File I/O Examples (II)

- Read character from one file, and write uppercase version to another

```
fCopyRaiseChar :: FilePath -> FilePath -> IO ()
fCopyRaiseChar fromf tof
 = do ff <- openFile fromf ReadMode
      c <- hGetChar ff
      hClose ff
      tf <- openFile tof WriteMode
      hPutChar tf (toUpper c)
      hClose tf
```

More do-notation

- Anywhere we invoke an action we can put a general Haskell expression, provided it evaluates to an I/O-action
  e.g. `if cond then actionIfTrue else actionIfFalse`
- We can define local value using a special form of `let`-expression:
  e.g. `let v = any-expression`
  - Note there is no `in` keyword at the end
  - `any-expression` need not be an I/O-action

File I/O Examples (III)

- Copy *all* characters from one file, writing uppercase versions to another

```
fCopyAllChars :: FilePath -> FilePath -> IO ()
fCopyAllChars fromf tof
 = do ff <- openFile fromf ReadMode
      str <- readWholeFile ff
      hClose ff
      tf <- openFile tof WriteMode
      let ustr = map toUpper str
      writeWholeFile tf (ustr)
      hClose tf
```

- The two utilities, `readWholeFile` and `writeWholeFile` are on the next slides

File I/O Examples (IV)

- These two utility functions read and write a whole file at once.
-
```
writeWholeFile :: Handle -> String -> IO ()
writeWholeFile _ [] = return ()
writeWholeFile h (x:xs) = do
  hPutChar h x
  writeWholeFile h xs
```

-
```
readWholeFile :: Handle -> IO String
readWholeFile h
 = do eof <- hIsEOF h
      if eof then return []
             else do c <- hGetChar h
                     str <- readWholeFile h
                     return (c:str)
```

Tail Recursion and I/O

- Recursively-defined functions doing I/O should really be tail-recursive.
- On the previous slide, `writeWholeFile` is, but `readWholeFile` isn't.

File I/O Examples (V)

- ```
  readWholeFile :: Handle -> IO String
  readWholeFile h = getWholeFile h []
  ```
- ```
  getWholeFile :: Handle -> String -> IO String
  getWholeFile h cs
   = do eof <- hIsEOF h
        if eof then return $ reverse cs
               else do c <- hGetChar h
                       getWholeFile h (c:cs)
  ```

File I/O Examples (VI)

- The Prelude has the following two functions:

  ```
  readFile :: FilePath -> IO String
  writeFile :: FilePath -> String -> IO ()
  ```
- So our program can in fact be written as

  ```
  fCopyAllChars :: FilePath -> FilePath -> IO ()
  fCopyAllChars fromf tof
   = do str <- readFile fromf
        writeFile tof (map toUpper str)
  ```

Escape ?

- How do we "get out" of I/O actions ?
- i.e. is there a function with type `IO a -> a` ?
- So I could write

  ```
  myfun :: Int -> Int
  myfun x = (ioescape getnum) + x
   where
     getnum :: IO Int
     getnum = do f <- openFile "y.dat" ReadMode
                 c <- hGetChar f
                 hClose f
                 return (ord c)
  ```

  where `ioescape` was this special function.

Escape ! But . . .

- No there isn't . . .
- Well, actually there is:
  `unsafePerformIO`
- This is unsafe because bad use of this will break referential transparency.
- i.e. a Haskell program using it is *impure*.
- for use as a "backdoor" by experts

REPL Code

- A common programming style is the so-called REPL idiom (Read-Eval/Execute-Print-Loop)
- We can do this in Haskell
  using `putStr` and `putStrLn` for output
  using `getLine` for input
- A simple dumb program that shouts back at you:

```
shout = do putStr "Say something: "
           utterance <- getLine
           putStrLn ("You said : "++map toUpper utterance)
           if null utterance then return () else shout
```

REPL Code

- Slighty less dumb — it checks for no utterance first.

```
shout2
 = do putStr "Say something: "
      said <- getLine
      if null said
      then putStrLn "I CAN'T HEAR YOU! I'M OFF !!"
      else do putStrLn ("You said : "++map toUpper said)
              shout2
```

R-Eval-PL Template

There is a common pattern to most RE(eval)PL programs:

- Issue a prompt
- Get user input
- Evaluate user input
- Print result
- Look at result and decide either to continue, or exit

We can capture this as the following code

```
revpl prompt eval print done
 = do putStr prompt
      userinp <- getLine
      let result = eval userinp
      print result
      if done result
       then return ()
       else revpl prompt eval print done
```

Using R-Eval-PL Template

We can now write

```
shout1 = revpl "Say something :"
               (map toUpper)
               print1
               null

print1 res = putStrLn ("You said : "++res)
```

R-Execute-PL Template

There is another common pattern to most RE(execute)PL programs:

- Issue a prompt
- Get user input
- Parse input and perform requested action
- Look at outcome and decide either to continue, or exit

We can capture this as the following code

```
rexpl prompt execute
 = do putStr prompt
      usercmd <- getLine
      done <- execute usercmd
      if done then return ()
              else rexpl prompt execute
```

Using R-Execute-PL Template

We can now write

```
shout3 = rexpl "Say something :"  doshout3
doshout3 utt
  = if null utt
    then do putStrLn "I CAN'T HEAR YOU! I'M OFF !!"
            return True
    else do putStrLn ("You said : "++map toUpper utt)
            return False
```

These examples show how easy it is to "grow our own" control structures.

Most REPLs need state

- Consider implementing a simple "Totting-up" program:
  - User enters numbers one at a time
  - These are added up, and a running total is displayed
  - An empty line terminates the process
- This cannot be implemented using `revpl` or `rexpl` Try it!
- Given that there is a state being updated (here the running total), it makes sense to view this as being a R-Execute-PL, rather than Eval.

Totting-Up REPL

- We need to initialise the running total:

```
totup = dototting 0.0
```

- We then implement the REPL loop, passing the total (state) in as an argument

```
dototting tot
 = do putStr("["++show tot++"]\n:- ")
      numtxt <- getLine
      if null numtxt
       then putStrLn ("\nTotal = "++show tot)
       else dototting (tot+read numtxt) -- state update!
```

- We use `read :: Read a => String -> a` here, which has numeric instances.
- Again, we can build a HOF that abstracts this pattern

State REPL

- State REPL builder:

```
srepl prompt done exit execute state
 = do prompt state
      cmd <- getLine
      if done cmd
       then exit state
       else
        let state' = execute cmd state
        in srepl prompt done exit execute state'
```

- Haskell derives the following type, where `t` denotes the state type:

```
srepl :: (t -> IO a)          -- prompt
      -> (String -> Bool)     -- done
      -> (t -> IO b)          -- exit
      -> (String -> t -> t)   -- execute
      -> t                    -- state
      -> IO b
```

"Totting-Up" using `srepl`

We can focus on the four key processing steps: prompting,

```
totpr tot = putStr("["++show tot++"]\n:- ")
```

checking if done,

```
null
```

exiting cleanly,

```
totxit tot = putStrLn ("\nTotal = "++show tot)
```

and computing the next state

```
totexe cmd tot = tot+read cmd
```

We then invoke the REPL-generator with these and the starting state:

```
totup2 = srepl totpr null totxit totexe 0.0
```

REPL with `putStr` and `getLine`

- Building REPL code using `getLine :: IO String` is very convenient
- Unfortunately, keys such as delete or backspace are not handled properly (on Unix-based systems at least — it seems to work fine on Windows!).
- There are modules that help
  - Best is probably: `System.Console.Haskeline`
    Careful: uses monad transformers
  - An alternative `System.Console.Readline`
    Interfaces to GNU readline, but has restricted portability