第11章 Interactive Programming

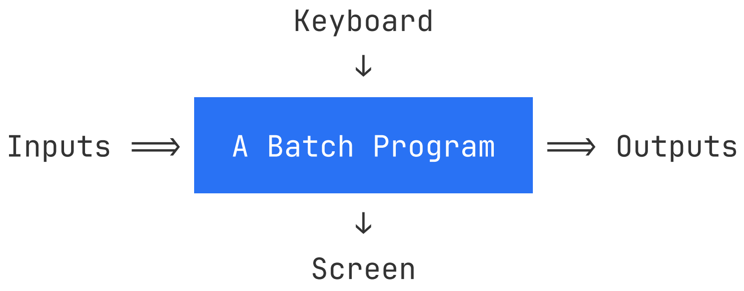
✦ Batch Programs

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.

Inputs ⟹ A Batch Program ⟹ Outputs

✦ Interactive Programs

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.



Keyboard

↓

Inputs ⟹ A Batch Program ⟹ Outputs

↓

Screen

✦ Interactive Programs in Haskell: Difficulties

Haskell programs are pure mathematical functions.
As a result, Haskell programs have no side effects.

However, interactive programs (i.e., reading from the keyboard and writing to the screen) have side effects.

✦ A solution that looks perfect

An interactive program can be viewed as **a pure function** that

- takes ***the current state of the world*** as its argument, and
- produces ***a modified world*** as its result.

```
type IO = World -> World
```

To represent a returning result in addition to performing side effects, we generalize the type to:

```
type IO a = World -> (a, World)
```

So, interactive programs are written in Haskell by using types to distinguish pure expressions from impure actions that may involve side effects.

IO a The type of actions
that returns a value of type a.

For example: IO Char IO ()

✦ Some IO Actions exported by Prelude

The action getChar :: IO Char
(1) reads a character from the keyboard,
(2) echoes it to the screen, and
(3) returns the character as its result value.

The function putChar :: Char -> IO ()

- accepts a character, and returns an action that
    - writes the character to the screen, and
    - returns no result value.

The function return :: a -> IO a

- accepts a value of type a, and returns an action that
    - simply returns the value, without performing any interaction

✦ do a sequence of actions

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

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

✦ Some IO Actions exported by Prelude

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)
```

Writing a string to the screen:

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

Writing a string to the screen and move to a new line:

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

✦ A Simple 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"
```

```
ghci> strlen
Enter a string: Haskell
The string has 7 characters
```

✦ An Example: Hangman 游戏

The rules:

- One player secretly types in a word.
- The other player tries to deduce the word, by entering a

sequence of guess.

- For each guess, the computer indicates which letters in the secret word occur in the guess.
- The game ends when the guess is correct.

```
ghci> hangman
Think of a word:
-------
Try to guess it:
? pascal
-as--ll
? rust
--s----
? haspell
has-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: "
             -- get a string secretly
             word <- sgetLine
             putStrLn "Try to guess it:"
             play word -- play the game
```

The action sgetLine reads a line of text from the keyboard, echoing each character as a dash:

```
sgetLine :: IO String
sgetLine = do
     x <- getCh -- get a char without echoing
     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:

```haskell
import System.IO (hSetEcho, stdin)

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

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

✦ An Example: Nim 游戏

The Rules:

- 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.

Board的表示和显示:

```haskell
type Board = [Int]

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

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

```haskell
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
```

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

```
ghci> putBoard initial
1: * * * * *
2: * * * *
3: * * *
4: * *
5: *
```

游戏中的一次操作：从某行删除若干个星号

- 判断一次操作是否合法

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

-- (!!) :: [a] -> Int -> a
-- List index (subscript) operator, starting from 0
-- (exported by Prelude)
```

- 进行一次操作

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

游戏入口函数:

```
nim :: IO
nim = play initial 1

play :: Board → Int → IO ()
play board player =
   do newline
      putBoard board
      newline
      if finished board then
         do putStr "Player "
            putStr $ show $ next player
            putStrLn " wins!!"
      else
         do putStr "Player "
            putStrLn $ show player
            row ← getDigit "Enter a row number: "
            del ← getDigit "Stars to remove: "
            if valid board row del then
               play (move board row del) (next player)
            else
               do newline
                  putStrLn "ERROR: Invalid move"
                  play board player
```

作业01

Define an action adder :: IO () that reads a given number of integers from the keyboard, one per line, and displays their sum.

For example:

```
ghci> adder
How many numbers? 5
1
3
5
7
9
The total is 25
```

作业02

Download the source codes of the two games (hangman and nim) from the following website:

http://www.cs.nott.ac.uk/~pszgmh/pih.html

read the codes carefully, and run them using ghci.