COMP105: Programming Paradigms
Lab Sheet 4

Filename: Lab4.hs

This is a *challenge lab* on recursion, which means that the questions in this lab are focused on building a useful program.

Hidden tests. The Codegrade settings for this lab are closer to those that will be used for the final exam. This means that each function will have some visible tests, as in previous labs, but also some *hidden* tests, which will not be shown to you until after the deadline has passed.

These tests might contain trickier inputs, so think carefully about whether your functions are comprehensive: do they return the correct answer for *all* valid inputs?

Restrictions. Since this lab is focused on recursion, the code that you write in this lab **must not use any library functions except `head` and `tail`, you must not use list comprehensions, and you must not import any other modules**. To be clear, other than this restriction, you can use all other normal Haskell syntax such as:

- Any operator that is infix by default. For example `+`, `++`, `&&`, and `:` are all allowed, but `` `min` `` and `` `max` `` are not.
- `if` expressions, `let` expressions, `where` syntax.
- List ranges.
- Pattern matching.

Codegrade will refuse to mark your function if it detects the use of a non-allowed library function.

Repeat encodings. In this lab, we will implement the *repeat encoding*. If a string contains repeated letters, such as `"aaaa"`, then we can replace this with a shorter repeat-encoded string `"a4"`, which says that the letter `a` should be repeated four times. To *encode* a string into the repeat encoding, all repeated characters should be transformed into their repeat encodings. So, when the string `"aaaabbb"` is given to the encoder, it should return `"a4b3"`. The *decoder* does the opposite: it takes repeat encoded strings and returns the original. So when the string `"a2b2c2"` is given to the decoder, it will return `"aabbcc"`.

To make things easier, in Parts A and B we will consider a *simplified* repeat encoding with the following properties:

- No character is ever repeated more than nine times.
- Single instances of a character will still be repeat encoded. This means that the string `"abc"` will be encoded to `"a1b1c1"`, even though the encoding is longer than the original.

These two restrictions mean that *every* encoded string has the format:

```
<character> <number> <character> <number> ...
```

where all of the numbers are represented by single characters.

Here are some examples of the simplified repeat encoding.

Original	Encoded
`"too"`	`"t1o2"`
`"hello"`	`"h1e1l2o1"`
`"bookkeeper"`	`"b1o2k2e2p1e1r1"`
`"aaaaaaaaabbbbccc"`	`"a9b4c3"`

Part A

Question 1. Write a function `char_to_int :: Char -> Integer` that takes a character representing a number between zero and nine, and returns the corresponding integer. So `char_to_int '1'` should return the integer `1`, and `char_to_int '2'` should return `2`, and so on.
Hints:

- You only need to handle single characters as input to this function. So the only inputs will be the characters `'0'` through `'9'`. You *do not* need to handle strings with multiple numbers. It does not matter what your function returns if the input is not a number.
- There is no need to use recursion for this function.
- There is an example in Lecture 7 that you can adapt to implement this function. Although we saw some more advanced ways to work with characters in Lecture 10, those examples used library functions that are not allowed in this lab.
- There is no special trick here, it is fine if your function takes quite a few lines.

Question 2. Write a function `repeat_char :: Char -> Integer -> [Char]` that takes a character `c` and an integer `n` and returns a string that contains `n` copies of the character `c`. So `repeat_char 'a' 3` should return `"aaa"` and `repeat_char 'b' 9` should return `"bbbbbbbbb"`.
Hints:

- You will need to use recursion to do this.
- First think about the base case: when should we stop the recursion? It doesn't make sense to call `repeat_char c (-1)`, so we should probably stop before then. What should we return for the base case?

- Next think about the recursive step: how do we break `repeat_char c n` into something that is closer to the base case? Once we've made a recursive call, how should we transform the output to get what we want?
- Look at the examples in Lecture 8 if you are stuck here. The `down_from` example is also a function that builds a list using an integer argument.

Question 3. Write a function `decode :: [Char] -> [Char]` that decodes a string in the simple repeat encoding. So `decode "a1b2"` should return `"abb"` and `decode "t1h1e1"` should return `"the"`.
Hints:

- As always, start with the base case. What is the smallest possible input to `decode`? What should the output be for that input?
- The recursive rule will be a little complex. You will need to pull the first *two* characters from the front of the string. Pattern matching can do this, and there is an example in Lecture 8 demonstrating this.
- Because we are using the simple repeat encoding, we know that the first character in the string will be a letter, and the second character will be a number.
- So there are two things that we need to do: turn the second character into an integer, and then repeat the first character that many times. We have already written functions to do both of these things.
- Then we just need to put our repeated characters at the front of the decoded string. The Haskell operator for joining two strings can do this.

Part B

In Part B we will build an encoder for the simple repeat encoding. There are no hints in Part B. You can answer the first three questions in any order, so if you are stuck on one question, try another one. You will probably need to answer Questions 4–6 before you can tackle Question 7.

Question 4. Write a function `int_to_char :: Integer -> Char` that takes an integer between `0` and `9` and returns the corresponding character. So for example `int_to_char 1` should return `'1'`, and `int_to_char 2` should return `'2'`, and so on.

Question 5. Write a function `length_char :: Char -> [Char] -> Integer` that takes a character `c` and a string `string` and returns the number of times that `c` occurs at the start of `string`. So `length_char 'a' "aaabaa"` should return `3`, and `length_char 'c' "cbcac"` should return `1`.

Question 6. Write a function `drop_char :: Char -> [Char] -> [Char]` that takes a character `c` and a string `string` and returns a version of `string` without any leading instances of `c`. A leading instance of `c` is any instance that would have been counted by `length_char`. So `drop_char 'a' "aaabaa"` should return `"baa"`, and `drop_char 'c' "cbcac"` should return `"bcac"`.

Question 7. Write a function `encode :: [Char] -> [Char]` that encodes a string in the simple repeat encoding. So `encode "aaabbbccc"` should return `"a3b3c3"` and `encode "abcd"` should return `"a1b1c1d1"`.

(*) Part C

Part C is as an optional final challenge. Part C requires you to build an encoder and decoder for the *non-simple* repeat encoding. This means that the encoded strings may contain more than nine repeats , eg. `"a11b11"`, and that single characters will be encoded without a number, so `"hello"` encodes to `"hel2o"`. You can assume that the un-encoded strings are entirely formed from non-numeric characters.

In Part C, you may use the library functions `mod`, `div`, `length`, `reverse`, `head`, `tail`, `last`, `init`, and `fromIntegral`. It is expected (though not required) that you will implement several helper functions to answer these questions.

Question 8. Write a function `complex_encode :: [Char] -> [Char]` that returns the non-simple repeat encoding of `string`.

Question 9. Write a function `complex_decode :: [Char] -> [Char]` that takes a non-simple repeated encoded string, and returns the original un-encoded string.