CSE 322
Intro to Formal Models in CS
Homework #8
Due: Friday 12 Mar 2010, and *not* accepted more than 36 hours late

W. L. Ruzzo 28 Feb 10

A *lexical analyzer*, *lexer* or *scanner* for short, is the main input routine of many programs. A lexer translates a stream of characters into a stream of *tokens*, groups of characters at a somewhat higher logical level. For example, the lexer in a C compiler might report that the 25 input characters `halfagadro += 6.02e23 /2;` constitute 6 tokens: an identifier `halfagadro`, two operators `+=` and `/`, a double constant `6.02e23`, an int constant `2`, and the punctuation `;`. The 3 whitespace characters are ignored.

A *parser* analyses a stream of tokens checking its validity with respect to a context-free grammar, and creating a corresponding *structured* description (parse tree). Continuing the above example, a parser might conclude that the above token sequence is a valid assignment statement consisting of an identifier, an assignment operator, and an expression.

Regular expressions can conveniently describe the lexical units required for many application. The unix utility *lex* and derivatives will semi-automatically construct a lexer from regular expressions. *Yacc* and derivatives are companion utilities that convert a context-free grammar into a parser. To be slightly more precise, lex & yacc convert source files into C (C++, Java, ...) programs incorporating tables built from the regular expressions and grammar, plus some library modules and various snippets of code that you provide with each regular expression and grammar rule. The code is then compiled, linked, and run.

In this assignment you'll use lex and yacc (or equivalents) to build a crude html formatter. Your program will produce formatted output reflecting the basic html commands: things like left-filled paragraphs, line breaks, unnumbered lists and headings. (You will be generating plain ASCII output, so you won't be doing font/size changes or other fancy stuff.) I have provided most of the formatting code, so you can concentrate on the lex/yacc end. Maybe we shouldn't call it "html"; a better view is that you'll actually be processsing **R**uzzo's **A**wesome **T**ext **M**arkup **L**anguage. The following summarizes ratml syntax to be used in this assignment.

At the lexical level, a ratml file contains *text*, *tags*, and *escapes*. Escapes are the simplest: `&`, `<`, and `>` in the midst of text specify the ampersand, less than, and greater than characters, &, <, and >, respectively. Escapes are case sensitive; e.g., you must use `&`, not `&AMP;`.

Tags consist of certain text enclosed in < and > brackets. Tags control formatting. We distinguish two types of tags.

Comment tags begin with the sequence `<!--` and end with the *nearest* subsequent `-->`. Note that this precludes nested comments. The enclosed string may include newline characters. Hint: this regular expression is a variant of the solution to problem 1.22 in homework 4.

Simple tags look like `<xyz>` or `</xyz>`, where the tag name xyz is a letter, followed by zero or more letters or digits, and many be upper case, lower case, or mixed, but is case-sensitive, i.e., `<p>` ≠ `<P>`. No whitespace is allowed within `< >`.

Finally, text is everything else—everything in the file that isn't either part of a tag or escape. Text is broken into "words" by escapes, tags, or whitespace characters (space, tab, newline). The whitespace is neither a token, nor part of one. (For our purposes, a word is a consecutive sequence of non-whitespace, non-escape, non-tag characters. E.g., `Yo!&<P>Y o!` contains 5 tokens: `Yo!`, `&`, `<P>`, `Y`, and

o!.) Note that an ampersand not followed by one of the recognized escape sequences, or a < bracket not followed by a tag name, should be returned as text tokens.

That's it for the lex level: successive calls to `yylex()` (the lexer's entry point) should return successive tokens of the input ratml file, until EOF.

For the yacc part, here's a brief description of ratml, and what the output should look like.

By the phrase "X is enclosed in a Y wrapper", I mean the source looks like <Y>X</Y>.

Each ratml document is enclosed in the HTML wrapper, and consists of a header section (enclosed in a HEAD wrapper), and a body section enclosed in a BODY wrapper. The header just consists of a TITLE wrapper around a text string that is the document title (the string shown in the browser's title bar, for example). You should print the title on a line (or lines) by itself.

The body contains a sequence of words, ratml escapes, paragraph separators <P>, and unordered lists. These may be nested. Escapes should be rendered as the indicated symbol, i.e., & < > should print as &, <, >, respectively. Successive words/escapes should be printed as filled paragraphs, with one space between each word, according to the prevailing left margin. Maximum line width is 80 characters. <P> should end a paragraph and leave a blank line. (<P> is not a wrapper; there's no </P>.) The line break tag <BR> should end the current line, but not insert a blank line.

The unordered list wrapper UL causes the text within it to be indented 4 spaces farther to the right (i.e., the prevailing left margin is increased by 4). Within the list, list item tags <LI> cause a line break, and place a * two spaces to the left of the prevailing left margin, to produce a "bulleted list". Lists may be nested.

Unrecognized ratml tags should just be treated as plain text.

At the end of any wrapper, the prevailing rendering parameters (e.g., margin) should revert to their values as set before the start of the wrapper. E.g.,

```
Specifically:
<UL><LI>item 1
  <LI>item 2 <UL><LI>inner list</UL>
  <LI>item 3
</UL>
```

becomes something like:

```
Specifically:
  * item 1
  * item 2
      * inner list
  * item 3
```

As mentioned earlier, most of the formatting code is provided. It walks through a tree structure reflecting the nested list of page elements, and prints them appropriately. So your job is really to

- build a lexer to break the input file into appropriate tokens and
- build a grammar allowing you to parse the resulting token stream and to build the corresponding tree.

Format of the tree is documented in the skeleton file provided to you. Note that for full credit, your solution must not generate lex/yacc errors, even if the resulting code produces the "correct" formatted output. In particular, clean up all "shift/reduce" and "reduce/reduce" errors reported by yacc. These are usually, but not always, symptoms of grammar ambiguity.

Where to start: the course web page links to a .zip file with Makefile, lex/yacc examples and skeletal lex/yacc programs that illustrate the communication between lexer and parser, as well as providing some convenient utility routines for this assignment and the core of the formatting code.

Please, name your files `ratml.l, ratml.y`, and include your name and student number in comments in your program, as indicated in the skeleton files. You may develop on any convenient machine. To turn in your files, use "make clean" or equivalent to delete object files etc. that we can rebuild, bundle the rest into a tar or zip archive, then follow the link on the course web to upload it.

Here's a suggested plan:

- Run the skeleton once as provided just to see what it does
- Then work on the lex part.
  - change the LEXDEBUG flag in the Makefile so it just prints out the tokens found by lex.
  - run the skeleton again to see what it does in this mode
  - read the lex man page, concentrating on the section that describes the patterns
  - add *just one or two* new patterns at a time to html.l
  - *make up your own test file* in parallel, to test the new features as you add them. *Keep it simple*. I will *not* help debug anything on my test file until you can show me a working run on your test file...
  - if you get stuck on any new feature for more than 30 minutes, leave it for later. E.g., if you can't figure out how to handle escapes like `&` or comments, then omit them; come back later if time permits. The world will not end.
  - if all you have time to do is get some of the lex part done, that's ok. Write a short readme file explaining what works and what doesn't and turn it in. The lex half is half the asssignment.
- once you get a lexer that handles most or all of the tags, then start working on the grammar.
  - again, *only change a little at a time*, and *build your own, simple, test files*.
  - don't worry about building the tree at first; just get the grammar right (in particular, one with no yacc "conflicts").
  - I'd start by changing it to recognize the various tokens your lexer produces, but don't worry about structure, matching begin/end tags, etc. yet. Just build a grammar that says "a valid string is a list of tokens."
  - then start worrying about structure, e.g. `<title>...</title>`.
  - finally add stuff to build and print the tree.
- Don't hesitate to use the class email list to ask/answer questions.

Extra Credit: Try to do some or all of the following. Get the basic assignment working first, *and save a copy*, before starting any extra credit.

Case-insensitive tags: "real" html ignores case in tags. Mimic this. Hint: it's actually somewhat of a nuisance to do it at the regexp level; use the action routines in your lexer to help.

Other simple tags: Add support for html headings (`<H1>`, etc.), centered text, or other interesting tags. (I'm more interested in the regexp/grammar end of things than fancy C code, so pick features that impact those; e.g. allowing new tags to nest within each other and old tags. You'll find the code I provided already has hooks for some of this.)

Complex tags: are like simple tags, but have *parameters* between the tag name and the closing > bracket, e.g., `<P size=15 font="diamond<>jubilee">`. Parameters can be any set of keyword=value pairs, where values other than simple integers should be quoted. Parameters are largely irrelevant for our purposes, except that they complicate locating the end of the tag. The tag ends with the nearest > not enclosed in quotes (`"`); it may be on a different line from the start of the tag.

For purposes of showing that your solution works, have your parser (not lexer) print info about the tag and its parameters whenever you *first* do a reduction involving that tag (so as not to mess up your pretty formatting). Including an input line number in this printout would be nice.

Hint: look at "start states" in the lex documentation.

Long Comments: Most lex implementations have relatively small buffers for the tokens it processes (a few hundred characters). This is generally fine, except for comments, which can run to tens of thousands of characters. Change your handling of comments so the body of the comment isn't collected as a potential token. (Hint: start states again.)

Ordered Lists: Add `<OL>` and `</OL>` tags, perhaps with `<OL START=?>` (for the whole list) and `<LI VALUE=?>` (this and subsequent items) parameters. Add corresponding formatting code, and make your grammar enforce obvious rules like not ending a `<UL>` with an `</OL>`.

Other: If there are other things you would like to try, ask. As noted, parts involving the lex/yacc rules are of more interest than adding straight C code.

When you turn in your assignment, put the basic solution in one subdirectory and your extra credit solution in another, including sources, a Makefile, etc. *Include a README-EC* describing what you've done, and one or more *test files* to show off its capabilities. Don't include .o or other generated files. Bundle all in a zip or tar archive and turn it in all at once.