CS457/557 Functional Languages, Spring 2017 **Homework 4**

Due: At the start of class on Thursday, May 11.

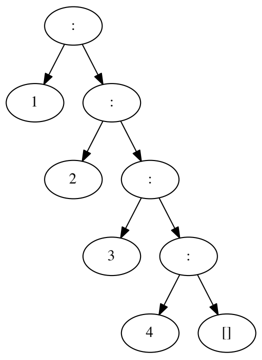
Note: This assignment is written in a more whimsical style than the ones that you've seen previously. I hope that this will make it more fun for you to read. But for anyone who finds this style irritating, I promise I'll get back to the more usual style next time!

In this document, places where answers are expected are indicated by leaving blank space (in many cases, more blank space than you'll likely need, just to be sure). But if you prefer not to print out and complete this document by hand, you're welcome to prepare your solutions as normal in a Haskell source file, together with the additional graphic file that is requested at the end.

Introduction: Professor Kram Senoj is teaching a functional programming class using Haskell, but finds it terribly dull to see the output from the interpreter that he used in class presented as a plain, flat string:

```
Main> [1..4]
[1,2,3,4]
Main>
```

Instead, Prof. Senoj would much prefer to see the output for expressions like this displayed in a more appealing visual form. Wouldn't it be so much nicer, he says, if the same result could be displayed as a pretty picture something like the following:



Somehow, Prof. Senoj has heard that you have the Haskell skills to help out, and he mumbled that he'd give you something you wanted (it sounded like "credit towards a homework assignment") if you could build something he could use, at least for simple examples, with a range of basic datatypes including integers, lists, and perhaps a few more.

However, pictures like the one above can be generated quite easily using the GraphViz library from `http://www.graphviz.org` (which also happens to be ready installed on the departmental Linux machines). And if you're working in Haskell, it probably makes sense to use a little library, like the one in `~mpj/fun/Treedot.lhs`, that can produce dot format descriptions of tree like data structures.

Given these tools, a sensible strategy for helping Prof. Senoj would be:

1) First, create a new Haskell source file and define a tree-like type, `VizTree`, for capturing the structure of data values. Something like the following would be a good choice: it would allow you to specify a `String` label for every node in the tree, and it would also allow each tree node to have zero or more subtrees:

```
data VizTree = VizNode String [VizTree]
```

Of course, if you're going to use this type with the `Treedot` library, then there are a couple of other things we're going to have to write out. What might they be, for instance?

2) Second, define the set of visualizable types, meaning types whose values can be converted into appropriate trees of type `VizTree`. As you quickly realize, there's a classy way to do this:

```
class Viz a where
  toVizTree :: a -> VizTree

instance Viz Integer where
  toVizTree n = VizNode (show n) []

instance Viz Char where
  toVizTree n = VizNode (show n) []

instance Viz a => Viz [a] where
  toVizTree []     = VizNode "[]" []
  toVizTree (x:xs) = VizNode ":"  [toVizTree x, toVizTree xs]
```

When he sees the last line of code in that box, Prof. Senoj seems at first confused. How is it possible to have `toVizTree x` and `toVizTree xs` in the same list? Given that `x` comes from the front of a list, and `xs` comes from the tail, wouldn't they have different types? How can we apply the same function to both values? I'm sure Prof. Senoj would be much happier if you could explain why this code is actually ok.

In fact, I bet Prof. Senoj would be even happier if you would show him how to write out some more instance declarations that would add, say, `Bool`, `Int`, pairs, triples, and `Maybe` types to the `Viz` class. In fact, I'd encourage you to do that right now:

And for some more (slightly mind-bending) fun, how would you visualize a `VizTree`? There are probably several ways to do this, but since Prof. Senoj hasn't specified a particular approach, he will probably be satisfied with any reasonable option. And while you might not have an immediate use for this, but perhaps it will come in handy later on …

But I hope you won't have come away from this thinking that Prof. Senoj isn't following along. Not at all! He realizes now that the way to connect the two items described above in a way that will solve his original problem is to define a function that looks something like this:

```
viz :: Viz a => a -> IO ()
viz  = writeFile "tree.dot" . toDot . toVizTree
```

He suspects you might not be familiar with all the details here just yet, but knows that, with the definition of `viz` in place, he can type expressions like:

```
 viz [1..4::Integer]
```

or even

```
 viz (filter odd [1..10::Integer])
```

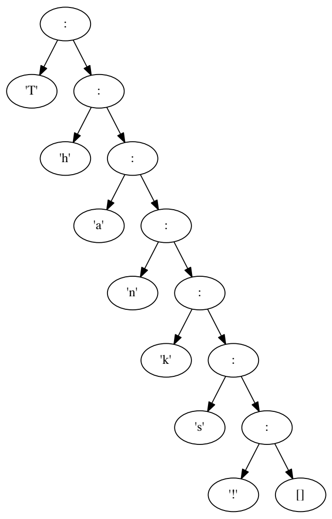
at the interpreter prompt, and then the machine will automatically spit out a pretty picture of that list (or at least, a dot file called `tree.dot` that could be used to draw a pretty picture).

One unfortunate detail here: Prof. Senoj wishes that he didn't have to include that type annotation, `::Integer`, in the examples above. But things don't work correctly when he leaves it out; can you explain why it is necessary in these cases?

It looks like Prof. Senoj will be pretty happy with this mechanism. (Let's call him Kram now; I think we've known him for long enough to be on a first name basis.) But one thing that catches him off guard is when he tries to do:

```
viz "Thanks!"
```

and sees a long list of individual characters in the output instead of a single bubble labelled "Thanks!". In other words, he sees the diagram on the left, but would prefer to see the one on the right:



Recognizing that you've already done a lot to help, Kram doesn't want to bother you again and instead posts a question on a Haskell mailing list, asking for advice. A helpful Haskell guru by the name of Apsus Tudent gets back to him almost immediately with the following suggestion:

- Add three extra lines to the definition of the `Viz` class:

```
class Viz a where
  toVizTree       :: a -> VizTree
  toVizList       :: [a] -> VizTree
  toVizList []     = VizNode "[]" []
  toVizList (x:xs) = VizNode ":"  [toVizTree x, toVizTree xs]
```

- Add an extra line to the instance of `Viz` for the `Char` type:

```
instance Viz Char where
  toVizTree n = VizNode (show n) []
  toVizList s = VizNode ("\\\""++s++"\\\"") []
```

- Replace the instance declaration for `Viz` on lists with:

```
instance Viz a => Viz [a] where
  toVizTree = toVizList
```

Kram is ecstatic. This does exactly what he wanted. He only wishes he knew how it worked. Can you explain this for him, commenting on the particular language features that have been used here, and explaining why it is not necessary to change any other parts of the code?

In the longer term, what aspects of the implementation of `viz` do you think Kram will find most limiting if he continues to use this code while teaching his classes?

On behalf of Prof. Senoj, and his students, thanks for your help in bringing a more visual way to understand functional values to the classroom! Doubtless in all the testing you've done, you've come up with some example outputs that are much more interesting than the ones shown in this document. It would be a good idea to attach one of those diagrams here, wouldn't it, to provide a more impressive demonstration of what the code can do. But there's not a lot of space left here, so you'll need an extra page for that!