Due:

This program is a layout engine for the ordered list portion of HTML. Let us say that the grammar for the part of HTML that describes ordered lists looks like the following. HTML tags, the tokens of the language, look like the symbols usually used in BNF for non-terminals. So in our grammar something in angle brackets is a terminal, and something not in angle brackets is a non-terminal.

\[
\begin{align*}
\text{list} & ::= \ <\text{OL}> \ \text{listBody} \ </\text{OL}> \\
\text{listBody} & ::= \ <\text{LI}> \ \text{item} \ \{ \ <\text{LI}> \ \text{item} \ \} \\
\text{item} & ::= \ \text{stuff} \ \{ \ \text{list} \ \} \\
\text{stuff} & ::= \ \text{all characters, including white space, between any two HTML tags, however a token cannot consist of just white space, nor can it contain the character “<“.}
\end{align*}
\]

Your program will input a text file that contains HTML lists and output a file with the lists nicely formatted. For example, here is some sample input:

\[
\begin{align*}
<\text{OL}> <\text{LI}> \text{wake up} <\text{LI}> \text{eat} <\text{OL}> <\text{LI}> \text{Capt. Crunch} <\text{LI}> \text{Crunchy Frogs} </\text{OL}> \\
<\text{LI}> \text{watch TV} <\text{LI}> \text{drive to school} \\
<\text{LI}> \text{study} <\text{OL}> <\text{LI}> \text{Mickey Mouse} <\text{LI}> \text{Basket Weaving} \\
<\text{OL}> <\text{LI}> \text{open weave} <\text{LI}> \text{closed weave} </\text{OL}> \\
<\text{LI}> \text{Computers} </\text{OL}> \\
<\text{LI}> \text{drive home} \\
<\text{LI}> \text{sleep} \\
</\text{OL}>
\end{align*}
\]

The output should be something like:

1. wake up
2. eat
   i. Capt. Crunch
   ii. Crunchy Frogs
3. watch TV
4. drive to school
5. study
   i. Mickey Mouse
   ii. Basket Weaving
      a. open weave
      b. closed weave
   iii. Computers
6. drive home
7. sleep
Step I. Write a scanner that tokenizes the input. Our tokens will be null-terminated strings. For the above input, the stream of tokens is:

<OL>
<LI> wake up
<LI> eat
<OL>
<LI> Capt. Crunch
<LI> Crunchy Frogs
</OL>
<LI> watch TV

and so on. Notice that the tokens for “stuff” can contain spaces. The scanner should be:

```c
int getToken( char* token );
```

Regard anything that is enclosed in angle brackets as a “tag” token. Return any such token without complaint; it will be up to the parser to say if a token matches what is expected. When the scanner is gathering characters for “stuff” it will quite when it reads a “<“. It will then need to do an `ungetc()`. The example scanner on the web should be especially helpful. The start state should discard white space (blanks, tabs, newlines) until it changes state to the stuff-state or the tag-state.

Step II. Write a recursive descent parser for the language. Assuming that the scanner is working, this should be easy. (If your scanner is not working, then create an input file that consists of one token per line, so that `getToken()` is trivial.) The parser should print an error message where it gets a token it does not expect, and then just quit. Remember to write the parser in the style of the book: each recognition routine on entry expects an un-consumed token in a global variable, `token`. Each recognition routine ensures that this remains true for any routine that is called after it. You will need a recognition routine for each production of the grammar: `listR( )`, `listBodyR( )`, `itemR( )`, and `stuffR( )`.

Step III. Comment the grammar with action symbols that show where the output routines will be called. The following will work:

```
list ::= <OL> #startList listBody </OL> #endList
listBody ::= <LI> #addItem item { <LI> #addItem item }
item ::= stuff { list }
stuff ::= all characters, including spaces, between any two HTML tags #printStuff
```

Now write the output routines and call them from the appropriate places in the parser. The output routines will need some global variables such as `currentIndentLevel`, and a stack of item counters (since lists can be contained within lists you need a stack.) The top of the stack is the item counter for the currently active list.
Notes:

1. This is (possibly) easier than it looks. Write this in modular fashion; the scanner, the parser functions, and the output functions are separate pieces. Most of them are fairly short. The program will be about 300 lines total.

2. Don’t worry too much about Roman numerals and alphabetical item numbering. Just use ordinary numbers to start with and add more sophistication later if you have time. Naive Roman numerals are OK (ie. Print “X” for each 10 followed by “V” for each remaining 5, followed by “I” for each remaining unit, so 29 comes out XXVIII, which is not really correct.)

3. Follow each item number with a tab char ‘\t’ so items line up, if you want.

4. For nesting levels greater than three, just use “*” for each item.