COMP304 Programming Languages (2017)
Assignment 1 : Basic Haskell
(Due midnight, Wednesday 9 August)
Revised Version

This assignment asks you to write some small programs in Haskell. The aim is to help you to understand how you can use the basic constructs of Haskell we have looked at so far. You should think about how the programs can be implemented using these constructs, try to make your programs simple and easy to read, and to be reasonably efficient (e.g. do not use a quadratic time algorithm if a straightforward linear time one can be used instead). If you can see different ways of implementing/expressing the solution, briefly describe a couple of possibilities and explain why you chose a particular one, or show a couple of versions in detail and compare them.

You should submit Haskell source code for all of the programs, using the online submission system. All Haskell code must be suitably commented to explain how it is intended to work, and should include code for test cases. Other discussion of what you have done and the results you obtained, may be be embedded as comments (preferably using literate scripts), or presented in a separate document.

Note: Your assignment should be submnitted as a single literate Haskell script, using the function names specified below.

1. Extracting words from text.

   (a) Write a function (`lineToWords`) that takes a string (a list of characters) and returns a list of the words occurring in that string (in the same order that they occur in the input), where a word is defined to be a maximal string of letters and/or digits. For example, if the input is

   ```
   "For example, this sentence has 7 words!"
   ```

   the output should be

   ```
   ["For", "example", "this", "sentence", "has", "7", "words"]
   ```

   Note that the maximality requirement ensures that a sequence of letters and/or digits is not broken into multiple words.

   A nice way to design this function is to pass it a function which it uses to read a word from the input, or perhaps to determine whether the next character is part of a word, so that it is easy to change the definition of what constitutes a word to suit different applications (e.g. to allow underscores or hyphens, include quote marks or decimal points in numbers).

   (b) Extend your function from (a) so that the input is a list of strings, and the output is again a list of all words occurring in the input (in the same order that they occur in the input).

   You may read the input from a file, so the file is treated as a list of strings (one string per line), if you want to find out how to do it.

   This function should also be called `linesToWords`. If you write a version that reads frmo a file, as suggested above, you should still submit a version that takes a list of strings as an argument and returns a list of strings as a result.

(c) Further extend your function from (b) so that the output is a list of the words occurring in the input, along with the line number and position within that line, at which it occurs. That is, the output is a list of triples, each consisting of a word, and the line number and position within that line (both counting from 1), at which it occurs. For example, if the input is

```
["For example,", "in this sentence,", "there are 9 words."]
```

the output should be

```
[("For", 1, 1), ("example", 1, 5), ("in", 2, 1), ("this", 2, 4),
("sentence", 2, 9), ("there", 3, 1), ("are", 3, 7), ("9", 3, 11),
("words", 3, 13)]
```

This function should be called `posOfWords`.

2. Text formatting

(a) Write a function (`wrapLines`) which takes as input a natural number, N (where $N > 1$), and a list of strings, and formats the second input according to the following rules:

- The output is a list of lines, where a line is a string, containing the same words as the input, in the same order. In this case we will define a word to be a maximal sequence of non-space characters, so that punctuation etc. is retained.
- No line may contain more than N characters.
- Lines do not start or end with spaces.
- Consecutive words on the same line are separated by a space.
- Every word is entirely contained in one line, unless it contains more than N characters, in which case the first $N-1$ characters are placed on one line and followed by a hyphen ("-"), and the rest of the word is placed on the next line (following the same rule, so that a really long word may be broken over several lines).
- Each line is filled as much as possible, so a new line is started only when the next word will not fit on the current line.

(b) Extend your function in (a) so that each line of the output, except the last, is padded with extra spaces to that lines are right justified (i.e. the line has exactly N characters and the last character is not a space).

The extra spaces should be spread out, so that you don't get big blocks of white space.

You might also try to avoid creating vertical "channels" of white space running over multiple lines.

This function should be called `justifyLines`.

3. Text encoding

A simple way of encoding text, which will give reasonable compression if there are lots of repeated words, is to translate the text into a lexicon (a list containing one occurrence of each words that occur in the text), and a list in which each word of the text is represented by its index in the lexicon.

For example, if the input is

```
["The more I learn, the more I know.",
"The more I know, the more I forget."]
```

the output might be

```
(["The", "more", "I", "learn", "the", "know.", "know,", "forget."],
[1, 2, 3, 4, 5, 2, 3, 6, 1, 2, 3, 7, 5, 2, 3, 8])
```

In this case, we are treating words as maximal sequences of non-space characters, and the lexicon simply lists words in the order of their first occurrence.

Write a basic encode function, which takes a text (list of lines), and encodes it as illustrated above, and a decode function, which takes the output of the encode function and reconstructs the initial text. The result need not break the text into lines in the same way — or you may chose to encode line breaks as well.

Think about ways in which you might improve on this basic approach. Would it help to store the lexicon in alphabetical order or perhaps frequency order (i.e. most frequent words first)? Can you reduce duplication in the lexicon, e.g. by storing punctuation separately?

These functions should be called `encode` and `decode`. If you want to write versions that keep track of new lines, they should be called `encodeLines` and `decodeLines`.