The aim of this practical is to introduce lists and polymorphism. Try to write the types of all your functions, so as to get used to polymorphism. It is also best to write your programs as literate scripts, with ample comments in between functions. Remember that programming requires practice – just by working through these practical sheets as we go along, you will find that you need not do much more work for the final exam.

1. Define a function `doubleElts`, which takes a list of integers, and doubles them. Thus, `doubleElts [1,2,3]` would return `[2,4,6]`.

2. Write a function `sumList`, which given a list of integers, returns the sum of the elements.

3. Define `lastElt`, which returns the last element of a list.

4. Define the functions, `takeN` and `dropN` which, given an integer `n` and list `xs`, return a list containing the first `n` elements of `xs` and a list containing all the elements of `xs` beyond the first `n`, respectively. Therefore, `takeN 2 [1,2,3]` would return `[1,2]`, while `dropN 2 [1,2,3]` would return `[3]`. If `n` is larger than the length of the given list, `takeN` should return the whole list, while `dropN` should return the empty list.

5. Define the function `join`, which is given two lists and pairs up the elements. For example, `join [1,2,3] ['a','b','c']` would return `[(1,'a'), (2,'b'), (3,'c')]`. If one of the lists is shorter, ignore the extra trailing part. `join [1,2] ['a']` would thus return `[(1,'a')]`.

6. Write a function `oddElts`, which given a list, returns the odd-positioned (first, third, fifth ... ) elements of the list.

7. The aim of this question is to construct code to manipulate and format text. Strings in Haskell may include characters such as tabs and newlines. A tab character is written as \t, while a newline is written as \n. At the Hugs prompt, a string returned may have such control characters appearing. If you want to see the string with these characters interpreted to cause a newline or tab, you may use the function `putStr`. For example, if someone wrote a function `hello` which returns a string which includes newlines, you can see its result using `putStr`:

```
Hugs> hello 3
"a\naa\naaa\n"

Hugs> putStr (hello 3)
a
aa
aaa
```

(a) Write a function `skipWhiteSpace`, which given a string, returns the same string, but with initial white space skipped.

Haskell has a useful function called `isSpace`, which returns whether or not a given character is white space. To use this function you have to import the Haskell character library, by starting the source code with:
import Char

(b) Hence or otherwise, write a function \texttt{getWord}, which given a string, returns two strings — the first word in the given string, and the rest of the given string. For example, 
\texttt{getWord " Hello world, goodbye!"} returns ("Hello", " world, goodbye!").

(c) Using \texttt{getWord}, write a function \texttt{intoWords}, which given a string, returns a list of strings — each of which was a word (separated by white space) in the original input.

(d) Define a function \texttt{getLine}, which given a list of strings, and an integer \texttt{width} representing the maximum line length, returns a string containing the first words (separated by a space, and whose length does not exceed \texttt{width}, and such that adding another word would make the line longer than \texttt{width}), and a list of strings (the remaining words in the text). For example, \texttt{getLine ["The","sky","is","not","blue"] 12} would return ("The sky is",["not","blue"]).

(e) Hence or otherwise, define a function \texttt{leftJustified}, which given an integer \texttt{width} and a string \texttt{s}, splits the string into a list of lines, each of which is no longer than \texttt{width}, and without splitting a word in half.

(f) Define a function \texttt{justified}, which given an integer \texttt{width} and a string \texttt{s}, splits the text into lines as in the case of \texttt{leftJustified} but adds additional spaces between words so as to ensure that all the lines are exactly \texttt{width} characters wide.