Code Guidelines

While implementing this exercise, please make sure that you don't change the definitions given in the template. In particular:

- Don't change the package declaration or the object name. If you are working in Eclipse or IntelliJ, the package and file names in the IDE must match the ones in the code.

- Don't change the function names.

- Don't change the given function signatures.

Respecting these guidelines allows us to be tremendously more efficient in correcting the submissions. Thank you for your collaboration.

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 gutsy gibbon
jumps over
the feisty fawn."""
```

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: List[Char] = sample.toList
```

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

```
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` and `\r`).

1. Define a function `lines` that converts an arbitrary list of characters into a list of lines. The following three subsequences have to be considered as line breaks: `List(\n)`, `List(\r)` and `List(\r, \n)`. If there are successive line breaks, the result should include empty lines. Your function should have the following signature:

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

The semantics is illustrated by the following example:

---

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

```
val foo = """feisty

fawn
"""
println(lines(foo.toList))
```
should print out
```
List(List(f,e,i,s,t,y),List(),List(f,a,w,n),List())
```

2. Define a function `unlines` that turns a list of lines into a list of characters inserting a `\n` character between each two lines.

The following example illustrates the function `unlines`:
```
unlines(List(List('f','e','i','s','t','y'),List('f','a','w','n')))
```
should yield
```
List('f','e','i','s','t','y','\n','f','a','w','n')
```

3. *Bonus:* Assume that `xs` is a list of characters in which line breaks are represented as `\n`. Show that for all such lists `xs`, your implementations of `lines` and `unlines` satisfy the following equivalence:
```
unlines(lines(xs)) = xs
```
Hint: use structural induction as shown in the lecture. Note that purely recursive definitions (that is, not using library functions such as `foldRight` or `span`) are easier to prove 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:
```
val t = List(' ','w','o','r','d',' ',' ','p','u','n',' ')
words(ch => ch == ' ')(t)
```
should yield
```
List(List('w','o','r','d'),List('p','u','n'))
```

2. Define a function `wordCount` that counts the number of words in a text. Use a predicate which defines whitespace, newline and basic punctuation characters (such as . , : ;) as delimiters.
```
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 the 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 in the text. More specifically, each occurence is represented by a pair consisting of the word and the corresponding line number.

   Use the functions `lines` and `words` defined in the previous section to extract the lines and words from the text, and use the same delimiter predicate as in `wordCount`.

   Hints: Using for comprehensions simplifies this task. For pairing the elements of a list with their (0-based!) indices, the List class defines a function `zipWithIndex` (see Scala API documentation).

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. Collapse 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), (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
   ...
   ```

Printing a calendar

In this part we are interested in printing a calendar. More specifically, we want to print an overview of a given month that shows which date falls on which day of the week. For example, in 2008, the First of October was a Wednesday.

The month of October 2008 should be printed as follows:
```
Su Mo Tu We Th Fr Sa
          1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31
```

Leap years, the First of January and all that

To be able to print a monthly overview, we first have to determine on which weekday falls the first day of the given month. We provide you with the following function definitions to simplify this task:

```
/** The weekday of January 1st in year y, represented
 *  as an Int. 0 is Sunday, 1 is Monday etc. */
def firstOfJan(y: Int): Int = {
  val x = y - 1
  (365*x + x/4 - x/100 + x/400 + 1) % 7
}

def isLeapYear(y: Int) =
  if (y % 100 == 0) (y % 400 == 0) else (y % 4 == 0)

def mlengths(y: Int): List[Int] = {
  val feb = if (isLeapYear(y)) 29 else 28
  List(31, feb, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31)
}
```

With the help of these functions, define a function `firstDay` that calculates the weekday of the first day of a given month:

```
def firstDay(month: Int, year: Int): Int = ...
```

How to picture that?

Picturing data with a non-trivial layout such as a calendar can be tricky. Therefore, we want to use a compositional approach where larger, more complex pictures are composed of smaller, simpler pictures.

In our design, pictures are represented as instances of the `Picture` case class:

```
case class Picture(height: Int, width: Int, pxx: List[List[Char]]) {
  def showIt: String = unlines(pxx).mkString("")
}
```

As we can see, a picture has a height and width, and contents `pxx` which is character data represented as a list of rows, where each row is a list of characters. The `showIt` method turns the picture into a list of characters using the `unlines` function defined in the first part.

The following function `pixel` creates a simple picture of height and width 1 that contains a given character:

```
def pixel(c: Char) = Picture(1, 1, List(List(c)))
```

From pictures as simple as that, we want to compose larger ones using composition operators.

1. Define a method `above` for class `Picture` that returns a new picture where the argument picture is put below `this`:

   ```
   case class Picture(...) {
     def above(q: Picture): Picture = ...
   }
   ```

   For instance, the following code

   ```
   println((pixel('a') above pixel('b')).showIt)
   ```

   should print

   ```
   a
   b
   ```

   Give an error message (using the predefined `error` function) when the pictures do not have the same width.

2. Define a method `beside` for class `Picture` that returns a new picture where the argument picture is put on the right side of `this`:

```
case class Picture(...) {
  def beside(q: Picture): Picture = ...
}
```

Give an error message (using the predefined `error` function) when the pictures do not have the same height.

3. Define functions `stack` and `spread` that arrange a list of pictures above and beside each other, respectively, producing a single resulting picture. For `stack`, the picture at the head of the argument list should be the topmost picture in the result. Similarly for `spread`, the head of the list should be the leftmost picture in the result.

```
def stack(pics: List[Picture]): Picture = ...
def spread(pics: List[Picture]): Picture = ...
```

4. Define a function `tile` that arranges a list of rows of pictures in a rectangular way using the `stack` and `spread` functions:

```
def tile(pxx: List[List[Picture]]): Picture = ...
```

5. Define a function that takes a width `w` and a list of characters, and produces a picture of height 1 and width `w` where the given characters are justified on the right border:

```
def rightJustify(w: Int)(chars: List[Char]): Picture = ...
```

Give an error message if `chars.length > w`.

6. Define a function `group` that splits a list into sublists. The function takes an integer as argument that indicates the split indices (e.g. split every 7 elements). We intend to use this function to split a list representing a whole month into a list of weeks. Note that this function is parameterized which means that it can be used with lists of any element type.

```
def group[T](n: Int, xs: List[T]): List[List[T]] = ...
```

7. Define a function `dayPics` that takes the number of the first day and the number of days of a month and produces a list of 42 pictures. In this list the first `d` pictures are empty (i.e. the character data is a list of spaces) if the number of the first day is d (d==0: Sunday, d==1: Monday etc.). The trailing pictures that correspond to days of the next month are empty, too. Using this function, a picture of a calendar can be produced by grouping and tiling the result of `dayPics`.

```
def dayPics(d: Int, s: Int): List[Picture] = ...
```

8. Using the functions defined in the previous steps, define a function `calendar` that produces a picture of a calender that corresponds to the given year and month.

```
def calendar(year: Int, month: Int): Picture = ...
```

Bonus 1: Add a version of the calendar that prints a month with the days of the week running from Monday to Sunday.

Bonus 2: Under what conditions does the law

```
(p above q) beside (r above s) == (p beside r) above (q beside s)
```

hold?