COMP105 Assignment 1: Recursion

Assessment Number	1 (of 4)
Weighting	25%
Date Circulated	Wednesday the 19th of October (week 4)
Deadline	Wednesday the 2nd of November (week 6) at 12:00 midday
Submission Mode	Electronic only
Learning outcomes assessed	• Apply recursion to solve algorithmic tasks. • Write programs using a functional programming language.
Marking criteria	Correctness: 100%
Submission necessary in order to satisfy module requirements?	No
Late Submission Penalty	Standard UoL penalty applies
Date of advantageous feedback	Tuesday the 8th of November (anticipated)

Important: Please read all of the instructions carefully before starting the assignment. There is an FAQ at the end of this document. Please read it before asking any questions.

Repeat encodings. In this assignment, you 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"`

The assignment. In this assignment, you will implement an encoder and decoder for the repeat encoding in Haskell. In Part A (30%) you will create a decoder for the simplified encoding, and in Part B (35%) you will create an encoder for the simplified encoding. Note that since parts A and B combined give 65%, you can get a "2:1" mark on the assignment from only these two parts. Ambitious students who would like to push for 100% should attempt Part C (35%), where the task is to implement an encoder and decoder for the non-simplified repeat encoding.

Restrictions. The purpose of this assignment is to test your skills at creating recursive functions. For this reason, you should implement your functions using recursion wherever possible, **you should not use any library functions except `head` and `tail`**, and you should not use list comprehensions. To be clear, all normal Haskell syntax is allowed, including:

- 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.

The only things that are disallowed are calling are calling named library functions except for `head` and `tail`, and list comprehensions. In part C, a few more library functions can be used, as described in that section.

The template file. There is a template file called `Assignment1.hs`, which is available for download from the COMP105 website, and your solutions must be written into that file. The first line of the file, that starts with `"module Assignment1..."` should not be altered.

For each question, the template has two lines like so:

```
char_to_int :: Char -> Integer
char_to_int = error "Not implemented"
```

The first line is a type annotation, which tells Haskell what types the function takes as arguments, and what type the function returns. We will see these in lectures soon. **The type annotations should not be removed or modified in any way.** Note that if your function takes or returns the wrong type, then the type annotation will cause a compilation error, so in this case, the `char_to_int` function must take a `Char` and must return an `Integer`.

The second line is a stub implementation that just returns an error. You should remove the stub, and implement your own version of each of the functions. Note that all of the stubs take no arguments, so you will need to add the appropriate number of arguments to the function. These arguments can be named whatever you like, and pattern matching is allowed. Your code will fail to compile if there is no implementation of one of the functions, so make sure to reinstate the stub implementation in the event that you do not solve a problem.

You may create other functions in the template file if you wish, and in part C it is expected that you will create several helper functions in order to solve the problems.

Marking. Marks will be awarded for correct function implementations only. No marks will be gained or lost for coding style. Each function that you write will be tested by an automated tester on a number of test cases. While some test cases are given for each question, these will not be the same test cases used by the automated marker.

If you are not able to fully implement a function, you may still get marks for demonstrating an understanding of recursion. If you have a non-working function that you would like to be looked at by a human marker, then include the following comment above the function.

```
-- PLEASE READ
```

Please make sure that the comment is *exactly* as shown above including capitalization. Functions that do not include this comment will be marked by the automated marker only. If you have multiple non-working functions that you would like to be looked at, then please include the comment above each one.

Do not include the comment above working functions – no extra feedback will be given for those other than what is produced by the automated marker.

Although there are no marks available for coding style, you are still advised to produce readable code. This means that you should comment your code appropriately, and make use of Haskell constructs like `let` and pattern matching, which can help to make code more readable. This will help you to develop the code, and help the human marker in the event that some marks are awarded to non-functional code.

Submission. Submit `"Assignment1.hs"` (please do not change the name or capitalization) to the assessment task submission system:

`https://sam.csc.liv.ac.uk/COMP/Submissions.pl`

Since your functions will be called from the automated marker, please **make sure that your code compiles without errors before submitting.** There will be a 5% penalty for any submission that fails to compile, or for any submission where the type annotations have been modified. You can use the checker file (see the section at the end of this document) to check whether your submission will get this penalty.

If you have some code that does not compile, but you would like the marker to see it, then comment out the code (but *not* the type annotation), and explain in another comment what the code is supposed to do, and include the `PLEASE READ` comment following the instructions above. Remember that you will need to reinstate the stub implementation after you comment out your code, or the file will still fail to compile.

Academic Integrity. The university takes academic integrity very seriously. The work that you submit for this assessment must be your own. You should not *collude* with another student, meaning that you should not see another COMP105 student's submission, or copy any part of it. You should also not *plagiarise* anyone else's work, meaning that code written by someone else that you find online, or anywhere else, should not be submitted as part of the assignment.

Deadline. The deadline for this task is:

Wednesday the 2nd of November (week 6) at 12:00 midday

The standard university late penalty will be applied: 5% of the *total marks available* for the assessment shall be deducted from the assessment mark for each 24 hour period after the submission deadline. Late submissions will no longer be accepted after five calendar days have passed.

Getting help. There are a number of ways of getting help on the assignment. There is an FAQ section at the end of this document that might answer your question. If you are still stuck, then you can send an email to the module coordinator at `john.fearnley@liverpool.ac.uk`. Please **attach your entire source file** (not screenshots) if you are asking for help.

Part A (worth 30%)

In part A we will build a decoder for the simple repeat encoding.

Question 1. Write a function `char_to_int char` 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 Assignment 1.
- There is no special trick here, it is fine if your function takes quite a few lines.

Question 2. Write a function `repeat_char c n` that 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 string` that decodes a string in the simple repeat encoding. So `decode "a1b2"` should return `"abb"` and `decode "t1h1e1"` should return `"the"`
Hints:

- Note that, in all of the questions in the assignment, the arguments names are not prescribed. So you can change the name of the argument `string` to whatever you like, and you can use pattern matching if you wish.
- 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 (worth 35%)

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 int` that takes a number between `0` and `9` and returns the corresponding character. So `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 c string` 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 c string` 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 string` that encodes `string` in the simple repeat encoding. So `encode "aaabbbccc"` should return `"a3b3c3"` and `encode "abcd"` should return `"a1b1c1d1"`.

Part C (worth 35%)

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`, and `init` without penalty. All other functions should be implemented by you. It is expected (though not required) that you will implement several helper functions to answer these questions.

Question 8. Write a function `complex_encode string` that returns the non-simple repeat encoding of `string`.

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

The Checker

You can download `Checker.hs` from the Canvas site. This file is intended to help you check whether your code will run successfully with the automated marker. To run the checker, follow these instructions.

1. Place `Checker.hs` and your completed `Assignment1.hs` into the *same folder*. The checker will not work if it is not in the same folder as your submission.
2. You have two options to run the checker.
   (a) Navigate to the folder in Windows explorer, and double-click on `Checker.hs` to open the file into ghci.
   (b) In a ghci instance, first use the `:cd` command to change to the directory containing the checker, and then run `:load Checker.hs`

   You **cannot** load the checker directly with something like `:l M:\Checker.hs`, because you must first change the directory so that ghci can also find `Assignment1.hs`. Both of the methods above will do this.

If the checker compiles successfully, then you are guaranteed to not receive the 5% penalty for non-compilation. You can run the `compileCheck` function in ghci to confirm that the checker has been compiled successfully.

You can also run the `test` function to automatically test your code using some simple test cases. Passing these test cases does not guarantee any marks in the assessment, since different test cases will be used to test your code after submission. You are encouraged to test your functions on a wide variety of inputs to ensure that they are correct.

FAQ

What should I do if the user calls a function with an invalid argument? For example, what if the user calls `repeat_char 'a' (-3)`?

> You only need to return values for valid arguments. Your code can do *anything* for invalid arguments: it can return a nonsensical value, it can throw an exception using `error`, or it can go into an infinite loop. Your code will only be tested on valid arguments, so your response to invalid arguments will not affect your mark.

Can I create helper functions?

Yes. You can create any helper functions that you like. You will probably need to create some helper functions in part C, but you are also allowed to create them for parts A and B if you wish.

Can I call functions from parts A and B in my code for part C?

Yes, you may call any function that you've implemented yourself.

Can I use `show` or `read` in part C?

No. These functions are not on the approved list.

My functions return strange strings. For example, I'm getting `["aaa", "bb"]` instead of `"aaabb"`

The value `["aaa", "bb"]` is a list of strings (which is a list of lists of characters). The value `"aaabb"` is a string (which is a list of characters). You are probably creating a list of strings by trying to join two strings with the `:` operator. The code `"hi" : []` will return `["hi"]`, which is a list of strings. You should use the `++` operator to join strings, eg., `"hello" ++ "world"` will return "`"helloworld"`.

Some questions give names to their arguments, does this mean that pattern matching is not allowed?

Pattern matching is allowed, and you can also name your arguments whatever you wish.

The `test` function in the checker reports that some of my functions are incorrect, throw exceptions, or timeout. Will I receive the penalty?

No. The 5% penalty is only for code that *doesn't compile* when the checker is loaded into ghci. If you can successfully load the checker into ghci, then you will not receive a penalty. You are likely to receive fewer marks for a question if your function doesn't work, however.

The checker doesn't compile with an error like `Could not find module 'Assignment1'`

There are three things to check:

- Are you following the instructions on how to run the checker? Note that just doing something like `:load M:\Checker.hs` or similar **will not work**, because you need to change the directory first.
- Is your file name *exactly* `Assignment1.hs`? If it is misnamed, the compiler won't be able to find it.
- Did you alter or remove the `module Assignment1 ...` line at the top of the template? This line needs to be in place unaltered for the checker to load it in.

If you are still having difficulty, then send an email to get help.

My `Assignment1.hs` file compiles, but the checker does not compile and shows one of the assignment functions in the error. What should I do?

The most likely culprits are:

- You removed or altered the `module Assignment1 ...` line at the top of the template.
- You removed or altered one of the type annotations in the template file.

Make sure that these are all present. If they are, then your source file should also fail to compile, so you can fix that error. If you are still having difficulty, then send an email to get help.