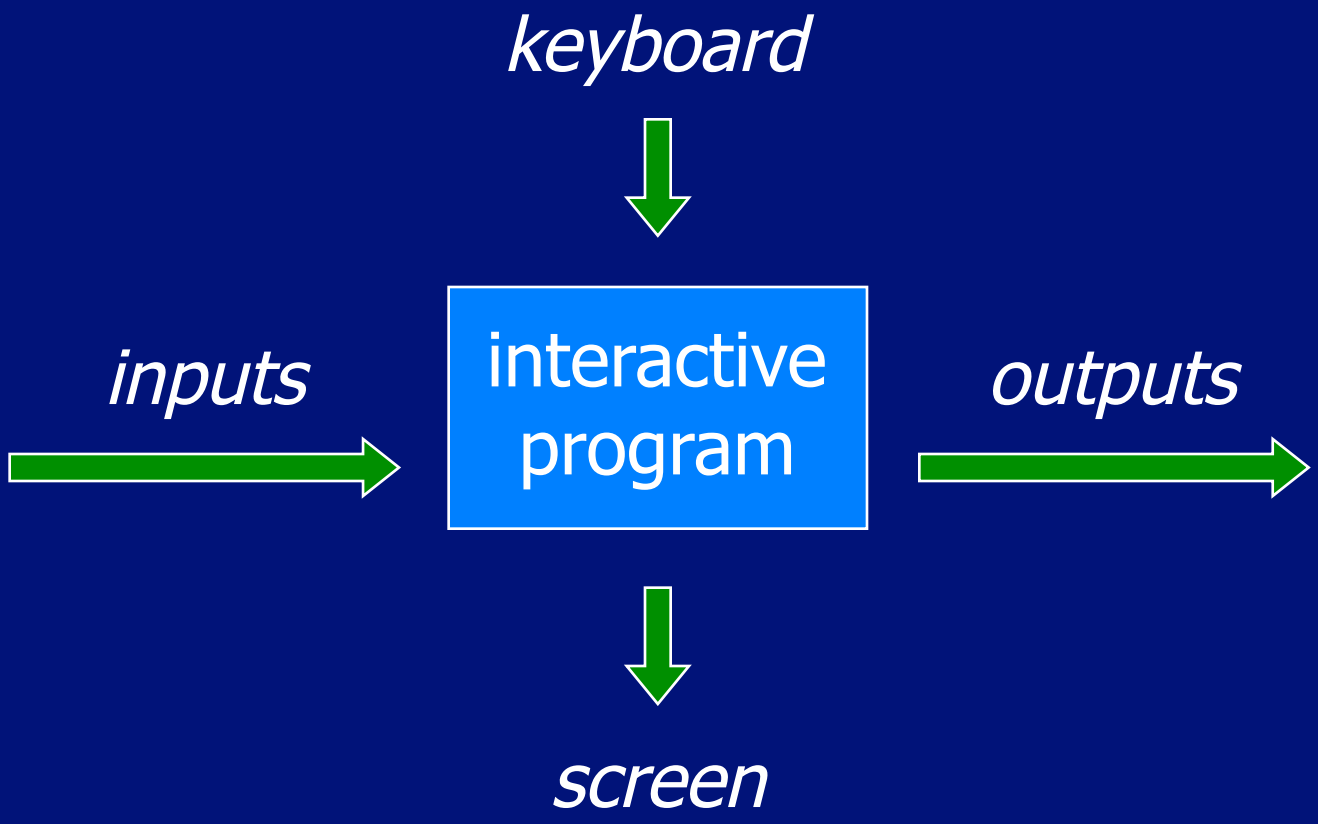
PROGRAMMING IN HASKELL

Chapter 10 - Interactive Programming

Introduction

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, for example, read from the keyboard and write to the screen, as they are running.



The Problem

Haskell programs are pure mathematical functions:

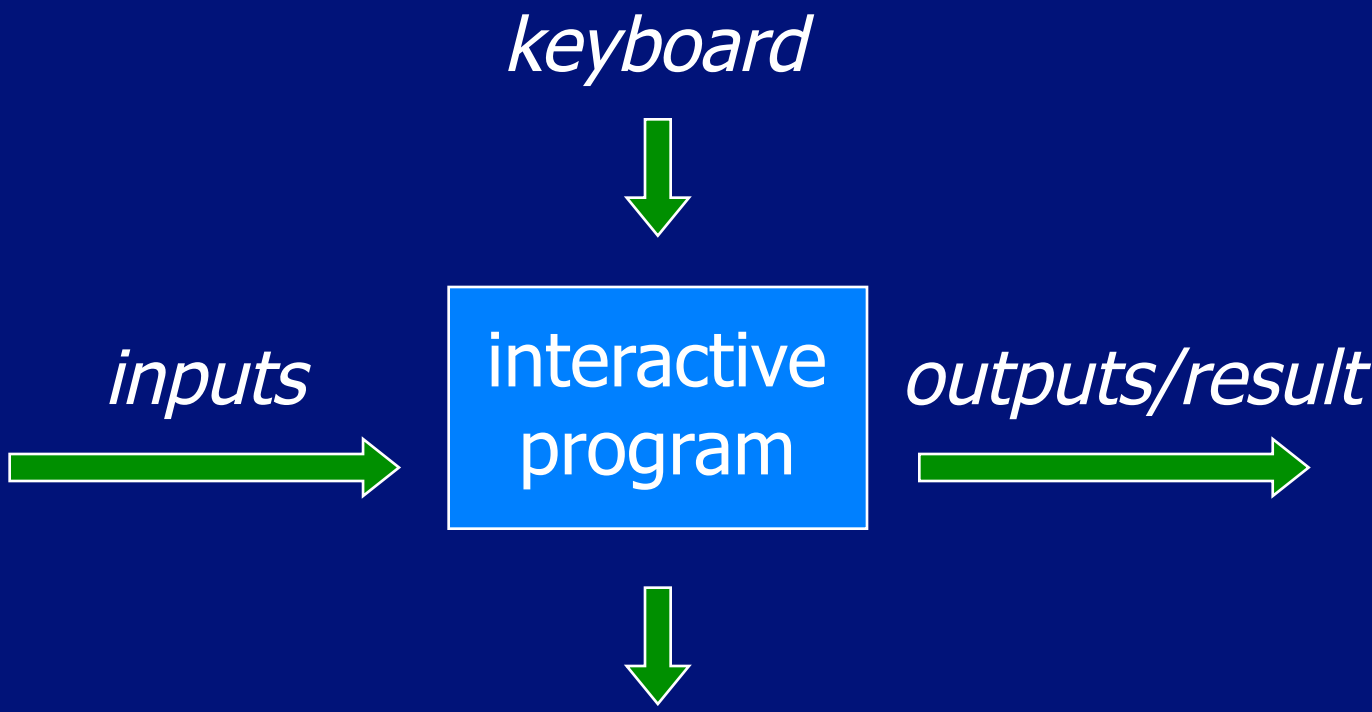
- Haskell programs have no side effects.

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

- Interactive programs have side effects.

There can be a range of sources to read from as well as side effect results, not just keyboard and screen.

Note that the keyboard and screen are part of the interaction; keyboard is input; screen is the side effect. The output is the functional result of applying the functions to the inputs/keyboard.



The Solution

Interactive programs can be viewed as a pure function that takes a current state of the world as its argument and produces a modified world as its result.

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

The Solution

Interactive programs generally return a result in addition to the side effects.

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

Expressions of type IO are called actions.
a is the type of the result.

The Solution

Interactive programs can be 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 (e.g. show something on screen) that return no result value (e.g. output of function).

Note:

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

Basic Actions

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
```

- This function bridges between pure expressions without side effects and impure expressions with side effects.

Sequencing

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

```
do v1 ← a1
   v2 ← a2
    .
    .
    .
   return (f v1 v2 ... vn)
```

do action a1 with result v1, etc. In the end, apply the function f to the results of all the actions. f combines the results to one result for the expression.

Sequencing

vn ← an are generators.

```
do v1 ← a1
   v2 ← a2
   .
   .
   .
   return (f v1 v2 ... vn)
```

Sequencing

Example: reads in three characters, discards the second (no result), and returns the first and third.

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

Derived Primitives

- Reading a string from the keyboard, terminated by \n. Recursion is used to read in each character:

```
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"
```

show takes a value of basic types and converts it to strings of 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.

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.

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 to keep the word secret:

```
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 the 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, where match word guess :

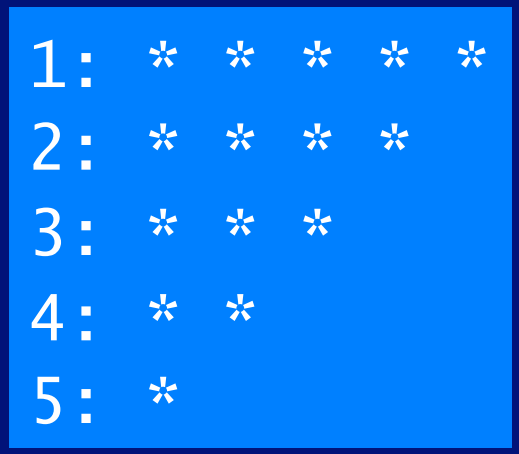
```
> match "haskell" "pascal"

"-as--ll"
```

Note “haskell” matches l twice in “pascal”:

Exercise

Implement the game of nim in Haskell, where the rules of the game are as follows:

- 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.

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].