Exercises
Interactive Programming

Exercise 1

Write an I/O program which will read a line of input and test whether the input is a palindrome. The program should 'prompt' the user for its input and also output an appropriate message.

Exercise 2

Write an I/O program which will read two integers, each on a separate line and output their sum. The program should prompt for input and explain its output.

Exercise 3

Define a function

```
putNtimes :: Integer -> String -> IO ()
```

so that the effect of

```
 putNtimes n str
```

is to output a string *str*, n times, one per line.
Hint: You can use recursion in the definition.

Exercise 4

Write an I/O program which will first read a positive integer, n, and then read n integers and write their sum. The program should prompt for input and explain its output.
Hint: use auxillary functions, e.g.

```
 getInteger :: String -> IO Integer
 sumNInts ::       -- .... which sums n ints
```

** Exercise 5

Without looking at the definitions from the standard Prelude, define the following library functions on lists using recursion:

1. Decide is all logical values in a list are ***True***

   ```
   myAnd :: [Bool] -> Bool
   ```

2. Concatenate a list of lists

   ```
   myConcat :: [[a]] -> [a]
   ```

3. Produce a list with ***n*** identical elements

   ```
   myReplicate :: Int -> a -> [a]
   ```

4. Select the n^{th} element of a list

   ```
   myNth :: [a]-> Int -> a
   ```

5. Decide if an value is an element of a list

   ```
   myElem  :: Eq a => a -> [a] -> Bool
   ```

*** Exercise 6

Using the five-step process, construct the library functions that:

1. calculate the ***sum*** of a list of numbers;
2. ***take*** a given number of elements from the start of a list;
3. select the ***last*** element of non-empty list.

*** Exercise 7

Define a recursive function

```
merge :: Ord a => [a] -> [a] -> [a]
```

that merges two sorted lists to give a single sorted list. Note : Your definition should not use other functions on sorted lists such as ***insert*** or ***isort***, but should be defined using explicit recursion.

**** Exercise 8

Using ***merge***, define a function

```
msort :: Ord a => [a] -> [a]
```

that implements *merge sort*, in which the empty list and singleton lists are already sorted, and any other list is sorted by merging together the two lists that result from sorting the two halves of the list separately.

Hint 1: First define a function

```
halve :: [a] -> ([a] , [a])
```

that splits a list into two halves whose lengths differ by at most one.
Hint 2: You can use the following functions (though you may not need to)

```
fst :: (a,b) -> a
snd :: (a,b) -> b
fst (x,y) = x
snd (x,y) = y
```

Solutions

Solutions to exercise 1

```
interactivePalCheck :: IO ()

interactivePalCheck
  = do putStr "Input␣a␣string␣for␣palindrome␣check:␣"
       st <- getLine
       if st == reverse st
          then putStr "Palindrome.\n"
          else putStr "Not␣a␣palindrome.\n"
```

Solutions to exercise 2

```
interactiveIntSum :: IO ()

interactiveIntSum
  = do putStr "Input␣an␣integer␣(followed␣by␣Return):␣"
       st1 <- getLine
       let int1 = (read st1) :: Int
       putStr "Input␣another␣integer␣(followed␣by␣Return):␣"
       st2 <- getLine
       let int2 = read st2 :: Int
       putStrLn ("The␣sum␣of␣these␣integers␣is␣"++ show (int1+int2))
```

Solutions to exercise 3

```
putNtimes :: Integer -> String -> IO ()

putNtimes n st
  = if n<=0
       then return ()
       else do putStrLn st
               putNtimes (n-1) st
```

Solutions to exercise 4

```
-- Instead of solving this as a single function, worth thinking about how you ca
-- decompose the problem: write a function to get an integer, and another
-- to do the summing.
-- Useful auxiliary function, taking the prompt as parameter.

getInteger :: String -> IO Integer

getInteger prompt
  = do putStr prompt
       st <- getLine
       return (read st :: Integer)

-- Sum N integers: prompt, number to sum and and "sum so far" are the parameters

sumNints :: String -> Integer -> Integer -> IO Integer

sumNints prompt n s
  = if n<=0
       then return s
       else do m <- getInteger prompt
               sumNints prompt (n-1) (s+m)


--- The function itself

getNints :: IO ()

getNints
  = do bound <- getInteger "Input␣the␣number␣of␣integers␣to␣add:␣"
       sum <- sumNints "Input␣an␣integer:␣" bound 0
       putStrLn ("The␣sum␣of␣these␣integers␣is␣"++ show sum)
```

Solutions to exercise 5

1. Decide is all logical values in a list are ***True***

   ```
   myAnd :: [Bool] -> Bool
   myAnd [] = True
   myAnd (b:bs) = b && myAnd (bs)
   ```

2. Concatenate a list of lists

   ```
   myConcat :: [[a]] -> [a]
   myConcat []    = []
   myConcat (x:xs)   = x ++ (myConcat xs)
   ```

3. Produce a list with ***n*** identical elements

   ```
   myReplicate :: Int -> a-> [a]
   ```

```haskell
myReplicate 0 _ = []
myReplicate n x = x: myReplicate (n-1) x
```

4. Select the n^{th} element of a list

   ```haskell
   myNth ::[a] -> Int -> a
   myNth (x:xs)  0      = x
   myNth (x:xs) n = myNth xs (n-1)
   ```

5. Decide if an value is an element of a list

   ```haskell
   myElem :: Eq a => a -> [a] -> Bool
   myElem x [] = False
   myElem  x' (x:xs)  | x' == x = True
                      | otherwise = myElem x' xs
   ```

Solutions to exercise 6

1. calculate the ***sum*** of a list of numbers;

   ```haskell
   sum' :: Num a => [a] -> a
   sum' [] = 0
   sum' (x:xs) = x + sum xs
   ```

2. ***take*** a given number of elements from the start of a list;

   ```haskell
   take' :: Int-> [b] -> [b]
   take' 0 _ = []
   take' _ [] = []
   take' n (x:xs) = x: take' (n-1) xs
   ```

3. select the ***last*** element of non-empty list.

   ```haskell
   last' :: [a] -> a
   last' [x] = x
   last' (_:xs) = last xs
   ```

Solutions to exercise 7

```haskell
merge :: Ord a => [a] -> [a] -> [a]
merge xs [] = xs
merge [] ys = ys
merge (x:xs) (y:ys) | x <= y = x: merge xs (y:ys)
                    | otherwise = y: merge (x:xs) ys
```

Solutions to exercise 8

```
halve :: [a] -> ([a], [a])
halve [x] = ([x], [])
halve xs = (firsthalf, secondhalf)
   where
    firsthalf = take half xs
    secondhalf = drop half  xs
    half = div (length  xs) 2

msort :: Ord a => [a] -> [a]
msort [] = []
msort [x] = [x]
msort xs = merge  (msort left)  (msort right )
    where
     (left, right) = halve xs
```