# 1 Extra: Word Wrap

## Exam question

The following Word Wrap exercise is a question from a previous exam.

## Some practicalities

- In the folder `TODO`, you'll find the files `MyHaskell.hs`, `MyHaskellTest.hs`, and `Testing.hs`.
  - The file `MyHaskell.hs` contains a template for your solution.
  - You can also **import extra functions and types**.
  - For each assignment, a number of functions and type classes have already been defined with corresponding type signatures. These functions and type signatures **may not be modified**. Replace all occurrences of `error "..."` with your implementation. You can add arguments in front of the equals sign, but, when possible, try to write the function *point-free*. It is of course also permitted to add extra helper functions.
  - You are allowed to use the slides of the lectures on Toledo, to use e-systant, hoogle (http://www.haskell.org/hoogle/ and the accompanying hackage documentation.
  - Even though you will develop a whole program in this exam, you can make most assignments separately. **Whenever you're stuck on an assignment, try the next**.
  - You can test your solution using `MyHaskellTest.hs`. Run the following command in the directory you put the three `.hs`-files in:

    ```
    runhaskell MyHaskellTest.hs
    ```

    **N.B.** the fact that all your tests pass doesn't mean that your program is completely correct, nor that you will get the maximum score.
- Hand in your solution (`MyHaskell.hs`) on Toledo.

# Word Wrap

In this assignment we will split a text into lines with a maximal width such that the text will fit on a page with a certain width.

Take for example the text below.

> Leverage agile frameworks to provide a robust synopsis for high level overviews. Iterative approaches to corporate strategy foster collaborative thinking to further the overall value proposition. Organically grow the holistic world view of disruptive innovation via workplace diversity and empowerment.

This text, that was originally written on one line, has been split into lines by the word processor this assignment has been made with, namely LaTeX. LaTeX has a smart algorithm to split lines. It tries to keep the width of the lines as even as possible and it intelligently splits words. In this assignment we will keep it simple: we will not split words and we will use a simple *greedy* algorithm that places as many words as possible on the line until it is full, after which we go to the next line until we run out of words. When the original text contains a newline, this newline will occur at the same place in the split text (think of \\ in LaTeX).

If we run this algorithm with the text above using line width 50, we get the result below. **N.B.** the dashed first line and the bar at the end of each line were added to this figure to make it easier to see the line width. Your solution will **not** have to draw these dashes and bars.

```
----+----1----+----2----+----3----+----4----+----5|
Leverage agile frameworks to provide a robust     |
synopsis for high level overviews. Iterative      |
approaches to corporate strategy foster           |
collaborative thinking to further the overall     |
value proposition. Organically grow the holistic  |
world view of disruptive innovation via workplace |
diversity and empowerment.                        |
```

If we run the algorithm using line width 32, we get the result below:

```
----+----1----+----2----+----3--|
Leverage agile frameworks to    |
provide a robust synopsis for   |
high level overviews. Iterative |
approaches to corporate strategy|
foster collaborative thinking to|
further the overall value       |
proposition. Organically grow   |
the holistic world view of      |
disruptive innovation via       |
workplace diversity and         |
empowerment.                    |
```

## Approach

The approach in this assignment is as follows: we split the original text (`String`) in a list of `LineItem`s, a new data type. We perform the splitting of lines by a number of transformations of lists of `LineItem`s Finally, we convert these lists of `LineItem`s back to a `String`.

## Part 1: Line Items

**Assignment 1.1:** Define a data type named `LineItem`. A `LineItem` is either a space (`" "`), a newline (`"\n"`), or a word (a `String`). A word may not contain spaces or newlines, but it may contain punctuation. Multiple spaces in the input will result in multiple space `LineItem`s, the same is true for newlines. **Think carefully about which fields the constructors for a space and a newline should contain.**

See assignment 1.3 for some examples of `LineItem`s.

**Assignment 1.2:** Implement the functions `mkSpace :: LineItem`, `mkNewline :: LineItem`, and `mkWord :: String -> LineItem` that create the respective `LineItem`s.

**Assignment 1.3:** Make a `Show` instance for `LineItem`.

A space is shown as `" "`, a newline as `"\n"`, and a *word* as `"word"`. **N.B.** the double quotes are part of the `String`.

**Hint:** `show "foo" = "\"foo\""`.

**Examples:**

```
> mkSpace
" "
> mkNewline
"\n"
> mkWord "Hello"
"Hello"
> map show [mkSpace, mkNewline, mkWord "Hello"]
["\" \"","\"\\n\"","\"Hello\""]
```

**Assignment 1.4:** Implement the function `toLineItems :: String -> [LineItem]` which splits a `String` into a list of `LineItem`s.

**Examples:**

- The sentence "`See ya, John.`" consists of 5 `LineItem`s: `["See"," ","ya","," ","John."]`.
- The sentence "`Hint: read the\nassignment carefully!`" consists of 9 `LineItem`s: `["Hint:"," ","read"," ","the","\n","assignment"," ","carefully!"]`.
- The sentence "`! oops  \n\n `" consists of 8 `LineItem`s: `["!"," ","oops"," "," ","\n","\n"," "]`.

**Assignment 1.5:** Implement the function `fromLineItems :: [LineItem] -> String` which turns a list of `LineItem`s into a `String`.

**Property:** for each `String s`, it must be that `fromLineItems (toLineItems s) == s`.

## Part 2: Word Wrap

Description of the algorithm:

- Place as many words on a line as possible until there is no more place left on the line. Start with a new line afterwards. Repeat this until there are no more words left.
- If there is a newline in the input, it should occur at the same place in the output.
- Multiple consecutive spaces will be replaced by a single space.
- Spaces at the start or end of a line are omitted.
- No line may be longer than the maximal line width, unless it contains a word longer than the line width. In this case, that word will be the only word on the line.

We split the algorithm in multiple simpler steps.

1. First we remove all spaces. Because we know there should be a space between every two words, we don't have to keep track of them explicitly. This makes the rest of the implementation easier. This also makes sure that multiple consecutive spaces will be replaced by a single space, just like required by the algorithm. Furthermore, this makes sure that no spaces will occur at the start and end of a line.
2. Next, we split the text into lines. Pay attention: these lines originate from the newlines that are already present in the text. We can then split each of those lines individually, without having to worry about preserving the newlines in the original text.
3. Afterwards, we put words longer than the maximal line width on their own lines. Later, during the actual splitting, we won't have to make sure no other words are placed on the same line before such a word.
4. Next, we will perform the actual splitting of the lines: we split a list of words into lines by putting as many words on a line as possible until there is no more place left on the line. While doing this, we remember to take the width of the space between each two words into account.
5. In the next step We restore the spaces between the words on each line.
6. Afterwards, we join the lines together by placing newlines between each two lines.

**Assignment 2.1:** Implement the function `removeSpaces :: [LineItem] -> [LineItem]` which removes all spaces from a list of `LineItem`s.

**Examples:**

```
> removeSpaces [mkWord "hello", mkSpace, mkWord "world", mkNewline, mkWord "Bye", mkSpace]
["hello","world","\n","Bye"]
> removeSpaces [mkWord "hi", mkSpace, mkSpace, mkNewline, mkSpace, mkNewline, mkSpace]
["hi","\n","\n"]
```

**Assignment 2.2:** Implement the function `splitInLines :: [LineItem] -> [[LineItem]]` which splits a list of `LineItem`s whenever a newline occurs. The result will no longer contain any newlines.

You may assume the input will not contain spaces.

**Examples:**

```
> splitInLines [mkWord "hi", mkNewline, mkWord "bye"]
[[mkWord "hi"], [mkWord "bye"]]
> splitInLines [mkNewline, mkWord "hi", mkNewline, mkNewline, mkWord "bye", mkNewline]
[[], [mkWord "hi"], [], [mkWord "bye"], []]
> splitInLines []
[[]]
> splitInLines [mkNewline]
[[], []]
> splitInLines [mkNewline, mkNewline]
[[], [], []]
> splitInLines [mkWord "foo", mkNewline]
[[mkWord "foo"], []]
```

**Assignment 2.3:** Implement the function `separateTooLongWords :: Int -> [LineItem] -> [[LineItem]]` which splits a list of `LineItem`s whenever a word is longer than the maximal line width. Each list of `LineItem`s represents a line, so this function splits one line into a list of lines. Whenever a word is too long, the line is split in three lines (lists): a line of the words coming before the too long word, a line with just the too long word, and a line with the words coming after the too long word. When no words come before the too long word, the first list is omitted. When no words come after the too long word, the third list is omitted. This process is repeated for each too long word.

You may assume the input will only contain words, no spaces or newlines.

**Examples:**

```
> separateTooLongWords 6 [mkWord "look", mkWord "a", mkWord "brontosaurus",
                          mkWord "over", mkWord "there"]
[["look","a"],["brontosaurus"],["over","there"]]
> separateTooLongWords 3 [mkWord "Yuuuuge", mkWord "amazing"]
[["Yuuuuge"],["amazing"]]
> separateTooLongWords 3 [mkWord "Banana"]
```

```
[["Banana"]]
> separateTooLongWords 100 [mkWord "Banana"]
[["Banana"]]
```

**Assignment 2.4:** Implement the function `wrap :: Int -> [LineItem] -> [[LineItem]]` which splits a list of `LineItem`s every time the maximal line width is reached. Each list of `LineItem`s represents a line, so this function splits one line into a list of lines. Recall the algorithm: *place as many words on a line as possible until there is no more place left on the line. Start with a new line afterwards.* Do not forget to account for the space between each two words. Also do not forget that some words may be longer than the maximal line width.

You may assume the input will only contain words, no spaces or newlines.

**Examples:**

```
> wrap 7 [mkWord "foo", mkWord "bar", mkWord "qu", mkWord "u", mkWord "x", mkWord "banana"]
[["foo","bar"],["qu","u","x"],["banana"]]
> wrap 4 [mkWord "gyronef"]
[["gyronef"]]
```

**Assignment 2.5:** Implement the function `joinLineWithSpaces :: [LineItem] -> [LineItem]` which puts a space between each two words in a list of `LineItem`s.

You may assume the input will only contain words, no spaces or newlines.

**Examples:**

```
> joinLineWithSpaces [mkWord "so", mkWord "much", mkWord "space"]
["so"," ","much"," ","space"]
```

**Assignment 2.6:** Implement the function `joinLinesWithNewlines :: [[LineItem]] -> [LineItem]` which joins the given list of lines into one list of `LineItem`s where each line is separated by a newline.

You may assume the input will only contain words and spaces, no newlines.

**Examples:**

```
> joinLinesWithNewlines [[mkWord "hi", mkSpace, mkWord "there"],[mkWord "bye"]]
["hi"," ","there","\n","bye"]
```

**Result** The above mentioned functions are combined in the function `wordWrap`. This function is given, so you do not have to write it yourself. Using this function, it is easy to test whether you have implemented the algorithm correctly by trying the examples below.

**Examples:**

- ```
  > wordWrap 5 "a b c d e f g"
  "a b c\nd e f\ng"
  > wordWrap 4 "a b c d e f g"
  "a b\nc d\ne f\ng"
  > wordWrap 5 "a\nb c d e f g"
  ```
  ```

```
"a\nb c d\ne f g"
> wordWrap 4 "\n food"
"\nfood"
> wordWrap 4 "a b c delta e f g"
"a b\nc\ndelta\ne f\ng"
> wordWrap 7 "foo  bar  "
"foo bar"
> wordWrap 20 "  foo  \n  \n  bar  "
"foo\n\nbar"
```

- See the two examples from the introduction (`putStrLn $ wordWrap 50 "..."` and `putStrLn $ wordWrap 32 "..."`).

**Hint:** using the test suite you can test all the examples above at once.

## Part 3: Interactive Word Wrapping

Part 3 uses IO which has not been covered yet. Therefore, it is not part of this exercise, meaning that if you finished everything of parts 1 and 2, you are done with this exercise. Part 3 has been included in case you want to experiment with IO on your own.

**Assignment 3.1:** Implement the function `getLines :: IO String` which reads a text (`String`) that may contain newlines. The user enters the text line by line, where each line is terminated by a press of the Enter key. This continues until the user types in `STOP` (followed by Enter).
**Hint:** use `getLine` to read one line. Don't forget the newline following each line.

**Examples:** The line following `STOP` is the result of the call to `getLines`.

```
> getLines
Hello
STOP
"Hello\n"
> getLines
Hello, there

Bye now
STOP
"Hello, there \n\nBye now\n"
```

**Assignment 3.2:** Implement the function `interactiveWrapper :: IO ()` which asks the user a line width and a text, after which the text will be 'wrapped' using the line width. Read the text using the `getLines` function from the previous assignment. You may assume the user enters a valid number for the line width.

**Examples:**

```
> main
```

```
Please enter a line width: 4
Please enter a text to wrap:
a
b c d e f g
STOP
a
b c
d e
f g
```