Chapter 11 - Interactive Programs, Declaring Types and Classes
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:

- Haskell programs have no side effects.

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

- 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.

\[ \text{IO } a \]

The type of actions that return a value of type \( a \).
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:

\[
\text{IO ()}
\]

\text{() is the type of tuples with no components.}
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 v` simply returns the value `v`, 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:

```haskell
a :: IO (Char,Char)
a  = do x ← getChar
      getChar
      y ← getChar
      return (x,y)
```
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:

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

Writing a string and moving to a new line:

```
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
            putStr "The string has "
            putStr (show (length xs))
            putStrLn " characters"
```
For example:

> strlen

Enter a string: abcde
The string has 5 characters

Note:

- Evaluating an action executes its side effects, with the final result value being discarded.
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:"
       guess word
```
The action `sgetLine` reads a line of text from the keyboard, echoing each character as a dash:

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

---

15  

---

---
Note:

- The action `getCh` reads a character from the keyboard, without echoing it to the screen.

- This useful action is not part of the standard library, but is a special Hugs primitive that can be imported into a script as follows:

  ```haskell
  primitive getCh :: IO Char
  ```
The function **guess** is the main loop, which requests and processes guesses until the game ends.

```haskell
guess :: String -> IO ()
guess word =
  do putStrStr "> 
      xs ← getline
      if xs == word then
        putStrLnLn "You got it!"
      else
        do putStrLnLn (diff word xs)
          guess word
```
The function `diff` indicates which characters in one string occur in a second string:

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

For example:

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

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

☐ The board comprises five rows of stars:

<table>
<thead>
<tr>
<th></th>
<th>*</th>
<th>*</th>
<th>*</th>
<th>*</th>
<th>*</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>2</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>3</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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].
Type Declarations

In Haskell, a new name for an existing type can be defined using a type declaration.

```
type String = [Char]
```

String is a synonym for the type `[Char]`. 
Type declarations can be used to make other types easier to read. For example, given

\[
\text{type Pos} = (\text{Int,Int})
\]

we can define:

\[
\begin{align*}
\text{origin} & \quad :: \text{Pos} \\
\text{origin} & \quad = (0,0) \\
\text{left} & \quad :: \text{Pos} \to \text{Pos} \\
\text{left} (x,y) & \quad = (x-1,y)
\end{align*}
\]
Like function definitions, type declarations can also have parameters. For example, given

type Pair a = (a,a)

we can define:

```
mult :: Pair Int → Int
mult (m,n) = m*n
```

```
copy :: a → Pair a
copy x = (x,x)
```
Type declarations can be nested:

```
type Pos   = (Int,Int)
type Trans = Pos → Pos
```

However, they cannot be recursive:

```
type Tree = (Int,[Tree])
```
Data Declarations

A completely new type can be defined by specifying its values using a data declaration.

```
data Bool = False | True
```

Bool is a new type, with two new values False and True.
Note:

- The two values False and True are called the constructors for the type Bool.

- Type and constructor names must begin with an upper-case letter.

- Data declarations are similar to context free grammars. The former specifies the values of a type, the latter the sentences of a language.
Values of new types can be used in the same ways as those of built in types. For example, given

```haskell
data Answer = Yes | No | Unknown
```

we can define:

```haskell
answers :: [Answer]
answers = [Yes,No,Unknown]

flip :: Answer -> Answer
flip Yes = No
flip No = Yes
flip Unknown = Unknown
```
The constructors in a data declaration can also have parameters. For example, given

```haskell
data Shape = Circle Float |
             Rect Float Float
```

we can define:

```haskell
square :: Float → Shape
square n = Rect n n

area :: Shape → Float
area (Circle r) = pi * r^2
area (Rect x y) = x * y
```
Note:

- Shape has values of the form Circle \( r \) where \( r \) is a float, and Rect \( x \ y \) where \( x \) and \( y \) are floats.

- Circle and Rect can be viewed as functions that construct values of type Shape:

\[
\begin{align*}
\text{Circle} & \;::\; \text{Float} \rightarrow \text{Shape} \\
\text{Rect} & \;::\; \text{Float} \rightarrow \text{Float} \rightarrow \text{Shape}
\end{align*}
\]
Not surprisingly, data declarations themselves can also have parameters. For example, given

```
data Maybe a = Nothing | Just a
```

we can define:

```
safediv :: Int → Int → Maybe Int
safediv _ 0 = Nothing
safediv m n = Just (m `div` n)
safehead :: [a] → Maybe a
safehead [] = Nothing
safehead xs = Just (head xs)
```
Recursive Types

In Haskell, new types can be declared in terms of themselves. That is, types can be recursive.

\[
\text{data Nat} = \text{Zero} \mid \text{Succ Nat}
\]

Nat is a new type, with constructors Zero :: Nat and Succ :: Nat → Nat.
A value of type Nat is either Zero, or of the form Succ n where n :: Nat. That is, Nat contains the following infinite sequence of values:

- Zero
- Succ Zero
- Succ (Succ Zero)
- ...
- ...
We can think of values of type Nat as **natural numbers**, where Zero represents 0, and Succ represents the successor function $1+$. 

For example, the value

$$\text{Succ (Succ (Succ Zero))}$$

represents the natural number

$$1 + (1 + (1 + 0)) = 3$$
Using recursion, it is easy to define functions that convert between values of type Nat and Int:

```
nat2int :: Nat → Int
nat2int Zero = 0
nat2int (Succ n) = 1 + nat2int n

int2nat :: Int → Nat
int2nat 0 = Zero
int2nat (n+1) = Succ (int2nat n)
```
Two naturals can be added by converting them to integers, adding, and then converting back:

```haskell
add    :: Nat → Nat → Nat
add m n = int2nat (nat2int m + nat2int n)
```

However, using recursion the function add can be defined without the need for conversions:

```haskell
add Zero     n = n
add (Succ m) n = Succ (add m n)
```
For example:

\[
\begin{align*}
\text{add} & \ (\text{Succ} \ (\text{Succ} \ \text{Zero})) \ (\text{Succ} \ \text{Zero}) \\
= & \ \text{Succ} \ (\text{add} \ (\text{Succ} \ \text{Zero}) \ (\text{Succ} \ \text{Zero})) \\
= & \ \text{Succ} \ (\text{Succ} \ (\text{add} \ \text{Zero} \ (\text{Succ} \ \text{Zero}))) \\
= & \ \text{Succ} \ (\text{Succ} \ (\text{Succ} \ \text{Zero}))
\end{align*}
\]

Note:

- The recursive definition for add corresponds to the laws \(0+n = n\) and \((1+m)+n = 1+(m+n)\).
Consider a simple form of expressions built up from integers using addition and multiplication.
Using recursion, a suitable new type to represent such expressions can be declared by:

```haskell
data Expr = Val Int
         | Add Expr Expr
         | Mul Expr Expr
```

For example, the expression on the previous slide would be represented as follows:

```haskell
Add (Val 1) (Mul (Val 2) (Val 3))
```
Using recursion, it is now easy to define functions that process expressions. For example:

\[
\begin{align*}
\text{size} & : \text{Expr} \rightarrow \text{Int} \\
\text{size} (\text{Val} \ n) & = 1 \\
\text{size} (\text{Add} \ x \ y) & = \text{size} \ x + \text{size} \ y \\
\text{size} (\text{Mul} \ x \ y) & = \text{size} \ x + \text{size} \ y \\
\text{eval} & : \text{Expr} \rightarrow \text{Int} \\
\text{eval} (\text{Val} \ n) & = n \\
\text{eval} (\text{Add} \ x \ y) & = \text{eval} \ x + \text{eval} \ y \\
\text{eval} (\text{Mul} \ x \ y) & = \text{eval} \ x \ast \text{eval} \ y
\end{align*}
\]
Note:

- The three constructors have types:

  \[
  \begin{align*}
  & \text{Val} :: \text{Int} \to \text{Expr} \\
  & \text{Add} :: \text{Expr} \to \text{Expr} \to \text{Expr} \\
  & \text{Mul} :: \text{Expr} \to \text{Expr} \to \text{Expr}
  \end{align*}
  \]

- Many functions on expressions can be defined by replacing the constructors by other functions using a suitable `fold` function. For example:

  \[
  \text{eval} = \text{fold id (+) (*)}
  \]
In computing, it is often useful to store data in a two-way branching structure or binary tree.
Using recursion, a suitable new type to represent such binary trees can be declared by:

```haskell
data Tree = Leaf Int
          | Node Tree Int Tree
```

For example, the tree on the previous slide would be represented as follows:

```
Node (Node (Leaf 1) 3 (Leaf 4))
  5
(Node (Leaf 6) 7 (Leaf 9))
```
We can now define a function that decides if a given integer occurs in a binary tree:

\[
\begin{align*}
\text{occurs} & \quad :: \quad \text{Int} \rightarrow \text{Tree} \rightarrow \text{Bool} \\
\text{occurs} \ m \ \text{(Leaf} \ n) & \quad = \quad m == n \\
\text{occurs} \ m \ \text{(Node} \ l \ n \ r) & \quad = \quad m == n \\
& \quad \quad \quad \quad \quad \quad \quad \quad \quad || \ \text{occurs} \ m \ \text{l} \\
& \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad || \ \text{occurs} \ m \ \text{r}
\end{align*}
\]

But… in the worst case, when the integer does not occur, this function traverses the entire tree.
Now consider the function **flatten** that returns the list of all the integers contained in a tree:

\[
\text{flatten} :: \text{Tree} \rightarrow [\text{Int}]
\]

\[
\text{flatten} (\text{Leaf } n) = [n]
\]

\[
\text{flatten} (\text{Node } l \ n \ r) = \text{flatten } l
\]

\[
++ [n]
\]

\[
++ \text{flatten } r
\]

A tree is a **search tree** if it flattens to a list that is ordered. Our example tree is a search tree, as it flattens to the ordered list \([1,3,4,5,6,7,9]\).
Search trees have the important property that when trying to find a value in a tree we can always decide which of the two sub-trees it may occur in:

\[
\text{occurs } m \text{ (Leaf } n) = m == n \\
\text{occurs } m \text{ (Node } l \ n \ r) \mid m == n = \text{True} \\
\mid m < n = \text{occurs } m \ l \\
\mid m > n = \text{occurs } m \ r
\]

This new definition is more efficient, because it only traverses one path down the tree.
Exercises

(1) Using recursion and the function add, define a function that multiplies two natural numbers.

(2) Define a suitable function fold for expressions, and give a few examples of its use.

(3) A binary tree is complete if the two sub-trees of every node are of equal size. Define a function that decides if a binary tree is complete.