Case study

General format (observe layout!):

\[
\begin{align*}
\text{do} & \quad a_1 \\
& \quad \vdots \\
& \quad a_n
\end{align*}
\]

where each \( a_i \) can be one of

- an action
  - Effect: execute action
- \( x \leftarrow \text{action} \)
  - Effect: execute \text{action} :: 10 a, give result the name \( x :: a \)
- \text{let } x = expr
  - Effect: give \( expr \) the name \( x \)
  - Lazy: \( expr \) is only evaluated when \( x \) is needed!
Word: -a--
Missed: eg

Word: -a--
Missed: egsm

Word: -a--
Missed: egsmk
k
Word: -a--
Missed: egsmtky
YOU ARE DEAD: jazz
Input secret word: _

main :: IO ()
main = do putStrLn "Input secret word: "

main :: IO ()
main = do putStrLn "Input secret word: "
          word <- getWord ""
          clear_screen
          guess word
guess :: String -> IO ()
guess word = loop "" "" gallows
  loop :: String -> String -> [String] -> IO()
  loop guessed missed gals =

guess :: String -> IO ()
guess word = loop "" "" gallows where
  loop :: String -> String -> [String] -> IO()
  loop guessed missed gals =
    do let word' =
      map (\x -> if x `elem` guessed
        then x else '-')
        word
guess :: String -> IO ()
guess word = loop "" "" gallows where
  loop :: String -> String -> [String] -> IO()
  loop guessed missed gals =
    do let word' =
      map (\x -> if x 'elem' guessed
               then x else ' -')
        word
    writeAt (1,1)
      (head gals ++ "\n"
       ++ "Word: " ++ word' ++
guess :: String -> IO ()
guess word = loop "" "" gallows where
  loop :: String -> String -> [String] -> IO()
  loop guessed missed gals =
    do let word' =
      map (\x -> if x 'elem' guessed
             then x else '‐'
               )
         word
           writeAt (1,1)
           (head gals ++ "\n" ++ "Word: " ++ word' ++
             "\nMissed: " ++ missed ++ "\n")
       if length gals == 1
       then putStrLn ("YOU ARE DEAD: " ++ word)
       else if word' == word then putStrLn "YOU WIN!"
guess :: String -> IO ()
guess word = loop "" "" gallows where
    loop :: String -> String -> [String] -> IO()
    loop guessed missed gals =
        do let word' =
            map (\x -> if x 'elem' guessed
                    then x else ' - ')
                word
            writeAt (1,1)
                (head gals ++ "\n" ++ "Word: " ++ word' ++
                 "\nMissed: " ++ missed ++ "\n")
        if length gals == 1
            then putStrLn ("YOU ARE DEAD: " ++ word)
        else if word' == word then putStrLn "YOU WIN!"
        else do c <- getChar
                    let ok = c 'elem' word
                    loop (if ok then c:guessed else guessed)
                      (if ok then missed else missed++[c])
You cannot add I/O to a function without giving it an IO type

For example

\[
\begin{align*}
\text{sq} & \quad : \quad \text{Int} \to \text{Int} \\
\text{sq} \ x & = \ x \times x \\
\text{cube} & \quad : \quad \text{Int} \to \text{Int} \\
\text{cube} \ x & = \ x \times \text{sq} \ x
\end{align*}
\]

Let us try to make \text{sq} print out some message:

\[
\begin{align*}
\text{sq} \ x & = \ \text{do} \ \text{putStr}("I \ am \ in \ sq!") \\
& \quad \ \text{return}(x \times x)
\end{align*}
\]

What is the type of \text{sq} now?

\text{Int} \to \text{IO Int}
Haskell is a pure functional language
Functions that have side effects must show this in their type
I/O is a side effect

Separate I/O from processing to reduce I/O creep:

```
main :: IO ()
main = do s <- putStrLn
         let r = process s
         putStrLn r
         main
```

Separate I/O from processing to reduce I/O creep:

```
main :: IO ()
main = do s <- putStrLn
         let r = process s
         putStrLn r
         main

process :: String -> String
process s = ...
```
Separate I/O from processing to reduce IO creep:

```haskell
main :: IO ()
main = do 
  s <- getline
  let r = process s
  putStrLn r
  main

process :: String -> String
process s = ...
```

- `type FilePath = String`
- `type FilePath = String`
- `readFile :: FilePath -> IO String`
- type FilePath = String
- readFile :: FilePath -> IO String
  Reads file contents *lazily*,
  only as much as is needed
- writeFile :: FilePath -> String -> IO ()
  Writes whole file

- type FilePath = String
- readFile :: FilePath -> IO String
  Reads file contents *lazily*,
  only as much as is needed
- writeFile :: FilePath -> String -> IO ()
  Writes whole file
- appendFile :: FilePath -> String -> IO ()
The simple way

- `type FilePath = String`
- `readFile :: FilePath -> IO String`
  
  Reads file contents lazily, only as much as is needed

- `writeFile :: FilePath -> String -> IO ()`
  
  Writes whole file

- `appendFile :: FilePath -> String -> IO ()`
  
  Appends string to file

Handles

data Handle

Opaque type, implementation dependent
data Handle
Opaque type, implementation dependent

Haskell defines operations to read and write characters from and to files, represented by values of type Handle.

data IO

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

- openFile :: FilePath -> IOMode -> IO Handle
  Creates handle to file and opens file
• data IOMode = ReadMode \mid WriteMode
  \mid AppendMode \mid ReadWriteMode

• openFile :: FilePath \to IOMode \to IO Handle
  Creates handle to file and opens file

• hClose :: Handle \to IO ()
  Closes file

By convention
all IO actions that take a handle argument begin with h

• getChar :: IO Char
  Reads a Char from standard input, echoes it to standard output, and returns it as the result
• hGetChar :: Handle -> IO Char
• hGetLine :: Handle -> IO String

• hPutChar :: Handle -> Char -> IO ()
• hPutStr :: Handle -> String -> IO ()
• hPutStrLn :: Handle -> String -> IO ()
• stdin :: Handle
  stdout :: Handle

• stdin :: Handle
  stdout :: Handle
• getChar = hGetChar stdin
  putChar = hPutChar stdout

There is much more in the [Standard IO Library](#)