

计算概论A—实验班

函数式程序设计

Functional Programming

胡振江，张 伟

北京大学 计算机学院

2022年09~12月

# 第10章：交互式程序设计

## Interactive Programming

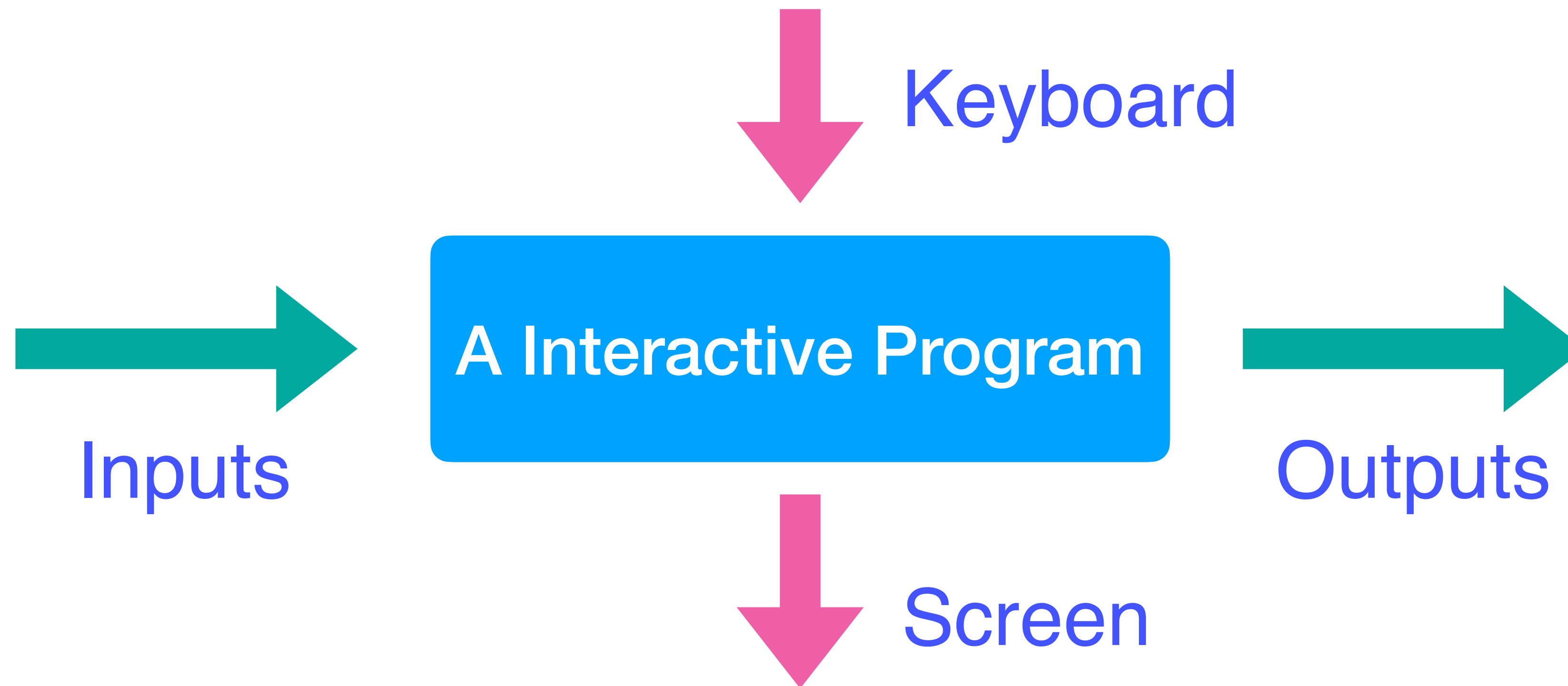
# Batch Programs

- ✿ 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.



# 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.



# Interactive Programs in Haskell: Difficulties

- ✿ 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**

# 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)
```

# A solution that looks perfect

- ❖ 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 return a value of type **a**.

- ❖ For example:

**IO** **char**

**IO** **()**

# Some IO Actions exported by Prelude

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

```
getChar :: IO Char
```

- ❖ The action `putChar c` (1) writes the character `c` to the screen, and (2) returns no result value:

```
putChar :: Char -> IO ()
```

- ❖ The action `return v` simply returns the value `v`, without performing any interaction :

```
return :: a -> IO a
```



# 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)
```

# Some IO Actions exported by Prelude

## 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 putStrLn "Enter a string: "
           xs <- getLine
           putStrLn "The string has "
           putStrLn (show (length xs))
           putStrLn " characters"
```

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

# 应用1：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!
```

# 应用1：Hangman 游戏

- ✿ 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
```

# 应用1：Hangman 游戏

- ❖ The action `sgetline` reads a line of text from the keyboard, echoing each character as a dash:

```
sgetline :: IO String
sgeline = do
  x <- getch -- get a char without echoing
  if x == '\n' then
    do putchar x
    return []
  else
    do putchar '-'
    xs <- sgetline
    return (x:xs)
```

# 应用1：Hangman 游戏

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

```
import System.IO (hSetEcho, stdin)
...
getCh :: IO Char
getCh = do hSetEcho stdin False
           x <- getChar
           hSetEcho stdin True
           return x
```



# 应用1：Hangman 游戏

- ✿ 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
```

```
match :: String -> String -> String
```

```
match xs ys =
```

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

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

# 应用2: 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

```
type Board = [Int]
```

```
initial :: Board
```

```
initial = [5,4,3,2,1]
```

```
finished :: Board -> Bool
```

```
finished = all (== 0)
```

# Board的表示和显示

```
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
```

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

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

# 游戏中的一步/一次操作：从某行删除若干个星号

## 判断一次操作是否合法

```
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
```

```
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
```

```
nim :: 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:

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



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.

# 第10章：交互式程序设计

## Interactive Programming

**就到这里吧**



























