You may work singly or in pairs on this lab: If you work with a partner, turn in a single solution with both names on it.

1 Lists

A list in Haskell is a sequence of values, all of which have the same type. As examples:

- \([1,3,5,6]\) is a 4-element list of \texttt{Ints}
- \(['a','b','c']\) is a 3-element list of \texttt{Chars}
- \([\text{True,True,False,False,True,False}]\) is a 5-element list of \texttt{Bools}.

Most of the lists that we’ll be using in this class will be finite. However, Haskell also allows us to define and use infinite lists; we’ll talk more about the use of infinite lists later this semester.

Haskell provides mechanisms for the easy definition of certain lists. Evaluate each of the following at the Ghci prompt:

\[
\begin{align*}
[15 .. 20] \\
[15 .. (20-2)] \\
['q' .. 'z'] \\
[14 .. 2]
\end{align*}
\]

You may have noticed the pattern here: for numbers, the expression \([m..n]\) yields the list \([m,m+1,m+2,\ldots,n]\). When \(n\) is less than \(m\) (such as the last example above), the empty list \([\text{}]\) is returned. (The same concept also works for characters.)

More generally, the expression \([m,p..n]\) yields the list

\[
[m,m+(p-m),m+2(p-m),m+3(p-m),\ldots,n']
\]

where \(n'\) is as close as possible to \(n\) without exceeding it. The textbook explains this notation a bit more precisely, but you should have the basic idea. Evaluate each of the following, to make sure you understand this notation:

\[
\begin{align*}
[1,3 .. 10] \\
['a','d' .. 'z'] \\
[10,8 .. -3] \\
[1,2 .. 0]
\end{align*}
\]

Without typing in the following example, determine what you think would happen if you were to evaluate it:

\[
[9,9 .. 10]
\]

You can try it out to check your answer if you want, but be prepared to type \texttt{C-c C-c} to stop the evaluation; if that doesn’t work, you may need to delete the Ghci buffer (\texttt{C-x k}).

2 List Comprehensions

Haskell provides a powerful mechanism known as \textit{list comprehension}, which provides a convenient way to construct new lists from existing lists.

A list comprehension typically has three parts: (1) a \textit{generator}, (2) zero or more \textit{tests} or \textit{filters}, and (3) a \textit{transformer}. The generator extracts elements from an existing list, the tests work as filters to select some of those extracted elements, and the transformer uses those selected elements to create the resulting list.

For example, consider the following expression:

\[
[ (x,x*x) | x <- [1..20], \text{even} x, x < 15 ]
\]

Here, \(x <- [1..20]\) is the generator, the tests are \texttt{even x} and \(x<15\), and \((x,x*x)\) is the transformer. This expression creates a list of all pairs of the form \((x,x*x)\), where \(x\) is between 1 and 20, even, and less than 15. (Recall that \texttt{even} is defined in Haskell’s standard Prelude.)

Evaluate this expression, and notice that the elements appear in the order that they were generated. Compare that result to the evaluation of the following:

\[
[ (x,x*x) | x <- [20, 19 .. 1], \text{even} x, x < 15 ]
\]

Once again, the elements appear in the order that they were generated.

As another example, consider the following:

\[
[ [m..n] | (m,n) <- [(3,6), (7,3), (8,4), (4,8)], n == 2*m ]
\]

This expression creates a list of lists of \texttt{Ints}, using only those pairs whose second component is exactly twice the first component. This expression also demonstrates how you can use patterns (e.g., \((m,n)\)) inside the generator to isolate particular pieces of data.

We can also use list comprehensions to define more general functions. For example, the function \texttt{sampleFun} is a generalization of the previous expression, and the functions \texttt{sumSquares} and \texttt{countVowels} are fairly self-explanatory:
3 Your Task (Read the Instructions Carefully!)

You will need to develop a series of lists using a list comprehension, with the list
[1..10] as the generator. That is, each of your answers should have the following
form (using the names one, two, et cetera), where you fill in the blanks as necessary:

one = [ _______ | x <- [1..10] ______________________ ]

The rules: The only changes you should make are to fill in the blanks (and, in some
cases, the second blank does not require filling). Thus, the following rules apply:

- Your answers must use the generator x <- [1..10].
- You must not embed the entire list comprehension within other function calls.

The preceding two rules mean that, for example, to create the list
[10,9,8,7,6,5,4,3,2,1], the following approaches are all forbidden:

[ x | x <- [10, 9 .. 1] ]
[ x | x <- reverse [1..10]]
reverse [ x | x <- [1..10]]

Despite those rules, however, you can use anything you want (including helper functions,
although they’re not actually required) to fill in the blanks themselves.

The Lists to Generate

1. [10,20,30,40,50,60,70,80,90,100]
2. [False,True,False,True,False,True,False,True]
3. [3,4,5,6,7,8,9]
4. [(5),[10],[15],[20],[25],[30]]
5. [(100,True),(200,True),(300,False),(400,False),(500,False)]
6. [(11,12),(13,14),(15,16),(17,18),(19,20)]
7. [[[10],[9],[8],[7],[6],[5],[4],[3],[2],[1]]
8. [[12],[10],[8],[6],[4],[2],[0]]
9. [[[2,3,4],[2,3,4,5,6],[2,3,4,5,6,7,8],[2,3,4,5,6,7,8,9,10]]
10. [30,24,18,12,6,0]

Hint: For the last few problems, you need to be a little creative. As a warmup, think
about transformers like 5*x-3 and [3*x..11]: what are their effects, and how might
you modify them to meet your needs?

What to Hand In: Hand in a single source code file that contains your answers (with
mnemonic names, such as one, two, and so on). You should also submit a transcript
that demonstrates convincingly that your code works as required. Generate your tran-
script only after figuring out the correct answers.

How to hand it in: This lab is due by noon on Friday, February 19. You should submit
the code and transcript to the labeled bin near CST 4-226.