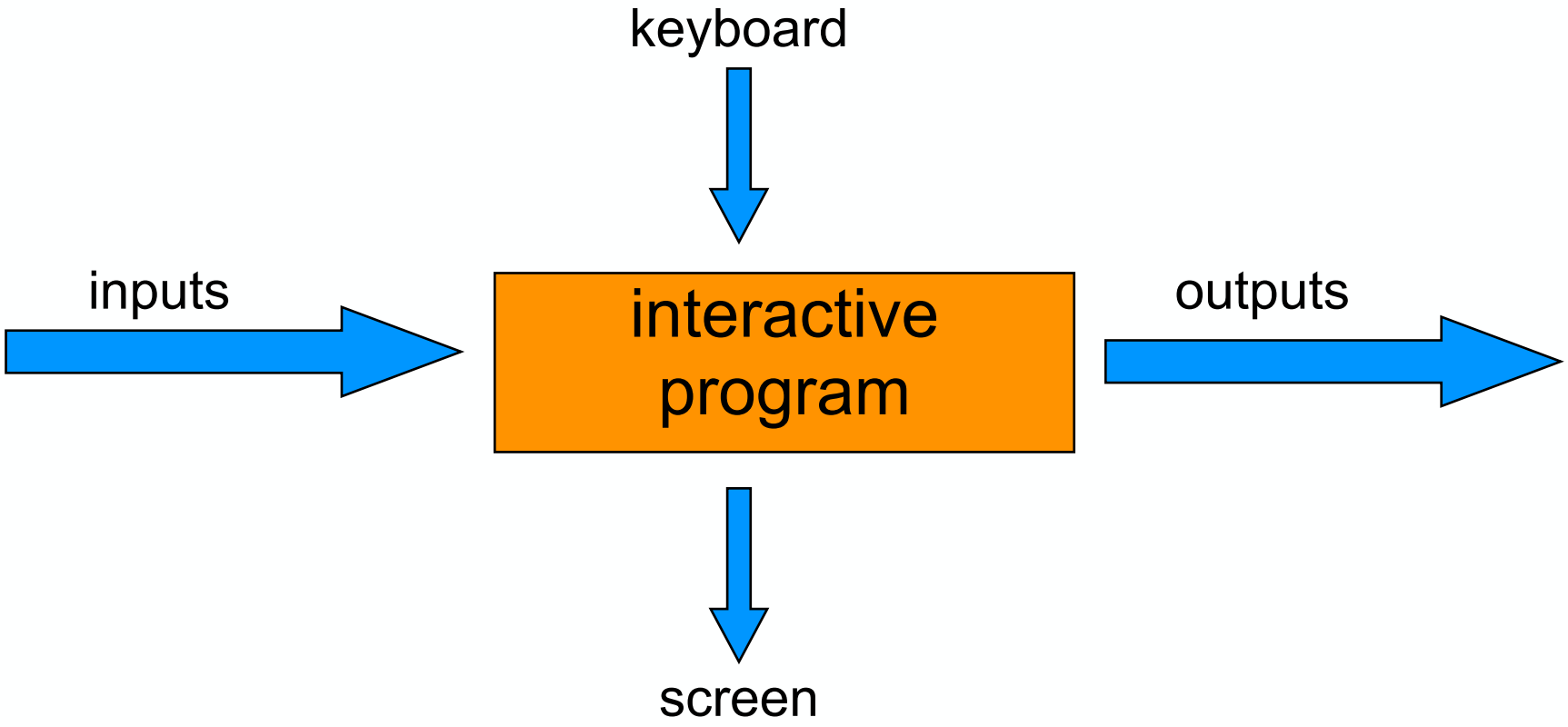
Chapter 10 - Interactive Programming

Strongly inspired by Graham Hutton's slides accompanying the book Programming in Haskell

Introduction

Haskell can be used to write ***batch*** programs that take all their inputs at the start and give all their outputs at the end.

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:

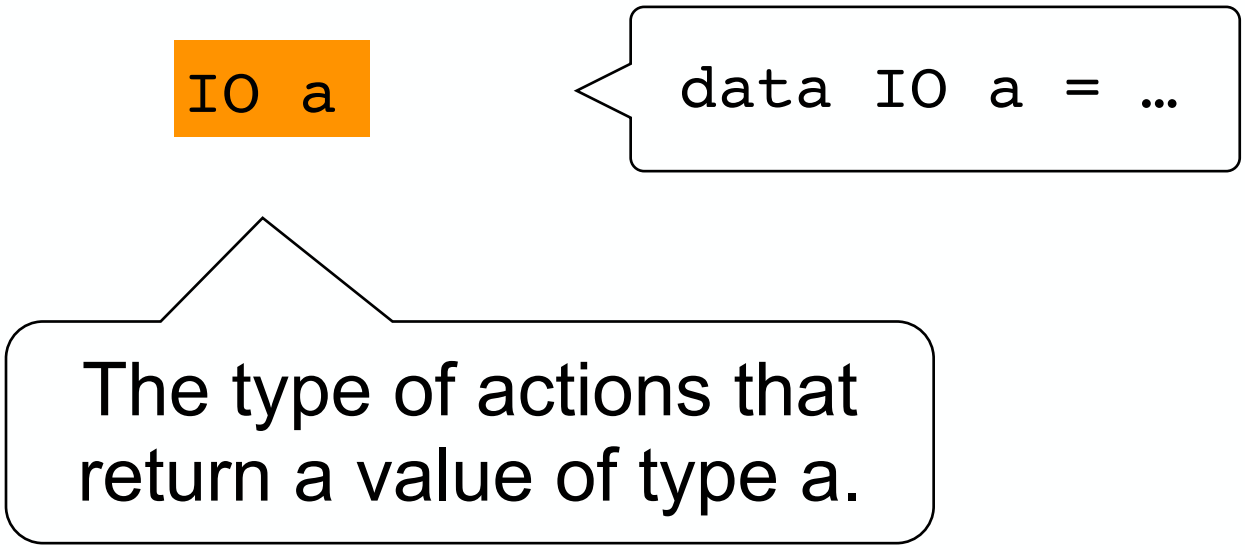
- Haskell programs ***have no side effects***.

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

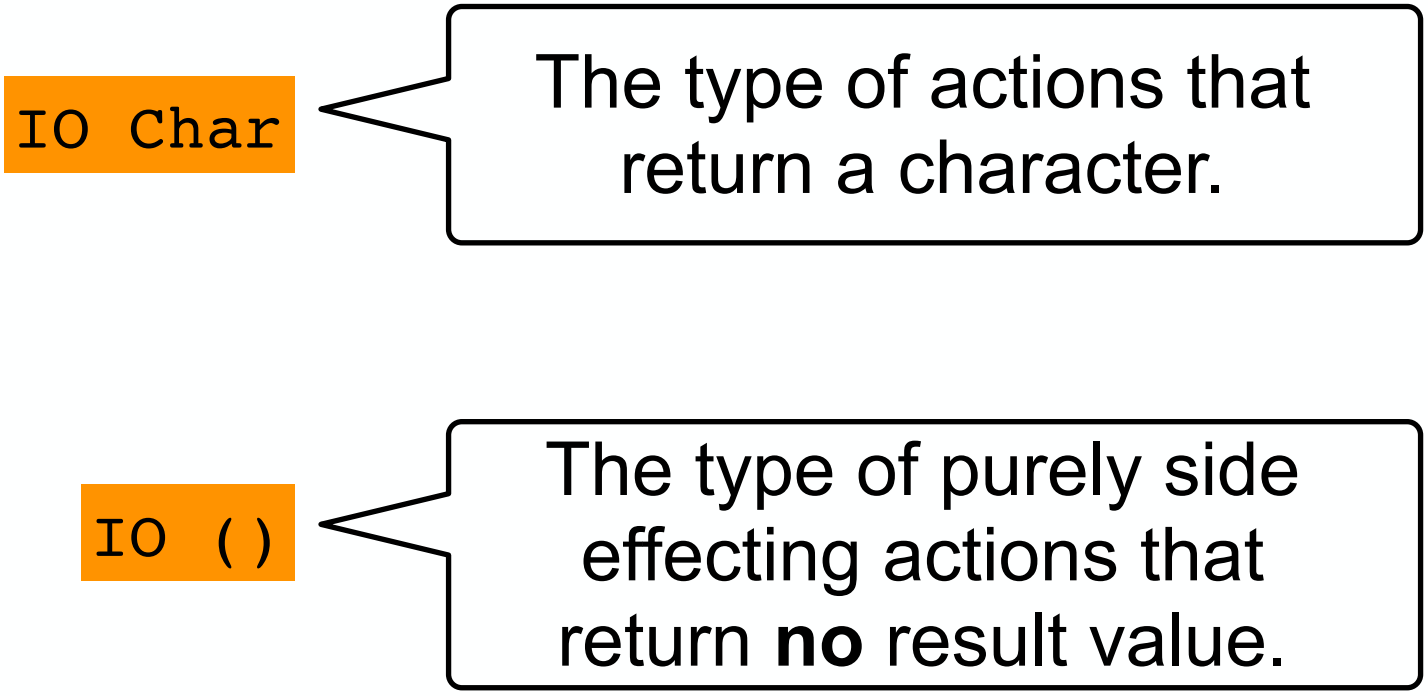
- Interactive programs ***have side effects***.

The Solution

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



For example:



Note:

() is the type of tuples with no components.

Basic Actions

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

```
getChar :: IO Char
```

```
putChar :: Char → IO ()
```

```
return :: a → IO a
```

- 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**.

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

Derived Primitives

Writing a string to the screen:

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

```
putStr' s = sequence_ [putChar c | c ← s]
```

Writing a string and moving to a new line:

```
putStrLn :: String → IO ()
putStrLn xs = do putStr xs
                 putChar '\n'
```

Derived Primitives

Reading a string from the keyboard:

```
getLine :: IO String
getLine = do x ← getChar
             if x == '\n' then
                return []
             else
                do xs ← getLine
                   return (x:xs)
```

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"
```

For example:

```
> 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*.

return

provides a bridge from expressions without side effects to impure actions with side effects.

```
hi = do putStr "Enter firstname: "
        firstName ← getLineputStr
        putStr "Enter lastname: "
        lastName ← getLine
        bfn ← return $ map toUpper firstName
        bln ← return $ map toUpper lastName
        putStrLn $ "Hi "++bfn++" "++bln
```

return

doesn't cause the end in execution.

```
hi = do putStr "Enter firstname: "
        firstName ← getLineputStr
        putStr "Enter lastname: "
        lastName ← getLine
        return "Don't stop"
        bfn ← return $ map toUpper firstName
        bln ← return $ map toUpper lastName
        putStrLn $ "Hi "++bfn++" "++bln
```

Using let

We can use let expression without the in (like in list comprehensions) :

```
hi = do putStr "Enter firstname: "
        firstName ← getLineputStr
        putStr "Enter lastname: "
        lastName ← getLine
        let bfn = map toUpper firstName
            bln = map toUpper lastName
        putStrLn $ "Hi "++bfn++" "++bln
```

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.

Example

```
> hangman
Think of a word:
-------
Try to guess it:
? million
-----ll
? maxwell
-a-ell
? haskell
You got it!
```

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

```
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:

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

The action *getCh* reads a single character from the keyboard, without echoing it to the screen:

```
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.

```
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 word guess =
  [if elem x guess then x else '-' | x ← word]
```

For example:

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

Exercise

Implement the classical version of **hangman**: twice as many choices as the length of the chosen word.

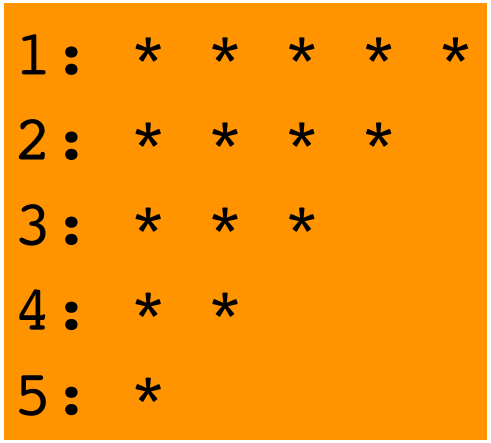
```
> hangman'
Think of a word:
-------
Try to guess it:
-------
a letter: g
-------
a letter: a
-a-----
a letter: e
-a--e--
a letter: l
```

```
a letter: l
-a--ell
a letter: p
-a--ell
a letter: h
ha--ell
a letter: k
ha-kell
a letter: s
You got it!
```

Exercise

Implement the game of **nim** in Haskell:

- The board comprises five rows of stars.
- 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.



Players are represented by integers:

```
next :: Int -> Int
next 1 = 2
next 2 = 1
```

Board is represented as a list of five integers that give the number of stars remaining on each row.

```
type Board = [Int]
initial :: Board
initial = [5,4,3,2,1]
```

Check if a game is finished

```
finished :: Board -> Bool
finished = all (== 0)
```

Move = line number + number of stars to remove

Check if a move is valid.

```
valid :: Board -> Int -> Int -> Bool
valid board row num = board !! (row-1) >= num
```

Make a move.

```
move :: Board -> Int -> Int -> Board
move board row num = [update r n |
                      (r,n) ← zip [1..] board]
     where update r n = if r == row then n-num
                                    else n
```

Print a row of the board.

```
putRow :: Int -> Int -> IO ()
putRow row num =
  do putStr (show row)
     putStr ": "
     putStrLn (concat (replicate num "* "))
```

Print a row of the board.

```
putBoard :: Board -> IO ()
putBoard [a,b,c,d,e] = do putRow 1 a
                          putRow 2 b
                          putRow 3 c
                          putRow 4 d
                          putRow 5 e
```

Read a character and convert it to a digit.

```
getDigit :: String -> IO Int
getDigit prompt = do putStr prompt
                     x ← getChar
                     putChar '\n'
                     if isDigit x then
                        return (digitToInt x)
                     else
                        do putStrLn "ERROR"
                           getDigit prompt
```

Hint:

`isDigit`: checks if a character represents a digit

`digitToInt`: converts Char to Int

```
play :: Board -> Int -> IO ()
```

```
play board player =
   do newline
      putBoard board
      if finished board then
         do putStrLn ((show (next player)) ++ " wins!!")
      else
         do putStrLn ("Player " ++ (show player))
            row ← getDigit "Enter a row number: "
            num ← getDigit "Stars to remove : "
            if valid board row num then
               play (move board row num) (next player)
            else
               do putStrLn "ERROR: Invalid move"
                  play board player
```