CS300-SP16 Exam 1

- Make sure you are sitting on the seat assigned to you.
- Total time is 5 hours and Maximum score is 15 points.
- Your startup code and a PDF of this exam is on LMS and the password of the zip file is “cs300isfun”.
- There are 15 parts that must be done in order. They have roughly equal score but the exact score per part will be assigned later. There is no partial credit so do not move to the next part before finishing a part.
- I have given a main that uses test cases near the end of file to tell you how many parts you have successfully done. Note that we will test it on different inputs for grading so do not hardcode for the given test cases. You only need to fill in the code for each part. Note that “Prelude.undefined” error means that you have not yet written the code for the next part in order.
- You must compile with “-Wall -Werror” otherwise your file will give errors when grading and you will lose marks.
- If you want to see the type of any function, you can use “:t” followed by the expression in “ghci”. If you want to print intermediate outputs, use “:l” to load the file and then use the previous functions to build the expression step by step.
- You can make any number of helper functions to solve your problem. Avoid functions that become too long as you’ll not be able to debug them. Use meaningful but short names
- No laptops, mobile phones, calculators, or electronic devices. Mobiles completely turned off and should not be on your work area but can stay in pockets, purses, backpacks that you’ll not access for the duration of the exam. Anything on your own USB or anything printed is fine.
- Do not talk to anyone during the exam and this is a strict rule. Raise your hand without shouting if you need TA help.
- No liquid containers without caps and no snacks that make crunching sounds. Please clean up after yourself.

High-level overview

A common type of text alignment in print media is "justification," where the spaces between words, are stretched or compressed to align both the left and right ends of each line of text. In this problem we’ll be implementing a text justification function for a monospaced terminal output (i.e. fixed width font where every letter has the same width).

The alignment is achieved by inserting blanks and hyphenating the words. For example, given a text

```
text = "He who controls the past controls the future. He who controls the present controls the past."
```

we want to be able to align it like this (to a width of, say, 15 columns):

```
He who controls
the  past cont-
rols  the futu-
re. He  who co-
ntrols the pre-
sent   controls
the past.
```

This problem is more intricate than it seems, so we’re going to approach it in a sophisticated way. We’ll design the solution step by step, defining a series of small, well-defined functions as we go.

We’ll represent a string as a list of tokens. We’ll refer to a list of tokens as a Line. Tokens may be words, hyphenated words, or inserted blanks.

```
data Token = Word String | Blank | HypWord String deriving (Eq,Show)
```

Part 1

Define a function to convert a string into a line. (You may assume that the input contains no hyphenated words! This, of course, is a gross oversimplification.) Note that blanks in the input do not convert into Blank tokens. The “words” function may be useful.

```
str2line :: String -> Line
str2line text ⇒ [Word "He", Word "who", Word "controls", ...]
```

Part 2

Define a function to convert back from a Line into a string. The “unwords” function might be useful.

```
line2str (str2line text) == text ⇒ True
```

Note: A hyphenated word should be displayed with a trailing hyphen. For example:

```
line2str [Word "He",Word "who",HypWord "cont",Word "rols"] ⇒ "He who cont- rols"
```

Part 3

Define a function to compute the length of a token.

```
tokLen (Word "He") ⇒ 2
tokLen (HypWord "cont") ⇒ 5
tokLen (Blank) ⇒ 1
```

Part 4

Define a function to compute the length of a line.

```
lineLen [Word "He",Word "who",Word "controls"] ⇒ 15
lineLen [Word "He",Word "who",HypWord "con"] ⇒ 11
```

Part 5

Define a function to break a line so that it’s not longer than a given width. The function should return a pair of Lines: first line is the broken-up line, and the second line is its continuation. This recursively. Patterns, guards, and let/in will make this part very easy.

```
breakLine 6 [Word "He",Word "who",Word "controls"] ⇒ ([Word "He",Word "who"],[Word "controls"])
```

Part 6

Define a helper function mergers that given all parts of a word, produces a pair of all possible ways to break up the word. This recursively. The “concat” function might help you.

```
mergers ["co","nt","ro","ls"] ⇒ [("co","ntrols"),("cont","rols"),("contro","ls")]
mergers ["co","nt"] ⇒ [("co","nt")]
mergers ["co"] ⇒ []
```

Part 7

To be able to align the lines nicely, we have to able to hyphenate long words. Although there are rules for hyphenation for each language, we will take a simpler approach here and assume that there is a list of words and their proper hyphenation. For example:

```
enHyp = [("controls",["co","nt","ro","ls"]), ("future",["fu","tu","re"]),
("present",["pre","se","nt"])]
```

Now, define a hyphenate function that breaks up a token in all possible ways defined by a hyphenation map. Use the mergers function defined previously.

Note: If the word has trailing punctuation, you need to remove it first, hyphenate the word, and then concatenate back the punctuation to the second part of the word. For example:

```
hyphenate enHyp (Word "future.")
  ⇒ [(HypWord "fu",Word "ture."), (HypWord "futu",Word "re.")]
```

The functions span, find, and isAlpha may help in this part.

Part 8

You can now define a function that breaks a line into different ways, by trying to hyphenate the last word in different ways. The broken-up line should not be longer than a given width. So you will start with the breakLine function and then attempt to breakup the next word using hyphenate and for each breakup option include it in output if the max length is satisfied. The functions map and filter may help you.

```
lineBreaks enHyp 12 [Word "He",Word "who",Word "controls"]
  ⇒ [([Word "He",Word "who"],[Word "controls"]),([Word "He",Word "who",HypWord "co"],[Word
"ntrols"]),([Word "He",Word "who",HypWord "cont"],[Word "rols"])]
```

Part 9

Define a helper function insertions that inserts a given variable at every possible place where it can be inserted. A helper recursive function with an accumulator may help.

```
insertions 'x' "abcd" ⇒ ["xabcd","axbcd","abxcd","abcxd","abcdx"]
```

Part 10

Using the insertions function, define a function that inserts a given number of blanks into the line and returns all possible insertions. Do not insert blanks at the very beginning and the end of the line! You may call the insertions function repeatedly, you may use filter to remove the ones with Blank at start and end, you may even use iterate function here. You'll also have to remove consecutive duplicate due to multiple Blanks.

```
insertBlanks 2 [Word "He",Word "who",Word "controls"]
  ⇒ [[Word "He",Blank,Blank,Word "who",Word "controls"],[Word "He",Blank,Word "who",Blank,Word
"controls"],[Word "He",Word "who",Blank,Blank,Word "controls"]]
```

Part 11

We want the inserted blanks to be spread out and evenly distributed. Define a function that computes the distances between inserted blanks, counted in number of in-between tokens. You should also take into account the distances to the start and end of line. Think recursively and span may help.

```
blankDistances [Word "He",Blank,Blank,Word "who",Word "controls"] ⇒ [1,0,2]
```

Part 12

Define two helper functions to compute the average and the variance of values in a list. Variance is average of the squared differences from the average of the numbers.
`var [1,0,2] ⇒ 0.6666666666666666`

Part 13

We're now capable of producing a number of candidate line splits, but we now need a mechanism to decide how good each line split is. To this end, we'll be defining a scoring function for a line break. The scoring function will be based on costs, defined as:

```
data Costs = Costs Double Double Double Double deriving (Eq,Show)
```

The first Double is blankCost and it is the cost of introducing a blank, the second is blankProxCost and it is the cost of having blanks close to each other, the third is blankUnevenCost and it is the cost of having blanks spread unevenly, and the last is hypCost and it is the cost of hyphenating the last word in a line.

Now, define a function to score a line based on a given cost. The total cost is computed simply as the sum of the individual costs. The blankProxCost equals the number of tokens minus the average blank distances if the line has blanks, and zero otherwise. The blank uneven cost is computed simply as the variance of blank distances.

```
lineBadness defaultCosts [Word "He",Blank,Word "who",Word "controls"] ⇒ 3.625
```

Part 14

We're getting closer to the solution. Define a function that computes the best line break given the costs, the hyphenation map, and the maximum line width. The best line break is the one that minimized the line badness score. If line break is not possible (because one word is longer than a specified width and cannot be hyphenated), return Nothing. You are near the end and this part will take some time while the last one is a piece of cake so concentrate and think the logic carefully. You may attach the badness to each line by making a list of tuples using the map function. Then you can find the minimum. Use let/in to keep the code neat.

```
bestLineBreak defaultCosts enHyp 12 [Word "He",Word "who",Word "controls"]
  ⇒ Just ([Word "He",Word "who",HypWord "cont"],[Word "rols"])
```

Part 15

Finally, define a function that justifies the line. You need to apply bestLineBreak iterativelly on each line. If a word gets hyphenated, you need to add the rest of it to the next line before you apply bestLineBreak again. Repeat that until reaching the last line, but do not justify the last line (justifiying the last line would look ugly). If any of the lines cannot be broken up, then return the line as it is.

```
justifyLine defaultCosts enHyp 8 (string2line text)
  ⇒ [[Word "He",Blank,Blank,Word "who"],[Word "controls"],[Word "the",Word "past"],[Word
"controls"],[Word "the"],...]
```

As a finishing touch, define a function that takes in a string and returns a list of strings justified to a specified width. If justification is not possible, return the original string.

```
justifyText defaultCosts enHyp 8 text ⇒ ["He   who","controls","the past",...]
```

Typing in `putStr.unlines.justifyText defaultCosts enHyp 15 text`
should give you exactly the example we started with.

GOOD LUCK