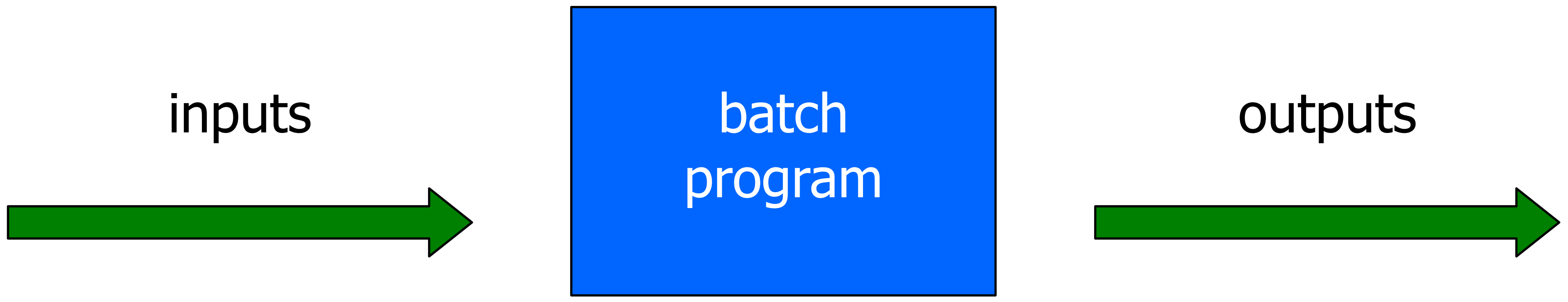
PROGRAMMING IN HASKELL

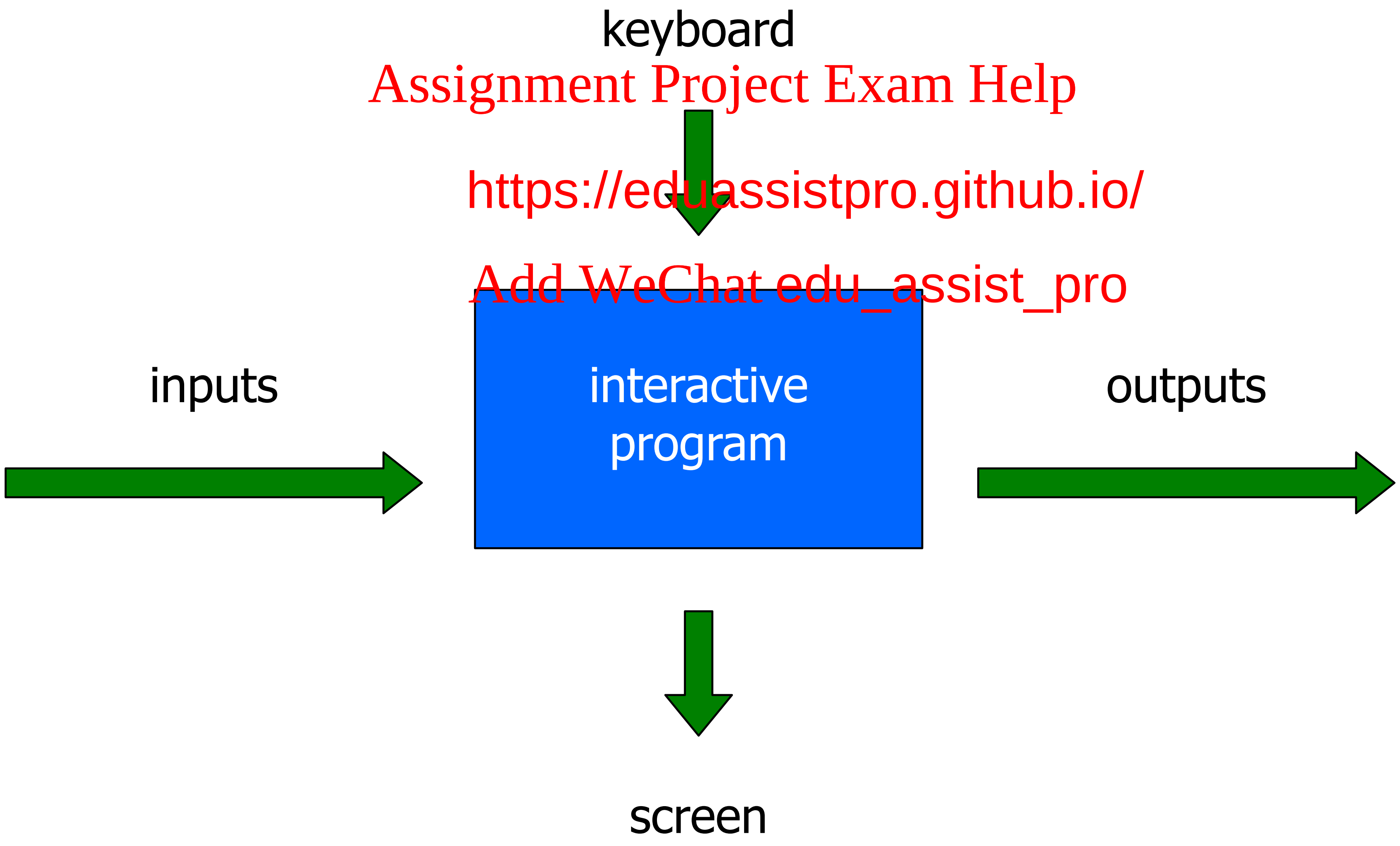
Chapter 9 - Interactive Programs

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.



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 <u>have no side effects</u>.

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

- 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 ()

T purely side effecting
ac eturn 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 yboard, echoes it to the screen, and returns the character as its r:

```
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 val ut 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:

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

do-notation again

IO also supports the do-notation and again, similarly to what happened with Parsers, we need to be careful understanding the typing of the ←

getChar :: IO Char

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

x :: Char

Derived Primitives

- Reading a string from the keyboard:

```
getLine :: IO
getLine  = d
           if x == '\n' the
              return []
           else
              do xs ← getLine
                 return (x:xs)
```

- Writing a string to the screen:

```
putStr        :: String → IO ()
putStr []      = ?
putStr (x: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
             xs ← getLine
             putStr "The string has "
             putStr (show (length xs))
             putStrLn " characters"
```

For example:

```
> strlen


Enter a string: abcde
The string h
```

Note:

Evaluating an action <u>executes</u> 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 ening a sequence of guesses.
- For each guess, the computer indicates rs 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 getCh reads a single character from the keyboard, without echoing it to the screen:

```
import System.IO

getCh :: IO C

getCh  = do hSetEcho s          e
            x ← getChar
            hSetEcho stdin True
            return x
```

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

```
sgetLine :: IO String
sgetLine  = ?
```

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

```
match :: String → String → String
match xs ys = ?
```

For example:

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

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

```
play      :: String → IO ()
play word =
   ?
```

Exercise

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

- The board comprises five

```
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].