FUNCTIONAL PROGRAMMING

INTERACTIVE PROGRAMS

Interactive programs can be written in Haskell by using types to distinguish pure expressions:

```
IO a
```

The type of actions that return a value of type a.

For example:

`IO Char`

The type of actions that return a character.

`IO ()`

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

Note:

() is the type of tuples with no components.

BASIC ACTIONS

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

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

```
getChar :: IO Char
```

- The action putChar c writes the character c to the screen, and returns no result value:

```
putChar :: Char -> IO ()
```

- The action return a simply returns the value a, without performing any interaction:

```
return :: a -> IO a
```

SEQUENCING

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

For example:

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

EXAMPLE

We can now define an action that prompts for a string to be entered and displays its length:

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

P.T.O

For example:

```
> strlen

Enter a string: abcde
The string has 5 characters
```

Note:

- Evaluating an action <u>executes</u> its side effects, with the final result value being discarded.

The function <u>diff</u> indicates which characters in one string occur in a second string:

```
diff         :: String -> String -> String
diff xs ys =
   [if elem x ys then x else '-' | x <- xs]
```

For example:

```
> diff "haskell" "pascal"

"-as--ll"
```

THE ZIP FUNCTION

A useful library function is zip, which maps two lists to a list of pairs of their corresponding elements.

```
zip :: [a] -> [b] -> [(a,b)]
```

For example:

```
> zip ['a','b','c'] [1,2,3,4]

[('a',1),('b',2),('c',3)]
```

Using zip we can define a function returns the list of all <u>pairs</u> of adjacent elements from a list:

```
pairs    :: [a] -> [(a,a)]
pairs xs = zip xs (tail xs)
```

For example:

```
> pairs [1,2,3,4]

[(1,2),(2,3),(3,4)]
```

Using pairs we can define a function that decides if the elements in a list are <u>sorted</u>:

```
sorted    :: Ord a => [a] -> Bool
sorted xs =
   and [x <= y | (x,y) <- pairs xs]
```

For example:

```
> sorted [1,2,3,4]
True

> sorted [1,3,2,4]
False
```

Using zip we can define a function that returns the list of all positions of a value in a list:

```
positions :: Eq a => a -> [a] -> [Int]
positions x xs =
   [i | (x’,i) <- zip xs [0..n], x == x’]
   where n = length xs - 1
```

For example:

```
> positions 0 [1,0,0,1,0,1,1,0]
[1,2,4,7]
```

PRACTICE

- Write a program in haskell where we can take four letters by pressing enter and then it will print the sequence:
  (a, b, c, d)

- Write a program in haskell where it can find the position of 1 in binary sequence like this [1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1].

- Write a program in haskell which will find out a sequence of numbers is in descending order or not.
  For example, [8, 4, 2, 1] is TRUE, [1, 3, 7, 9] is FALSE.