Adapted from Graham’s Lecture slides.

第十章：交互式程序设计

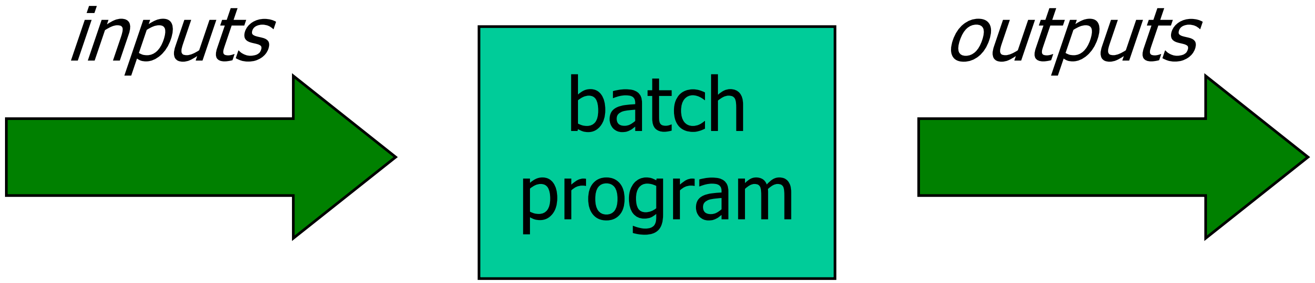
基本概念

基本/复合的交互操作

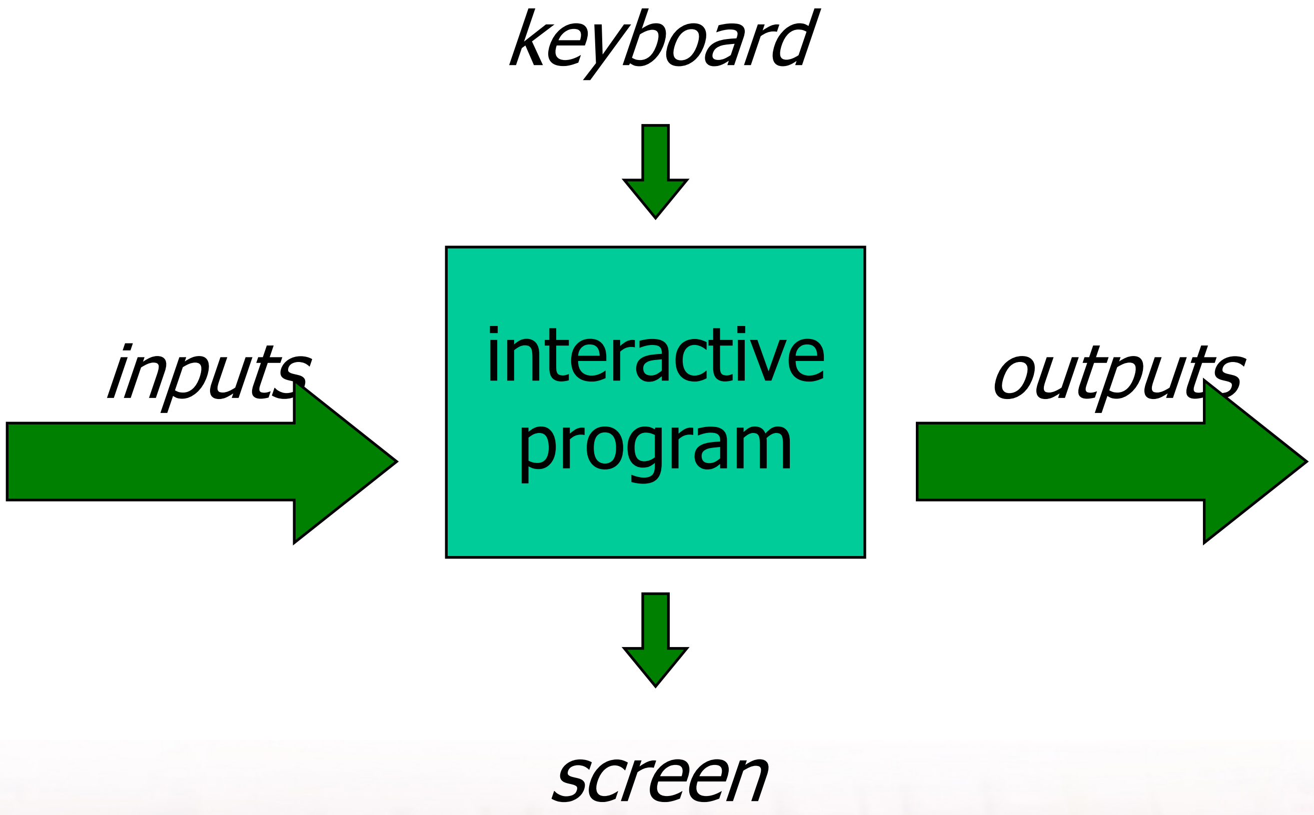
两个游戏（Hangman, Nim)

交互式编程

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.



用Haskell进行交互式编程的难点

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.

解决方法

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, interactove 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 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.

基本的交互操作

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
```

一组交互操作的组合：sequence

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)
```

- 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 and moving to a new line:

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

例子

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.

应用1: Hangman游戏

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:

```
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 xs ys =
   [if elem x ys then x else '-' | x ← xs]
```

For example:

```
> match "haskell" "pascal"

"-as--ll"
```

应用2: Nim游戏

Nim的游戏规则：

- 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

```
type Board = [Int]

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

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

```
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
```

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

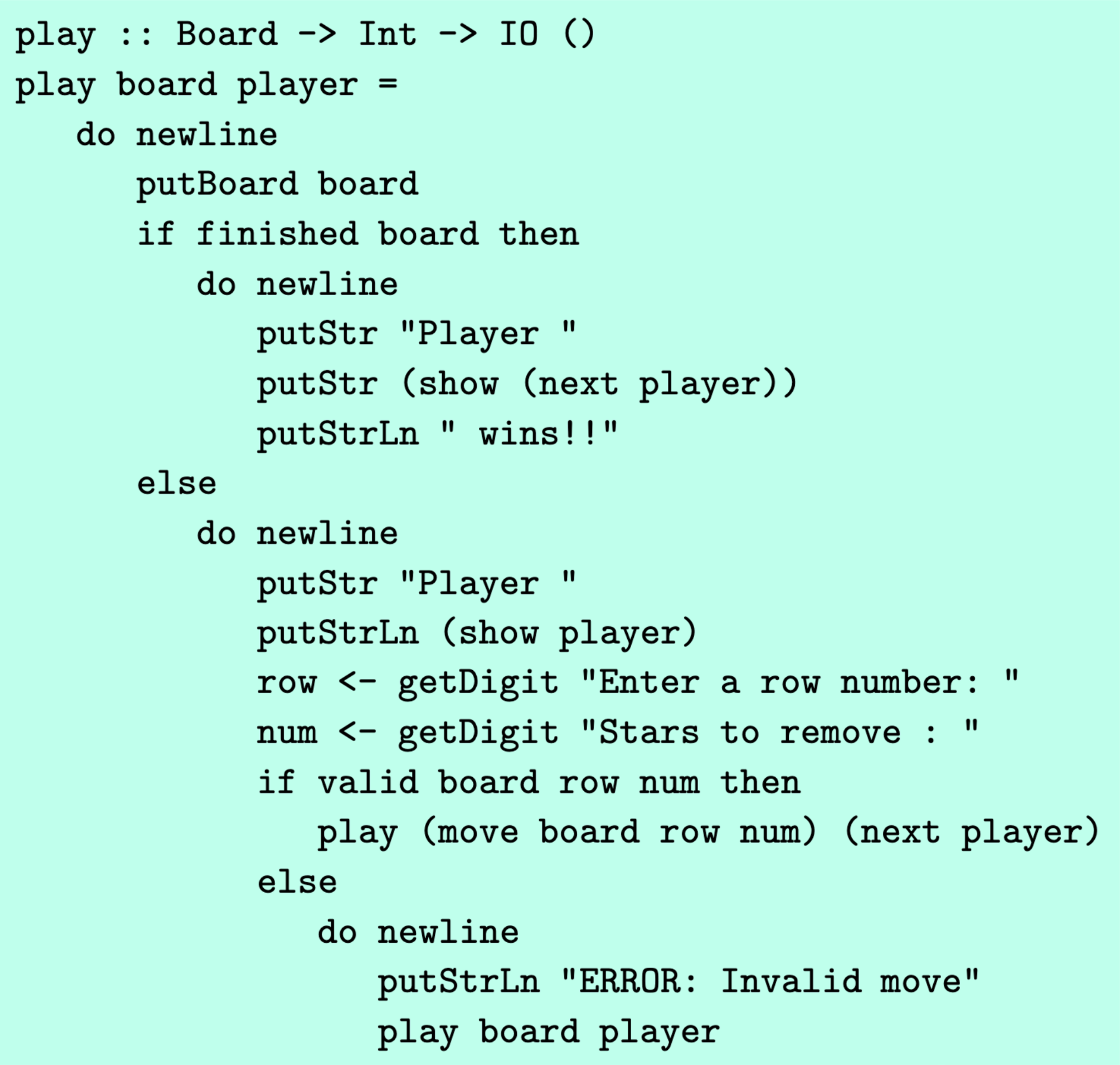
练习： 给出 putRow 的类型和定义。

游戏的一步（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
```

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

游戏的整体



```
play :: Board -> Int -> IO ()
play board player =
   do newline
      putBoard board
      if finished board then
         do newline
            putStr "Player "
            putStr (show (next player))
            putStrLn " wins!!"
      else
         do newline
            putStr "Player "
            putStrLn (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 newline
                  putStrLn "ERROR: Invalid move"
                  play board player
```

```
nim :: Board -> IO ()
nim = play initial 1
```

作业

10-1 Define an action adder :: IO () that reads a given number of integers from the keyboard, one per line, and displays their sum. For example:

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

Hint: start by defining an auxiliary function that takes the current total and how many numbers remain to be read as arguments. You will also likely need to use the library functions read and show.

10-2 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.