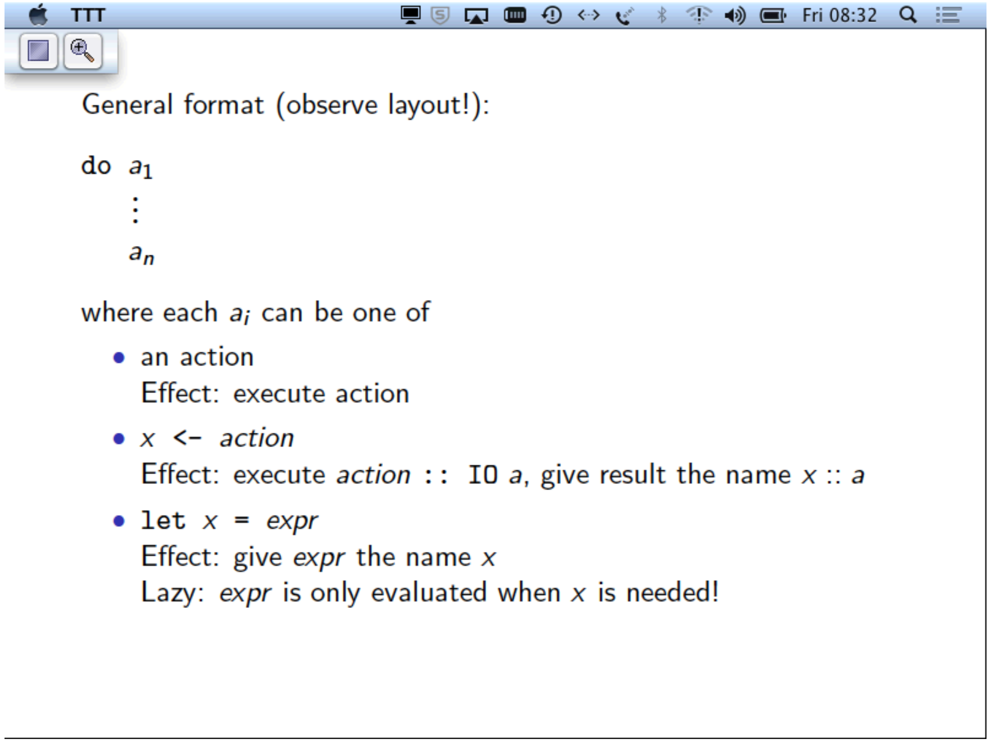
Script generated by TTT

Title: Nipkow: Info2 (12.12.2014)

Date: Fri Dec 12 08:31:03 CET 2014

Duration: 30:17 min

Pages: 62

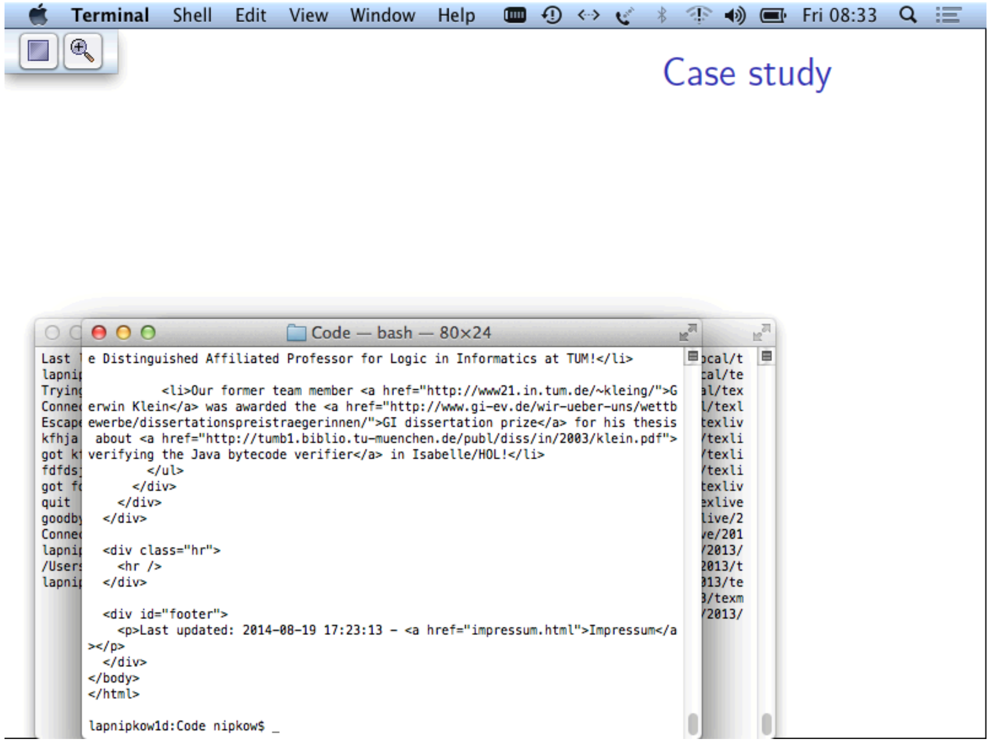


General format (observe layout!):

```
do a1
   ⋮
   an
```

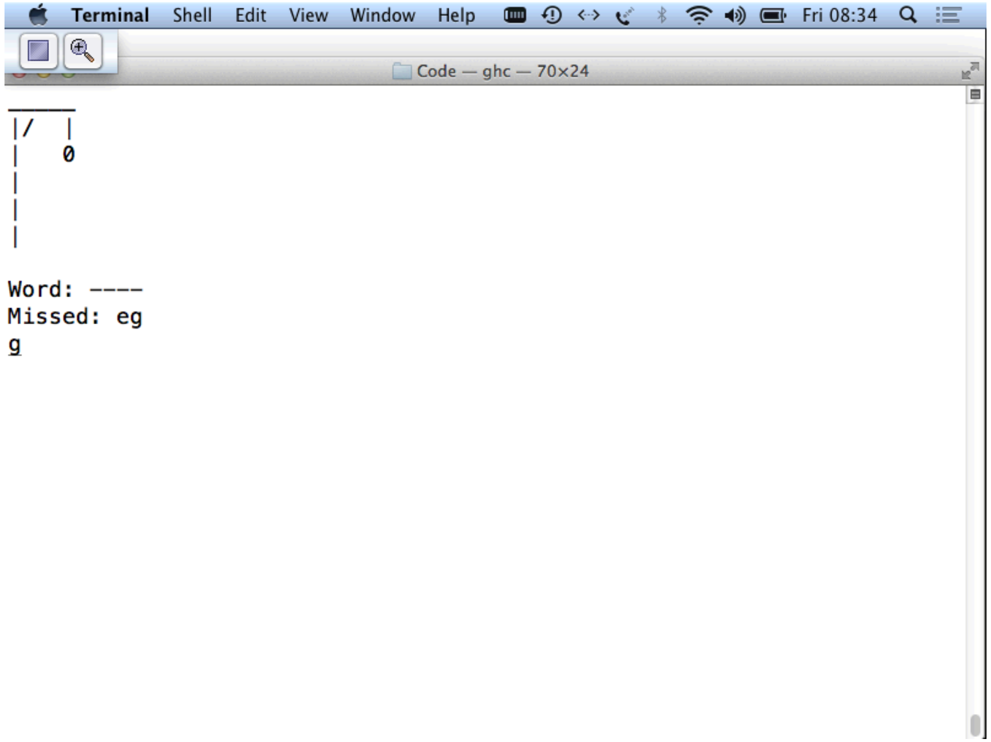
where each a_i can be one of

- an action
  Effect: execute action
- `x <- ` *action*
  Effect: execute *action* `:: IO a`, give result the name $x :: a$
- `let x = ` *expr*
  Effect: give *expr* the name x
  Lazy: *expr* is only evaluated when x is needed!

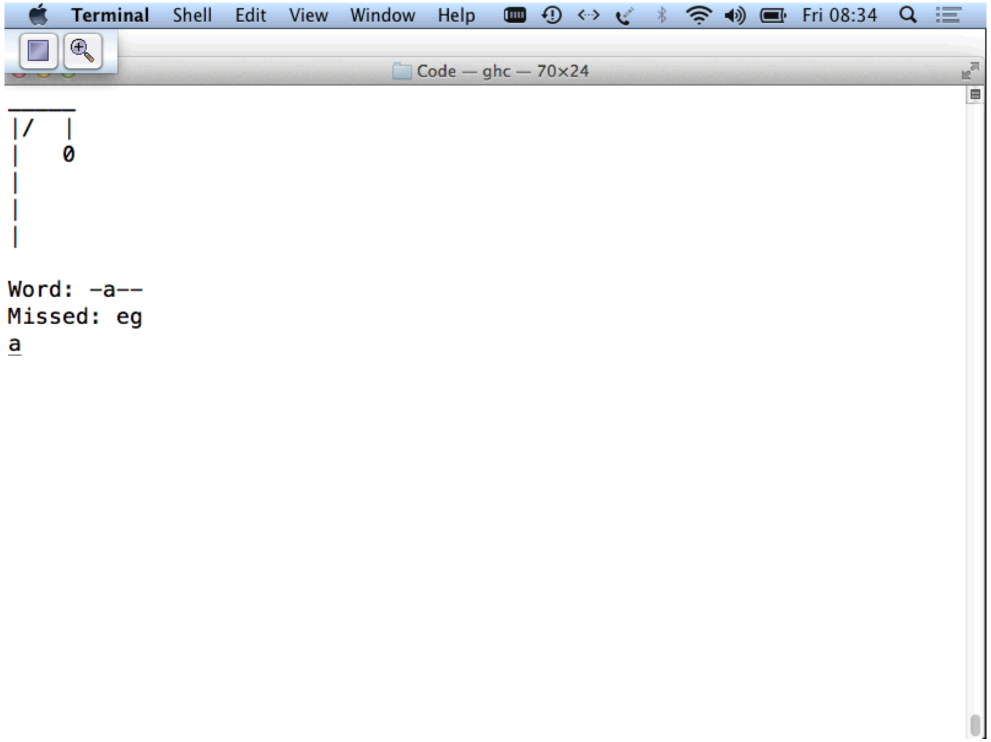


Case study

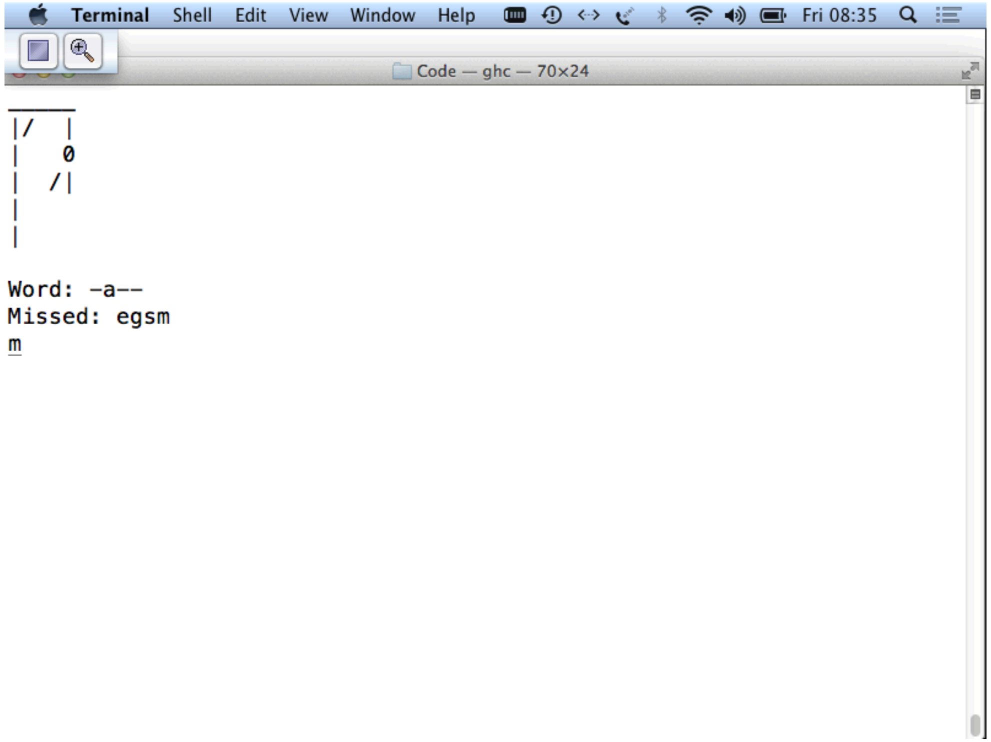
```
e Distinguished Affiliated Professor for Logic in Informatics at TUM!</li>

          <li>Our former team member <a href="http://www21.in.tum.de/~kleing/">G
erwin Klein</a> was awarded the <a href="http://www.gi-ev.de/wir-ueber-uns/wettb
ewerbe/dissertationspreistraegerinnen/">GI dissertation prize</a> for his thesis
 about <a href="http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2003/klein.pdf">
verifying the Java bytecode verifier</a> in Isabelle/HOL!</li>
        </ul>
      </div>
    </div>
  </div>

  <div class="hr">
    <hr />
  </div>

  <div id="footer">
    <p>Last updated: 2014-08-19 17:23:13 - <a href="impressum.html">Impressum</a
></p>
  </div>
</body>
</html>

lapnipkow1d:Code nipkow$ _
```



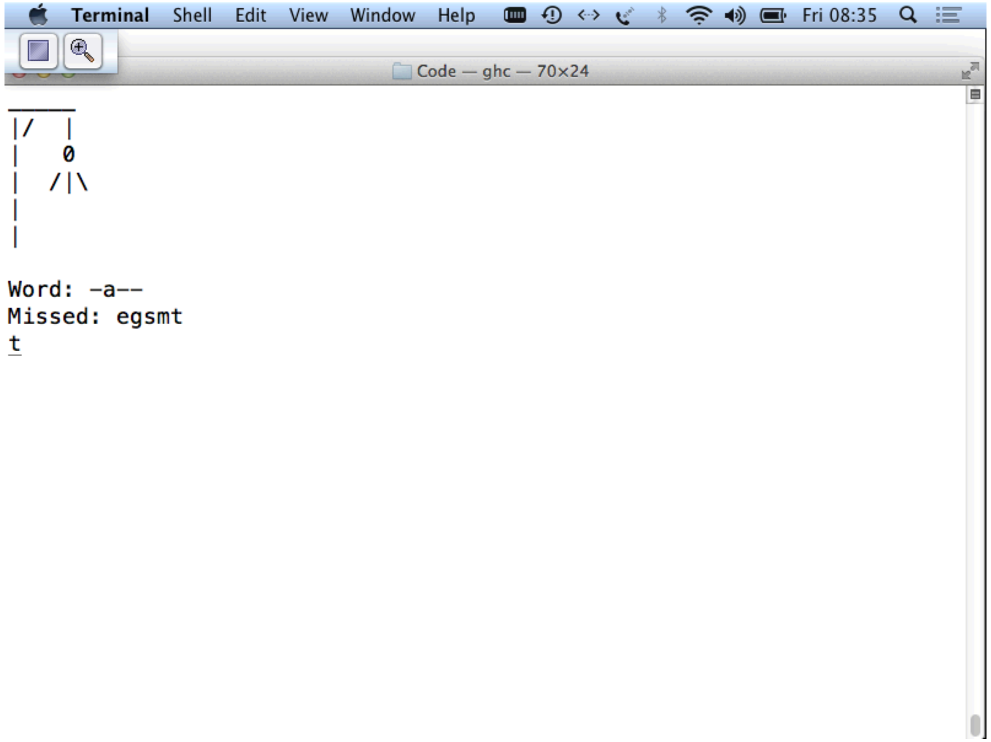
```
 ____
|/  |
|   0
|
|
|

Word: ----
Missed: eg
g
```



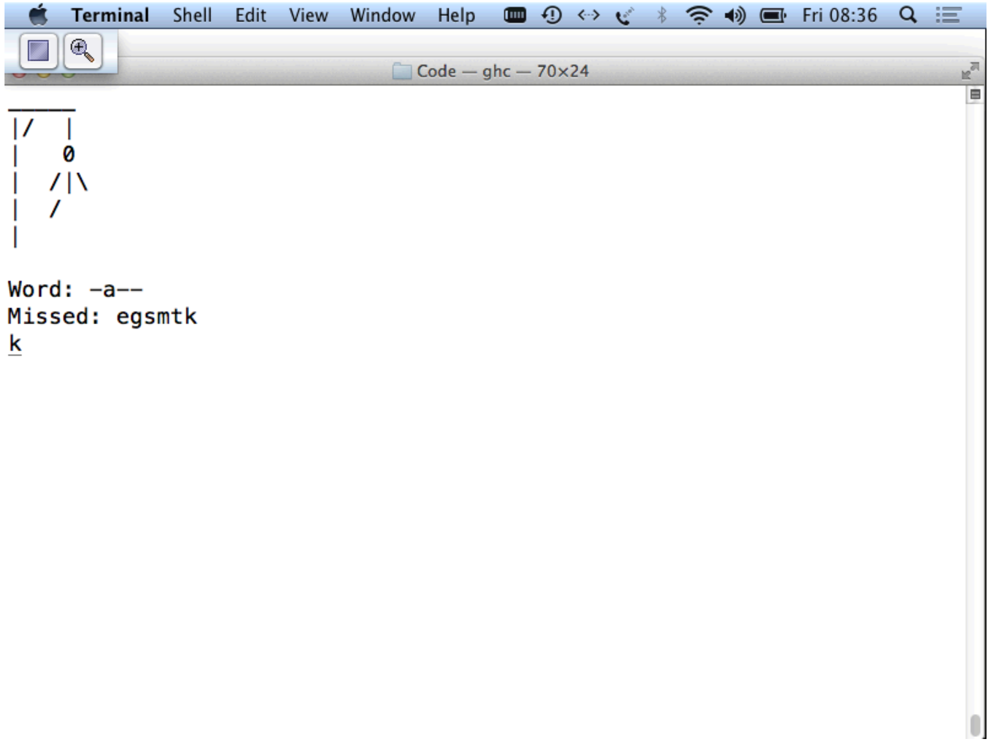
```
_____
|/  |
|   0
|
|
|

Word: -a--
Missed: eg
a
```



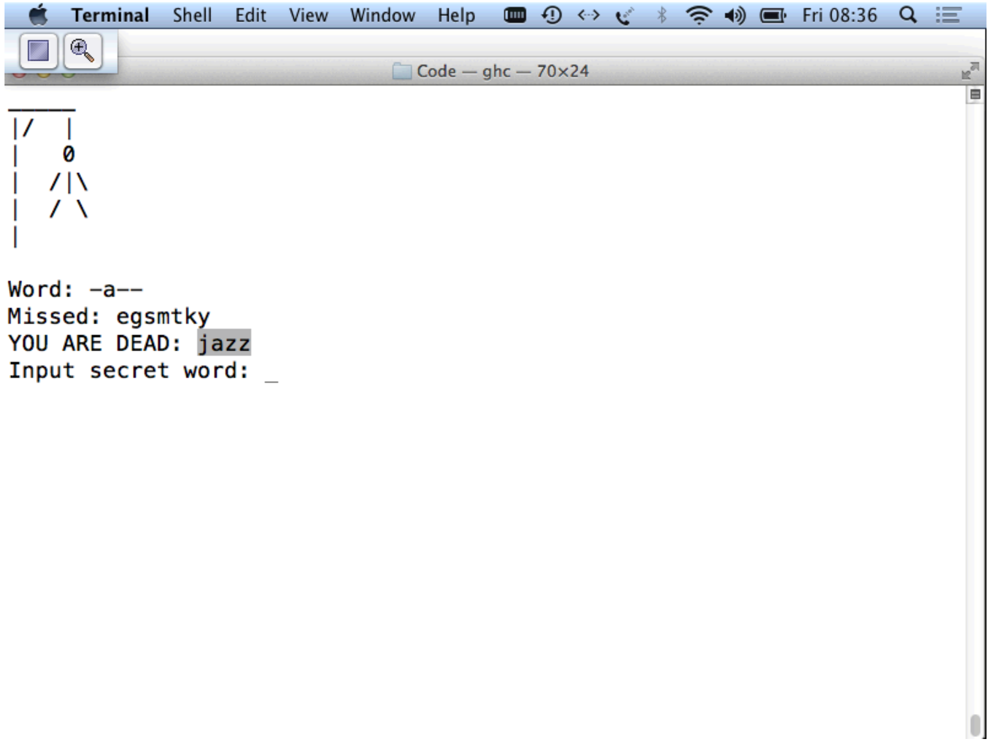
```
_____
|/  |
|   0
|  /|
|
|

Word: -a--
Missed: egsm
m
```



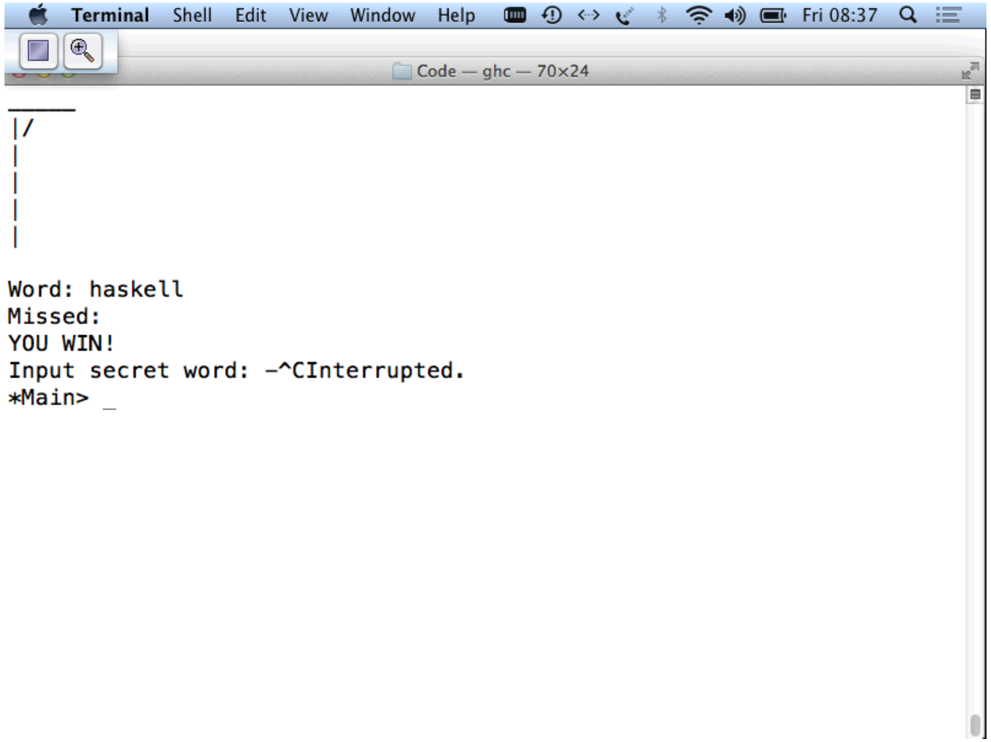
```
_____
|/  |
|   0
|  /|\
|
|

Word: -a--
Missed: egsmt
t
```



```
_____
|/  |
|   0
|  /|\
|  /
|

Word: -a--
Missed: egsmtk
k
```



```
______
|/   |
|    0
|   /|\
|   / \
|

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



```
______
|/
|
|
|
|

Word: haskell
Missed:
YOU WIN!
Input secret word: -^CInterrupted.
*Main> _
```

```
main :: IO ()
main = do putStr "Input secret word: "
```

```
main :: IO ()
main = do putStr "Input secret word: "
          word <- getWord ""
          clear_screen
          guess word
guess :: String -> IO ()
guess word = loop "" "" gallows
```

```
guess :: String -> IO ()
guess word = loop "" "" gallows  where
  loop :: String -> String -> [String] -> IO()
```

```
guess :: String -> IO ()
guess word = loop "" "" gallows  where
  loop :: String -> String -> [String] -> IO()
  loop guessed missed gals =
```

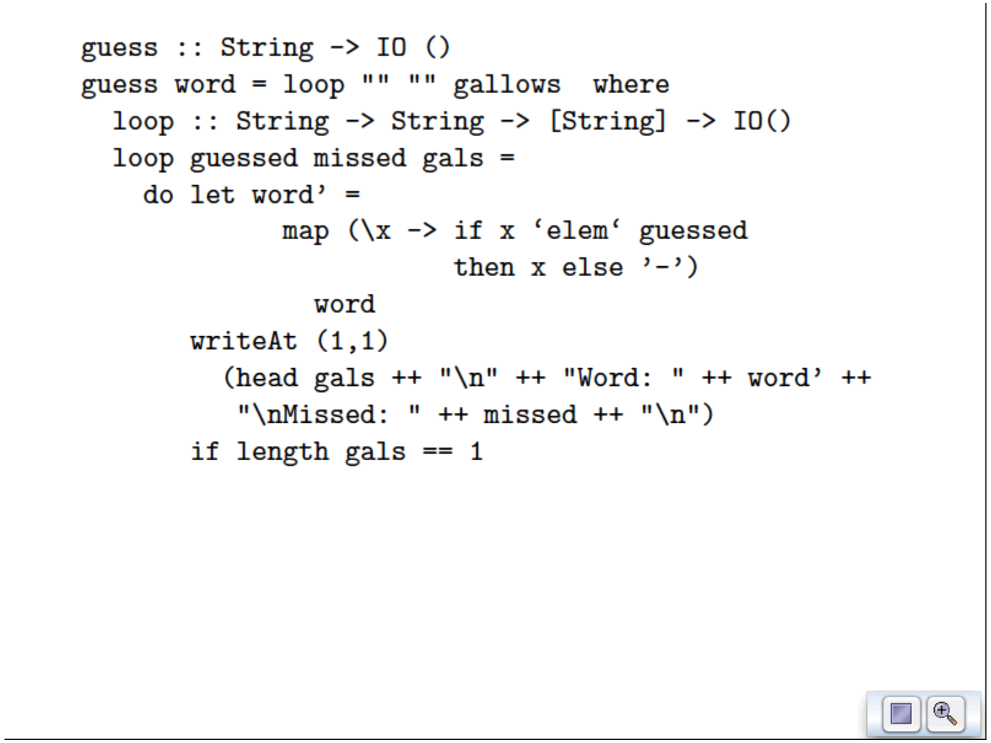
```
guess :: String -> IO ()
guess word = loop "" "" gallows  where
  loop :: String -> String -> [String] -> IO()
  loop guessed missed gals =
    do let word’ =
             map (\x -> if x ‘elem‘ guessed
                        then x else ’-’)
               word
```

```
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)
```

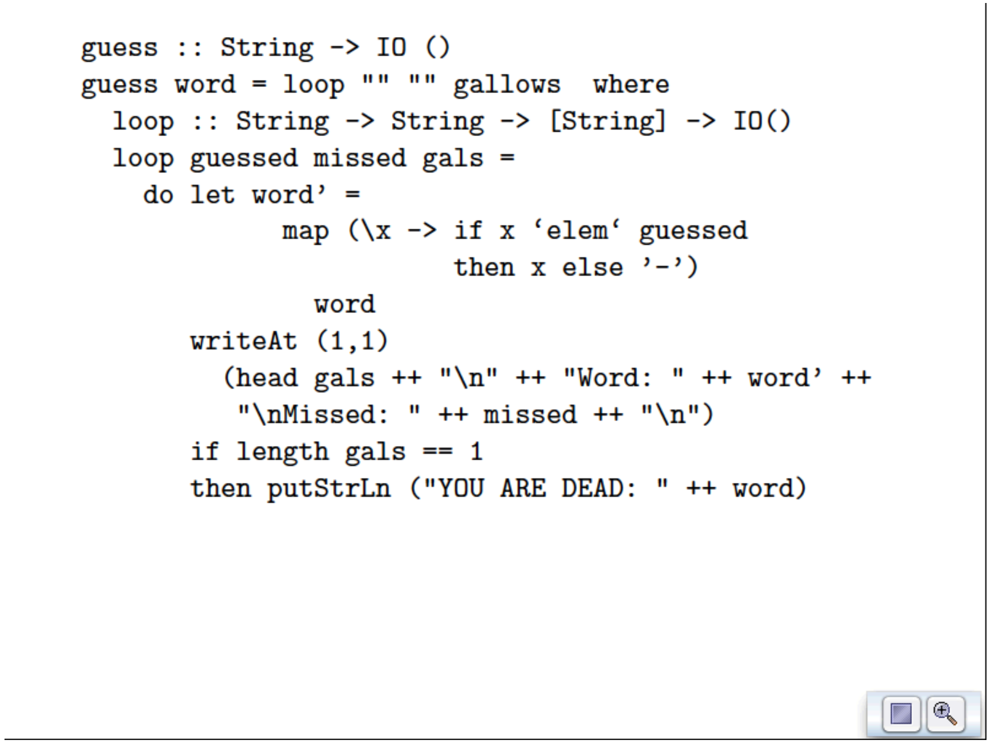
```
guess :: String -> IO ()
guess word = loop "" "" gallows  where
  loop :: String -> String -> [String] -> IO()
  loop guessed missed gals =
    do let word’ =
             map (\x -> if x ‘elem‘ guessed
                        then x else ’-’)
               word
       writeAt (1,1)
         (head gals ++ "\n"
```

```
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’ ++
```

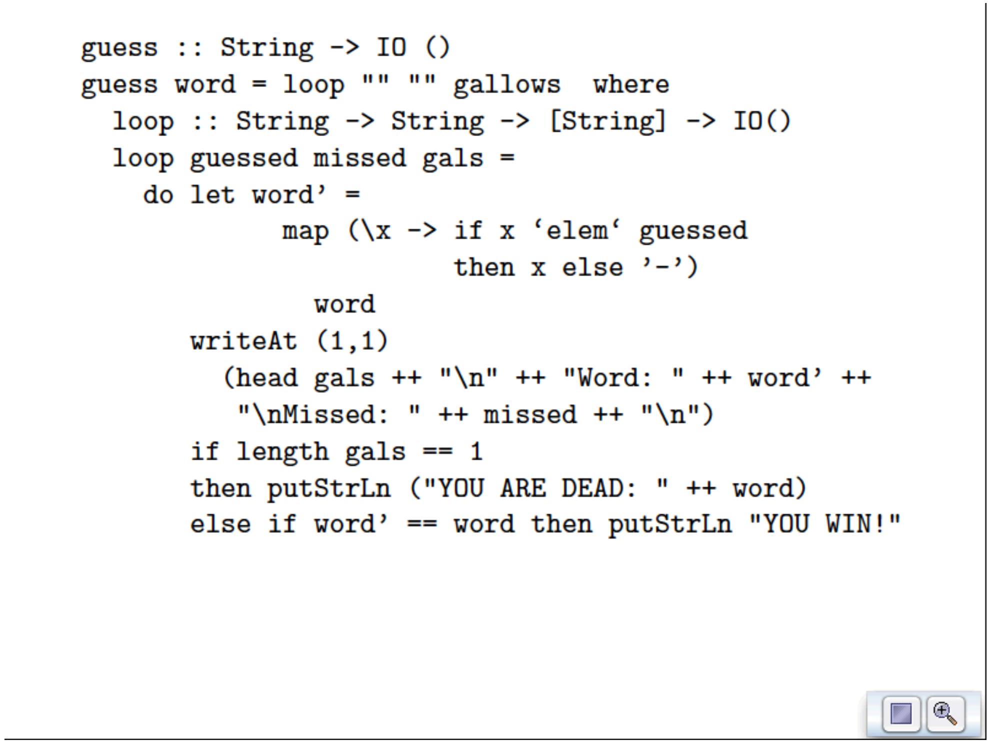
```
guess :: String -> IO ()
guess word = loop "" "" gallows  where
  loop :: String -> String -> [String] -> IO()
  loop guessed missed gals =
    do let word’ =
             map (\x -> if x ‘elem‘ guessed
                        then x else ’-’)
               word
       writeAt (1,1)
         (head gals ++ "\n" ++ "Word: " ++ word’ ++
          "\nMissed: " ++ missed ++ "\n")
```



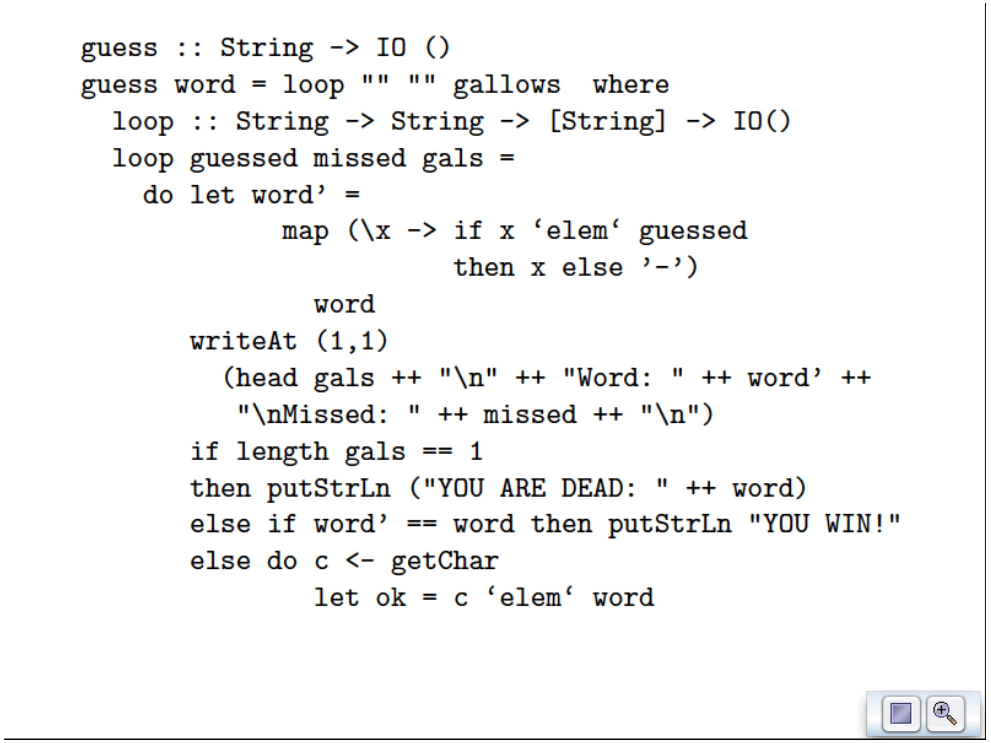
```
guess :: String -> IO ()
guess word = loop "" "" gallows  where
  loop :: String -> String -> [String] -> IO()
  loop guessed missed gals =
    do let word’ =
             map (\x -> if x ‘elem‘ guessed
                        then x else ’-’)
               word
       writeAt (1,1)
         (head gals ++ "\n" ++ "Word: " ++ word’ ++
          "\nMissed: " ++ missed ++ "\n")
       if length gals == 1
```



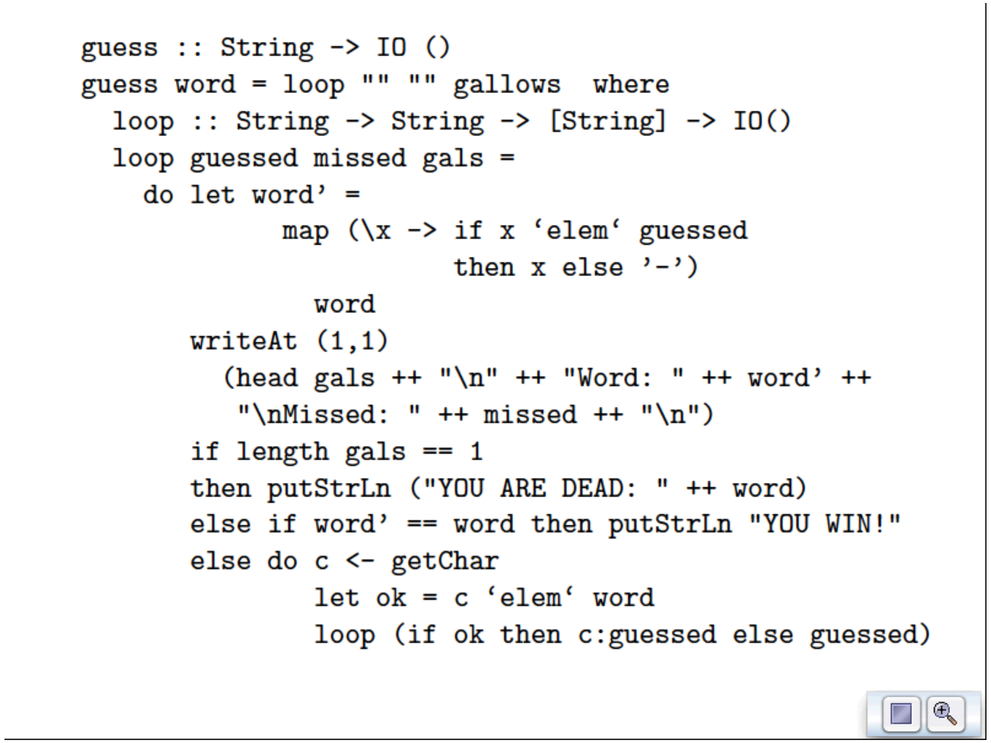
```
guess :: String -> IO ()
guess word = loop "" "" gallows  where
  loop :: String -> String -> [String] -> IO()
  loop guessed missed gals =
    do let word’ =
             map (\x -> if x ‘elem‘ guessed
                        then x else ’-’)
               word
       writeAt (1,1)
         (head gals ++ "\n" ++ "Word: " ++ word’ ++
          "\nMissed: " ++ missed ++ "\n")
       if length gals == 1
       then putStrLn ("YOU ARE DEAD: " ++ word)
```



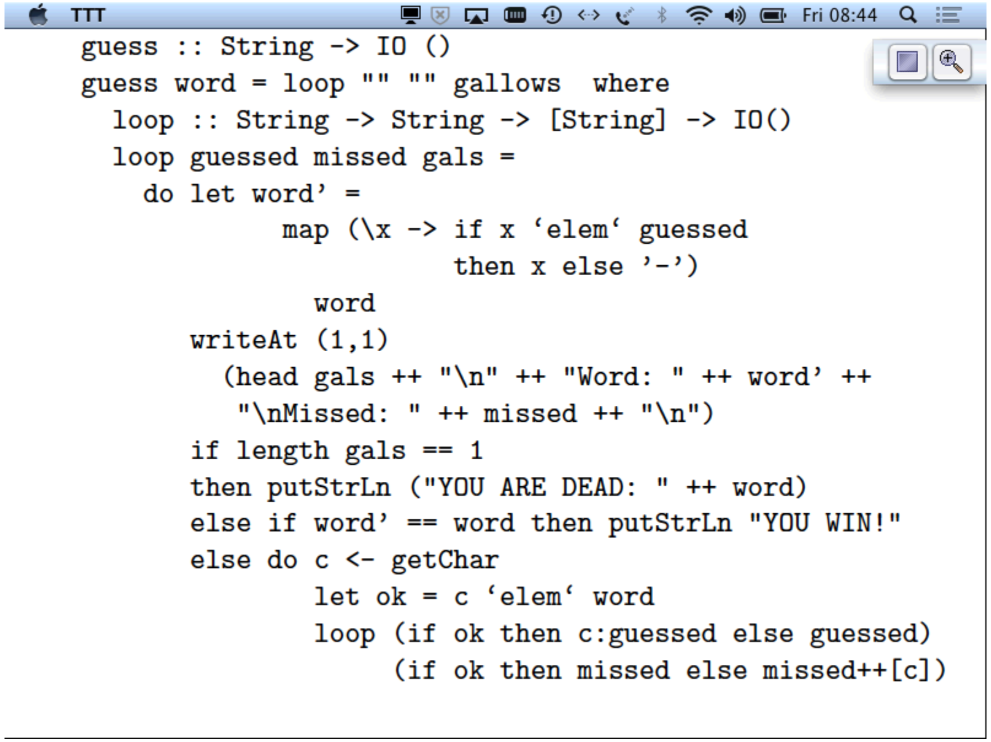
```
guess :: String -> IO ()
guess word = loop "" "" gallows  where
  loop :: String -> String -> [String] -> IO()
  loop guessed missed gals =
    do let word’ =
             map (\x -> if x ‘elem‘ guessed
                        then x else ’-’)
               word
       writeAt (1,1)
         (head 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!"
```



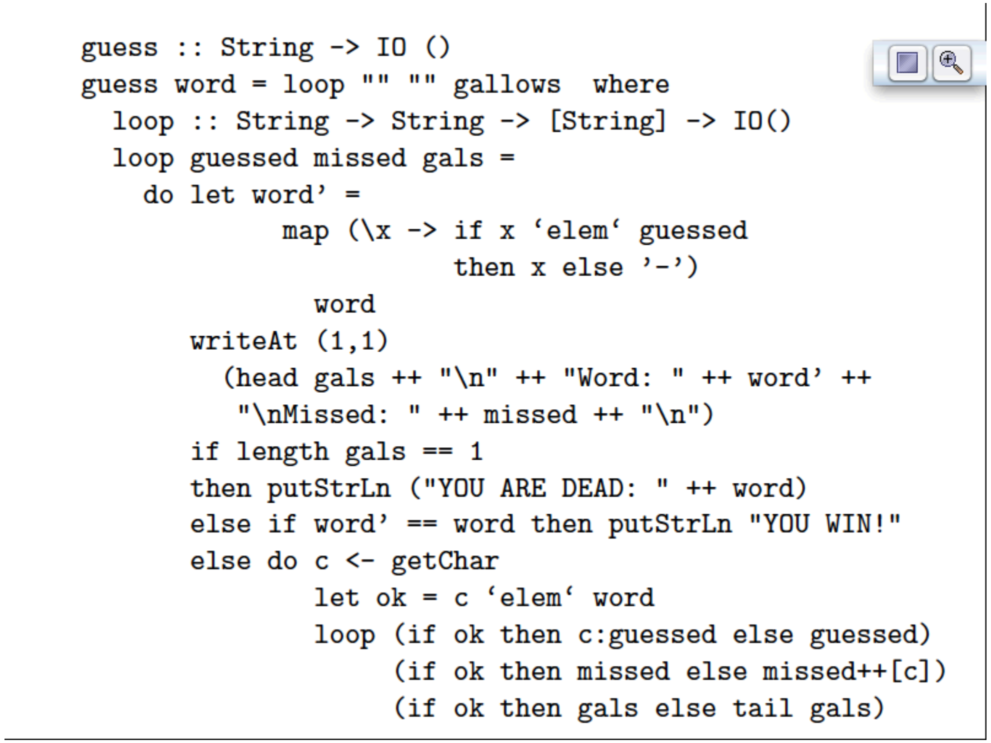
```
guess :: String -> IO ()
guess word = loop "" "" gallows  where
  loop :: String -> String -> [String] -> IO()
  loop guessed missed gals =
    do let word’ =
             map (\x -> if x ‘elem‘ guessed
                        then x else ’-’)
               word
       writeAt (1,1)
         (head 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
```



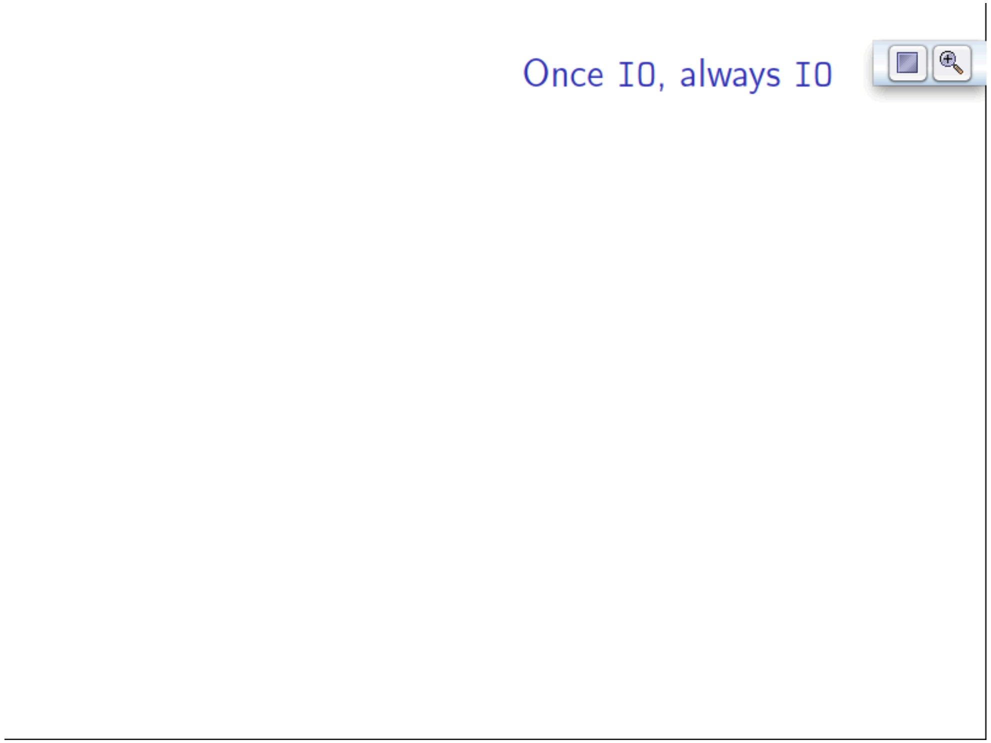
```
guess :: String -> IO ()
guess word = loop "" "" gallows  where
  loop :: String -> String -> [String] -> IO()
  loop guessed missed gals =
    do let word' =
             map (\x -> if x `elem` guessed
                        then x else '-')
               word
       writeAt (1,1)
         (head 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)
```



```
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])
```



```
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])
                    (if ok then gals else tail gals)
```



Once IO, always IO

Once IO, always IO

You cannot add I/O to a function without giving it an IO type

Once IO, always IO

You cannot add I/O to a function without giving it an IO type

For example

```
sq :: Int -> Int        cube :: Int -> Int
sq x = x*x              cube x = x * sq x
```

Once IO, always IO

You cannot add I/O to a function without giving it an IO type

For example

```
sq :: Int -> Int        cube :: Int -> Int
sq x = x*x              cube x = x * sq x
```

Let us try to make sq print out some message:

```
sq x = do putStr("I am in sq!")
          return(x*x)
```

What is the type of sq now?

Once IO, always IO

You cannot add I/O to a function without giving it an IO type

For example

```
sq :: Int -> Int        cube :: Int -> Int
sq x = x*x              cube x = x * sq x
```

Let us try to make sq print out some message:

```
sq x = do putStr("I am in sq!")
          return(x*x)
```

What is the type of sq now? `Int -> 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 `IO` creep:

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

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

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

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

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

Separate I/O from processing to reduce IO creep:

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

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

The simple way

The simple way

- `type FilePath = String`

The simple way

- `type FilePath = String`
- `readFile :: FilePath -> IO String`

The simple way

- `type FilePath = String`
- `readFile :: FilePath -> IO String`

  Reads file contents *lazily*,

The simple way

- `type FilePath = String`
- `readFile :: FilePath -> IO String`

  Reads file contents *lazily*,
  only as much as is needed

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

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 ()`

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

```
import System.IO
```

Handles

```
data Handle
```

Handles

```
data Handle
```

Opaque type, implementation dependent

Handles

```
data Handle
```

Opaque type, implementation dependent

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

Handles

```
data Handle
```

Opaque type, implementation dependent

> *Haskell defines operations to read and write characters from and to files, represented by values of type* `Handle`. *Each value of this type is a handle: a record used by the Haskell run-time system to manage I/O with file system objects.*

Files and handles

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

Files and handles

-
  ```
  data IOMode = ReadMode | WriteMode
              | AppendMode | ReadWriteMode
  ```
- `openFile :: FilePath -> IOMode -> IO Handle`

  Creates handle to file and opens file

Files and handles

- `data IOMode = ReadMode | WriteMode`
  `             | AppendMode | ReadWriteMode`
- `openFile :: FilePath -> IOMode -> IO Handle`

  Creates handle to file and opens file
- `hClose :: Handle -> IO ()`

Files and handles

- `data IOMode = ReadMode | WriteMode`
  `             | AppendMode | ReadWriteMode`
- `openFile :: FilePath -> IOMode -> IO Handle`

  Creates handle to file and opens file
- `hClose :: Handle -> IO ()`

  Closes file

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

Basic actions

- `getChar :: IO Char`

  Reads a `Char` from standard input,
  echoes it to standard output,
  and returns it as the result

In ReadMode

- `hGetChar :: Handle -> IO Char`
- `hGetLine :: Handle -> IO String`

In ReadMode

- `hGetChar :: Handle -> IO Char`
- `hGetLine :: Handle -> IO String`
- `hGetContents :: Handle -> IO String`

  Reads the whole file *lazily*

In WriteMode

- `hPutChar :: Handle -> Char -> IO ()`
- `hPutStr :: Handle -> String -> IO ()`
- `hPutStrLn :: Handle -> String -> IO ()`

In WriteMode

- `hPutChar :: Handle -> Char -> IO ()`
- `hPutStr :: Handle -> String -> IO ()`
- `hPutStrLn :: Handle -> String -> IO ()`
- `hPrint :: Show a => Handle -> a -> IO ()`

stdin and stdout

-
  ```
  stdin  :: Handle
  stdout :: Handle
  ```

stdin and stdout

-
  ```
  stdin  :: Handle
  stdout :: Handle
  ```

-
  ```
  getChar = hGetChar stdin
  putChar = hPutChar stdout
  ```

There is much more in the Standard IO Library