7.25 One list is a **sublist** of another if the elements of the first occur in the second, in the same order. For instance, `"ship"` is a sublist of `"Fish & Chips"`, but not of `"hippies"`.

A list is a **subsequence** of another if it occurs as a sequence of elements *next to each other*. For example, `"Chip"` is a subsequence of `"Fish & Chips"`, but not of `"Chin up"`.

Define functions which decide whether one string is a sublist or a subsequence of another string.

7.26 Write QuickCheck properties which test your implementations of the tests for 'sublist' and 'subsequence'.

From: Simon Thompson. Haskell: The Craft of Functional Programming Third Edition, Addison Wesley, 2011

7.6 Example: text processing

In word processing systems it is customary for lines to be filled and broken automatically, to enhance the appearance of the text. This book is no exception. Input of the form

```
The heat bloomed     in December
  as the   carnival  season
           kicked into  gear.
Nearly helpless with sun and glare, I avoided Rio's brilliant
sidewalks
  and glittering beaches,
panting in dark   corners
and waiting out the inverted southern summer.
```

would be transformed by **filling** to

```
The heat bloomed in December as the
carnival season kicked into gear.
Nearly helpless with sun and glare,
I avoided Rio's brilliant sidewalks
and glittering beaches, panting in
dark corners and waiting out the
inverted southern summer.
```

To align the right-hand margin, the text is **justified** by adding extra inter-word spaces on all lines but the last:

```
The heat bloomed in December as the
carnival  season  kicked into gear.
Nearly helpless with sun and glare,
I avoided Rio's brilliant sidewalks
and glittering beaches, panting  in
dark  corners  and  waiting out the
inverted southern summer.
```

An input file in Haskell can be treated as a string of characters, and so string-manipulating operations play an important role here. Also, since strings are lists, this example will exercise general list functions.

Overall strategy

In this section we give an example of bottom-up program development, thinking first about some of the components we will need to solve the problem, rather than decomposing the solution in a top-down way.

The first step in processing text will be to split an input string into **words**, discarding any white space. The words are then rearranged into lines of the required length. These lines can then have spaces added so as to justify the text. We therefore start by looking at how text is split into words.

Extracting words

We first ask, given a string of characters, how should we define a function to take the first word from the front of a string?

A word is any sequence which does not contain the **whitespace** characters space, tab and newline.

```
whitespace = ['\n','\t',' ']
```

In defining `getWord` we will use the standard function `elem`, which tests whether an object is an element of a list. For instance, `elem 'a' whitespace` is `False`.

To guide the definition, consider two examples.

- `getWord " boo"` should be `""` as the first character is whitespace;
- `getWord "cat dog"` is `"cat"`. We get this by putting `'c'` on the front of `"at"`, which is `getWord "at dog"`.

The definition is therefore given by:

```
getWord :: String -> String
getWord []     = []                                          (getWord.1)
getWord (x:xs)
  | elem x whitespace   = []                                 (getWord.2)
  | otherwise           = x : getWord xs                     (getWord.3)
```

Consider an example

```
getWord "cat dog"
~>  'c' : getWord "at dog"                          by (getWord.3)
~>  'c' : 'a' : getWord "t dog"                     by (getWord.3)
~>  'c' : 'a' : 't' : getWord " dog"                by (getWord.3)
~>  'c' : 'a' : 't' : []                            by (getWord.2)
~>  "cat"
```

In a similar way, the first word of a string can be dropped.

```
dropWord :: String -> String
dropWord []     = []
dropWord (x:xs)
  | elem x whitespace   = (x:xs)
  | otherwise           = dropWord xs
```

It is easy to check that `dropWord "cat dog" = " dog"`. We aim to use the functions `getWord` and `dropWord` to split a string into its constituent words. Note that before we take a word from the string `" dog"`, we should remove the whitespace character(s) from the front. The function `dropSpace` will do this.

```
dropSpace :: String -> String
dropSpace []     = []
dropSpace (x:xs)
  | elem x whitespace    = dropSpace xs
  | otherwise            = (x:xs)
```

How is a string `st` to be split into words? Assuming `st` has no whitespace at the start,

- the first word in the output will be given by applying `getWord` to `st`;
- the remainder will be given by splitting what remains after removing the first word and the space following it: `dropSpace (dropWord st)`.

The top-level function `splitWords` calls `split` after removing any whitespace at the start of the string.

use Word' to avoid name clash

```
type Word = String

splitWords :: String -> [Word]
splitWords st = split (dropSpace st)

split :: String -> [Word]
split [] = []
split st
  = (getWord st) : split (dropSpace (dropWord st))
```

Consider a short example.

```
splitWords "  dog cat"
⇝  split "dog cat"
⇝  (getWord "dog cat")
         : split (dropSpace (dropWord "dog cat"))
⇝  "dog" : split (dropSpace " cat")
⇝  "dog" : split "cat"
⇝  "dog" : (getWord "cat")
         : split (dropSpace (dropWord "cat"))
⇝  "dog" : "cat" : split (dropSpace [])
⇝  "dog" : "cat" : split []
⇝  "dog" : "cat" : []
⇝  [ "dog" , "cat" ]
```

Splitting into lines

Now we have to consider how to break a list of words into lines. As before, we look to see how we can take the first line from a list of words.

use Line' to avoid name clash

```
type Line = [Word]
getLine :: Int -> [Word] -> Line
```

`getLine` takes two parameters. The first is the length of the line to be formed, and the second the list from which the words are taken. The definition uses `length` to give the

length of a list. The definition will have three cases

- In the case that no words are available, the line formed is empty.
- If the first word available is `w`, then this goes on the line if there is room for it: its length, `length w`, has to be no greater than the length of the line, `len`.
  The remainder of the line is built from the words that remain by taking a line of length `len-(length w+1)`.
- If the first word does not fit, the line has to be empty.

renamed getLine2 to avoid name clash - replaced with getLine3 to correct bug in textbook code

```
getLine len []     = []
getLine len (w:ws)
  | length w <= len     = w : restOfLine
  | otherwise           = []
    where
    newlen      = len - (length w + 1)
    restOfLine  = getLine newlen ws
```

Why is the `rest` of the line of length `len-(length w+1)`? Space must be allocated for the word `w` *and* the inter-word space needed to separate it from the word which follows. How does the function work in an example?

```
getLine 20 ["Mary","Poppins","looks","like",...
⇝  "Mary" : getLine 15 ["Poppins","looks","like",...
⇝  "Mary" : "Poppins" : getLine 7 ["looks","like",...
⇝  "Mary" : "Poppins" : "looks" : getLine 1 ["like",...
⇝  "Mary" : "Poppins" : "looks" : []
⇝  [ "Mary" , "Poppins" , "looks" ]
```

A companion function,

```
dropLine :: Int -> [Word] -> Line
```

removes a line from the front of a list of words, just as `dropWord` is a companion to `getWord`. The function to split a list of words into lines of length at most (the constant value) `lineLen` can now be defined:

```
splitLines :: [Word] -> [Line]
splitLines [] = []
splitLines ws
  = getLine lineLen ws
      : splitLines (dropLine lineLen ws)
```

← use getLine3

This concludes the definition of the function `splitLines`, which gives filled lines from a list of words.

Conclusion

To fill a text string into lines, we write

```
fill :: String -> [Line]
fill = splitLines . splitWords
```

fill xs = splitLines (splitWords xs)

To make the result into a single string we need to write a function

```
joinLines :: [Line] -> String
```

This is left as an exercise, as is justification of lines.

Exercises

7.27 Define the function `dropLine` specified in the text.

7.28 Give a definition of the function

```
joinLine :: Line -> String
```

which turns a line into printable form. For example,

```
joinLine [ "dog" , "cat" ] = "dog cat"
```

7.29 Using the function `joinLine`, or otherwise, define the function

```
joinLines :: [Line] -> String
```

which joins together the lines, separated by newlines.

7.30 In this case study we have defined separate 'take' and 'drop' functions for words and lines. Redesign the program so that it uses 'split' functions – like the prelude function `splitAt` – instead.

7.31 [Harder] Modify the function `joinLine` so that it justifies the line to length `lineLen` by adding the appropriate number of spaces between the words.

7.32 Design a function

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

which when given a text string returns the number of characters, words and lines in the string. The end of a line in the string is signalled by the newline character, `'\n'`. Define a similar function

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

which returns the same statistics for the text *after* it has been filled.

7.33 Define a function

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

which tests whether a string is a palindrome – that is whether it is the same read both backwards and forwards. An example is the string

```
Madam I'm Adam
```

Note that punctuation and white space are ignored in the test, and that no distinction is made between capital and small letters. You might first like to develop a test which simply tests whether the string is exactly the same backwards and forwards, and only afterwards take account of punctuation and capital letters.

7.34 [Harder] Design a function

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

so that

```
subst oldSub newSub st
```

is the result of replacing the first occurrence in `st` of the substring `oldSub` by the substring `newSub`. For instance,

```
subst "much " "tall " "How much is that?"
  = "How tall is that?"
```

If the substring `oldSub` does not occur in `st`, the result should be `st`.

7.35 [Harder] Define QuickCheck properties which test the behaviour of your `subst` function, defined in the previous question.

Summary

This chapter has shown how functions can be defined by recursion over lists, and completes our account of the different ways that list-processing functions can be defined. In the chapter we have looked at examples of the design principles which we first discussed in Chapter 4, including 'divide and conquer' and general pieces of advice about designing recursive programs. The text processing case study provides a broadly bottom-up approach to defining a library of functions.