Text Processing

In this mini project we are interested in text processing. Some typical tasks are counting the number of words in a given text, or reformatting a paragraph of text such that it is pleasant to read. For creating sample input to test your functions, Scala's multi-line strings are especially useful. A multi-line string is created by enclosing a piece of text in triple quotes, like this[1]:

```
val sample = """The feisty fawn
jumps over
the edgy eft."""
```

Most of the functions that you are going to define operate on lists of characters. A Scala string can be converted to a list of characters by calling its `toList` method:

```
val chars = sample.toList
```

To print lists of characters in a readable way, we first convert them into strings using the `mkString` method[2]:

```
Console.println(chars.mkString(""))
```

Extracting lines

In the first part, we are interested in functions that convert an unstructured piece of text into lines and words. A line is represented by a value of type `List[Char]`. More specifically, it is a sequence of characters that does not contain line break characters (line break characters are `\n`, `\r`, and `\f`).

1. Define a function `lines` that converts an arbitrary list of characters into a list of lines. Your function should have the following signature:

```
def lines(chars: List[Char]): List[List[Char]]
```

The semantics is defined more precisely by the following example:

```
val foo = """feisty
fawn"""
Console.println(lines(foo.toList))
```

should print out

```
List(List(f,e,i,s,t,y),List(f,a,w,n))
```

2. Define a function `unlines` that turns a list of lines into a list of characters inserting line break characters between each two lines. The following example makes the semantics more precise:

```
unlines(List(List('f','e','i','s','t','y'),List('f','a','w','n')))
```

should yield

[1] Note that the resulting value is a normal Scala string.

[2] The single argument specifies the separation character between elements of the list.

```
List('f','e','i','s','t','y',\n,'f','a','w','n')
```

3. *Bonus:* Show that your implementations of `lines` and `unlines` satisfy the following equivalence:

```
unlines(lines(xs)) = xs
```

for all lists `xs` of characters. Hint: use structural induction as shown in the lecture! To simplify your proof, assume that line breaks are represented by a single `\n` character. Note that purely recursive definitions (that is, not using `span`, `foldRight` etc.) are easier to proof correct!

Counting words

In this part we are interested in counting the words contained in a given piece of text. A word is a non-empty sequence of characters that are not line break characters, white space, or delimiters such as `','` and `'.'`. We define `type Word = List[Char]`.

1. Define a function `words` that converts a list of characters into a list of words. Since word delimiters are often application-specific, the function takes a predicate which decides whether a particular character is considered as a delimiter. The signature is as follows:

```
def words(p: Char => Boolean)(text: List[Char]): List[Word]
```

For its precise semantics consider the following example:

```
words(ch => ch == ' ')(List('f','e','i','s','t','y', ' ', ' ', 'f','a','w','n'))
```

should yield

```
List(List('f','e','i','s','t','y'),List('f','a','w','n'))
```

2. Define a function `wordCount` that counts the number of words in a text.

```
def wordCount(text: List[Char]): Int
```

Building a word index

An index allows one to quickly find the places where a specific word occurs in a given text. It is structured as an alphabetically-sorted list of words indicating for each word the numbers of lines where it occurs. In Scala, we can represent an index as a value of the following type:

```
List[(Word, List[Int])]
```

The goal of this part is to define a function `buildIndex` that takes an arbitrary text as argument and returns an index of its words. Proceed in the following steps:

1. Convert a list of words into a list of pairs where each pair represents a single occurence of a word, pairing it with the line number where it occurs. Hints: the `indices` method of the `List` class returns a list of (integer) indices. For pairing, you can use the `zip` method (see Scala API documentation). Using for comprehensions simplifies this task.

2. Sort the list obtained in the first step according to the lexicographic order of the first component of each pair. One can access the components of a pair either by

   - pattern matching:

     ```
     myPair match {
       case (p, q) => ...
     }
     ```

- or by using the accessor methods `_1` and `_2`, as in `myPair._1`.

Note that characters are comparable using the `<` operator.

3. Collaps the list obtained in the previous step such that occurences of the same word are merged into a single component with the line numbers aggregated into a *sorted* list that contains *no duplicates*. For example,

```
(a, 7), (a, 5), (b, 7), (c, 6)
```

is collapsed into

```
(a, List(5, 7)), (b, List(7)), (c, List(6))
```

4. Define a `printIndex` method that prints indices generated in the last step in the following format:

```
...
testing: 4
text: 3, 5
...
```

Paragraph justification

In this part we are interested in justifying the lines of a paragraph, so that the resulting lines are well-balanced and the paragraph is pleasant to read.

Extracting paragraphs

We represent paragraphs as lists of words, instead of lists of lines, since we are going to rebuild their line structure. Paragraphs are separated by lines that do not contain any words (note that a line containing only whitespace is also separating). Moreover, paragraphs are non-empty, that is they contain at least one word.

1. Define a function `paras` that converts an arbitrary text into a list of its paragraphs:

```
def paras(text: List[Char]): List[List[Word]] = ...
```

Hint: the list methods `map`, `span`, `filter` are helpful. A list `xs` of lists can be "flattened" using `List.flatten(xs)`.

Rebuilding paragraphs

A particular strategy for *justification* is as follows: the words of a paragraph are arranged into a sequence of lines such that

- the length of each line does not exceed a fixed maximum n,
- the length of the longest word is less than or equal to n, and
- the number of lines is as small as possible[3].

[3] This property ensures that lines are non-empty.

1. Define a `take` function that takes a maximum line length and a list of words, and returns a maximally-sized list of those first words that still fit on a line (assume that words are separated by a single space character):

```
def take(max: int, p1: List[Word]): List[Word] = ...
```

2. Define a function `justify` that rebuilds the line structure of a paragraph according to the above strategy using the `take` function defined in the previous step.

```
def justify(n: int)(p: List[Word]): List[List[Word]] = ...
```

Bonus: Can you implement it using a single `foldLeft`? Here is a template:

```
def justify(n: int)(p: List[Word]): List[List[Word]] =
  p.foldLeft(??)(??).reverse
```

3. Define a function `justifyParas` that justifies all paragraphs in a given piece of text (it returns the text with paragraphs justified). Paragraphs should be separated by empty lines. The last word of the last paragraph is *not* followed by line break characters (or any other whitespace).

```
def justifyParas(n: int)(text: List[Char]): List[Char] = ...
```

Here is an example that shows `justifyParas` in action:

```
val sampleParas = """The feisty fawn
jumps
over the edgy eft."""

Console.println(justifyParas(12)(sampleParas.toList).mkString(""))
```

should print out

```
The feisty
fawn jumps
over the
edgy eft.
```