Weekly programming exercises

Week 1

Note - only the final part needs a submission! You're free to submit the other parts, but it's not required this week, they are just a warm-up to get you back into Haskell.

Part 1 (not graded)

Here's a short data type:

```
data Tree a = Leaf a
            | Node (Tree a) (Tree a)
      deriving Show
```

Write these three functions:

count :: Tree a -> Integer
which returns the number of nodes in the provided tree
depth :: Tree a -> Integer
which reports the maximum depth of the tree
flatten :: Tree a -> [a]
which converts the tree to a list (you can choose the order of the walk)

Part 2 (not graded)

Take the example Fibonacci programs from the lecture folder and try them out. See if you can reproduce the results described in the lecture (if you don't have a multicore machine available you should be able to use the lab machines via remote access)

Part 3 (1 mark)

Take a look at this implementation of quicksort:

```
quicksort []     = []
quicksort [x]    = [x]
quicksort (x:xs) =
```

```
 leftpartition `par` rightpartition `par` (leftpartition ++ (x:rightpartition))
  where leftpartition = quicksort [y|y <- xs, y < x]
        rightpartition = quicksort [y|y <- xs, y >= x]
```

It seems natural to think this would be a nicely parallelised version. Try it out and you will very likely find that you don't get much of a speedup at all when using multiple cores. The problem is similar to the one experienced with our naive first use of 'par' in the Fibonacci program.
Think about the effect of this helper function (which looks rather odd on the face of it -- surely it's a function that calculates no useful value?):

```
forceList :: [a] -> ()
forceList [] = ()
        forceList (x:xs) = x `seq` forceList xs
```

What role could this function have in improving the performance? Much more importantly, why - what do you think is going on here?

Submission: submit a short description of what you think is happening in the original parallel quicksort that is making is less useful than we'd like, and then comment on what effect you think forceList could have. If you have run this program and experimented with it then you'll probably have a better idea, so include your results in the report.

Week 2

This week we have been mainly looking at Monads. The connection between monads (especially as we see them in IO) and the notion that we are creating something that is related to general patterns of computation isn't always clear, so here are a few exercises that might help to highlight why we are spending so much time thinking about them.

- Write a function with this signature. The function should do _all_ the IO operations in the provided list, in the order that they are provided.
  f1 :: [IO a] -> IO a
- What should this print?
  main = f1 actions
        where actions = [putChar 'h', putChar 'e', putChar 'l', putChar 'l', putChar 'o']
- How about this?
  main = do
        let a = f1 actions
            b = if True then putChar 'a' else putChar 'b'
        putStr "Hi there"
       where actions = [do putChar 'h', ...]
  (and more importantly, why does it print what it does?)
- Can you create a new Haskell Control structure, with this signature? It should perform the provided action repeatedly until the action returns

false.
while :: IO Bool -> IO ()

- Finally, how about writing
  f2 :: [IO a] -> IO [a]
  such that we could write
  read10 :: IO String
  read10 = f2 $ take 10 actions
  where actions = getChar : actions
  to read ten characters.

What to submit

You should submit a single Haskell source file with your attempts. You can include in comments any additional notes about your code, including your idea of why things work the way they do.

Week 3

Part 1:

Lists in Haskell are an instance of monad. For this exercise please write your own instance of "Monad" for a list type.

Because lists get special syntax in Haskell and it would be very inconvenient to suppress the existing definitions, we will write our own list type.

Write an instance of the Monad class (and therefore also Functor and Applicative, of course) for the following type.

data List a = Nil | Cons a (List a)

To be confident the implementation is correct you should really prove the three Monad laws for your instance. If you're not used to this it's actually quite tricky, and I'm not going to make you do it all. But I do want you to think about how it might be done. You should be able to proceed by substituting the definitions of "return" and "bind" to transform one side of the equation in the law until it matches the other side, using the equational reasoning style Meertens used in his Functional Pearl paper. The only tricky case here is the associativity law, because you'll find yourself needing to unpack an inductive proof over the list structure, and it gets a little long winded. In Monday's lecture I'll introduce an alternative way to think about the monad laws that simplifies this, so wait until then to try proving this rule (it's a bit hairy and unless you have some experience building this kind of proof you won't enjoy it).

Part 2:

We met Functor and Applicative in the process of learning about Monad. It might seem pointless to have three classes when one would do. But in fact not everything that can be an instance of Functor can be an instance of Applicative, and not everything that can be an instance of Applicative can be a Monad. Yet they could still be useful.

Tuples are built in, like lists. But they are not Monads. Here's a data type that represents

2-tuples.
data Pair a b = P a b

Give an instance of Functor for Pair, and prove that the Functor laws hold for it.
Attempt to give an instance of Applicative for Pair, notice that you can't. Why?

Week 4

Part 1

A common pattern in programming is to need some form of logging or auditing for a process. In Haskell this is sometimes captured by the writer monad. Writer can be thought of as a variant of the State monad -- it maintains a state, which represents a kind of journal that the monadic computation can modify, but instead of "get" and "put" operations which allow the computation to inspect and arbitrarily update the state there is only one operation, tell, which appends a new entry to the journal. At the end of the computation there is a final value as usual and also the accumulated journal of log entries.

In this exercise you will design and implement a version of the writer monad. While the Writer can be parameterised so that the journal it's keeping is of arbitrary type you can just assume that it's a list of strings for this exercise.

For example, running this

```
 example :: Writer Int
example = do
  tell "entry 1"
  tell "entry 2"
  return (1 + 1)

should deliver (["entry 1", "entry 2"], 2)
```

Part 2

There are two ways you could generalize this monad:

Have the log be a list of any type, which would allow the programmer to create a type for individual log entries

Have the log be of any type at all, letting the programmer create a type for the log overall.

If you want to take the second approach you will run into a problem with your declarations as they stand. You don't have to solve the problem for this exercise, but try making the log type a parameter and determine what the problem is (hint: it is related to the operations that the Writer must perform on the log itself).

For part 2, just write a short note on what the problem you run into is, and speculate on how to solve it. A full answer will be in next weeks class!

Week 5

Attached Files:

- composing-fractals.pdf
  composing-fractals.pdf - Alternative Formats
  (200.182 KB)
- ShapeExample2.zip (7.712 KB)

Week 5 exercises

This week you will gain some experience with the JuicyPixels library which you will use in the assignment. Part 1–3 are a warm up (only part 3 should really require any thought). Part 4 is a little more challenging, but do as much as you can.

The attached Haskell project contains a library for a Shape language that is inspired by the Region and Pan languages we discussed in class (though it's much simpler).At the start of Friday's lecture I will do a quick run through this code to explain it, but it should be clear enough now from the DSL's we discussed. Your tasks:

- Build and run the program and confirm that the shape that's rendered matches the spec
- Experiment with changing the shapes that are drawn to confirm your understanding of how the rendering and drawing windows are structured.
- The rendering routine is very inefficient at the moment; the lookup routine is a O(n) operation, and it's run for each of the n pixels, resulting in an $O(N^2)$ approach to drawing an image. This is terrible: you should be able to get this down to O(n) in the number of pixels (at least -- there are even things that could get it closer to O(n) in the number of pixels used by the actual drawing, but that's far too much work for this exercise. You might consider it for the assignment)
- The attached paper contains (in sections 2 and 3) a description of a function to draw a Mandelbrot set. The form of that function is pretty well suited to rendering with JuicyPixels. Build a program that will render a Mandelbrot set fractal (in black and white, no need to implement the part that talks about colour palettes)

Submit your solutions to parts 3 and 4 (submit a zip file for each part containing your stack project so that I can run them).

Week 9

Attached Files:

- Interpreter_base.lhs (3.611 KB)

This weeks exercise is intended to give you a little experience working with programs written in the Monadic style. It's based on the "interpreter stack" introduced in the week 8 lectures.

Take the example code from the final interpreter in the lectures (I've attached a version

of it here. To simplify the code I removed the syntactic sugar code at the end, and I removed the 'run' function since you'll probably want to re-implement that anyway. Feel free to put them back using the model in "Interpreter 7" if you like, or even start with the Interpreter 7 from the lecture if that's simpler for you).
Your task is to modify the "Run" monad so that as well as running the program fully to completion it also logs each statement that it runs to a list maintained by a Writer monad. This is the sort of data we might collect in order to profile a program and look for hotspots where we could optimise it. For this exercise it's up to you how you handle compound statements like loops -- you can step one loop iteration at a time, or do a big step over the whole loop body, whichever is easier for you.
It's also up to you what sort of detail you log (I really just want to give you an opportunity to modify the set of monads that are being used in Run so that you get some practive with doing that)
Submit your modified interpreter and a short example trace showing it in action.
update: I included the interpreter as a literate haskell program (.lhs) - I should have checked that this was covered in the previous Haskell module, apologies. That link should explain what's going on!

Week 11

For the this weekly assignment I'd like you to get a little practice with ThreePenny.
Build a simple desktop calculator using ThreePenny GUI. It should handle four-function arithmetic (add, subtract, divide, multiply) and display at least 8-digit integers. Include a Clear button to reset the calculator. You don't have to implement anything fancy, and you can implement the actual calculation engine any way you want (including just "read"ing strings that are built up in the UI, or reusing the simple expression evaluator we saw in class).
References:
Overview: https://wiki.haskell.org/Threepenny-gui
API reference http://hackage.haskell.org/package/threepenny-gui