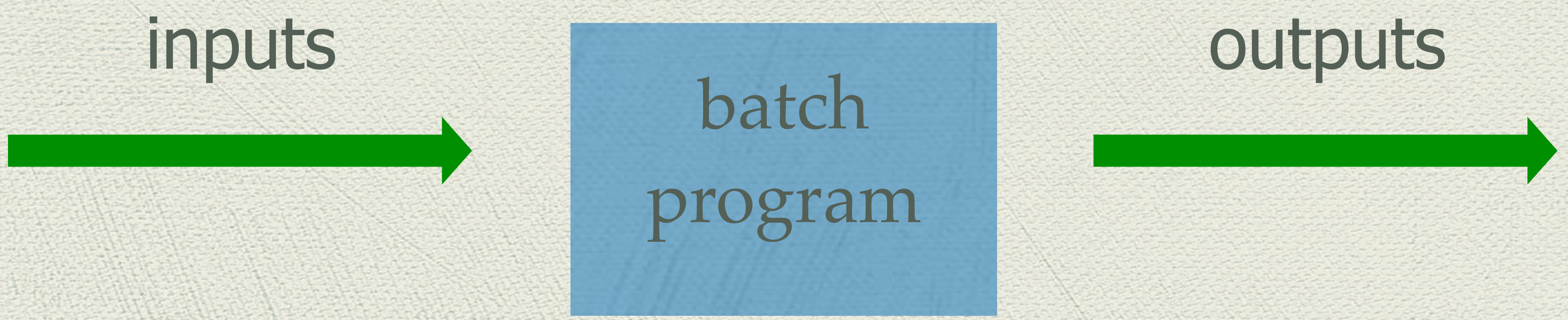
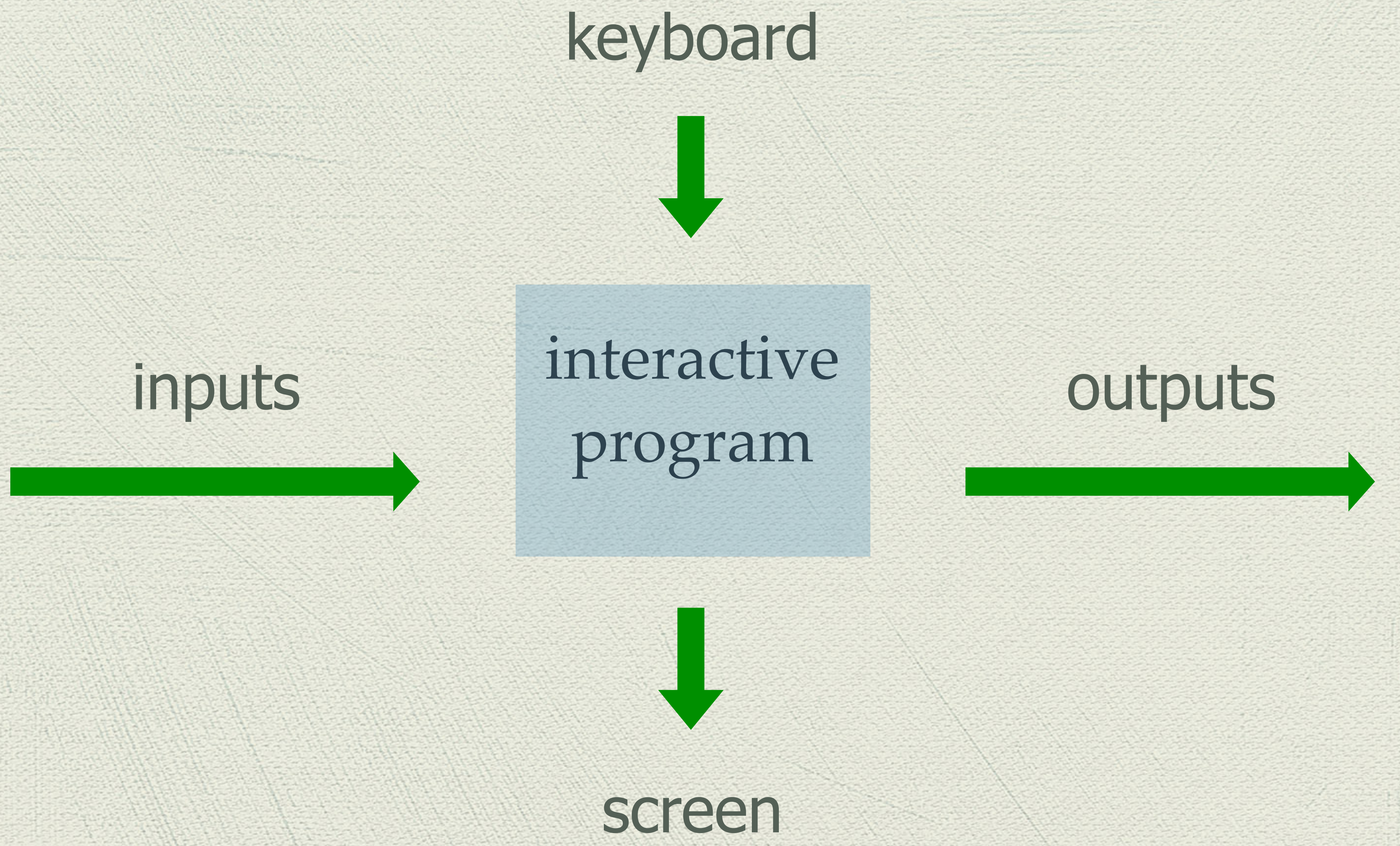
Interactive programs

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.



Interactive programs 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 <u>no</u> result value.

Note:

() is the type of tuples with no components.

Primitive 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 Actions

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

For example:

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

Note - in a sequence of actions:

- Each action must begin in precisely the same column. That is, the layout rule applies;

- The values returned by intermediate actions are discarded by default, but if required can be named using the ← operator;

- The value returned by the last action is the value returned by the sequence as a whole.

Other Library Actions

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: hello there
The string has 11 characters
```

Note:

? Evaluating an action executes its side effects, with the final result value being discarded.

doing the swapAround

```
swapAround = do line <- getLine
                if null line then return ()
                else do putStrLn $ reverseWords line


reverseWords :: String -> String
reverseWords = unwords . map reverse . words
```

unwords :: [String] -> String

words :: String -> [String]