CMPSC 461: Programming Language Concepts

Programming assignment 1: Recursive Descent Parsing
Prof. G. Tan
Total: 20 points.

Your assignment is to use Python to write a recursive descent parser for a simplified HTML language. The lexical syntax is specified by regular expressions:

Token	Extended regular expression definition
`STRING`	(LETTER | DIGIT)+
`KEYWORD`	`<body>` | `</body>` | `<b>` | `</b>` |`<i>` | `</i>` | `<ul>` | `</ul>` | `<li>` | `</li>`

In the above, LETTER is any lower or upper-case letter and DIGIT is any digit. An arbitrary number of whitespace can appear between tokens.

You may already know that `<b>...</b>` is the tag for bolded text in HTML, `<i>...</i>` for italicized text, `<ul>...</ul>` for an unordered list, and `<li>...</li>` for a list item.

Note in the above syntax we use a notation for non-terminals that is different from the notation we used in lectures. The reason is that symbols `<` and `>` are terminals in the HTML language; so we can no longer use the notation `<A>` for non-terminals. Instead, we use names in upper-case letters for non-terminals. Therefore, in the above token syntax, `KEYWORD` is a non-terminal, while `<body>` is a string of terminals that starts with terminal `<` and ends with terminal `>`.

Using the same notation, the syntax of the simplified HTML language is specified by the following E-BNF grammar, where `WEBPAGE` is the start non-terminal:

```
WEBPAGE -> <body> { TEXT } </body>
TEXT -> STRING | <b> TEXT </b> | <i> TEXT </i> | <ul> { LISTITEM } </ul>
LISTITEM -> <li> TEXT </li>
```

Note that { and } are meta-symbols in E-BNF.

An example expression in the language is as follows:

```
<body> google <b><i><b> yahoo</b></i></b></body>
```

This programming project is broken down into the following series of tasks.

1. (3 points) Write a class Token that can store the value and category of any token.

2. (7 points) Develop a lexical analyzer. You should start by drawing on paper a finite state automaton that can recognize types of tokens and then convert the automaton into code. Name the class Lexer and ensure that it has a public method nextToken(), which returns a Token. Lexer should have a constructor with the signature "__init__ (self, s)", where s is the string representing the string to be parsed. If nextToken() is called when no input is left, then it should return a token with category EOI (EndOfInput). If the next terminal string is not considered a token by the lexical syntax, then return a token with category Invalid.

   After you have coded your lexer, thoroughly test it using some test cases before coding the next part. Our example parser code posted in Canvas shows you how the lexer can be separately tested.

3. (10 points) Write a syntactic analyzer which parses the tokens given by Lexer using the recursive descent technique. Name this class Parser. Like Lexer, Parser should have a public constructor with the signature "__init__ (self, s)". This input string should be used to create a Lexer object. There should also be a method "run(self)" that will start the parse.

   As you parse the input, Parser should output the token that was matched in a new line with indentation reflecting the nesting structure. In particular, when there is a token nested inside a tag such as `<body>`, the token's indentation should be two spaces more than the indentation of the outer tag. An example will be provided shortly.

   Hint: to have proper indentation, one approach is to make your parser methods for non-terminals `WEBPAGE`, `TEXT`, and `LISTITEM` take an indentation level as a parameter. When a parser method is invoked inside another parser method, the indentation level should be increased by one.

   Whenever the parser comes across a token that does not fit the grammar, it should output a message of the form "Syntax error: expecting expected-token-category; saw token" and immediately exit.

Note that for this assignment you are allowed to base your code on the example lexer and parser we discussed in class.

```
<body>
  google
  <b>
    <i>
      <b>
        yahoo
      </b>
    </i>
  </b>
</body>
```

Figure 1: Sample output.

In order to test your file, you will have to create a test that creates a Parser object and then calls the run() method on the object. For example, the code:

```
parser = Parser ("<body> google <b><i><b> yahoo</b></i></b></body>");
parser.run();
```

should have the output given in Figure 1.

Submission format: Make sure that your code can be compiled and run on those Linux machines in the Westgate 135 lab (e5-cse-135-01.cse.psu.edu to e5-cse-135-38.cse.psu.edu). You can remotely access these machines via ssh (after VPN into the Penn State network). We will grade your submission on those machines. Make sure your code is well tested. We provided one test for your reference, but you should design your own test cases to test your code.

Please ensure that your file has a comment that includes your name, Penn State network user id, and a description of the purpose of the file. Please include comments for each method, as well as any complicated logic within methods.