Reading

1. Tackling the Awkward Squad, Sections 1 and 2, available from Courseware.
2. Real World Haskell, Chapter 7, available from Courseware.
3. Chapter 8 of the textbook, Sections 8.1 - 8.3 only.
4. Section 7.3 of The Haskell 98 Report, Exception Handling in the I/O Monad. See the link on the Wed, Oct 13 lecture reading assignment.
5. Chapter 3, Sections 3.3, 3.4.2, 3.4.3, 3.4.5, 3.4.8 only.

Problems

1. ................................................. Haskell IO monad programming

   Write a stand-alone Haskell program that takes the name of a file as a command-line argument, reads that file, sorts its lines in order of length, and then writes the result to another file named sorted.txt.

   If you like, you may use the following skeleton code to get started.

   ```haskell
   import System
   import Data.List

   comp :: String -> String -> Ordering
   comp x y = compare (length x) (length y)

   main :: IO ()
   main = do -- your code here
   ```

2. ................................................. bind- and do-Notation

   Recall that the types of the monadic bind (>>=), then (>>), and return (return) operators in Haskell are as follows:

   ```haskell
   (>>=) :: IO a -> (a -> IO b) -> IO b
   (>>) :: IO a -> IO b -> IO b
   return :: a -> IO a
   ```

   Also recall that there are two notations for monadic code in Haskell: do- and bind-notation. To convert between the two, we use the following rewrite rules:

   ```haskell
   do { x<-e; s } = e >>= \x -> do { s }
   do { e; s } = e >> do { s }
   do { e } = e
   ```

   (a) What is the return operator in Haskell used for? Explain briefly.
(b) Haskell provides the >>= operator only as a convenience. Show by example how to define
the >>= operator in terms of the >>>= operator. Specifically, assume that the actions \( m \) and \( n \)
are of monadic type \( \text{IO} \ a \) and \( \text{IO} \ b \) respectively. Write the expression
\[
m >>= n
\]
using the >>= operator (and without using >>). Recall that anonymous functions in Haskell
that ignore their first argument can be written using \( \_ \) (that is: a lambda followed by an
anonymous underscore variable).

(c) Convert the following code fragment, given in do-notation, to bind-notation. Your answer
can use >>= and should not have any instances of do.

```haskell
getLine :: IO [Char]
getLine = do {
c <- getChar ;
if c == '\n'
  then return []
  else
do {
    cs <- getLine;
    return (c:cs)
  }
}
```

3. ................................................................. Using the IO Monad

In this problem you will use the I/O Monad to implement a simple number guessing game. In this
game, the program selects a random integer in the set \{1\ldots10\} that the player has to guess. The
player guesses by entering a number at the prompt. If the player does not guess correctly, the
program tells the player if the guess was too high or low, and asks for another guess. Otherwise,
the program prints a congratulatory message and exits.

To get started, download the files Guesser.hs, Random.hs, and Guesser.py from Courseware.
Random.hs is a utility for generating random integers. Guesser.hs contains an outline of the
game's implementation. You only need to modify Guesser.hs, which you will submit to Course-
ware. Guesser.py is a reference implementation of the game in Python.

We will grade your code using automated scripts. Please make sure your code compiles properly
and contains only the changes that we told you to make. If it doesn't compile, you will get zero
points. Make sure the output and input capabilities of your program conform to that of the
Python reference implementation. Finally, our scripts cannot grade what you write in comments,
so please make sure to un-comment all of the code you want us to grade.

We will provide you with the QuickCheck grading sources after the submission deadline, but
before the mid-term exam so that you can self-grade your code.

(a) The game asks the player to guess until the player guesses correctly. To implement this
behavior, it is useful to have a control structure that executes an action until some condition
is true (in our case, the player guessing correctly.)

Fill in the code for untilIO in Guesser.hs. The function untilIO takes an IO action
producing a Bool and returns an IO action producing nothing: untilIO :: IO Bool ->
IO (). The function untilIO should first execute the loop body (the argument) and then
check the loop condition (the Bool produced by the argument). If the loop condition returns
True, untilIO should exit.

(b) Now implement the core of the game, doGuess :: Int -> IO Bool, which takes the
randomly-generated secret number as an argument, gets the player's current guess using
an IO action, prints whether the guess was low (Too low!), high (Too high!), or just right
(Congratulations!), and returns a Bool indicating whether the player finished the game.
There are two ways of throwing an exception in Haskell:

\[
\begin{align*}
\text{throw} & : \text{Exception} \to a \\
\text{throwIO} & : \text{Exception} \to \text{IO } a
\end{align*}
\]

According to the documentation, "Exceptions may be thrown from purely functional code, but may only be caught within the IO monad.", This question asks you to understand and explain this sentence and some of its consequences, in several steps. Note that the two functions above have different types, and the difference between "purely functional code" and code "within the IO monad" is a distinction enforced by the Haskell type system.

(a) Why does the type of \text{throw} allow this function to be used in "purely functional code"? Explain.

(b) Consider an expression of the form

\[
\text{if } x > 0 \text{ then } x + 1 \text{ else } (\text{throwIO } e)
\]

which tries to use \text{throwIO}. What keeps this from being acceptable Haskell code?

(c) Consider a function that takes a pair of integers and that may use the second integer only under certain circumstances, such as

\[
f(x, y) = \text{if } x > 0 \text{ then } x \text{ else } y
\]

Under lazy evaluation, when would \( f(x, \text{throw}...) \) throw an exception? Explain.

(d) Consider the following monadic code (with line numbers on the left):

\[
\begin{align*}
(1) & \quad \text{do } \{} \\
(2) & \quad \quad x \leftarrow \text{throwIO } e; \\
(3) & \quad \quad \text{return } f(3, x) \\
(4) & \quad \}\n\end{align*}
\]

Translate this \text{do} form to the \text{bind} form by filling in the missing parts below. You may need to write something in each of the three underlined spaces, or you may not need to fill in all three.

\[
(\text{throwIO } e) \quad \text{ >>= } \quad \text{ \underline{\_ \_ \_ \_ \_} } \quad \text{ -> } \quad \text{ \underline{\_ \_ \_ \_ \_} } \quad f(3, \quad \_ \_ \_ \_ \_ )
\]

(e) Explain why the code you wrote in the last part of this problem is type-correct, using the types of infix operator \( \text{ >>= } \) and \text{return} written here in case you do not remember them.

\[
\text{return} :: \text{a} \to \text{IO } a \\
(\text{>>=} ) :: \text{IO } a \to (\text{a} \to \text{IO } b) \to \text{IO } b
\]

(f) Even under lazy evaluation, some functions require their arguments to be evaluated before the result of that function can be computed. For example, a sum \( e_1 + e_2 \) is computed by first evaluating \( e_1 \) and \( e_2 \), then summing those values. Assuming that \( \text{ >>= } \) requires its first argument to be evaluated first, explain the order of evaluation of your translation of the the \text{do} construct above (from part (d)), and the resulting order of evaluation of lines of the original \text{do} code. State whether an exception is thrown and at what point.

(g) The Haskell \text{catch} function has the following type

\[
\text{catch} :: \text{IO } a \to (\text{Exception} \to \text{IO } a) \to \text{IO } a
\]

where the first argument is a computation to run normally, and the second is a function applied to any exception that is thrown in that computation.

This type has the form \( T \to (\text{Exception} \to U) \to V \).

(i) Why are \( T, U, V \) all the same type?

(ii) How does the type \( \text{IO } a \) in the type of \text{catch} relate to the statement that exceptions "may only be caught within the IO monad"?
In this question, we will examine exceptions in Javascript using the following code:

```javascript
1: var y=0;
2: var x=0;
3: function g(h,x) {
4:     try {
5:         h(6);
6:     }
7:     catch(e){ y=x; }
8: }
9: function b(h,x) {
10:    try {
11:        function f(x) {throw "exn";}
12:        h(f,4);
13:    }
14:    catch(e){ y=x; }
15: }
16: try{
17:    b(g,2);
18: }
19: catch(e) { y=x; }
```

Please use the convention that each top-level declaration or block creates a new activation record, although this is not standard Javascript semantics. Assume, for functions whose entire body consists of a single `try...catch` block, that the compiler creates a single activation record for both the function and the `try...catch` block. Also, remember that exception handlers are chosen by dynamic scoping.

(a) The following diagram captures the stack state right before the exception is thrown. Fill in all the blanks in the activation record stack and the closures.

Do not draw any arrows in the diagram; only write numbers and letters. Use the numbers provided in parentheses to refer to individual activation records, and the letters provided to refer to closures and compiled code (integer values can be filled in as-is). Fill in “exn handler” entries on the stack with the LINE NUMBER from the above code snippet representing code with an exception handler. Some “exn handler” entries may be incorrect and thus should be left empty.
Which activation records are popped off the stack after the exception is thrown, before control is passed to the exception handler?

To what value does \( y \) get set at the completion of the exception handler? Assume that exception handlers resolve variables by first searching the activation record in which they are found, and then following normal static scoping rules.

This question asks you to think about garbage collection in Lisp and compare our definition of garbage in the text to one given in McCarthy’s 1960 paper on Lisp. McCarthy’s definition is written for Lisp specifically, while our definition is stated generally for any programming language. Answer the question by comparing the definitions as they apply to Lisp only. Here are the two definitions.

**Garbage, our definition:** At a given point in the execution of a program \( P \), a memory location \( m \) is garbage if no continued execution of \( P \) from this point will access location \( m \).

**Garbage, McCarthy’s definition:** “Each register that is accessible to the program is accessible because it can be reached from one or more of the base registers by a chain of car and cdr operations. When the contents of a base register are changed, it may happen that the register to which the base register formerly pointed cannot be reached by a car-cdr chain from any base register. Such a register may be considered abandoned by the program because its contents can no longer be found by any possible program.”

(a) If a memory location is garbage according to our definition, is it necessarily garbage according to McCarthy’s definition? Explain why or why not.

(b) If a location is garbage according to McCarthy’s definition, is it garbage by our definition? Explain why or why not.
There are garbage collectors that collect everything that is garbage according to McCarthy’s definition. Would it be possible to write a garbage collector to collect everything that is garbage according to our definition? Explain why or why not.