CS 115: Functional Programming, Spring 2016

Assignment 4: `IO`, `IO`, it's off to work we go!

Due: *Monday, May 9, 02:00:00*

Coverage

This assignment covers the material up to lecture 14.

General notes

The theme of this assignment is working with the `IO` monad. We will do so in various ways: with some simple exercises, some simple stand-alone programs, and a more complicated program. When you've finished this assignment you should be very comfortable with the `IO` monad and with the `do` notation for monads.

Submitting your assignment

You will hand in four files for this assignment. The exercises will be in a file called `Lab4a.hs`. The three standalone programs should each be in separate files called `Reverse.hs`, `Columns.hs` and `Sudoku.hs`. Submit them all to csman as usual.

Writing and testing your code

As usual, all of your Haskell functions, even internal ones, should have explicit type signatures.

All of your code should be compiled with the `-W` (warnings enabled) command-line option (whether using `ghc` for standalone programs or `ghci` for interactive testing).

Your code (especially the code for the standalone programs) should be commented appropriately and neatly and reasonably formatted.

Naturally, all of your standalone programs should compile successfully and run correctly before you submit them, so make sure you test your code!

The Sudoku solver should successfully solve all the sample boards in a reasonable amount of time (certainly no more than 30 seconds for any given board).

Part A: Exercises

1. In order to work with the `IO` monad (or with any monad), you need to understand how to desugar monadic code using the `do` notation to code using the monad operators (`(>>=)` and `(>>)`). As we discussed in class, there are two ways to do this, and we will explore both of them here.

    1. **[5]** Consider the following function, which is a manual implementation of the `putStrLn` function:

    ```
    myPutStrLn :: String -> IO ()
    myPutStrLn "" = putChar '\n'
    myPutStrLn (c:cs) =
      do putChar c
         myPutStrLn cs
    ```

    Desugar this into a function that does not use the `do` notation. Either kind of desugaring will give the same answer.

    > *Note on coding style:* A very common way to write this function would be as follows:
    >
    > ```
    > myPutStrLn :: String -> IO ()
    > myPutStrLn "" = putChar '\n'
    > myPutStrLn (c:cs) = do
    >   putChar c
    >   myPutStrLn cs
    > ```
    >
    > This is perfectly acceptable, even though the `do` is situated far to the right of the lines inside the `do`. Haskell's indentation rules only require that these lines be indented relative to the *line* in which `do` is located, not relative to `do` itself. Feel free to use this style when writing your own code, though you don't have to. You can also write it in a non-indentation-dependent style as follows:
    >
    > ```
    > myPutStrLn :: String -> IO ()
    > myPutStrLn "" = putChar '\n'
    > myPutStrLn (c:cs) = do {
    >   ; putChar c
    >   ; myPutStrLn cs
    >   }
    > ```
    >
    > I rarely use this style myself, but some programmers prefer it.

    2. **[1]** Ben Bitfiddle wrote this code as a way to get familiar with the `IO` monad:

    ```
    greet :: String -> IO ()
    greet name = do putStrLn ("Hello, " ++ name ++ "!")
    ```

    Although this code works fine, it exhibits poor style because some of it is redundant. Can you simplify it a little without changing its behavior? *Hint:* We don't mean replacing the parentheses with a `$` operator, though you can do that too.

    3. **[15]** Here is a more complicated version of the `greet` function:

```
-- Ask the user for his/her name, then print a greeting.
greet2 :: IO ()
greet2 = do
  putStr "Enter your name: "
  name <- getLine
  putStr "Hello, "
  putStr name
  putStrLn "!"
```

Desugar this function in two ways, both described in lecture 11: the simple desugaring, and the more complicated desugaring involving an explicit `case` statement. In both cases, preserve the binding of the name to the identifier "`name`" (in other words, don't try to write it in a point-free manner; that's not what we're after here). Does the complex desugaring behave differently than the simple desugaring here? Write your answer in a comment. If there is a pattern match failure for the complex desugaring, use the error message `"Pattern match failure in do expression"`, which is close to what would actually be printed. Recall that we use the `fail` method from the `Monad` type class to handle pattern match failures in `do` expressions.

Note on coding style: Using indentation and formatting in a judicious manner will make the desugared code much easier to read. For instance, don't write the entire definition on one line, even if you can.

4. **[25]** Let's say we want to change the `greet` function so that it will capitalize the name given to it (in case the user entered his/her name without bothering to capitalize it correctly). That might lead to a function like this:

```
-- Need to import this to get the definition of toUpper:
import Data.Char

-- Ask the user for his/her name, then print a greeting.
-- Capitalize the first letter of the name.
greet3 :: IO ()
greet3 = do
  putStr "Enter your name: "
  (n:ns) <- getLine
  let name = toUpper n : ns
  putStr "Hello, "
  putStr name
  putStrLn "!"
```

Again, desugar this in the two different ways (also making sure to handle the `let` expression correctly in both cases). Does the more complex desugaring have any effects here? Write your answer in a comment. *Hint:* What kind of user input could be used to check which desugaring was being used? If there is a pattern match failure for the complex desugaring, again use the error message `"Pattern match failure in do expression"`.

Part B: Simple standalone programs

In this section you will write two fairly simple standalone programs that do input/output.

Note: Do not use `unsafePerformIO` at all in this assignment! If you think you need it, keep thinking.

1. **[30]** Write a program called `reverse` that reads in all the lines of a file and prints them, in reverse order, to standard output (*i.e.* to the terminal).

Usage:

Your program should be invoked as follows (`%` is the prompt):

```
% reverse filename
```

(for some file `filename`). If there are too many or too few command-line arguments, print out a usage message like so:

```
usage: reverse filename
```

and exit with a failure status. Upon successful completion, exit with a success status (see the documentation in the `System.Exit` module for more on exit status.)

Note: You may assume that the file exists and is a text file (you don't have to handle the error cases where this isn't true). This applies to all the programs in this section.

Example:

```
% cat myfile
This is my file.
It has five lines.
This is the third line.
To be or not to be,
that is the question.
% reverse myfile
that is the question.
To be or not to be,
This is the third line.
It has five lines.
This is my file.
```

Useful modules/functions:

You may want to look the following modules/functions up in Hoogle before writing your program.

1. `Prelude` module: `reverse`, `lines`, `mapM_`, `readFile`
2. `System.Environment` module: `getProgName`, `getArgs`
3. `System.Exit` module: `exitFailure`, `exitSuccess`

Writing and compiling the program:

Write the program in a file called `Reverse.hs`. The program should define a module called `Main` and there should be a function called `main` of type `IO ()`. All functions must have explicit type signatures! Compile the program by typing this line:

```
% ghc -W -o reverse Reverse.hs
```

(Without the `%` prompt, of course.) Any compiler warnings should be considered as errors (with some obscure exceptions that you're unlikely to run into; if in doubt, email me).

All other standalone programs described below should be compiled in a similar way.

Note: This program only needs to be a few lines long.

2. **[60]** Write a program called `columns` which will take some numbers and a filename as command-line arguments, and output the corresponding columns of the input to standard output. Column N of a line is defined to be the Nth item in a list which is obtained by splitting the line on whitespace (starting from 1 for the first item).

Usage:

Your program should be invoked as follows:

```
% columns n1 n2 ... filename
```

where `n1`, `n2`, etc. are positive integers (there should be at least one), and where `filename` is the name of the file to read from. If `filename` is `-` (a single dash character), then read from `stdin` (standard input *i.e.* the terminal).

Given invalid inputs (numbers that aren't numbers or are negative or 0), print a usage message and exit as in the previous program. Your usage message must indicate what the proper inputs need to be. Missing columns are not an error; just ignore them.

Column numbers may occur in the command line out of order or duplicated; print the correct value for each column number in the order that it occurs. For instance, if the command line was `columns 1 1 1 foobar` then each output line will have three copies of the first column of that line of the input file `foobar`, and `columns 3 2 1 foobar` would print the first three columns of the file `foobar` in reverse order.

Again, you may assume that the filename argument corresponds to a real text file; you don't have to check for nonexistent files or non-text files.

Examples:

```
% cat myfile
a b c d
foo bar baz
now is the time for all good men
these go to eleven
% columns 1 3 myfile
a c
foo baz
now the
these to
% cat myfile | columns 1 3 -
a c
foo baz
now the
these to
% columns 2 4 5 myfile
b d
bar
is time for
go eleven
```

```
% columns 2 1000 myfile
b
bar
is
go
```

Useful modules/functions:

In addition to the functions mentioned in the previous problem, you might want to look at:

1. `Prelude`: `read`, `words`, `all`, `mapM`
2. `Control.Monad`: `guard`
3. `Data.Char`: `isDigit`
4. `Data.List`: `intercalate`, `unwords`
5. `Data.Maybe`: `mapMaybe`
6. `System.IO`: `stdin`, `hGetContents`

You won't necessarily need to use all of these functions, but many of them should be useful. Note particularly the `read` function, which can parse arbitrary datatypes (as long as they are instance of the `Read` type class, which we haven't discussed in class). For instance, you can parse an integer this way (in `ghci` for illustration):

```
Prelude> read "1286" :: Integer
1286
```

Normally you don't need to put the `:: Integer` into the code, because the context requires the result to be of a particular type (here, an `Integer`).

Writing and compiling the program:

See the previous problem.

Our solution is about 70 lines long, including comments and blank lines.

Part C: Miniproject: Sudoku

Most of you probably already know how to play Sudoku. If by chance you don't, here is a link to teach you the rules. Sudoku puzzles range from quite easy to extremely difficult. However, computers have a much easier time solving Sudoku problems than humans do, and in this section you'll be writing a Haskell program to do just that. Your program should be able to solve the sample Sudoku puzzles we'll supply you with in no more than about 30 seconds per problem, though it may take longer on other Sudoku puzzles.

Program description

The program will read in a file containing a representation of a Sudoku problem, compute the solution to the problem, and print the solution to standard output (the terminal).

Solution algorithm

We want you to solve this problem imperatively. It is possible to write very concise and elegant Sudoku solvers in a purely functional manner (which is a good exercise) but that's not how we want you to do it here. Instead, the program will read the input board's contents into an array (specifically, an `IOArray`). From there, it will mutate the contents of the array until a solution is reached or until it is determined that there is no solution.

The solution algorithm works as follows. You will iterate over the 81 locations in the board row-by-row and column-by-column within rows. If the location is already filled (contains a number between 1 and 9), leave it alone and keep going. If it's empty (contains a 0), you have to pick a number to put into the location. Compute all the possible numbers that could be in this location (*i.e.* numbers that don't conflict with other numbers in the same row, column, or 3x3 box). For each of these numbers, set the location to contain that number, and try to (recursively) solve the board starting from the next location. If this works, you have your solution, so return `True` (the board will have mutated to the solution value). If not, unmake the move (write a zero back into the location) and try the next move. **Unmaking the move is very important!** If you forget to do this your program will not work! If you are at the last location and can't find a suitable number to put there, return `False`.

Inputs and outputs

The compiled program will be called `sudoku` and will take a single argument, which is the name of a file containing a representation of a board. An example board looks like this:

```
.........
....1..92
.86....4.
..156....
.....362.
......5.7
.3.....8.
.9.8.2...
..7..43..
```

The (`.`)s represent empty squares, and numbers from 1 to 9 mean that this location in the board has that (fixed) number. The file must contain nothing else but these characters (not even whitespace at the ends of lines, other than the necessary newline character).

The internal representation of the board is as a two-dimensional `IOArray` (indexed by (`Int`, `Int`) pairs) where an empty square is represented by a zero and a digit from 1 to 9 is represented by the corresponding `Int` value.

The job of the program is to fill in the locations marked with (`.`)s so that the board is solved. In this case, the program would output the following (unique) solution to the Sudoku puzzle:

```
412985763
753416892
986327145
271568934
549173628
368249517
634751289
195832476
827694351
```

This output is simply printed to standard output (the terminal).

Template file

To get you started, we are supplying you with a template file that contains some of the less interesting function definitions pre-written for you. This will allow you to concentrate on the interesting part (the board solving code). You are not absolutely required to use our code (if you think you can do things better all on your own, you may), but using it will probably cut down on the time requirement for this problem significantly. If you do use our code as a template, you should substitute your own code in the locations marked `TODO`. Please remove the `TODO` comments as well!

We are also including the type signatures and comments of all the helper functions we used in the `solveSudoku` function. You may use these to guide you in your solution, or write different helper functions, as you see fit. Note, though, that the solution has to be imperative, not functional. Nevertheless, it will be recursive, as described above.

Note that one thing you are not allowed to change is the representation of the Sudoku board *i.e.* the type definition:

```
type Sudoku = IOArray (Int, Int) Int
```

This problem is an exercise in doing imperative programming in Haskell, so we want you to solve this program imperatively, using the `IO` monad and `IOArray`s.

Another thing you are absolutely not allowed to do, of course, is to copy Sudoku-solving code written in Haskell you find somewhere on the internet or elsewhere.

Sample input boards

We are including a zipped file of ten input boards. You should unzip these and use them to test your program. None of them should take your program more than 30 seconds to solve, and most should take much less time than that.
