Homework 3, CSCE 240, Fall 2017

Assignment

You are given an input file with a large number of English sentences parsed using the code from Dan Jurafsky's group at Stanford. ("Parsing" is almost the same as "diagramming".)

There are four kinds of symbols in this file.

1. There is an open and close `<parse>` and `</parse>` because this comes from a package that outputs XML.

2. The open and close parentheses work just like parentheses do in an algebraic expression.

3. The symbols in all caps are the tags for parts of speech, parts of sentence, etc. For example, NP means "noun phrase", VP means "verb phrase", VB means "verb", NN means "noun", NNP means "proper noun", NNS means "noun plural", JJ means "adjective".

4. There are actually English words, which you should recognize.

You are to read these input lines and convert them to XML, with the tags becoming the XML tags. For example, an input line *"Food is enjoyable."* would parse as

```
<parse> (ROOT
          (S
            (NP
              (NNP Food)
            )
            (VP
              (VBZ is)
              (ADJP
                (JJ enjoyable)
              )
            )
            (PERIOD PERIOD)
          )
        ) </parse>
```

(Actually, you would get all this on one line, but I have indented to show the nested parentheses.)

For this sentence, your output should be

```
<ROOT level="1">
  <S level="2">
    <NP level="3">
      <NNP level="4">
        Food
      </NNP>
    </NP>
    <VP level="3">
      <VBZ level="4">
        is
      </VBZ>
      <ADJP level="4">
        <JJ level="5">
          enjoyable
        </JJ>
      </ADJP>
    </VP>
    <PERIOD level="3">
      PERIOD
    </PERIOD>
  </S>
</ROOT>
```

Basically, then, your program should read the parsed version, tokenize, and then output an XML version of this with proper XML tags and an attribute that indicates the level.

Filtering

You will probably find this easier to do if you first filter the input so that it is easier to tokenize. For example, the sample sentence

```
<parse> (ROOT (S (NP (NNP Food)) (VP (VBZ is) (ADJP (JJ enjoyable)))
    (PERIOD PERIOD))) </parse>
```

is going to be a pain to read through character by character to determine what kind of symbols exist. It is probably much easier if you first convert this to something you can tokenize on blank spaces, like
(ROOT (S (NP (NNP Food)) (VP (VBZ is) (ADJP (JJ enjoyable))) (PERIOD PERIOD)))
If you do this, then the utilities code `Next` function will break on blank spaces and return individual tokens.

This filtering process could be a function to which you pass the raw input string and that removes the "parse" XML items and then replaces open and close parentheses with the parenthesis surrounded by blank spaces. (Note that `Next` only cares about the existence of blank space to determine token boundaries, not the number of blank spaces.)

Parsing

The right way to deal with the nesting is with a stack, and there is a `stack` container in C++ that you should use for this purpose.

When you encounter an opening token (viz., an open parenthesis), you should create an opening XML tag like `<NP level="3">` and then push the closing XML tag `</NP>` onto the stack. Then, when you encounter a close parenthesis, that should match up with the most recent open parenthesis, and you can pop the closing XML tag from the stack.

And of course you need to keep track of the number of open parentheses you encounter, because that's what indicates the `level` that will go into the XML as an attribute. And that `level` value is what you can use to determine the number of spaces to indent.

Normally, if you were parsing putative proper XML, you would also use the stack to determine if the XML was properly nested. That is, if you hold on to the most recent open tag, and it doesn't match the close tag you pop from the stack, then the XML isn't properly nested.

In this assignment, *you can assume* that the XML is properly nested. We will trust Jurafsky's graduate students to have done this correctly.

Input and Output

You should read from an input file but then write to standard output.