Lesson 9

interactive programming

I/O

Chapter 10

so far we have seen batch programs in Haskell

input is given together with the program that executes and then prints the result

interactive programs ask input and give output possibly several times during the execution

since Haskell is functional we want to see also I/O as a function of type IO
type IO = World -> World

However IO actions are impure functions since they have side effects

Repeating the same action may give diffrent results

Provided that interactive IO is often necessary, Haskell programs contain both pure and impure functions

Is important to keep the interface very clear and limited

There is a main (impure) that does IO and calls the pure parts

but I/O actions may «return» a value
type IO a = World -> (a,World)
IO Char «returns» a Char (<- operator to «extract» it)
IO () only side effect

interactive programs may need input
It’s easy to model with currying
Char -> IO ()

what is World ?
in reality IO is a built-in type whose details are hidden
data IO a = ...

we start with some basic I/O actions
we will compose them to make more sophisticated interactive programs

--getChar :: IO Char
--putChar :: Char -> IO ()
--return :: a -> IO a

these actions are built into the GHC system

return transforms any expression into a IO action that «returns» that expression: from pure to impure

the type IO a is a **monad** and therefore we can use a special do-notation for composing I/O actions:

```
do v1 <- a1
     v2 <- a2
      ........
     vn <- an
     return (f v1 v2  ... vn)
```

mind the layout

each v <- a is a generator (is the inverse of return)

we use ai alone when vi doesn’t matter

example: an action that reads 3 Char and returns the 1st and 3rd

```
act :: IO (Char, Char)
act = do x <- getChar
         getChar
         y <- getChar
         return (x,y)
```

omitting the return would result in a type error

From the primitive IO actions => Derived IO actions

```
getLine :: IO String
getline = do x <- getChar
             if x == '\n' then
               return []
             else
               do xs <- getline  -- recursion
                    return (x:xs)
```

```
putStr :: String -> IO ()
putStr []       = return ()
putStr (x:xs) = do putChar x
                   putStr xs

putStrLn :: String -> IO ()
putStrLn xs = do putStr xs
                 putChar '\n'
```

example: an I/O action that prompts for a string and displays its length

```
strlen :: IO ()
strlen = do putStr «Enter a string: »
            xs <- getLine
            putStr «The string has »
            putStr (show (length xs))
            putStrLn « characters»
```

Hangman
is a game as follows.
-one player secretly enters a word
-another player tries to find the word through a series of guesses
-for each guess the program indicates which letters in the secret word occur in the guess and also in which positions of the secret word

```
hangman :: IO ()
hangman = do putStrLn «Think of a word:»
             word <- sgetLine
             putStrLn «Try to guess it :»
             play word
```

```
sgetLine :: IO String
sgetLine = do x <- getCh
                  if x == '\n' then
                     do putChar x
                          return []
                   else
                     do putChar '_'
                     xs <- sgetLine
                     return (x:xs)
```

getCh reads a Char without echo to the screen

```
import System.IO
getCh :: IO Char
getCh = do hSetEcho stdin False
           x <- getChar
           hSetEcho stdin True
           return x
```

```
play :: String -> IO ()
play word = do putStr «?»
               guess <- getLine
               if guess == word then
                 putStrLn «You got it!!»
                else
                 do putStrLn (match word guess)
                      play word

match :: String -> String -> String
match xs ys = [if elem x ys then x else ‘-’ | x <- xs]
```

Nim

1: * * * * *

2: * * * *

3: * * *

4: * *

5: *

The players eliminate some stars from a row. Wins the player who makes the board empty

Game Utilities

```
next :: Int ->Int
next 1 = 2
Next 2 = 1

type Board = [Int]

initial :: Board
initial = [5,4,3,2,1]

finished :: Board -> Bool
finished = all (== 0)
```

```
valid :: Board -> Int -> Int -> Bool
valid board row num = board !! (row – 1) >= num

move :: Board -> Int -> Int -> Board
move board row num = [update r n | (r,n) <- zip [1..] board]
    where update r n = if r == row the n-num else n
```

IO utilities

```
putRow :: Int -> Int -> IO ()
putRow row num = do putStr (show row)
                    putStr ‘’:  ‘’
                    putStrLn (concat (replicate num ‘’*  ‘’))

putBoard :: Board -> IO ()
putBoard [a,b,c,d,e] = do putRow 1 a
                          putRow 2 b
                          putRow 3 c
                          putRow 4 d
                          putRow 5 e
```

read a move

```
getDigit :: String -> IO Int
getDigit prompt = do putStr prompt
                     x <- getChar
                     newline
                     if isDigit x then
                       return (digitToInt x)
                     else
                       do putStrLn ''Error: invalid digit''
                          getDigit prompt
```

import Data.Char

```
newline :: IO ()
newline = putChar '\n'
```

```
play :: Boad ->Int -> IO ()
play board player =
    do newline
        putBoard board
        if finished board then
            do newline
                putStr ’’Player’’
                putStr (show (next player))
                putStrLn ‘’ wins !!’’
         else –game continues
```

```
else
    do newline
        putStr ‘’ Player ’’
        putStrLn (show player)
        row  <-getDigit ‘’Enter a row number: ‘’
        num <-getDigit ‘’Stars to remove: ‘’
        if valid board row num then
            play (move board row num) (next player)
        else
            do newline
                putStrLn ‘’ Error: invalid move ‘’
                play board player
```

2 remarks

1) play gets board and player as parameters and does not operate on mutable values: Haskell is pure

2) observe the separation and cooperation between IO (impure) parts and game utilities (pure)