Interactive Programming Examples

In this section we will implement a larger interactive game, Hangman. As we do so we will see how one structures such programs so as to separate the pure components from the unpure ones.

Hangman

The Hangman game we implement will be somewhat different from the version in the book, but it has many things in common. The game starts by asking the user for a string. It then offers the user a fixed number of chances to guess a single character from the string, and it visually shows all matched characters from the string.

Usually implementing a program like that consists of two levels:

- The user interaction component, starting from the main action and decomposing that further into smaller actions until you get to more basic components.
- The pure component of the application, which takes as input the kinds of elements that the user would be entering and produces suitable outputs. This may be one function or typically many smaller utility functions.

One can start working from either direction, and more often than not move back and forth between the two approaches.

We start with the top-level approach:

```
main :: IO ()
main = do
    word <- readHiddenWord
    guessWord word
    main
```

We need to implement two actions. One that reads a word from the user, without showing it, and another that interacts with the user as they are trying to guess it. The main function then simply performs these two operations, and calls itself again for an infinite set of games.

Reading The Word

The first half of the project has to do with implementing readHiddenWord. We will break that part further in smaller chunks:

```
readHiddenWord :: IO String
readHiddenWord = do
    putStrLn "Enter␣word␣to␣guess:␣"
    withoutEcho readWord
```

The first part in this action is fairly straightforward, we put in a prompt for the user. The second part is a call to the function withoutEcho, which we will write, and the action readWord which we will also write. withoutEcho takes as input an action, and performs that action returning its value. But before doing so it turns off the default "echo" process that is in place (namely that whenever you press a character in the keyboard that character also appears on the screen). readWord reads a character, then prints an asterisk in its place, and returns the character.

```
withoutEcho :: IO a -> IO a
withoutEcho action = do
    hSetEcho stdin False
    v <- action
    hSetEcho stdin True
    return v
```

The action hSetEcho is a built-in action that is given a "stream", standard input in this case, and a new boolean value, and it sets the echo-ing state of that stream.

readWord simply reads the character and puts an asterisk in its place. It must however still print the newline (and returns):

```
readWord :: IO String
readWord = do
    c <- getChar
    if c == '\n'
        then do putChar '\n'
                return ""
        else do putChar '*'
                cs <- readWord
                return (c:cs)
```

This finishes the first part of the application, that of reading in a guess from the user.

Interactive Guessing

The second part of the project requires that we implement the guessWord action. Let us think of what that might involve:

- We must keep track of the characters that the user has guessed. We could also keep a count of how many guesses the user has made, but we can simply find that out from the list of guessed characters. This means that our main function should take a second argument in addition to the word, namely the list of guesses, which probably will start life as empty.
- We must have a way of comparing the list of guessed characters with the word, and see if all characters in the word have been guessed. This will be a pure function.
- We must have a way of forming a visual representation of the word where all not-yet-guessed characters are replaced by an underscore. This will also be a pure function.

Let us see how guessWord would look like, with all this in mind. The heart of the matter is the guessLoop action (**Question**: Why did we write [Char] instead of String for the second parameter to guessLoop?).

```
guessWord :: String -> IO ()
guessWord word = guessLoop word []

guessLoop :: String -> [Char] -> IO ()
guessLoop word guesses = do
    putGuesses guesses
    putMaskedWord word guesses
    if isFullyGuessed word guesses
        then putStrLn "Congratulations, you guessed it!"
        else if length guesses >= 15
            then do putStrLn "I'm sorry, you have run out of guesses!"
                    putStrLn ("The word was: " ++ word)
            else readNextGuess word guesses

putGuesses :: [Char] -> IO ()
putGuesses guesses = do
    putStr "Guessed: "
    putEachGuess guesses
    putChar '\n'

putEachGuess :: [Char] -> IO ()
putEachGuess []     = return ()
putEachGuess (g:gs) = do
    putChar g
    unless (null gs) (putStr ", ")
    putEachGuess gs

putMaskedWord :: String -> [Char] -> IO ()
putMaskedWord word guesses = putStrLn $ maskedWord word guesses

readNextGuess :: String -> [Char] -> IO ()
readNextGuess word guesses = do
    hSetBuffering stdin NoBuffering
    guess <- getChar
    putChar '\n'
    if guess `elem` guesses
        then do putStrLn "Character already guessed!"
                guessLoop word guesses
        else guessLoop word (guess:guesses)
```

We used some function compositions there, and also the hSetBuffering action to set the input to do no buffering (so that we can immediately read every character typed rather than wait for the whole line). In order for this to work, we had to import the corresponding modules:

```
import System.IO
```

All that remains are the pure functions. We need one function to mask a word given some guesses, and one function to check if a word is fully guessed from its guesses. Both are simple:

```
isFullyGuessed :: String -> [Char] -> Bool
```

```
isFullyGuessed word guesses = and [c `elem` guesses | c <- word]

maskedWord :: String -> [Char] -> String
maskedWord word guesses = [handleChar c | c <- word]
    where handleChar c | c `elem` guesses   = c
                       | otherwise          = '_'
```

Now we have a working hangman implementation!

Variations

Here are some variations we can try on the game:

1. Have the number of guesses somehow depend on the length of the word. For example allow up to twice the number of guesses than the word length, with a minimum of 8 guesses.
2. Only count how many guesses were missed towards the allowed count of guesses. This would require that the count be kept as a separate parameter to the problem.