# PROGRAMMING IN HASKELL

Chapter 10 - Interactive Programming

Introduction

To date, we have seen how Haskell can be used to write <u>batch</u> programs that take all their inputs at the start and give all their outputs at the end.

inputs → batch program → outputs

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   interactive program   *outputs*
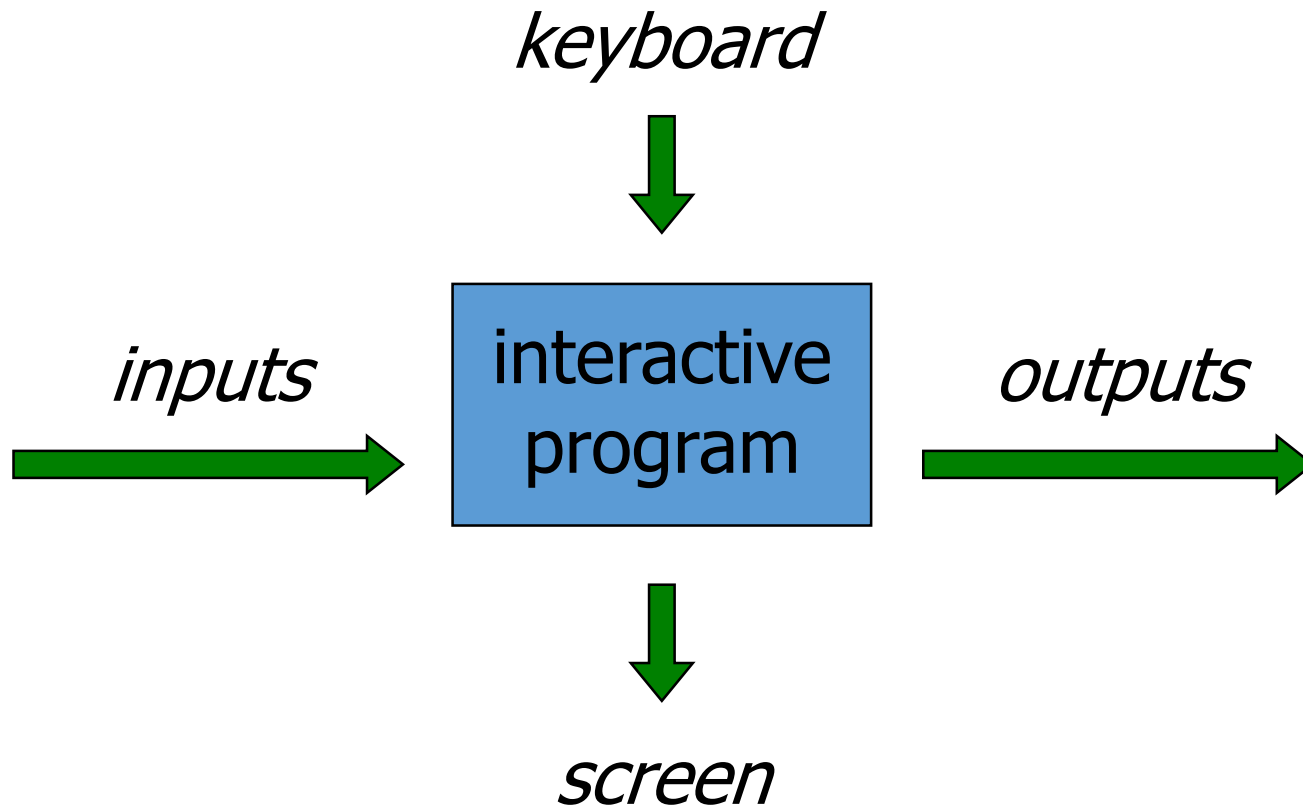
screen

The Problem

Haskell programs are pure mathematical functions:

    z Haskell programs <u>have no side effects</u>.

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

    z Interactive programs <u>have side effects</u>.

The Solution

Interactive programs can be written in Haskell by using types to distinguish pure expressions from impure <u>actions</u> 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 <u>no</u> result value.

Note:

z () is the type of tuples with no components.

Basic Actions

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

z The action <u>getChar</u> reads a character from the keyboard, echoes it to the screen, and returns the character as its result value:

```
getChar :: IO Char
```

z The action <u>putChar c</u> writes the character c to the screen, and returns no result value:

```
putChar :: Char → IO ()
```

z The action <u>return v</u> 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 <u>do</u>.

For example:

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

Derived Primitives

z 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'
```

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:

z Evaluating an action <u>executes</u> its side effects, with the final result value being discarded.

Hangman

Consider the following version of <u>hangman</u>:

- z One player secretly types in a word.

- z The other player tries to deduce the word, by entering a sequence of guesses.

- z For each guess, the computer indicates which letters in the secret word occur in the guess.

z The game ends when the guess is correct.

We adopt a <u>top down</u> 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 <u>sgetLine</u> 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 <u>getCh</u> 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 <u>play</u> 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
```

The function <u>match</u> 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"
```

Exercise

Implement the game of <u>nim</u> in Haskell, where the rules of the game are as follows:

z The board comprises five rows of stars:

```
1: * * * * *
2: * * * *
3: * * *
4: * *
5: *
```

z   Two players take it turn about to remove one or more stars from the end of a single row.

z   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].