**. Basic IO**

* We need side effects
* What is IO
* main + putStrLn + composing other functions
* >>
* >>=
* do notation
  + do
  + <-
  + let
* Some examples
* Read/Write to console
* Read/Write to file

**Input/Output**

The I/O system in Haskell is purely functional, yet has all of the expressive power found in conventional programming languages. In imperative languages, programs proceed via *actions* which examine and modify the current state of the world. Typical actions include reading and setting global variables, writing files, reading input, and opening windows. Such actions are also a part of Haskell but are cleanly separated from the purely functional core of the language.

Haskell's I/O system is built around a somewhat daunting mathematical foundation: the *monad*. However, understanding of the underlying monad theory is not necessary to program using the I/O system. Rather, monads are a conceptual structure into which I/O happens to fit. It is no more necessary to understand monad theory to perform Haskell I/O than it is to understand group theory to do simple arithmetic. A detailed explanation of monads is found in Section [9](https://www.haskell.org/tutorial/monads.html#tut-monads).

The monadic operators that the I/O system is built upon are also used for other purposes; we will look more deeply into monads later. For now, we will avoid the term monad and concentrate on the use of the I/O system. It's best to think of the I/O monad as simply an abstract data type.

Actions are defined rather than invoked within the expression language of Haskell. Evaluating the definition of an action doesn't actually cause the action to happen. Rather, the invocation of actions takes place outside of the expression evaluation we have considered up to this point.

Actions are either atomic, as defined in system primitives, or are a sequential composition of other actions. The I/O monad contains primitives which build composite actions, a process similar to joining statements in sequential order using `;' in other languages. Thus the monad serves as the glue which binds together the actions in a program.

**Basic I/O Operations**

Every I/O action returns a value. In the type system, the return value is `tagged' with IO type, distinguishing actions from other values. For example, the type of the function getChar is:  
  
getChar                 ::   IO Char  
  
The IO Char indicates that getChar, when invoked, performs some action which returns a character. Actions which return no interesting values use the unit type, (). For example, the putChar function:  
  
putChar                 ::    Char -> IO ()  
  
takes a character as an argument but returns nothing useful. The unit type is similar to void in other languages.

Actions are sequenced using an operator that has a rather cryptic name: >>= (or `bind'). Instead of using this operator directly, we choose some syntactic sugar, the do notation, to hide these sequencing operators under a syntax resembling more conventional languages. The do notation can be trivially expanded to.

The keyword do introduces a sequence of statements which are executed in order. A statement is either an action, a pattern bound to the result of an action using <-, or a set of local definitions introduced using let. The do notation uses layout in the same manner as let or where so we can omit braces and semicolons with proper indentation. Here is a simple program to read and then print a character:  
  
main                    :: IO ()  
main                    =  do c <- getChar  
                              putChar c  
  
The use of the name main is important: main is defined to be the entry point of a Haskell program (similar to the main function in C), and must have an IO type, usually IO (). (The name main is special only in the module Main; we will have more to say about modules later.) This program performs two actions in sequence: first it reads in a character, binding the result to the variable c, and then prints the character. Unlike a let expression where variables are scoped over all definitions, the variables defined by <- are only in scope in the following statements.

There is still one missing piece. We can invoke actions and examine their results using do, but how do we return a value from a sequence of actions? For example, consider the ready function that reads a character and returns True if the character was a `y':  
  
ready                   :: IO Bool  
ready                   =  do c <- getChar  
                              c == 'y'  -- Bad!!!  
  
This doesn't work because the second statement in the `do' is just a boolean value, not an action. We need to take this boolean and create an action that does nothing but return the boolean as its result. The return function does just that:  
  
return                  ::   a -> IO a  
  
The return function completes the set of sequencing primitives. The last line of ready should read return (c == 'y').

We are now ready to look at more complicated I/O functions. First, the function getLine:  
  
getLine     :: IO String  
getLine     =  do c <- getChar  
                  if c == '\n'  
                       then return ""  
                       else do l <- getLine  
                               return (c:l)  
  
Note the second do in the else clause. Each do introduces a single chain of statements. Any intervening construct, such as the if, must use a new do to initiate further sequences of actions.

The return function admits an ordinary value such as a boolean to the realm of I/O actions. What about the other direction? Can we invoke some I/O actions within an ordinary expression? For example, how can we say x + print y in an expression so that y is printed out as the expression evaluates? The answer is that we can't! It is *not* possible to sneak into the imperative universe while in the midst of purely functional code. Any value `infected' by the imperative world must be tagged as such. A function such as  
  
f    ::  Int -> Int -> Int  
  
absolutely cannot do any I/O since IO does not appear in the returned type. This fact is often quite distressing to programmers used to placing print statements liberally throughout their code during debugging. There are, in fact, some unsafe functions available to get around this problem but these are better left to advanced programmers. Debugging packages (like Trace) often make liberal use of these `forbidden functions' in an entirely safe manner.

**Programming With Actions**

I/O actions are ordinary Haskell values: they may be passed to functions, placed in structures, and used as any other Haskell value. Consider this list of actions:  
  
todoList :: [IO ()]  
  
todoList = [putChar 'a',  
            do putChar 'b'  
               putChar 'c',  
            do c <- getChar  
               putChar c]  
  
This list doesn't actually invoke any actions---it simply holds them. To join these actions into a single action, a function such as sequence\_ is needed:  
  
sequence\_        :: [IO ()] -> IO ()  
sequence\_ []     =  return ()  
sequence\_ (a:as) =  do a  
                       sequence as  
  
This can be simplified by noting that do x;y is expanded to x >> y .

This pattern of recursion is captured by the foldr function (see the Prelude for a definition of foldr); a better definition of sequence\_ is:  
  
sequence\_        :: [IO ()] -> IO ()  
sequence\_        =  foldr (>>) (return ())  
  
The do notation is a useful tool but in this case the underlying monadic operator, >>, is more appropriate. An understanding of the operators upon which do is built is quite useful to the Haskell programmer.

The sequence\_ function can be used to construct putStr from putChar:  
  
putStr                  :: String -> IO ()  
putStr s                =  sequence\_ (map putChar s)  
  
One of the differences between Haskell and conventional imperative programming can be seen in putStr. In an imperative language, mapping an imperative version of putChar over the string would be sufficient to print it. In Haskell, however, the map function does not perform any action. Instead it creates a list of actions, one for each character in the string. The folding operation in sequence\_ uses the >> function to combine all of the individual actions into a single action. The return () used here is quite necessary -- foldr needs a null action at the end of the chain of actions it creates (especially if there are no characters in the string!).

The Prelude and the libraries contains many functions which are useful for sequencing I/O actions. These are usually generalized to arbitrary monads; any function with a context including Monad m => works with the IO type.

**Exception Handling**

So far, we have avoided the issue of exceptions during I/O operations. What would happen if getChar encounters an end of file? (We use the term *error* for *\_|\_*: a condition which cannot be recovered from such as non-termination or pattern match failure. Exceptions, on the other hand, can be caught and handled within the I/O monad.) To deal with exceptional conditions such as `file not found' within the I/O monad, a handling mechanism is used, similar in functionality to the one in standard ML. No special syntax or semantics are used; exception handling is part of the definition of the I/O sequencing operations.

Errors are encoded using a special data type, IOError. This type represents all possible exceptions that may occur within the I/O monad. This is an abstract type: no constructors for IOError are available to the user. Predicates allow IOError values to be queried. For example, the function  
  
isEOFError       :: IOError -> Bool  
  
determines whether an error was caused by an end-of-file condition. By making IOError abstract, new sorts of errors may be added to the system without a noticeable change to the data type. The function isEOFError is defined in a separate library, IO, and must be explicitly imported into a program.

An *exception handler* has type IOError -> IO a. The catch function associates an exception handler with an action or set of actions:  
  
catch                     :: IO a -> (IOError -> IO a) -> IO a  
  
The arguments to catch are an action and a handler. If the action succeeds, its result is returned without invoking the handler. If an error occurs, it is passed to the handler as a value of type IOError and the action associated with the handler is then invoked. For example, this version of getChar returns a newline when an error is encountered:  
  
getChar'                :: IO Char  
getChar'                =  getChar `catch` (\e -> return '\n')  
  
This is rather crude since it treats all errors in the same manner. If only end-of-file is to be recognized, the error value must be queried:  
  
getChar'                :: IO Char  
getChar'                =  getChar `catch` eofHandler where  
    eofHandler e = if isEofError e then return '\n' else ioError e  
  
The ioError function used here throws an exception on to the next exception handler. The type of ioError is  
  
ioError                 :: IOError -> IO a  
  
It is similar to return except that it transfers control to the exception handler instead of proceeding to the next I/O action. Nested calls to catch are permitted, and produce nested exception handlers.

Using getChar', we can redefine getLine to demonstrate the use of nested handlers:  
  
getLine'        :: IO String  
getLine'        = catch getLine'' (\err -> return ("Error: " ++ show err))  
        where  
                   getLine'' = do c <- getChar'  
                         if c == '\n' then return ""  
                                            else do l <- getLine'  
                                                    return (c:l)

The nested error handlers allow getChar' to catch end of file while any other error results in a string starting with "Error: " from getLine'.

For convenience, Haskell provides a default exception handler at the topmost level of a program that prints out the exception and terminates the program.

**Files, Channels, and Handles**

Aside from the I/O monad and the exception handling mechanism it provides, I/O facilities in Haskell are for the most part quite similar to those in other languages. Many of these functions are in the IO library instead of the Prelude and thus must be explicitly imported to be in scope. Also, many of these functions are discussed in the Library Report instead of the main report.

Opening a file creates a *handle* (of type Handle) for use in I/O transactions. Closing the handle closes the associated file:  
  
type FilePath         =  String  -- path names in the file system  
openFile              :: FilePath -> IOMode -> IO Handle  
hClose                :: Handle -> IO ()   
data IOMode           =  ReadMode | WriteMode | AppendMode | ReadWriteMode  
  
Handles can also be associated with *channels*: communication ports not directly attached to files. A few channel handles are predefined, including stdin (standard input), stdout (standard output), and stderr (standard error). Character level I/O operations include hGetChar and hPutChar, which take a handle as an argument. The getChar function used previously can be defined as:  
  
getChar                = hGetChar stdin  
  
Haskell also allows the entire contents of a file or channel to be returned as a single string:  
  
getContents            :: Handle -> IO String  
  
Pragmatically, it may seem that getContents must immediately read an entire file or channel, resulting in poor space and time performance under certain conditions. However, this is not the case. The key point is that getContents returns a "lazy" (i.e. non-strict) list of characters (recall that strings are just lists of characters in Haskell), whose elements are read "by demand" just like any other list. An implementation can be expected to implement this demand-driven behavior by reading one character at a time from the file as they are required by the computation.

In this example, a Haskell program copies one file to another:  
  
main = do fromHandle <- getAndOpenFile "Copy from: " ReadMode  
          toHandle   <- getAndOpenFile "Copy to: " WriteMode   
          contents   <- hGetContents fromHandle  
          hPutStr toHandle contents  
          hClose toHandle  
          putStr "Done."  
  
getAndOpenFile          :: String -> IOMode -> IO Handle  
  
getAndOpenFile prompt mode =  
    do putStr prompt  
       name <- getLine  
       catch (openFile name mode)  
             (\\_ -> do putStrLn ("Cannot open "++ name ++ "\n")  
                       getAndOpenFile prompt mode)  
           
  
By using the lazy getContents function, the entire contents of the file need not be read into memory all at once. If hPutStr chooses to buffer the output by writing the string in fixed sized blocks of characters, only one block of the input file needs to be in memory at once. The input file is closed implicitly when the last character has been read.

**7.5  Haskell and Imperative Programming**

As a final note, I/O programming raises an important issue: this style looks suspiciously like ordinary imperative programming. For example, the getLine function:  
  
getLine         = do c <- getChar  
                     if c == '\n'  
                          then return ""  
                          else do l <- getLine  
                                  return (c:l)  
  
bears a striking similarity to imperative code (not in any real language) :  
  
  
function getLine() {  
  c := getChar();  
  if c == `\n` then return ""  
               else {l := getLine();  
                     return c:l}}  
  
So, in the end, has Haskell simply re-invented the imperative wheel?

In some sense, yes. The I/O monad constitutes a small imperative sub-language inside Haskell, and thus the I/O component of a program may appear similar to ordinary imperative code. But there is one important difference: There is no special semantics that the user needs to deal with. In particular, equational reasoning in Haskell is not compromised. The imperative feel of the monadic code in a program does not detract from the functional aspect of Haskell. An experienced functional programmer should be able to minimize the imperative component of the program, only using the I/O monad for a minimal amount of top-level sequencing. The monad cleanly separates the functional and imperative program components. In contrast, imperative languages with functional subsets do not generally have any well-defined barrier between the purely functional and imperative worlds.