More Input and Output

Thumrongsak Kosiyatrakul
tkosiyat@pitt.edu

Input Redirection

- So far, inputs to our program come from keyboard via the `getLine` function
- Consider the following text file named `sentences.txt`

```
Getting up at dawn is for the birds.
Let me help you with your baggage.
On each full moon.
```

- Suppose we want each line from the above file as the input to our program
- This can be done via **input redirection**

Input Redirection

- Example: `toUpper.hs`

```
import Control.Monad
import Data.Char

main = forever (do
    aLine <- getLine
    putStrLn (map toUpper aLine))
```

- Compile and run:

```
$ ghc --make toUpper.hs
[1 of 1] Compiling Main             ( toUpper.hs, toUpper.o )
Linking toUpper ...

$ ./toUpper < sentences.txt
GETTING UP AT DAWN IS FOR THE BIRDS.
LET ME HELP YOU WITH YOUR BAGGAGE.
ON EACH FULL MOON.
toUpper: <stdin>: hGetLine: end of file
```

Input Stream

- Instead of dealing with input streams directly, Haskell provides some I/O actions that make accessing input stream easier
- They allow us to treat an input stream as a normal string
- `getContents` reads everything from the standard input until it encounters an end-of-file character

```
ghci> :t getContents
getContents :: IO String
```

- `getContents` is a lazy I/O
- `foo <- getContents` will not cause the `getContents` function to read the file
- In previous example, we use `forever` and `getLine` but `getContents` will take care of end-of-file for us
- Once `getContents` encounters end-of-file, it turns its content to a string

The `getContents` Function

- Here is a new version of `toUpper`, `newToUpper.hs`:

```
import Data.Char

main = do
    aLine <- getContents
    putStrLn (map toUpper aLine)
```

- Here is a couple runs:

```
$ ./newToUpper < sentences.txt
GETTING UP AT DAWN IS FOR THE BIRDS.
LET ME HELP YOU WITH YOUR BAGGAGE.
ON EACH FULL MOON.

$ ./newToUpper
Hello
HELLO
Haskell
HASKELL
```

- Without input redirection, we still can use keyboard to enter inputs
- Use Ctrl-D to end the input

The `lines` and `unlines` Functions

- We deal with lines in a text file quite a lot
- The `lines` turns a string containing multiple lines into a list of strings of each line

```
ghci> lines "Hello\nWorld\nHow are you?\nI'm fine"
["Hello","World","How are you?","I'm fine"]
```

- The `unlines` takes a list of strings and turns them into a single string by separating them with newline

```
ghci> unlines ["Hello","World","How are you?","I'm fine"]
"Hello\nWorld\nHow are you?\nI'm fine\n"
```

- Note that `unlines` put the newline at the end of the last string

The `interact` Function

- `interact` takes a function of type `String -> String` and returns an I/O action that
  1. takes some input
  2. runs that function on the input
  3. prints out the result
- Here is its type:

```
ghci> :t interact
interact :: (String -> String) -> IO ()
```

- Here is `anotherToUpper.hs` using `interact`:

```
import Data.Char

main = interact myToUpper

myToUpper str = map toUpper str
```

Exercise

- Suppose I have the following text file (`words.txt`):

```
hello
racecar
elephant
rotor
ottoman
madam
```

- Write a Haskell program named `isPalindrome` such that when executing with the following command:

```
$ ./isPalindrome < words.txt
```

the output will be

```
hello - not a palindrome
racecar - palindrome
elephant - not a palindrome
rotor - palindrome
ottoman - not a palindrome
madam - palindrome
```

Exercise

- Check whether a given string is a palindrome:

```
isPalindrome xs = xs == reverse xs
```

- Turn a given string `str` into `str -` followed by whether it is a palindrome

```
genResult xs = if isPalindrome xs
               then xs ++ " - palindrome"
               else xs ++ " - not a palindrome"
```

- Here is the `main`:

```
main = do
    content <- getContents
    putStrLn $ (unlines . map genResult . lines) content
```

- Notes:
  - `content` will be the whole file
  - `lines` splits lines into list of strings
  - `unlines` turns list of strings into one by separated them with `'\n'`

Reading Files

- Instead of input redirection, we can read the file directly in Haskell
- The `openFile` function takes a file path as a string, and an IO mode, and returns an IO handle

```
ghci> :m + System.IO
ghci> :t openFile
openFile :: FilePath -> IOMode -> IO Handle
```

- The `hGetContents` takes a `Handle` and and returns an `IO String`

```
ghci> :t hGetContents
hGetContents :: Handle -> IO String
```

- Here is how to read the file `sentences.txt` and print its content to the console screen:

```
import System.IO

main = do
    handle <- openFile "sentences.txt" ReadMode
    content <- hGetContents handle
    putStrLn content
    hClose handle
```

Reading File

- The `FilePath` is simply a string
  - Absolute path:
    `"/home/john/Documents/Haskell/aFile.txt"`
  - Relative path: `"../../aDirectory/aFile.txt"`
- `IO Mode` is just an enumerate type as shown below:

```
data IOMode = ReadMode | WriteMode | AppendMode | ReadWriteMode
```

- Similar to Java and C, we need to close the file (handle) when we are done:

```
    hClose handle
```

The `withFile` Function

- Another way is to use the `withFile` function:

```
ghci> :t withFile
withFile :: FilePath -> IOMode -> (Handle -> IO r) -> IO r
```

- The third argument is a function that takes a handle and returns an I/O action
- Here is another version of the previous program using `withFile`:

```
import System.IO

main = withFile "sentences.txt" ReadMode readAndPrint

readAndPrint aHandle = do
    content <- hGetContents aHandle
    putStrLn content
```

- `withFile` also makes sure that the file is closed
- Even if an exception occurs, it makes sure to close the file

The `bracket` Function

- Exceptions can happen while a file is open or trying to acquire a resource
- Let's take a look at the `bracket` function from `Control.Exception` module:

```
ghci> :t bracket
bracket :: IO a -> (a -> IO b) -> (a -> IO c) -> IO c
```

- The `bracket` function takes three parameters:
  - An I/O action that acquire a resource
  - The function that releases the resource if an exception has been raised
  - The function that do something with the resource
- Whether or not an exception occurs during the I/O action of the third argument, the I/O action of the second argument will be performed
- You can think about `bracket` as `try..catch..finally` in Java

Work with Handles

- Note that `gGetContents` is pretty much the same as `getContents` except that it works on a specific file (handle)
- We also have `hGetLine`, `hPutStr`, `hPutStrLn`, and so on

```
ghci> :t hGetLine
hGetLine :: Handle -> IO String
ghci> :t hPutStr
hPutStr :: Handle -> String -> IO ()
ghci> :t hPutStrLn
hPutStrLn :: Handle -> String -> IO ()
```

- If a function name starts with `h`, most likely it works on a handle
- Haskell also provides the `readFile` function:

```
ghci> :t readFile
readFile :: FilePath -> IO String
```

- Here is another version using `readFile`:

```
import System.IO

main = do
    content <- readFile "sentences.txt"
    putStrLn content
```

The `writeFile` Function

- To write to a file, we can use the `writeFile` function from `System.IO`:

```
ghci> :t writeFile
writeFile :: FilePath -> String -> IO ()
```

- It takes a file path and a string to write to the file
- Here is an example of a program that read the file `sentences.txt`, turns every character to an uppercase, and write the output to a new file named `allCaps.txt`:

```
import System.IO
import Data.Char

main = do
    content = readFile "sentences.txt"
    writeFile "allCaps.txt" (map toUpper content)
```

The `appendFile` Function

- To append a string to a file, we can use the `appendFile` function from `System.IO`:

```
ghci> :t appendFile
appendFile :: FilePath -> String -> IO ()
```

- It takes a file path and a string to append to the end of the file
- Here is an example of a program that append the string `"- The End -"` at the end of `aFile.txt`:

```
import System.IO

main = appebdFile "aFile.txt" "-- The End --"
```

- If the file does not exists, the `appendFile` function will create a new file

Some Useful Functions

- The `delete` function from `Data.List`:
  - `delete` takes an item and a list, and remove the first occurrence of the item from the list

```
ghci> :t delete
delete :: Eq a => a -> [a] -> [a]
```

  - Example:

```
ghci> delete 5 [3,5,1,5,2,4]
[3,1,5,2,4]
```

- The `removeFile` from `System.Directory`
  - `removeFile` takes a string (`FilePath`) and remove the file

```
ghci> :t removeFile
removeFile :: FilePath -> IO ()
```

  - Example:

```
ghci> :t removeFile
    removeFile "aFile.txt"
```

Some Useful Functions

- The `renameFile` from `System.Directory`
  - `renameFile` renames the first `FilePath` to the second `FilePath`:

```
ghci> :t renameFile
renameFile :: FilePath -> FilePath -> IO ()
```

  - Example:

```
    renameFile "oldName.txt" "newName.txt"
```

- The `openTempFile` from `System.IO`
  - `openTempFile` takes a path to a temporary directory and a template name

```
ghci> :t openTempFile
openTempFile :: FilePath -> String -> IO (FilePath, Handle)
```

    - It returns a given name with random characters to ensure that it will not overwrite an existing file
    - It also returns a handle associate with the returned file
  - Example:

```
    (tempFile, tempHandle) <- openTempFile "." "temp"
```

Cleanup

- Let's look at the type of `bracketOnError` from `Control.Exception`:

```
ghci> :t bracketOnError
bracketOnError :: IO a -> (a -> IO b) -> (a -> IO c) -> IO c
```

  - The first argument is the computation to run first ("acquire resource")
  - The second argument is the computation to run last ("release resource")
  - The thrid argument is the computation to run in-between
- Like `bracket`, but only performs the final action (second argument) if there was an exception raised by the in-between computation
- You can think about `bracket` as `try..catch` in Java

Command-Line Arguments

- The `System.Environment` module provides the functions related to command-line arguments
- The `getArg` function:

```
ghci> :t getArgs
getArgs :: IO [String]
```

- The `getProgName` function:

```
ghci> :t getProgName
getProgName :: IO String
```

- Example:

```
main = do
    args <- getArgs
    name <- getProgName
    :
```

  - `args` is a list of strings (can be empty)
  - `name` is a string (name of the program)

Exercise: To-Do Lists

- Let's create a command-line program that can perform the following tasks:
  1. View task:

```
$ ./myToDo view aToDo.txt
```

  2. Add task:

```
$ ./myToDo add aToDo.txt "Start Assignment 6"
```

  3. Remove task number 5 (0 is the first task):

```
$ ./myToDo remove aToDo.txt 5
```

- Each task is a line in a to-do list file

```
Take CS15xx quiz
Submit CS1699 assignment 4
Study CS15xx Chapter y
```

Exercise: To-Do Lists

- Required Modules:

```
import System.Environment     -- command-line argument
import System.Directory       -- removeFile and renameFile
import System.IO
import Data.List
import Control.Exception
```

- Obviously, the first thing to do is to get the command-line argument which can be one of these three format:
  - `view aFileName`
  - `add aFileName aTask`
  - `remove aFileName aNumber`
- Recall that the `getArg` function returns a list of arguments (list of strings)
- The first string (`head`) will tell us what to do (command)
- The rest (`tail`) will be the arguments for the given command

Exercise: To-Do Lists

- A common way to do is to call a function based on a given command
- Consider the following action:

```
    (command:args) <- getArg
```

- `command` will be the head of the arguments and `args` will be the tail
- Now, we can call a function based on the string `command`

```
sendArguments "view" args = view args
sendArguments "add" args = add args
sendArguments "remove" args = remove args
```

The view Command

- Suppose a to-do list file contains the following content:

```
Take CS15xx quiz
Submit CS1699 assignment 4
Study CS15xx Chapter y
```

- The view command should print the following on the console screen:

```
0 - Take CS15xx quiz
1 - Submit CS1699 assignment 4
2 - Study CS15xx Chapter y
```

- Numbered tasks will allow a user to correctly remove a task
- What to do?
  1. Read file as a long string
  2. Split the string based on newline characters
  3. Append `"n - "` in front of each line where `n` is a number starting from 0
  4. Put all lines back to one single string
  5. Print it on the console screen

The `view` Command

- Here is the `view` Function:

```
view args = do
    contents <- readFile (head args)
    let result = unlines $ zipWith
                           (\line num -> show num ++ " - " ++ line)
                           (lines contents)
                           [0..]
    putStr result
```

- `lines contents` is a list of tasks (list of strings)
- Attach a number in front of each string by `zipWith`
- Put them back into one string using `unlines`

The `add` Command

- Arguments to the `add` function are:
  1. A to-do list filename of type string
  2. A task to add of type string
- Ideally, we can just simply append it to the end of the file

```
add args = do
    let [file,task] = args
    appendFile file task
```

- But what if the end of the file does not contain the newline character
  - Append `"Get a new iPhone"` to

```
Take CS15xx quiz
Submit CS1699 assignment 4
Study CS15xx Chapter y
```

results in

```
Take CS15xx quiz
Submit CS1699 assignment 4
Study CS15xx Chapter yGet a new iPhone
```

The `add` Command

- For simplicity, let's assume that a user can only add new task from our program
- We will ensure that the file will always ends with the newline character
- Here is a simple `add` function using `appendFile`

```
add args = do
    let [file,task] = args
    appendFile file (task ++ ['\n'])
```

- A hard way is to read the whole file and check the last character before appending a new task

The `remove` Command

- Arguments to the `remove` function are:
  1. A to-do list filename of type string
  2. A number (as a string)
- For simplicity, if the number is out-of-bound, we will do nothing
- Obtains the arguments and read the file:

```
    let file = args !! 0
        index = read (args !! 1) :: Int
    toDo <- readFile file
```

- Create a new list of tasks by removing the item at the index

```
    let taskList = filter (/="") (lines toDo)
        item = taskList !! index
        newTaskList = delete item taskList
```

- `filter (/="")` to filter out the empty lines (not necessary)

The `remove` Command

- Create a temp file and put the new list of tasks in there:

```
(tempFile, tempHandle) <- openTempFile "." "temp"
hPutStr tempHandle (unlines newTaskList)
hClose tempHandle
```

- Do not forget to close the file
- Remove the file and rename the temp file

```
removeFile file
renameFile tempFile file
```

- A test:

```
$ ./myToDo add todo01.txt "Pick up new movie"
$ ./myToDo add todo01.txt "Get a piece of salmon"
$ ./myToDo view todo01.txt
0 - Pick up new movie
1 - Get a piece of salmon
$ ./myToDo add todo01.txt "Publish assignment 7"
$ ./myToDo view todo01.txt
0 - Pick up new movie
1 - Get a piece of salmon
2 - Publish assignment 7
$ ./myToDo remove todo01.txt 1
$ ./myToDo view todo01.txt
0 - Pick up new movie
1 - Publish assignment 7
```

Random Number Generator

- A way to generate a random number in Haskell is to use the `random` function from `System.Random`

```
ghci> :t random
random :: (Random a, RandomGen g) => g -> (a, g)
```

- `random` takes a `RandomGen` and returns a random number and a new `RandomGen`
- The `RandomGen` type class is for types that can act as sources of randomness

```
ghci> :i RandomGen
class RandomGen g where
  next :: g -> (Int, g)
  genRange :: g -> (Int, Int)
  split :: g -> (g, g)
  {-# MINIMAL next, split #-}
        -- Defined in 'System.Random'
instance RandomGen StdGen -- Defined in 'System.Random'
```

- From the information, `StdGen` type is an instance of `RandomGen`

Random Number Generator

- The `mkStdGen` function from `System.Random` takes an integer and returns a value of type `StdGen`

```
ghci> :t mkStdGen
mkStdGen :: Int -> StdGen
```

- Let's try a couple:

```
ghci> mkStdGen 1234
1235 1
ghci> mkStdGen 35129
35130 1
```

- Let's generate a couple of random numbers:

```
ghci> random (mkStdGen 1234)
(-4311284599041119727,636612013 2103410263)
ghci> random (mkStdGen 4321)
(-1031626213265511161,1554948163 2103410263)
```

- But if we try to generate a random number with the same argument, we get the same result (pure)

```
ghci> random (mkStdGen 1234)
(-4311284599041119727,636612013 2103410263)
ghci> random (mkStdGen 4321)
(-1031626213265511161,1554948163 2103410263)
```

Random Number Generator

- We can use type annotation to get different types:

```
ghci> random (mkStdGen 1234) :: (Int, StdGen)
(-4311284599041119727,636612013 2103410263)
ghci> random (mkStdGen 1234) :: (Integer, StdGen)
(-4311284599041119727,636612013 2103410263)
ghci> random (mkStdGen 1234) :: (Float, StdGen)
(0.44221336,1698564100 1655838864)
ghci> random (mkStdGen 1234) :: (Double, StdGen)
(0.3512572432712791 6,636612013 2103410263)
ghci> random (mkStdGen 1234) :: (Bool, StdGen)
(True,49417290 40692)
ghci> random (mkStdGen 1234) :: (Char, StdGen)
('\355669',49417290 40692)
```

Random with Range

- The `randomR` takes a range `(low, high)` and a `RandomGen` and returns a random value **uniformly** distributed in the closed interval `[low, high]`

```
ghci> :t randomR
randomR :: (Random a, RandomGen g) => (a, a) -> g -> (a, g)
```

- Examples:

```
ghci> randomR (1,6) (mkStdGen 324134)
(6,85036512 40692)
ghci> randomR (1,6) (mkStdGen 20983450298345)
(3,1396743909 40692)
```

- The `randomRs` generates an infinite list of random range:

```
ghci> :t randomRs
randomRs :: (Random a, RandomGen g) => (a, a) -> g -> [a]
```

- Example:

```
ghci> take 6 $ randomRs (1,6) (mkStdGen 324134)
[6,6,3,1,2,1]
ghci> take 10 $ randomRs (1,10) (mkStdGen 20983450298345)
[7,9,1,4,6,6,2,1,3,10]
```

Random and I/O

- So far, we need to manually pick the first number to generate the first generator
  - If we use the same generator, we get the same sequence of random numbers
  - Unfortunately, this is not random

```
ghci> :t randoms
randoms :: (Random a, RandomGen g) => g -> [a]
```

- The `getStdGen` function asks the system for a random number generator
- Consider the following program (`random01.hs`):

```
import System.Random

main = do
    stdgen <- getStdGen
    print stdgen
```

- If you run this program twice, you get two different generators:

```
$ ./random01
140756677 1
$ ./random01
1790449925 1
```

- Problem: Let's consider the following program (`random02.hs`):

```
import System.Random

main = do
    stdgen <- getStdGen
    print stdgen
    stdgen <- getStdGen
    print stdgen
```

- Here is the output:

```
$ ./random02
95243183 1
95243183 1
```

- Calling `getStdGen` multiple times in the same program results in the same generator

Exercise

- 

Binary?

- 

- 

-