Functional Programming / Funkcinis programavimas

Exercise set 3

Solutions to be sent until October 27th

Exercise 1. Design a function

```
subst :: String -> String -> String -> String,
```

so that `subst oldSub newSub st` is the result of replacing all the occurrences in `st` of the substring `oldSub` by the substring `newSub`. If `oldSub` does not occur in `st`, the function result should be `st`.

Exercise 2. Define a function

```
isPalin :: String -> Bool,
```

which tests whether a string is a palindrome – that is whether it is read the same way both backwards and forwards. Note that punctuation and white spaces (see the text processing examples in the slides of Lecture 6) should be ignored in the test, and that no distinction should be made between capital and small letters.

Exercise 3. Design a function

```
count:: String -> (Int,Int,Int),
```

which for a given text string returns the number of characters, words, and lines in the string (again, see the text processing examples in the slides of Lecture 6). The end of the line in the string is signalled by the newline character `'\n'`.

Exercise 4. Write a function

```
justify:: String -> Int -> String,
```

which justifies the given text string (the first argument) to the new line length (the second argument). In other words, the newline characters should be added or removed to reformat the string so that all the lines in the resulting string are as close as possible (but not exceeding) the given line length. Note that the newline character `'\n'` can be added only between the string words. If any single word exceeds the given line length, the corresponding error message should be returned.

Exercise 5. Extend the `Shape` datatype definition (see the slides of Lecture 6) by, for each shape, adding its position (`(x,y)` coordinates) as an extra argument or arguments.

Write a function

```
overlaps::  Shape -> Shape -> Bool,
```

that checks whether two given shapes are overlapping or not.

Exercise 6. In a library database, each book copy is stored by its name and numerical id (i.e., there could several different copies of a book with the same name). Moreover, the system stores, for each book copy, its current loan status. The loan status can be one of three values: `Loaned`, `Free`, or `Locked`. The last value means that this copy cannot be loaned.

Additionally, a separate library database stores which persons are currently borrowing which books. Both databases are implemented as Haskell lists containing the elements of appropriate types.

Define all the necessary types (for persons, books, the book and loan statuses, etc.) using `data` or `type` commands.

Write a function

```
loan ::  Person -> Book -> ([??],[???])  -> ([??],[???]),
```

which, for the given person and book copy, loans this book to the person (if possible) and correspondingly updates the book and loan databases. Replace `??` and `???` with appropriate your defined types.