Assignment 3

In this assignment we will do a bit of "lexical analysis" on a text. We will read a text and record information as we encounter it. We will concentrate on the following:

1. Words
2. Punctuation marks
3. end of line marks

Our goal would be to refine such a text, as well as provide some information about it, as follows:

1. Convert three consecutive periods into an "ellipsis", two consecutive dashes into an "em-dash" which is a special longer-than normal dash (you can read all about hyphens, en-dashes and em-dashes and their uses in any good writing-style document).
2. Convert "programmer's quotes: `"`" into more fancy open-close quotation marks.
3. Count the number of words, sentences, and paragraphs.
4. Reformat the text to a given width limit, with two variants: One preserves any earlier newline characters (i.e. shorter lines allowed) the other tries to make all lines as full as possible (without trying to add extra spaces to really fill it all out).

We start by reading a large string of the entire text, then breaking it into "terms". Here is the definition of the `term` type:

```
data Term = Word String | Punc Char | Space | NewL | Para deriving (Eq, Show)
type Text = [Term]
```

So a term is either a word (containg a string information) or a punctuation (containing a character information), a space (marked by the space character), a newline (marked by the newline character) or a paragraph mark (a paragraph consists of 2 or more consecutive newline character). As a quick example, the string: `"Hello there!\nNew"` will become:

```
[Word "Hello", Space, Word "there", Punc '!', NewL, Word "New"]
```

Your assignment is to provide code that does a number of processing steps to a string. You can run the automated tests with:

```
# Run in shell
# Compile first
ghc assignment3
./assignment3 tests
```

You should ADD YOUR OWN TESTS; The ones provided are by no means exhaustive.

Remember that you will need to create STUB implementations for all your functions in advance, in order for the tests to compile.

You can test how your script performs on a particular text file (once you have implemented the needed functions) with the following shell commands:

```
# Run in shell
# Compile first
ghc assignment3
./assignment3 stats < inputfile
./assignment3 long < inputfile   > outputfile
```

This assignment is meant to exercise your skills in writing pattern-matching functions. Please avoid other techniques for writing these functions.

First, a brief overview of the functions you need to implement and/or are given:

1. processText is a function that is provided to you, but which uses functions you will need to write. It takes as input a string and returns a Text value for that string, which is the result of the initial processing of the string followed by a number of post-read steps.

2. readText is a function you will need to write. It takes in a string and reads it through, turning it into a Text value by converting each character to an appropriate term. The rules are as follows:

   - Any punctuation character becomes a Punc value. You can use the function isPunctuation to determine if a given character is a punctuation character or not.
   - A space character is converted to a Space value. (we will not consider tabs as characters that occur in our texts, but if you want to consider them then treat them as 4 spaces.)
   - The newline character is converted to the NewL value.
   - Any other character is considered a word character, any a as-large-as-possible sequence of consecutive such characters forms the string in a Word value. You can use the next function to help you in handling this case.

3. combineChar is a helper function to be used with question 2. It takes as input a character and a Text value which represents the read rest of the string. For example if we were reading the string "cat?" then the combineChar function may be called to operate on the character 'c' and the Text value [Word "at", Punc '?']. The desired behavior in this case would be to combine the 'c' character with the "at" string and result in the text value [Word "cat", Punc '?']. A little bit deeper in to the recursive calls it may be called to operate on the character 's' and the text value [Punc '?'], in which case it would produce [Word "s", Punc '?']. Make sure you understand these two examples.

   Here is how combineChar is meant to behave on a character ch and a text value txt:

- If the first element in `txt` is a `Word s` value, then this means we are in the middle of reading a word, and we need to replace that value with one where the new character has been prepended to the string `s` in the `Word` value.
- In all other cases, this character is meant to start a new word and we must forms a new `Word` value with that character and place it at the front of the `Text` list. You can handle the empty list case together with this case in one clause.

4. The big workhorse is the `commonSubstitutions` method, which recursively goes through a `Text` value and returns a `Text` value back, attempting to perform various transformation along the way. You will spend a lot of time on this function incrementally adding behavior. It will be a big list of pattern-match cases each handling a different kind of transformation then recursively continuing on the rest.

   - Start with a basic recursive function which returns an empty string when given an empty string, and which simply recursively traverses the elements and not losing any elements in the process. At this point the result of your function should basically be a new list that looks just like the original one (but actually had each element "visited" by the function). You should add all the following cases before the trivial recursive step, using that as a catchall: If the current term does not need any transformation done to it, we continue looking at the remaining terms.
   - The first thing you should add is a transformation that converts three consecutive periods into a single ellipsis element. There is a variable `ellipsis` that you can use for that, which uses the special Unicode character for an ellipsis.
   - Next you should add a transformation that converts two consecutive dashes/minuses into an "emdash". Again there is a `Term` value defined for you for that called `emdash` which uses the special Unicode character for an emdash.
   - Next up you need to handle cases of words separated by a dash. For example the two-word combination `"twenty-five"` would have been parsed as the three terms `Word "twenty"`, `Punc '-'` and `Word "five"` in that order. Your pattern should match such combinations and replace them with a single `Word "twenty-five"` combination. Make sure the recursive call allows you to catch the case of two dashes like in `"five-and-twentieth"` by combining the first dash into a compound word but allowing that word to be part of the recursive match.
   - Next up you need to handle apostrophes. There are three cases you need to handle: Where the apostrophe is in the middle of the word (like `"isn't"`, when it is at the end of the word (`"parents'"`) or at the beginning indicating omitted text (`"'re"`). You should handle all three cases, and be careful about the order. Note: You can represent the single quote/apostrophe character as `'\''`.
   - Next up you need to handle periods which are parts of abbreviations. We will handle only a specific set of abbreviations here, so you'll need a case for each of Mr, Mrs, Ms, Dr and Prof. If any one of those is immediately followed

by a period, you need to combine that word and that punctuation mark into one word.

- Next up we have a case to handle roman numerals. A roman numeral for us is a sequence of any of the letters IXVL, either uppercase or lowercase, with possible repetitions, like VII or ix. You should write the helper method isNumeral which takes a string and returns a boolean as to whether that string is a numeral. Then use this helper to handle the cases of a word followed by a period: If that word is a numeral then combine it with the period into a new word, otherwise keep the word as is and continue recursively with the rest (make sure that you still recursively examine the period, in case it is part of an ellipsis for example).
- Lastly, you will need two more cases to handle newline/paragraph related topics. Namely: Any sequence of two or more newline elements NewL needs to be replaced by a single NewL followed by a single Para. You can do this with one case turning a NewL pair into a NewL and Para combination, and another case that reads any NewL which follows a Para and simply skips it.

5. Next up, you need to implement a method called smartQuotes, which takes a Text and replaces the normal quotes with nicer open/close quotation marks, returning a Text value. Use the provided openQuote and closeQuote values. You will need a helper method that uses an extra boolean parameter to remember whether you are in the middle of a quotation (in which case the next quotation mark you see should be a closing one) or outside (in which case the next quotation mark you see should be an opening on).

6. Next up you are asked to implement a series of "counting" functions. Given a text, these functions count various things and return the integer count:

   - countWords simply counts how many word terms there are.
   - countSentences counts how many sentences there are. For us, a sentence is any sequence that ends in a period, question mark or exclamation point.
   - countLines counts how many lines there are. A line happens when the NewL term is encountered (we do not count the empty lines formed by paragraphs. It also occurs at the very last term, *unless* that term is a Para term.
   - countParagraphs counts how many paragraphs there are. A paragraph happens when the Para term is encountered or at the very last term (even if that term is not a Para term).

7. Next up you should write a printStats method. It takes as input a Text value and produces an IO () action which prints stat information. You should be producing output that looks like this:

```
Words: 234
Sentences: 23
Lines: 12
Paragraphs: 5
```

This will be a simple do sequence of actions, calling on the functions you wrote on the previous step.

8. We will now put together a set of functions whose goal is to print out the text to produce a string.

   - termToString should take as input a Term and convert it to a String. Word terms result in the corresponding word, Punc terms produce a string containing just that punctuation, Space terms become a single space string " ", and NewL and Para terms both become a string containing a single newline character, "\n".
   - toString takes a whole Text and converts it to a string, by simply using the termToString method to turn each term into a string, then concatenating those.
   - eliminateNewlines eliminates the newline terms NewL as follows: A NewL term that is followed by a paragraph term is simply eliminated, while a NewL term that is not followed by a paragraph term is replaced by a Space term. Don't forget to recursively traverse the entire term list.
   - splitOnParagraph takes a Text value and returns a list of Text values by splitting on the Para terms. The resulting Text values should not contain the Para terms. Make sure to NOT create an extra empty Text value if the last term is a Para term.
   - toParagraphs combines these as follows: It takes a Text value and must result in a list of strings. It does this by first using eliminateNewlines followed by splitOnParagraph, and then it applies the toString function to each of the result Text elements to produce corresponding String elements. You can use a list comprehension for part of this function if you find it helpful.
   - Lastly, printParagraphs takes as input a list of strings and produces an IO () action which prints those strings as paragraphs as follows: It prints the string/paragraph followed by a newline character; then if we are not at the end of the list it prints an extra newline character to create an empty line, then recursively prints the rest of the list. A simple do statement should work for this.

Here are initial file contents:

```
module Main where

import Test.HUnit
import Data.Char (isAlpha, toUpper, isPunctuation)
import System.Environment (getArgs)

data Term = Word String | Punc Char
          | Space | NewL | Para deriving (Eq, Show)
type Text = [Term]


openQuote = Punc '\8220'
```

```
closeQuote = Punc '\8221'
ellipsis = Punc '\8230'
emdash = Punc '\8212'


processText :: String -> Text
processText = smartQuotes . commonSubstitutions . readText

tests = TestList [
   TestCase $ assertEqual "combineChar"
      [Word "cat", Punc '!'] (combineChar 'c' [Word "at", Punc '!']),
   TestCase $ assertEqual "combineChar"
      [Word "t", Punc '!'] (combineChar 't' [Punc '!']),
   TestCase $ assertEqual "combineChar"
      [Word "t", Space] (combineChar 't' [Space]),
   TestCase $ assertEqual "commonSubstitutionsEmdash"
      [Word "say", emdash, Word "hello"] (processText "say—hello"),
   TestCase $ assertEqual "commonSubstitutionsEllipsis"
      [Word "some", Space, Word "ellipsises", ellipsis, Punc '.']
      (processText "some ellipsises...."),
   TestCase $ assertEqual "commonSubstitutionsMrAndApostrophe"
      [Word "Mr.", Space, Word "Smith's", Space, Word "work"]
      (processText "Mr. Smith's work"),
   TestCase $ assertEqual "commonSubstitutionsDashed"
      [Word "seventy-five"]
      (processText "seventy-five"),
   TestCase $ assertEqual "commonSubstitutionsDashedTwice"
      [Word "five-and-twentieth"]
      (processText "five-and-twentieth"),
   TestCase $ assertEqual "isNumeral" False (isNumeral ""),
   TestCase $ assertEqual "isNumeral" False (isNumeral "IGF"),
   TestCase $ assertEqual "isNumeral" True (isNumeral "ILX"),
   TestCase $ assertEqual "commonSubstitutionsNumerals"
      [Word "I.", Space, Word "II.", Space, Word "III.", Space,
       Word "IV.", Space, Word "V.", Space, Word "VI.", Space,
       Word "vii.", Space, Word "IX.", Space, Word "X.", Space,
       Word "XI.", Space, Word "Normal", Punc '.']
      (processText "I. II. III. IV. V. VI. vii. IX. X. XI. Normal."),
   TestCase $ assertEqual "commonSubstitutionsParagraphs"
      [Word "word", NewL, Para]
      (processText "word\n\n\n\n"),
   TestCase $ assertEqual "commonSubstitutionsParagraphs"
      [Word "word", NewL, Para, Word "More"]
      (processText "word\n\n\n\n\n\n\nMore"),
   TestCase $ assertEqual "smartQuotes"
      [Word "here", Space, Word "be", Space, openQuote,
       Word "double", Space, Word "quotes", closeQuote]
      (processText "here be \"double quotes\""),
   TestCase $ assertEqual "countLines1"
      3
      (countLines $ processText "one\n\ntwo\nthree"),
   TestCase $ assertEqual "countLines2"
      3
      (countLines $ processText "one\n\ntwo\nthree\n"),
   TestCase $ assertEqual "countLines3"
      3
```

```
      (countLines $ processText "one\n\ntwo\nthree\n\n")
   ]

main :: IO ()
main = do
   args <- getArgs
   s <- getContents
   let txt = processText s
   case args of
      ("tests" : _) -> do runTestTT tests
                          return ()
      ("stats" : _) -> printStats txt
      ("long" : _)  -> (printParagraphs . toParagraphs) txt
      _             -> (printParagraphs . toParagraphs) txt
```