tag's alphabetical order.
2. The entry corresponding to the most popular tag should be printed again as the last line, with the string "Most popular -> " prepended. If 2 tags appear the same number of times, the most popular is the first in alphabetical order. If the file has no HTML tags, don't print anything.

As an example, if the HTML file has 2 `html` tag, 2 `body` tags, and 1 `br` tag, the right output will be:

```
body, 2
br, 1
html, 2
Most popular -> body, 2
```

Write a correct `Makefile` that compiles and links `html_parser.c`. The compile process should use the `-Wall` and `-pedantic` options, and produce no warnings in `nova.eecs`

Submit `html_parser.c` and `Makefile` before Wednesday, September 22nd, at 11:59pm.

4 Implementation Details

- There can multiple blank characters (blank implying spaces, tabs, or new line characters) between the enclosing angle-brackets and the tag. In other words, the strings `<table>` and `<  table >` are 2 occurrences of the `table` tag.
- HTML tags are case-insensitive. In other words, it doesn't matter whether they are in upper (`<TABLE>`), lower (`<table>`), or mixed case (`<TaBlE>`). Consequently, your program should no differentiate tags based on the case. Moreover, when outputting results, your program should print tags in lower case
- You can assume that tag names will never be more than 1023 characters long. Once this has been said, we will take points out if the dynamic structure that you use to keep track of tag occurrence uses more bytes that needed
- Tag names can be composed of the following characters only: a-z, A-Z, 0-9, hyphen ("-"), and underscore ("_")
- Tag names that don't appear in any HTML specification must be counted as well
- Empty tags (angle-brackets with nothing in the middle, or only blanks) are illegal. They should cause your parser to print the following message (and after that exit):

  ```
  Error in input file: empty tag in line 34
  ```

5 Other Notes

To help you fine-tune your program, we provide you with an oracle, i.e., a valid version of the HTML parser

that works. You may want to use it to compare its output to that of your program. The oracle has been compiled for several different architectures. Choose the one that corresponds to your architecture (run `uname -a` from your shell to get your architecture.)

We also provide a valid HTML file (`index.html`)that you can use to test your program.

References

[RLJ98] D. Raggett, A. Le Hors, and I. Jacobs. "HTML 4.0 Specification." *Available on-line at* http://www.w3.org/TR/1998/REC-html40-19980424

[Mey95] E. A. Meyer. "Introduction to HTML." *Available on-line at* http://www.cwru.edu/help/introHTML/toc.html

[FC00] M. Fellows, N. Casey. "The Alphabets, Words, and Languages of Finite State Machines." LANL Megamath. *Available on-line at* http://www.c3.lanl.gov/megamath/workbk/machine/mabkgd.html