Chapter 10 - Interactive Programming
Introduction

To date, we have seen how Haskell can be used to write batch programs that take all their inputs at the start and give all their outputs at the end.
However, we would also like to use Haskell to write interactive programs that read from the keyboard and write to the screen, as they are running.
The Problem

Haskell programs are pure mathematical functions:

z Haskell programs **have no side effects**.

However, reading from the keyboard and writing to the screen are side effects:

z Interactive programs **have side effects**.
Interactive programs can be written in Haskell by using types to distinguish pure expressions from impure actions that may involve side effects.
For example:

\[ \text{IO Char} \]

- The type of actions that return a character.

\[ \text{IO ()} \]

- The type of purely side effecting actions that return no result value.

Note:

\[ z () \] is the type of tuples with no components.
Basic Actions

The standard library provides a number of actions, including the following three primitives:

1. The action `getChar` reads a character from the keyboard, echoes it to the screen, and returns the character as its result value:

\[
\text{getChar :: IO Char}
\]
The action `putChar c` writes the character `c` to the screen, and returns no result value:

\[
\text{putChar} :: \text{Char} \rightarrow \text{IO} \ ()
\]

The action `return v` simply returns the value `v`, without performing any interaction:

\[
\text{return} :: a \rightarrow \text{IO} \ a
\]
Sequencing

A sequence of actions can be combined as a single composite action using the keyword `do`.

For example:

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

z Reading a string from the keyboard:

```haskell
getLine :: IO String
getLine = do x ← getChar
            if x == '\n' then
                return []
            else
                do xs ← getLine
                return (x:xs)
```
Writing a string to the screen:

```haskell
putStr :: String → IO ()
putStr [] = return ()
putStr (x:xs) = do putChar x
               putStr xs
```

Writing a string and moving to a new line:

```haskell
putStrLn :: String → IO ()
putStrLn xs = do putStr xs
                putChar '\n'
```
We can now define an action that prompts for a string to be entered and displays its length:

```haskell
strlen :: IO ()
strlen = do putStr "Enter a string: 
  xs ← getLine
  putStrLn "The string has 
  putStrLn (show (length xs))
  putStrLn " characters"
```
For example:

\[
> \text{strlen}
\]

Enter a string: Haskell
The string has 7 characters

Note:

- Evaluating an action **executes** its side effects, with the final result value being discarded.
Hangman

Consider the following version of hangman:

- One player secretly types in a word.
- The other player tries to deduce the word, by entering a sequence of guesses.
- For each guess, the computer indicates which letters in the secret word occur in the guess.
The game ends when the guess is correct.

We adopt a **top down** approach to implementing hangman in Haskell, starting as follows:

```haskell
hangman :: IO ()
hangman = do putStrLn "Think of a word: "
             word <- sgetLine
             putStrLn "Try to guess it:"
             play word
```
The action `sgetLine` reads a line of text from the keyboard, echoing each character as a dash:

```haskell
sgetLine :: IO String
sgetLine = do x ← getCh
              if x == \n then
                do putChar x
                return []
              else
                do putChar '-'
x:xs ← sgetLine
return (x:xs)
```
The action `getCh` reads a single character from the keyboard, without echoing it to the screen:

```haskell
import System.IO

getCh :: IO Char
getCh = do hSetEcho stdin False
            x ← getChar
            hSetEcho stdin True
            return x
```
The function `play` is the main loop, which requests and processes guesses until the game ends.

```haskell
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
```
The function `match` indicates which characters in one string occur in a second string:

```
match :: String → String → String
match xs ys = [if elem x ys then x else '-' | x ← xs]
```

For example:

```
> match "haskell" "pascal"
"-as--ll"
```
Exercise

Implement the game of nim in Haskell, where the rules of the game are as follows:

1. The board comprises five rows of stars:

   1: * * * * *
   2: * * * *
   3: * * *
   4: * *
   5: *
Two players take it turn about to remove one or more stars from the end of a single row.

The winner is the player who removes the last star or stars from the board.

Hint:

Represent the board as a list of five integers that give the number of stars remaining on each row. For example, the initial board is [5,4,3,2,1].