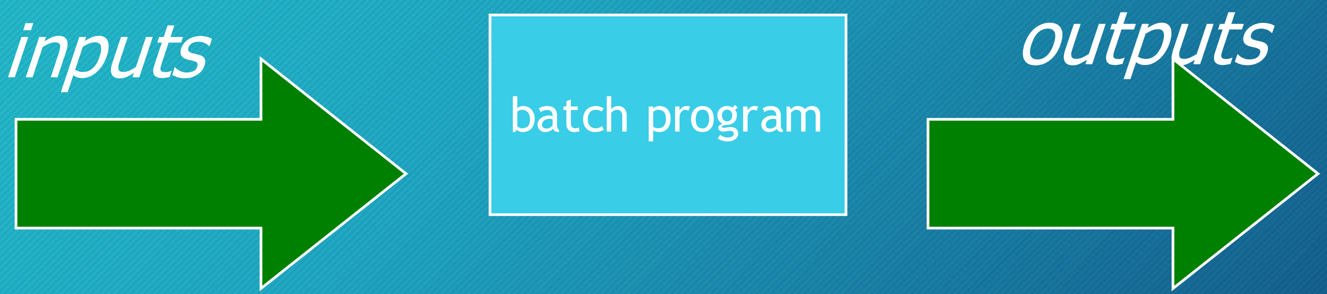
PROGRAMMING IN HASKELL

Chapter 9- Interactive Programming

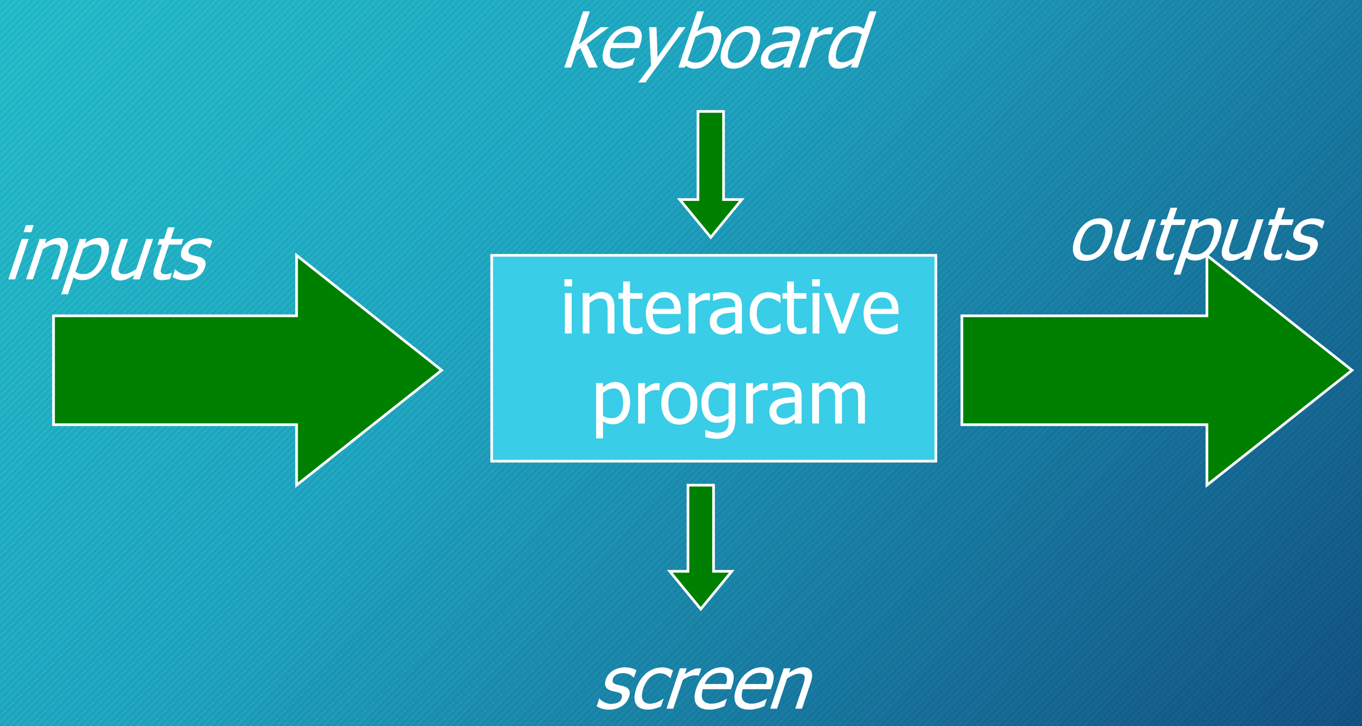
Introduction

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.



Introduction

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.



mystery's in Java ..


```java
public static int mystery1(int val1, int val2) {
 int val3 = 3;
 return (val1+val2+val3)^2;
}
```

Both have the same signature

```java
public static int mystery2(int val1, int val2) {
 int val3 = 3;
 Scanner in = new Scanner(System.in);
 System.out.println("Enter a number: ");
 try {
     val3 = in.nextInt();
 } catch (Exception e) {
     e.printStackTrace();
 }
 return (val1+val2+val3)^2;
}
```

mystery2 depends on the outside world.

Specifically, if I call mystery2(4,5) today and tomorrow, I may get different results.

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.

IO Types

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.

What is an Action?

Actions:

- Have the type IO t
- Are first-class values in Haskell – fit in seamlessly
- Produce an effect when they are performed, but not when evaluated. i.e. they produce an effect only when called by something else in an I/O context.

What is an Action?

Actions:

- Any expression may produce an action as its value, but the action will not perform I/O until it is executed inside another I/O action (or it is main)

- Performing (executing) an action of type
    - IO t

  may perform I/O and will ultimately deliver a result of type t.

The IO Type

Actions:

- Values of type

  IO a

  are

  - *descriptions of* effectful computations,
  - which, if executed would (possibly) perform some effectful I/O operations and (eventually) produce a value of type a.

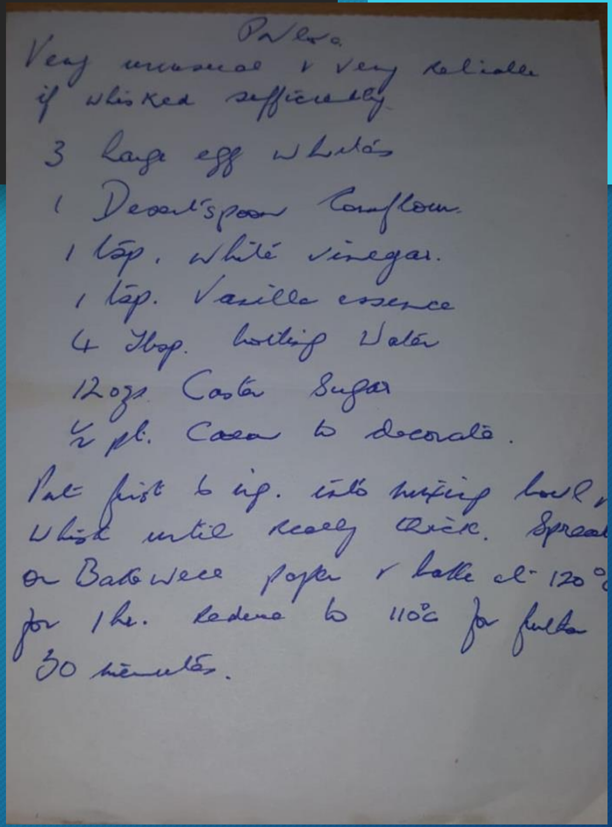
The IO Type


```
c :: Cake
```

The IO Type

Alternatively we can have

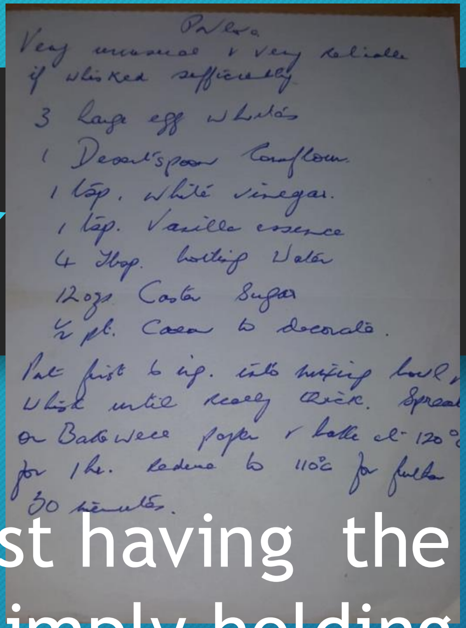
```
c :: Recipe Cake
```



What do you have? A cake?
No, you have some *instructions* for how to make a cake, just a sheet of paper with some writing on it.

The IO Type

`c :: Recipe Cake`



Not only do you not actually have a cake, just having the recipe has no effect on anything else whatsoever. Simply holding the recipe in your hand does not, for instance:

- Cause your oven to get hot or
- Put all ingredients in the bowl or
- Mix the ingredients

To actually produce a cake, the recipe must be followed (ingredients bought, causing flour to be spilled, ingredients mixed, the oven to get hot, *etc.*).

The IO Type

In the same way, a value of type IO a is just a “recipe” for producing a value of type a (and possibly having some effects along the way). Like any other value, it can be passed as an argument, returned as the output of a function, stored in a data structure, or (as we will see shortly) combined with other IO values into more complex recipes.

Click to see video

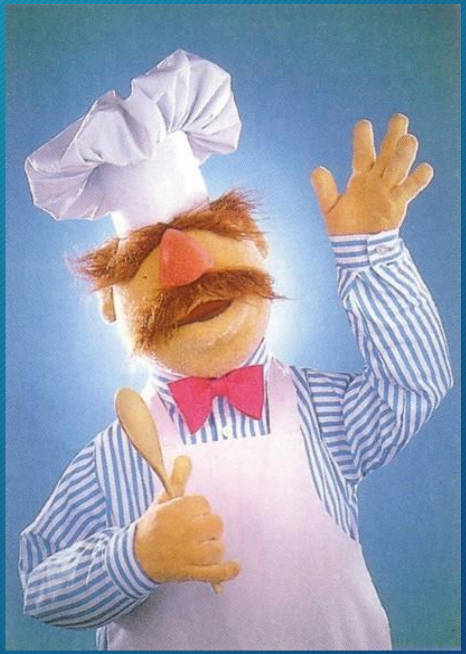
main

- So, how do values of type IO a actually ever get executed? (Apart from when we use the interactive GHCI)
- There is only one way: the Haskell compiler looks for a special value
- which will actually get handed to the runtime system and executed. That's it!

```
main :: IO ()
```

main

Think of the Haskell runtime system as a master chef who is the only one allowed to do any cooking.



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
```

Basic Actions


```
getChar :: IO Char
```

- Look at it's type – it looks like a value rather than a function – actually, getChar stores an I/O action. When it's performed you get a Char.

- The <- operator is used to "pull out" the result from performing the I/O action and store it in the variable.

Basic Actions

- 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 <u>do</u>.

For example:

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

do notation

The do notation give us a way of building IO programs from the components that we have seen. It does two things :

- It is used to sequence I/O programs
- It is used to name the values returned by the IO actions – this means that the later actions can depend on values captured earlier in the program

Derived Primitives

- Reading a string from the keyboard:

if .. then .. else

```
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'
```

Other examples

- Writing a string four times:

```
put4times :: String → IO ()
put4times str
      = do putStr str
           putStr str
           putStr str
           putStr str
```

Spaces not tabs!

Other examples

Reading input

```
read2lines :: IO ()
read2lines
      = do getLine
      getLine
      putStrLn "two lines read"
```

read2lines is applied to 2 arguments (the two lines that are typed in)

Other examples

- Using the input

```
getNput :: IO ()
getNput = do line <- getLine
             putStrLn line
```

- Note that in Haskell, each

  var <-

  creates a new variable – so, this 'single assignment' is allowed (rather than 'updatable assignment')

- line <- getLine acts as a local definition

Other examples

We can’t change line but we can use it differently:

```
reverse2lines :: IO ()
reverse2lines =
        do line1 <- getLine
           line2 <- getLine
           putStrLn(reverse line1)
           putStrLn(reverse line2)
```

Other examples

- We can also use local definitions:

Only used to name IO actions

```
reverse2lines2 :: IO ()
reverse2lines2 =
      do line1 <- getLine
         line2 <- getLine
         let rev1 = reverse line1
         let rev2 = reverse line2
         putStrLn(rev1)
         putStrLn(rev2)
```

Local definitions

Pure Versus I/O

Pure code ensures that Haskell functions return the same result when given the same input and have no side effects. We use the execution of I/O actions to avoid these rules

Pure	Impure
Always produces the same result when given the same parameters	May produce different results for the same parameters
Never has side effects	May have side effects
Never alters state	May alter the global state of the program , system or world

Why purity matters

- In other languages (e.g. C, Java) cannot be sure of no side effects (need to read documentation, hope it's accurate).
- Many bugs are caused by unanticipated side effects.
- This can cause cascading effects when using multi-threading, other forms of parallelisms.
- In Haskell, isolating side effects into I/O actions provides a clear boundary between no side effects and potential side effects.

Back to mystery's in Java .. recap


```java
public static int mystery1(int val1, int val2) {
 int val3 = 3;
 return (val1+val2+val3)^2;
}
```

```java
public static int mystery2(int val1, int val2) {
 int val3 = 3;
 Scanner in = new Scanner(System.in);
 System.out.println("Enter a number: ");
 try {
     val3 = in.nextInt();
 } catch (Exception e) {
     e.printStackTrace();
 }
 return (val1+val2+val3)^2;
}
```

mystery's in Haskell ..


```
mystery1:: Int -> Int -> Int
mystery1 val1 val2 = (val1 + val2 + val3)^2
         where val3 = 3
```

No IO involved here

It is clear where IO is involved

```
mystery2 :: Int -> Int-> IO Int
mystery2 val1 val2 = do
       putStrLn "Enter a number"
       val3Input <- getLine
       let val3 = read val3Input::Int
       return ((val1 + val2 + val3)^2)
```

IO involved here

More Examples

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"
```

More Examples


```
> 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.

Hangman

- 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
```

Hangman

The action sgetLine 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)
```

Hangman

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
```

Hangman

The function play is the main loop, which requests and processes 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
```

Hangman

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:

```
> match "haskell" "pascal"

"-as--ll"
```

Reading values in general

Haskell contains the class Read with the function

```
read :: Read a => String -> a
```

This can be used to parse a string representing a value of a particular type into that value

Example using read

- Suppose that we want to read an I/O program to read in an integer value.
- To read an integer from a line of input, we start by

  `do line <- getLine`

- Then we need to sequence this with an I/O action to return the *line* interpreted as an *Integer*.
- We can convert the *line* to an integer by the expression

  `read line`

Example using read

- What we need is the IO Integer action which returns this value – this is the purpose of return.
- Our program to read an integer is

```
getInt :: IO Integer
getInt =
    do line <- getLine
       return (read line)
```

The compiler can figure out it’s an Integer from this

Compiler is expecting an Integer to be returned so needs return

Being more specific

More generally, remember that

```
Read a => String -> a
```

So we can specify directly into which type we wish to cast:

```
read "1" :: Int
```

Or.. E.g. Inside a do block ..

```
..
st1 <- getLine
let int1 = read st1 :: Int
..
```

Add two numbers using show

- Recall that to use a non-string type as a string, we use **show**

- Add two numbers and print result

```
add :: IO ()
add = do   putStrLn "Enter two numbers"
       numstr1 <- getLine
       numstr2 <- getLine
       let num1 = read numstr1 :: Int
       let num2 = read numstr2 :: Int
       putStrLn ("Sum is "++ show (num1 + num2))
```

Alternative using getInt

Using getInt (which already uses read to make into Integer)

```
add :: IO ()
add = do  putStrLn ”Enter two numbers"
      num1 <- getInt
      num2 <- getInt
      putStrLn ("Sum is "++show (num1+num2))
```

References

- Programming in Haskell, Graham Hutton
- https://www.seas.upenn.edu/~cis194/fall16/index.html
- Get Programming with Haskell, Will Kurt, Mannning, ISBN 9781617293764.


ANY
QUESTIONS?