Well, all I can say for ch 10 is, you should definitely read that chapter and understand the example since its one of the shortest chapter after so long, but goes in depth how to tackle a bigger program.

Chapter 8/9: Input and Output:

Summary:

The introduction to the chapter is a great summary of this chapter: "you're going to learn how to receive input from the keyboard and print stuff to the screen," can't get simpler than that. Using the do syntax allows you to use several I/O actions that can also use pure data that is grabbed using Haskell other functions. All I/O actions that fall into the main function are performed which can then be transmitted onto the screen, such as, Hello, world! That's not all you can do with I/O functions, you can also interact with files, deal with command-line arguments, and much more.

Notes: (holy moly, 50 pages, will be super lazy)

It seems that they began with a Hello, world example.

steps to print things out is:

1. define some main in a file (e.g. main = putStrLn "hello, world" # file name is helloworld.hs)
2. compile the program (e.g. ghc --make helloworld)
3. now you can run the program on the terminal (e.g. using ./helloworld)

Examining putStrLn

```
ghci> :t putStrLn
putStrLn :: String -> IO ()
ghci> :t putStrLn "hello, world"
putStrLn "hello, world" :: IO ()
```

putStrLn takes a string and returns an *I/O action* that has a result type of () (that is, the empty tuple, also known as *unit*).

Gluing I/O Actions Together:

You can use the do syntax to glue together several I/O actions. Example:

```
main = do
        putStrLn "Hello, what's your name?"
        name <- getLine
        putStrLn ("Hey " ++ name ++ ", you rock!")
```

<-getLine reads a line from the input and stores into a variable

```
ghci> :t getLine
getLine :: IO String
```

Using let Inside I/O Actions

you can use let syntax to bind pure values to names. the <- is used to perform I/O actions and bind their results to names. let is used when you want to give names to normal values inside I/O actions.

Uses of return do is make I/O actions that yield a result, which is then thrown away because it isn’t bound to a name.

```
main = do
        let a = "hell"
            b = "yeah"
```

```
putStrLn $ a ++ " " ++ b
```

Useful IO functions: (for examples, just look in the book)
putStr is similar to putStrLn and takes a string as a parameter but it doesn't jump to a newline and will continually place into the same line.

The putChar function takes a character and returns an I/O action that will
print it to the terminal

print takes a value of any type that's an instance of Show (meaning that
we know how to represent it as a string), applies show to that value to
"stringify" it, and then outputs that string to the terminal. Basically, it's
just putStrLn .

tl:dr basically a glorified if statement w/o the need of an else
The when function is found in Control.Monad (to access it, use import Control.Monad). It's interesting because in a do block, it looks like a flowcontrol statement, but it's actually a normal function. when takes a Bool and an I/O action, and if that Bool value is True, it returns the same I/O action that we supplied to it. However, if it's False, it returns the return () action, which doesn't do anything

The sequence function takes a list of I/O actions and returns an I/O action that will perform those actions one after the other. The result that this I/O action yields will be a list of the results of all the I/O actions that were performed

mapM takes a function and a list, maps the function over the list, and
then sequences it. mapM_ does the same thing, but it throws away the result
later.

The forever function takes an I/O action and returns an I/O action that just
repeats the I/O action it got forever. It's located in Control.Monad. The following
little program will indefinitely ask the user for some input and spit it
back in all uppercase characters

forM (located in Control.Monad) is like mapM, but its parameters are switched
around. The first parameter is the list, and the second is the function to map
over that list, which is then sequenced.

The (\a -> do ...) lambda is a function that takes a number and returns
an I/O action.

I/O Action Review
Let's run through a quick review of the I/O basics. I/O actions are values much like any other value in Haskell. We can pass them as parameters to functions, and functions can return I/O actions as results. What's special about I/O actions is that if they fall into the main function
(or are the result in a GHCi line), they are performed. And that's when they get to write stuff on your screen or play "Yakety Sax" through your speakers. Each I/O action can also yield a result to tell you what it got from the real world.

CAPTHER 9: rMoe Putin nad rMoe tuPout (More Input and More Output)

TL:DR Interact with files, make random numbers, deal with command-line args and more!

Files and Streams:

A *stream* is a succession of pieces of data entering or exiting a program over time. For instance, when you're inputting characters into a program via the keyboard, those characters can be thought of as a stream.

Input redirection:

To get the input by feeding the contents of a text file to the program. To achieve this, we use *input redirection*

Here's how to do it, have some .txt file ready and then make some .hs file that can forever take in some user <- getLine input. Compile that program and then on the terminal you can type in:

```
./textreader < something.txt
```

You add a < character after the program to specify the file that you want to act as the input.

Getting Strings from Input Streams

getContents makes processing input streams easier by allowing us to treat them as normal strings getContents reads everything from the standard input until it encounters an end-of-file character.

Ehhh more useful Functions: (I wanna go back to sleep TT_TT)

interact takes a function of type String -> String as a parameter and returns an I/O action that will take some input, run that function on it, and then print out the function's result. Example:

```
main = interact shortLinesOnly

shortLinesOnly :: String -> String
shortLinesOnly = unlines . filter (\line -> length line < 10) . lines
```

there's also an interesting palindrome example!

Reading and Writing Files

One way to think about reading from the terminal is that it's like reading from a (somewhat special) file. The same goes for writing to the terminal— it's kind of like writing to a file. We can call these two files *stdout* and *stdin*, meaning standard output and standard input, respectively

This prints out the content of the file:

```
import System.IO
        main = do
        handle <- openFile "girlfriend.txt" ReadMode
        contents <- hGetContents handle
        putStr contents
        hClose handle
```

openFile takes a file path and an IOMode and returns an I/O action that will open a file and yield the file's associated handle as its result. FilePath is just a type synonym for String. Then they start defining its type which leads to more defining … and so on.

openFile returns an I/O action that will open the specified file in the specified mode. If we bind that action's result to something, we get a Handle, which represents where our file is. We'll use that handle so we know which file to read from.

hGetContents takes a Handle, so it knows which file to get the contents from, and returns an IO String—an I/O action that holds contents of the file as its result

A handle just points to our current position in the file. The contents are what's actually in the file.

withFile function, which has the following type signature:
withFile :: FilePath -> IOMode -> (Handle -> IO a) -> IO a
It takes a path to a file, an IOMode, and a function that takes a handle and returns some I/O action.

```
import System.IO
main = do
        withFile "girlfriend.txt" ReadMode (\handle -> do
                contents <- hGetContents handle
                putStr contents)
```

(\handle -> ...) is the function that takes a handle and returns an I/O action, and it's usually done like this, with a lambda.

Because bracket is all about acquiring a resource, doing something with it, and making sure it gets released, implementing withFile is really easy:

```
withFile :: FilePath -> IOMode -> (Handle -> IO a) -> IO a
withFile name mode f = bracket (openFile name mode)
        (\handle -> hClose handle)
        (\handle -> f handle)
```

The first parameter that we pass to bracket opens the file, and its result is a file handle. The second parameter takes that handle and closes it. bracket makes sure that this happens even if an exception is raised. Finally, the third parameter to bracket takes a handle and applies the function f to it, which takes a file handle and does stuff with that handle, like reading from or writing to the corresponding file

Loading files and then treating their contents as strings is so common that we have three nice little functions to make our work even easier:
readFile, writeFile, and appendFile. (read it up on your own time)
example: easier version of girlfriend

```
import System.IO
main = do
        contents <- readFile "girlfriend.txt"
        putStr contents
```

another to modify:

```
import System.IO
import Data.Char
main = do
        contents <- readFile "girlfriend.txt"
        writeFile "girlfriendcaps.txt" (map toUpper contents)
```

Well, this section here I'm giving small descriptions

shows you how to use appendFile function to append things to some file (e.g. .txt file)
shows you how to separate contents from file and give each items a number and user can delete off

To make sure our temporary file is cleaned up in case of a problem, we're going to use the bracketOnError function from Control.Exception. It's very similar to bracket, but whereas the bracket will acquire a resource and then make sure that some cleanup always gets done after we've used it, bracketOnError performs the cleanup only if an exception has been raised.

Command-Line Arguments
Everything previously is an interactive program.
The System.Environment module has two cool I/O actions that are useful for getting command-line arguments: getArgs and getProgName. Examples:

```
import System.Environment
import Data.List
main = do
        args <- getArgs
        progName <- getProgName
        putStrLn "The arguments are:"
        mapM putStrLn args
        putStrLn "The program name is:"
        putStrLn progName
```

testing it:

```
$ ./arg-test first second w00t "multi word arg"
The arguments are:
first
second
w00t
multi word arg
The program name is:
arg-test
```

A lotta examples with To-Do Lists
The things on dealing with Bad Inputs
Then random data through functions
Bytestrings:
When the first element of the list is forcibly evaluated (say by printing it), the rest of the list 2:3:4:[] is still just a promise of a list, and so on. We call that promise a *thunk*. A thunk is basically a deferred computation. Haskell achieves its laziness by using thunks and computing them only when it must, instead of computing everything up front.

That overhead doesn't bother us most of the time, but it turns out to be a liability when reading big files and manipulating them. That's why Haskell has *bytestrings*. Bytestrings are sort of like lists, only each element is one byte (or 8 bits) in size. The way they handle laziness is also different.

Bytestrings come in two flavors: strict and lazy. Strict bytestrings reside in Data.ByteString, and they do away with the laziness completely. There are no thunks involved. A strict bytestring represents a series of bytes in an array.

The other variety of bytestrings resides in Data.ByteString.Lazy. They’re lazy, but not quite as lazy as lists. Since there are as many thunks in a list as there are elements, they are kind of slow for some purposes. Lazy bytestrings take a different approach. They are stored in chunks (not to be confused with thunks!), and each chunk has a size of 64KB. So if you evaluate a byte in a lazy bytestring (by printing it, for example), the first 64KB will be evaluated. After that, it’s just a promise for the rest of the chunks.

Then a whole buncha examples with bytestrings

Chapter 10: Functionally Solving Problems:

Summaries:
The chapter mainly ties everything we learned so far in the form of two big examples. What I saw is that they wrote out the problem then listed out steps of how to go about solving the problem. Then you had to find a way to represent all of the data in some way to pass into the functions. They gradually built the functions that would be used and finally a main function that ties together all of the defined functions.

Notes:
They gave an example of how to make an RPN calculator in Haskell.
Some Heathrow to London example.

and that's about it from the chapter, just look it over in the book because it's pretty in depth.