Title: Nipkow: Info2 (17.12.2013)
Date: Tue Dec 17 15:34:31 CET 2013
Duration: 67:28 min
Pages: 123

Code:
```
/
/
/
/
/

Word: -------
Missed:
```
```
/
/
/
/
/

Word: -a-----
Missed: a
```
Word: ha--ell
Missed: yzbmit

t

Word: haskell
Missed: yzbmit
YOU WIN!
Input secret word: _

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

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

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

(head gallops ++ "\n" ++ "Word: " ++ word' ++
  "\nMissed: " ++ missed ++ "\n")
if length gallops == 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)
    else if word' == word then putStrLn "YOU WIN!"
      else do c <- getChar
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)

Once IO, always IO
Once I0, always I0

You cannot add I/O to a function without giving it an I0 type

For example

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

Let us try to make sq print out some message:

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

What is the type of sq now? \(\text{Int} \to \text{IO} \text{Int}\)
Once IO, always IO

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

For example

```haskell
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:

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

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

And this is what happens to `cube`:

```haskell
cube x = do x2 <- sq x
    return(x * x2)
```

Haskell is a pure functional language

Separate I/O from processing to reduce IO creep:

```haskell
main :: IO ()
main = do s <- getLine
    let r = process s
    putStrLn r
    main
```
Separate I/O from processing to reduce I/O creep:

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

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

The simple way

- type FilePath = String
- readFile :: FilePath -> IO String
The simple way

- `type FilePath = String`
- `readFile :: FilePath -> IO String`
  
  Reads file contents *lazily*,

- `writeFile :: 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 ()`

Data Handles

- data Handle

- `appendFile :: FilePath -> String -> IO ()`
Handles

```haskell
data Handle
Opaque type, implementation dependent

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

```haskell
import System.IO
```

Files and handles

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

Handles

```haskell
data Handle
Opaque type, implementation dependent
```
Files and handles

- data IOMode = ReadMode | WriteMode | AppendMode | ReadWriteMode
- openFile :: FilePath -> IOMode -> IO Handle
  Creates handle to file and opens file
- hClose :: Handle -> IO ()

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

In ReadMode

- hGetChar :: Handle -> IO Char

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 ()`
- `hPrint :: Show a => Handle -> a -> IO ()`

stdin and stdout

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

- `getChar = hGetChar stdin`
- `putChar = hPutChar stdout`
There is much more in the **Standard IO Library**

(including exception handling for IO actions)

---

**Example (interactive cp: icp.hs)**

```haskell
main :: IO()
```

```haskell
main :: IO()
main =
    do fromH <- readOpenFile "Copy from: " ReadMode
```
Example (interactive cp: icp.hs)

```haskell
main :: IO()
main =
    do fromH <- readOpenFile "Copy from: " ReadMode
toH <- readOpenFile "Copy to: " WriteMode
    contents <- hGetContents fromH
    hPutStr toH contents
    hClose fromH
    hClose toH

readOpenFile :: String -> IOMode -> IO Handle
readOpenFile prompt mode =
    do putStrLn prompt
```

Example (interactive cp: icp.hs)

```haskell
main :: IO()
main =
    do fromH <- readOpenFile "Copy from: " ReadMode
toH <- readOpenFile "Copy to: " WriteMode
    contents <- hGetContents fromH
    hPutStr toH contents
    hClose fromH
    hClose toH

readOpenFile :: String -> IOMode -> IO Handle
readOpenFile prompt mode =
    do putStrLn prompt
```
Example (interactive cp: icp.hs)

```haskell
main :: IO()
main =
  do fromH <- readOpenFile "Copy from: " ReadMode
toH <- readOpenFile "Copy to: " WriteMode
    contents <- hGetContents fromH
    hPutStr toH contents
    hClose fromH
    hClose toH

readOpenFile :: String -> IOMode -> IO Handle
readOpenFile prompt mode =
  do putStrLn prompt
     name <- getLine
     handle <- openFile name mode
     return handle
```

Executing xyz.hs

If `xyz.hs` contains a definition of `main`:
- `runhaskell xyz`

If `xyz.hs` contains a definition of `main`:
- `runhaskell xyz`
or
- `ghc xyz` → executable file `xyz`
main :: IO()
main =
  do fromH <- readFile "Copy from: " ReadMode
      toH <- readFile "Copy to: " WriteMode
      contents <- hGetContents fromH
      hPutStrLn toH contents
      hClose fromH
      hClose toH

readOpenFile :: String -> IOMode -> IO Handle
readOpenFile prompt mode =
  do putStrLn prompt
     name <- getLine
     handle <- openFile name mode
     return handle

import System.IO
- data Socket
- data PortId = PortNumber PortNumber | ...

- data Socket
- data PortId = PortNumber PortNumber | ...
- data PortNumber
  instance Num PortNumber

- data Socket
- data PortId = PortNumber PortNumber | ...
- data PortNumber
  instance Num PortNumber

- listenOn :: PortId -> IO Socket
  Create server side socket for specific port
**Server functions**

- **listenOn**: PortId -> IO Socket
  Create server side socket for specific port

- **accept**: Socket -> IO (Handle, ..., ...)
  → can read/write from/to socket via handle

- **sClose**: Socket -> IO ()
  Close socket

---

**Initialization for Windows**

```haskell
withSocketsDo :: IO a -> IO a
```

Standard use pattern:
```haskell
main = withSocketsDo $ do ...
```
Initialization for Windows

\[ \text{withSocketsDo :: IO a -> IO a} \]

Standard use pattern:
\[ \text{main = withSocketsDo \$ do ...} \]

Does nothing under Unix

---

main :: IO ()
main = withSocketsDo \$ do
  sock <- listenOn \$ PortNumber 9000

withSocketsDo :: IO a -> IO a

Standard use pattern:
\[ \text{main = withSocketsDo \$ do ...} \]

Does nothing under Unix
main :: IO ()
main = withSocketsDo $ do
  sock <- listenOn $ PortNumber 9000
  (h, _, _) <- accept sock
  hSetBuffering h LineBuffering
  loop h
  sClose sock
Example (pingPong.hs)

```haskell
main :: IO ()
main = withSocketsDo $ do
    sock <- listenOn $ PortNumber 9000
    (h, _, _) <- accept sock
    hSetBuffering h LineBuffering
    loop h
    sClose sock

loop :: Handle -> IO ()
loop h = do
    input <- hGetLine h
    if take 4 input == "quit"
      then do hPutStrLn h "goodbye!"
              hClose h
      else do hPutStrLn h ("got " ++ input)
             loop h
```

Example (pingPong.hs)

```haskell
main :: IO ()
main = withSocketsDo $ do
    sock <- listenOn $ PortNumber 9000
    (h, _, _) <- accept sock
    hSetBuffering h LineBuffering
    loop h
    sClose sock

loop :: Handle -> IO ()
loop h = do
    input <- hGetLine h
    if take 4 input == "quit"
      then do hPutStrLn h "goodbye!"
              hClose h
      else do hPutStrLn h ("got " ++ input)
             loop h
```

Client functions

```haskell
xxx.hs
lapproy100:Code nipkow$ less xxx.hs
import System.IO

main :: IO()
main = do
    fromH <- readOpenFile "Copy from: " ReadMode
    toH <- readOpenFile "Copy to: " WriteMode
    contents <- hGetContents fromH
    hPutStrLn toH contents
    hClose fromH
    hClose toH

readOpenFile :: String -> IOMode -> IO Handle
readOpenFile prompt mode =
    do putStrLn prompt
       name <- getline
       handle <- openFile name mode
       return handle
lapproy100:Code nipkow$
lapproy100:Code nipkow$
lapproy100:Code nipkow$ runhaskell pingPong
```
readOpenFile :: String -> IOHandle -> IO Handle  
readOpenFile prompt mode =  
do putStrLm prompt  
  name <- getLine  
  handle <- openfile name mode  
return handle  
lapbroy100:Code ntpkow$  
lapbroy100:Code ntpkow$  
lapbroy100:Code ntpkow$ runhaskell pingPong

Last login: Tue Dec 17 16:21:51 on ttys003  
lapbroy100:- nipkow$ telnet localhost 9000  
Trying ::1...  
Connected to localhost.  
Escape character is '^]'.

hClose sock  

Last login: Tue Dec 17 16:22:28 on ttys002  
lapbroy100:- nipkow$ cd Teaching/FP/1314/Code/  
lapbroy100:Code ntpkow$ runhaskell pingPong

^\C{l}apbroy100:Code ntpkow$  
lapbroy100:Code ntpkow$ runhaskell pingPong
Last login: Tue Dec 17 16:21:51 on ttys003
lapbroy100:~ nипkow$ telnet localhost 9000
Trying ::1...
Connected to localhost.
Escape character is '^]'.
+\^H
got +

Client functions
Client functions

- `type HostName = String`
  
  For example "haskell.org" or "192.168.0.1"

- `connectTo :: HostName -> PortId -> IO Handle`
  
  Connect to specific port of specific host

Example (wGet.hs)

```haskell
main :: IO()
main = withSocketsDo $ do
```
Example (wGet.hs)

```haskell
main :: IO()
main = withSocketsDo $ do
  putStrLn "Host?"
  host <- getLine
  h <- connectTo host (PortNumber 80)
  hSetBuffering h LineBuffering
  putStrLn "Resource?"
  res <- getLine
  hPutStrLn h ("GET " ++ res ++ " HTTP/1.0\n")
```

Example (wGet.hs)

```haskell
main :: IO()
main = withSocketsDo $ do
  putStrLn "Host?"
  host <- getLine
  h <- connectTo host (PortNumber 80)
  hSetBuffering h LineBuffering
  putStrLn "Resource?"
  res <- getLine
  putStrLn h ("GET " ++ res ++ " HTTP/1.0\n")
```
Example (wGet.hs)

```haskell
main :: IO()
main = withSocketsDo $ do
  putStrLn "Host?"
  host <- getline
  h <- connectTo host (PortNumber 80)
  hSetBuffering h LineBuffering
  putStrLn "Resource?"
  res <- getline
  putStrLn h ("GET " ++ res ++ " HTTP/1.0\n")
  s <- hGetContents h
  putStrLn s
```

For more detail see

http://hackage.haskell.org/package/network/docs/
Network.html

http://hackage.haskell.org/package/network/docs/
Network-Socket.html
http/1.1 302 Found
Date: Tue, 17 Dec 2013 15:33:17 GMT
Server: Apache
Location: http://www21.in.tum.de/teaching/info2/WS1314/
Content-Length: 291
Connection: close
Content-Type: text/html; charset=iso-8859-1

<title>302 Found</title>
</head>
<body>
<h1>Found</h1>
<p>The document has moved <a href="http://www21.in.tum.de/teaching/info2/WS1314/">here</a>.</p>
</body>

lapbroy100:Code niphkow$ runhaskell wget
Host?

http/1.0 302 Found
Cache-Control: private
Content-Type: text/html; charset=UTF-8
Location: http://www.google.de/?gfe_rd=cr&ei=D2-wUq-1Ho2c_wbRxIHIBQ
Content-Length: 258
Date: Tue, 17 Dec 2013 15:34:39 GMT
Server: GFE/2.0
Alternate-Protocol: 80:quic

<TITLE>302 Moved</TITLE>

The document has moved
<A HREF="http://www.google.de/?gfe_rd=cr&amp;ei=D2-wUq-1Ho2c_wbRxIHIBQ">here</A>.
</BODY>

lapbroy100:Code niphkow$ runhaskell wget
Host?
<IDCDOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="initial-scale=1, minimum-scale=1, width=device-width">
  </head>
  <title>Error 404 (Not Found)!!1</title>
  <style>*{margin:0;padding:0}html{font:15px/22px arial,sans-serif}body{margin:7%auto;max-width:390px;min-height:180px;padding:30px 0 15px}body{background:url('https://www.google.com/images/errors/robot.png') 100% 5px no-repeat;padding-right:0;}p{margin:11px 0 22px;overflow:hidden}a img{border:0}@media screen and (max-width:772px){body{background:none;margin-top:0;max-width:none;padding-right:0}}
</style>
  <body>
    <a href="/cgi-bin/src/view/www.google.com/images/errrors/logo_sm.gif alt=Google/a">
      <p>The requested URL <code>/NID=67=kuaSY7udMAJcXVYIPy6_oJwMvljQi.x0jY9XdLlkVtbsUY9G0mImfcqLulb0TJw-GyFvYjJLqIo6dyB0CFCEe3T-5-z.a280XsMj1Ne5bHFwMCWTmzyHyqPSy</code> was not found on this server. <ins>ThatA4AAs all we know.</ins></p>
  
  <p>lapbro<sub>y</sub>100:Code nipkow</p>
  <p>lapbro<sub>y</sub>100:Code nipkow runhaskell wGet Host?</p>
  </body>
</html>