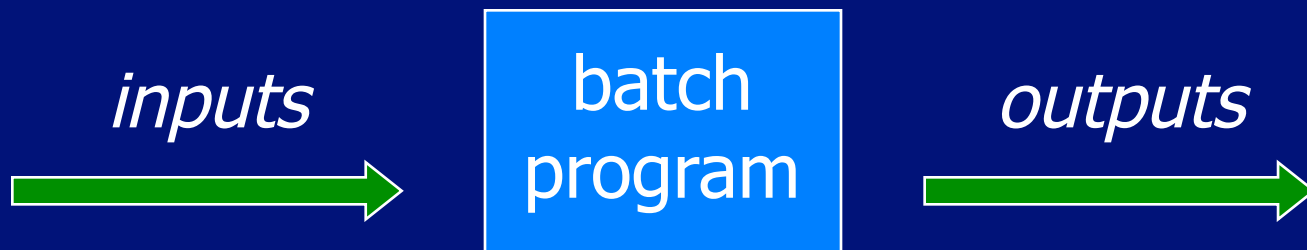
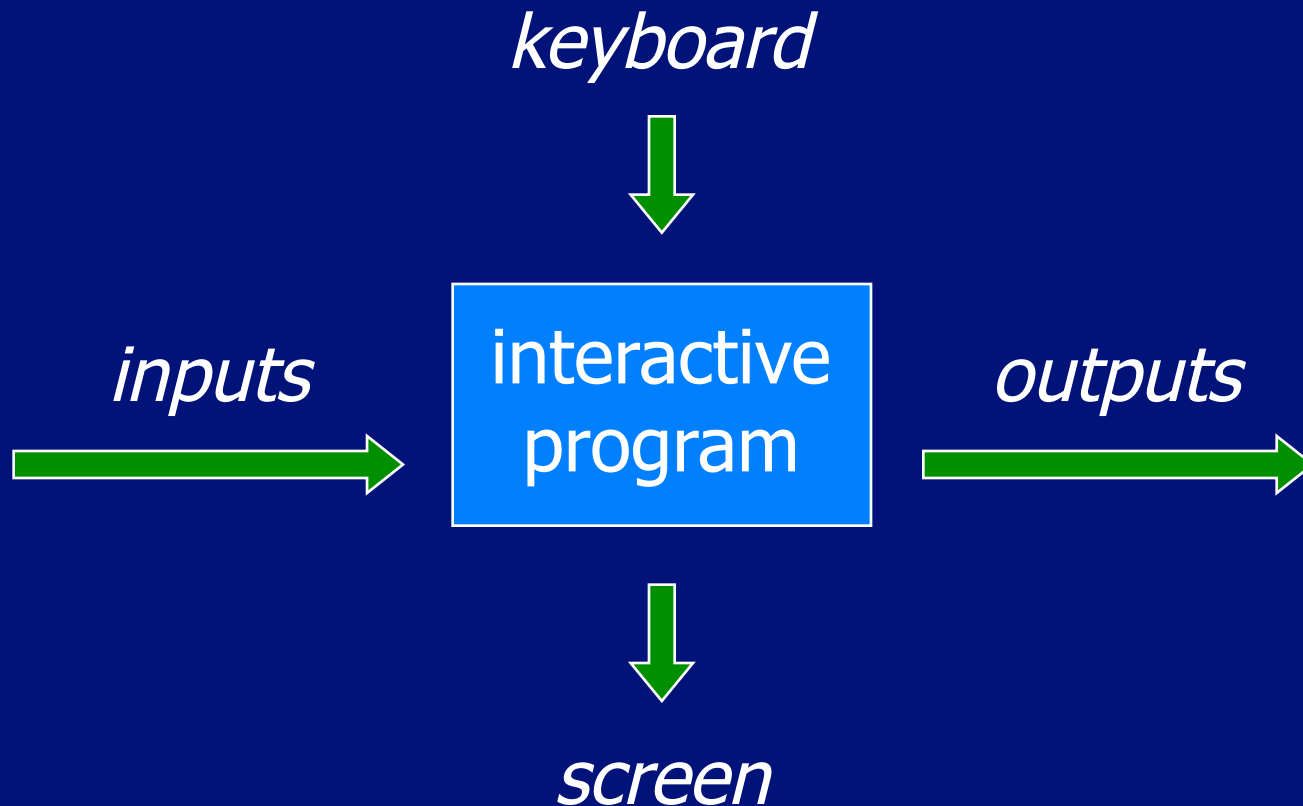


# Introduction

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 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.

# 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 that return no result value.

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
```

# Sequencing

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

For example:

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



# Derived Primitives

- 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: abcde
```

```
The string has 5 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:"
       guess word
```

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

```
sgetline :: IO String
sgeline  = 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
           c ← getChar
           hSetEcho stdin True
           return c
```



The function guess is the main loop, which requests and processes guesses until the game ends.

```
guess      :: String → IO ()
guess word =
    do putStr "> "
       xs ← getLine
       if xs == word then
           putStrLn "You got it!"
       else
           do putStrLn (diff word xs)
              guess word
```

The function diff indicates which characters in one string occur in a second string:

```
diff      :: String → String → String
diff xs ys =
    [if elem x ys then x else '-' | x ← xs]
```

For example:

```
> diff "haske11" "pasca1"
"-as--11"
```

# Exercise

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

- 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.

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].