Function Programming in Haskell

Prof. Madhavan Mukund and S. P. Suresh

Chennai Mathematical Institute

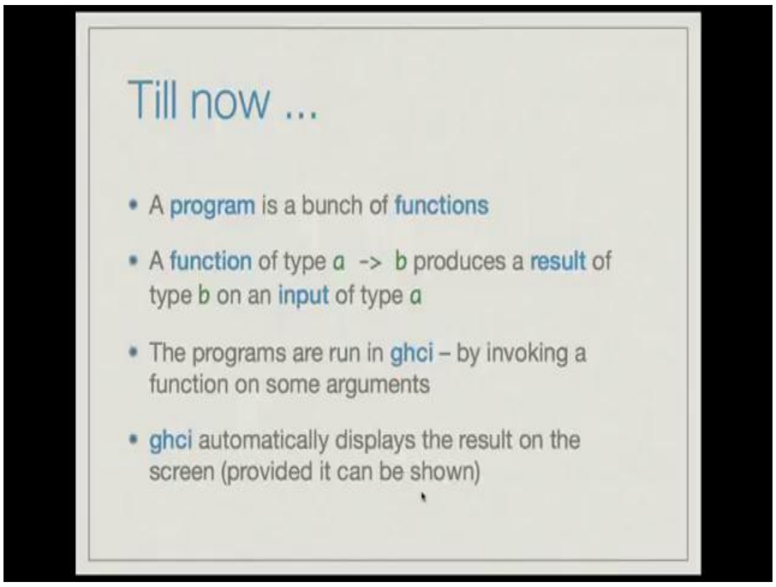
Module # 07

Lecture - 02

Input / Output

In this lecture, we shall study Input Output in Haskell.

(Refer Slide Time: 00:08)

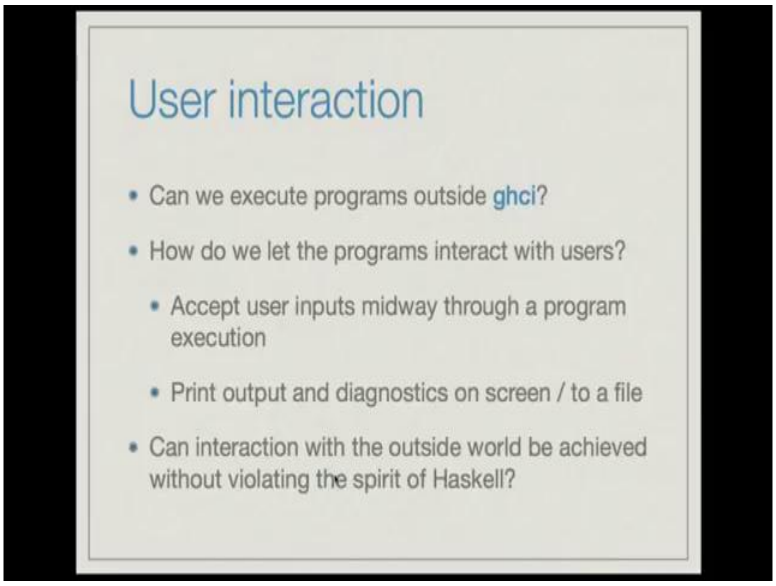


Till now ...

- A program is a bunch of functions
- A function of type a -> b produces a result of type b on an input of type a
- The programs are run in ghci – by invoking a function on some arguments
- ghci automatically displays the result on the screen (provided it can be shown)

Till now, the view we have taken is that a program is a bunch of functions, a function of type a arrow b produces a result of type b on an input of type a. The programs are running ghci by invoking a function on some arguments, ghci automatically displays the result on the screen provided the result can be shown or in other words, provided the result is a type that belongs to the type class show.

(Refer Slide Time: 00:46)

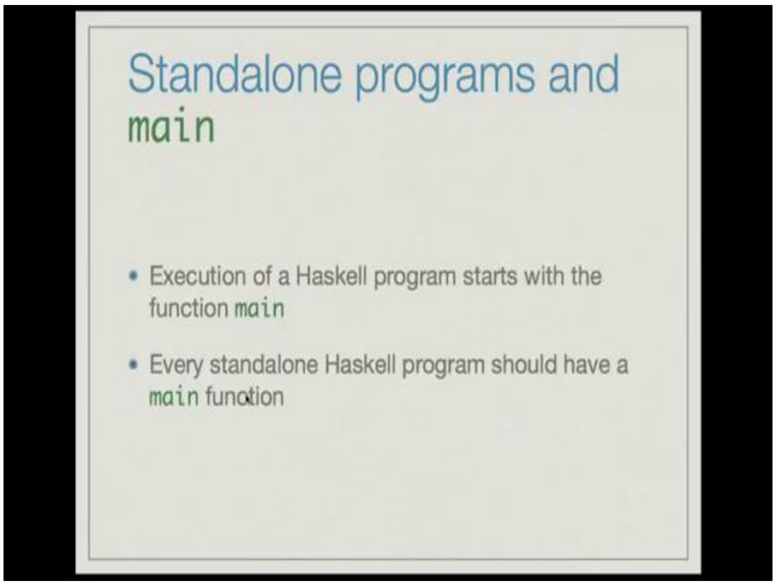


User interaction

- Can we execute programs outside ghci?
- How do we let the programs interact with users?
  - Accept user inputs midway through a program execution
  - Print output and diagnostics on screen / to a file
- Can interaction with the outside world be achieved without violating the spirit of Haskell?

But, this is the limited form of interaction with the user, we would like a slightly better user interaction model like another program languages. So, the questions we have can we execute programs outside ghci, how do you let the programs interact with users that is accept user inputs midway through a program execution. Print output and diagnostics on a screen or to a file, can interaction with the outside world be achieved without violating the spirit of Haskell. So, these are some other questions that we considered in this lecture.

(Refer Slide Time: 01:25)

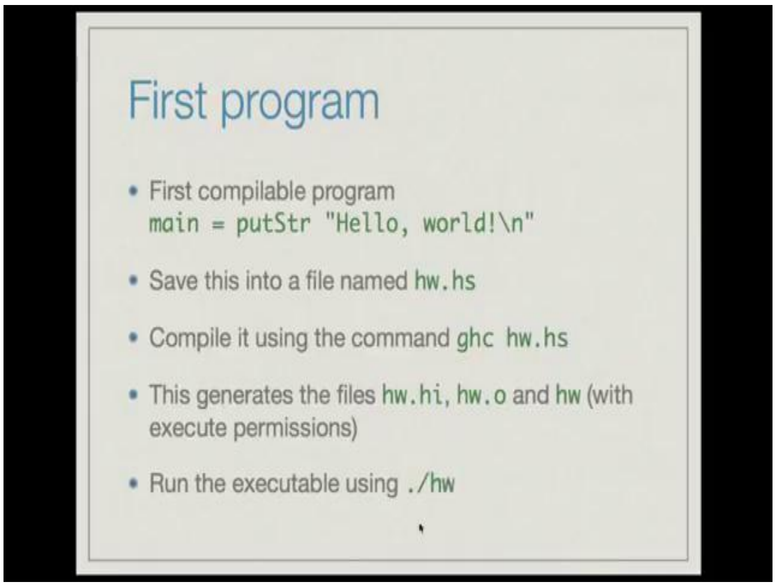


Standalone programs and main

- Execution of a Haskell program starts with the function main
- Every standalone Haskell program should have a main function

We shall first consider standalone programs, execution of a Haskell program starts with the function main, this is not something we have seen till now. Till now, our program was disturbance, the functions and ghci automatically interpreted any function that we entered in it. But, if you have to write a standalone program, execution has to start with some place and that is the function being, every standalone Haskell program should have a main function.

(Refer Slide Time: 02:03)



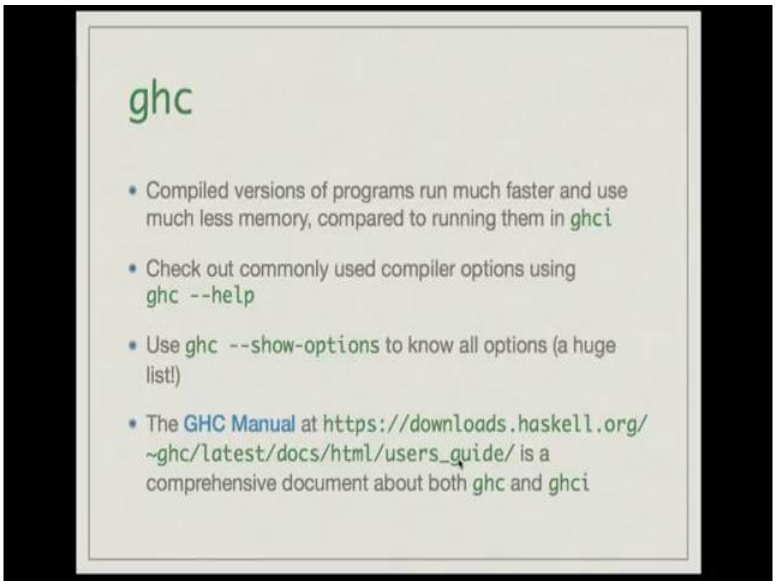
First program

- First compilable program
  `main = putStr "Hello, world!\n"`
- Save this into a file named `hw.hs`
- Compile it using the command `ghc hw.hs`
- This generates the files `hw.hi`, `hw.o` and `hw` (with execute permissions)
- Run the executable using `./hw`

Here is an example program, in fact this is the simplest compliable program that we can write, main equals putStr which stands for putStr Hello world with the new line character here. So, this as the name suggest put a string on the screen, so the way we run this is to save this in to a file named hw dot hs, hw standing for Hello world. We compile this file using the command ghc hw dot hs. Compiling this generate the following files, hw dot hi, hw dot o and hw, the file of interest was the hw without any extension.

If you see this on a unique terminal, we will see that the permissions for these two are retried, but hw has execute permissions or in other words 7 5 5. We can run the executable using dot slash hw, dot denotes the current directory, dot slash hw means that the executable files hw can be found in the current directory. If you add the path, where your Haskell executable reside your path environment, you can just invoke the functions using hw, but for now in the examples we will just use dot slash programming.

(Refer Slide Time: 03:45)



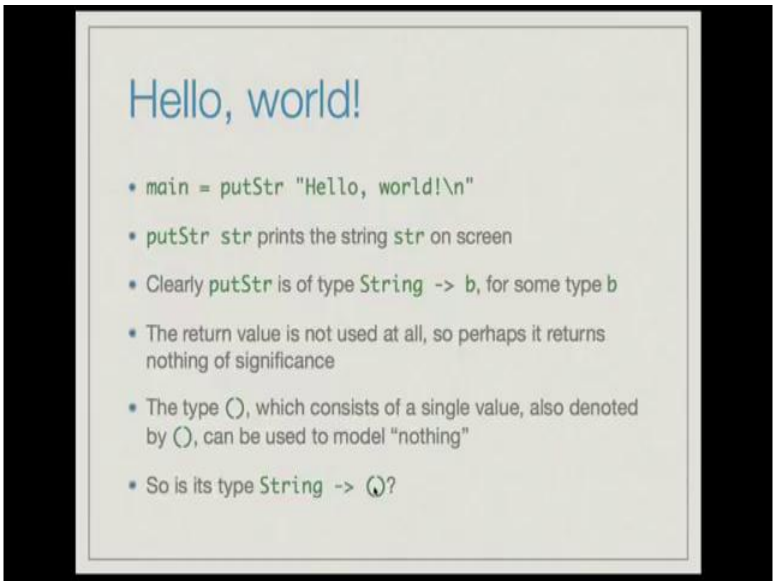
ghc

- Compiled versions of programs run much faster and use much less memory, compared to running them in ghci
- Check out commonly used compiler options using `ghc --help`
- Use `ghc --show-options` to know all options (a huge list!)
- The GHC Manual at `https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/` is a comprehensive document about both ghc and ghci

GHC is the Glasgow Haskell Compiler, ghci that we have been using till now is the interactive version of the complier, one can view ghci as an interpreter or a play ground in which to test your programs. If the program is indented for use by others, then it is usually written as a standalone program, complied using ghc and shipped. Complied versions of programs run much faster and use much less memory as compare to running them in ghci.

If you check the various options that ghc offers by typing ghc minus, minus help in the terminal, you can use ghc minus, minus show options to know all the options that you can provide, all the complier options that you may provide to ghc, but this is a huge list. If you want to know more about ghc and ghci, you can consult the GHC Manual at this url, which is the part of the official Haskell page. So, we have learnt how to write a simple program which standalone and turns on it and compile it and run it.

(Refer Slide Time: 05:05)

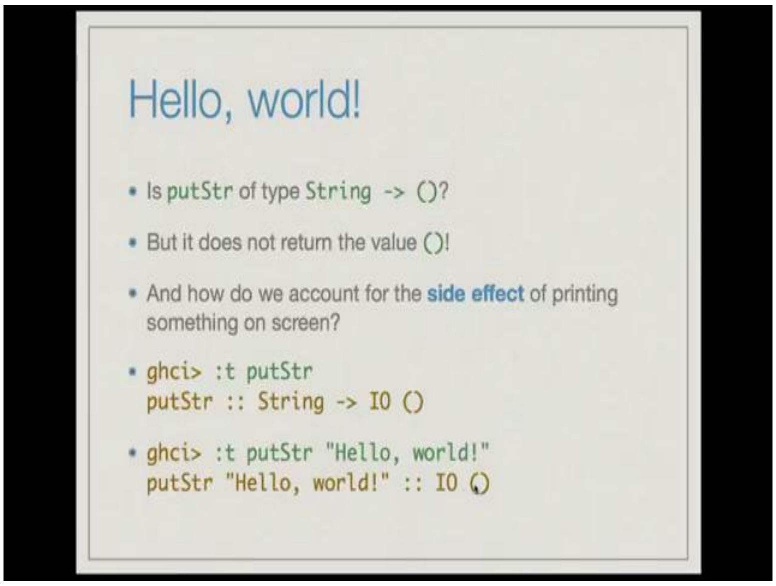


Hello, world!

- `main = putStr "Hello, world!\n"`
- `putStr str` prints the string `str` on screen
- Clearly `putStr` is of type `String -> b`, for some type b
- The return value is not used at all, so perhaps it returns nothing of significance
- The type (), which consists of a single value, also denoted by (), can be used to model "nothing"
- So is its type `String -> ()`?

Let us study the program in more detail, main equals putStr Hello world with the new line character behind, put string is a function, the behavior is that putStr str putStr of string print the string str on screen. So, clearly putStr is of type string arrow b for some b, because the input is a string, but you notice that in this main program the return value is not used at all. So, perhaps we can say that putStr does not return anything of significance, the type null or empty as it is called denotes nothing or it can be used to model nothing. This type empty is denoted like an empty tuple and it consists of single value, which is also denoted by an empty tuple, so the question is, is the type of putStr string arrow empty.

(Refer Slide Time: 06:25)



Hello, world!

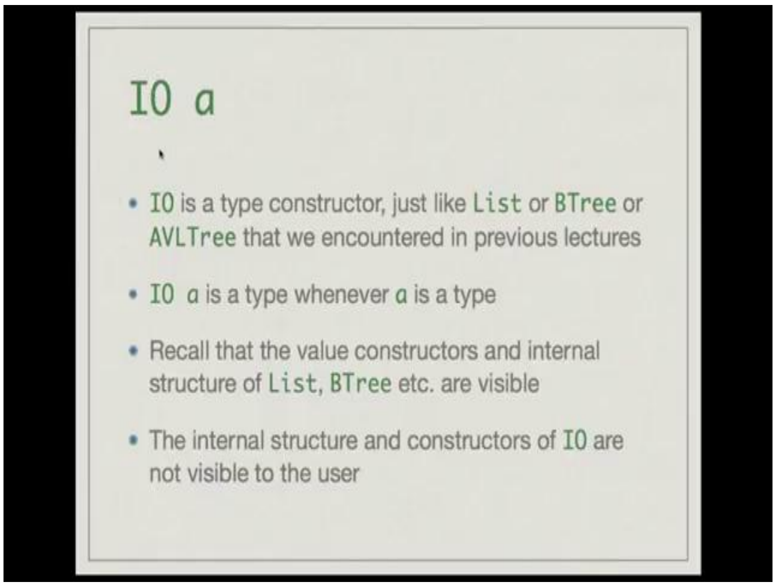
- Is putStr of type String -> ()?
- But it does not return the value ()!
- And how do we account for the **side effect** of printing something on screen?

-
  ```
  ghci> :t putStr
  putStr :: String -> IO ()
  ```

-
  ```
  ghci> :t putStr "Hello, world!"
  putStr "Hello, world!" :: IO ()
  ```

But, we notice that putStr is not an expression that returns a value and more over, it has a side effect, which is that of printing something on screen. So how do we account for the side effect? So, if you actually type colon t putStr in ghci, you will see that ghci says what the type of putStr is, the type is string arrow IO empty. If you check what the type of putStr Hello world is, you will see that, the type is IO empty.

(Refer Slide Time: 07:13)

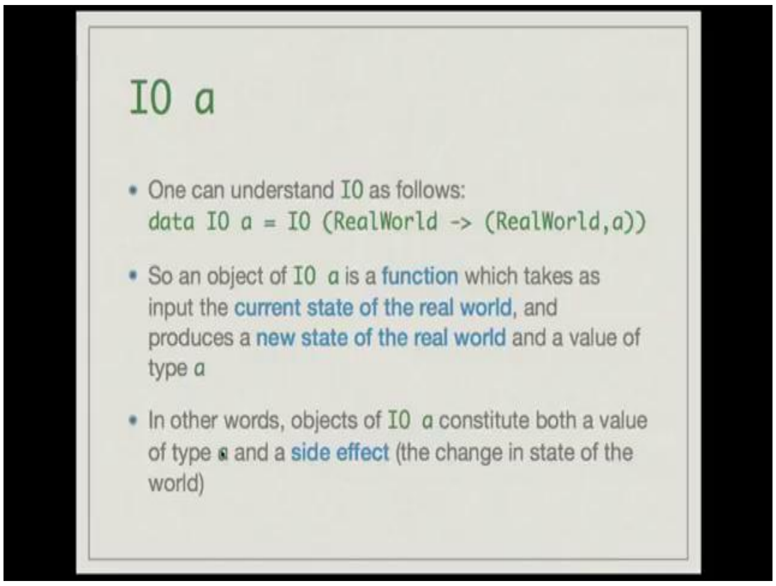


IO a

- IO is a type constructor, just like List or BTree or AVLTree that we encountered in previous lectures
- IO *a* is a type whenever *a* is a type
- Recall that the value constructors and internal structure of List, BTree etc. are visible
- The internal structure and constructors of IO are not visible to the user

So, what is this IO? IO is a type constructor, just like some other type constructors we have encountered in previous lectures like List or BTree or AVLTree, etcetera. So, therefore, IO a is a type whenever a is a type, but there are distraction. Recall that the value constructor and internal structure or user defined data types like list BTree etcetera are visible. But, the internal structure and constructor of IO are not visible to the user in other words the user cannot do any kind of pattern matching based on the constructors of IO.

(Refer Slide Time: 07:57)



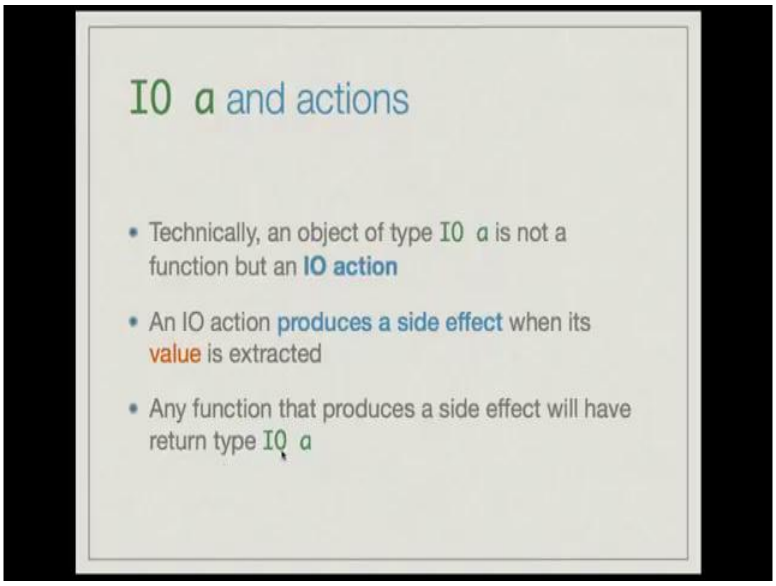
IO a

- One can understand IO as follows:
  `data IO a = IO (RealWorld -> (RealWorld,a))`
- So an object of IO a is a function which takes as input the current state of the real world, and produces a new state of the real world and a value of type a
- In other words, objects of IO a constitute both a value of type a and a side effect (the change in state of the world)

One can understand IO is as follows it is data its declaration can be thought of as data IO a equals IO the IO on the left is the type constructor the IO on the right is the value constructor. And the values are of type real world arrow the pair real world comma a real worlder is not an actual Haskell type, but it just one way we can understand, what IO means assume that there is a type, which represents all states of the real world.

So, we can thing of IO as taking as input the current state of the real world and producing the new state of real world due to side effects and also produce an value of type a. In other words object of IO a constitute both of value of type a and a side effect namely the change in state of the world.

(Refer Slide Time: 09:06)

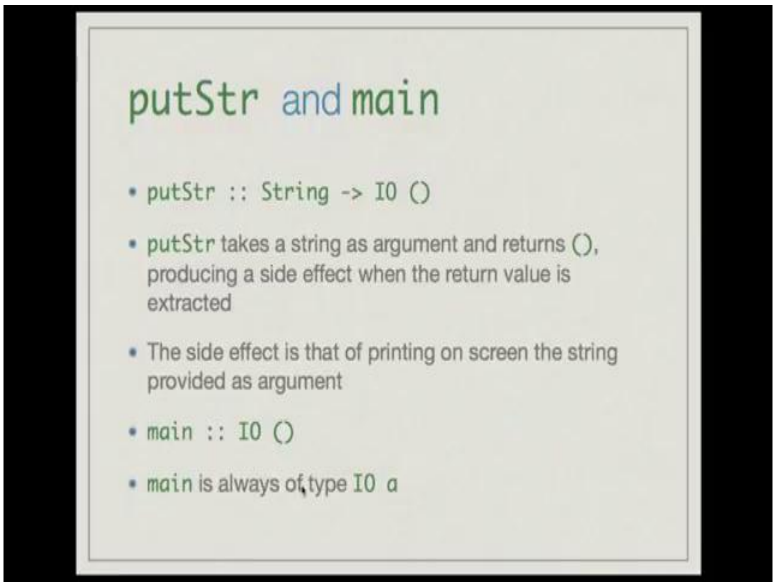


IO *a* and actions

- Technically, an object of type `IO a` is not a function but an **IO action**
- An IO action **produces a side effect** when its value is extracted
- Any function that produces a side effect will have return type `IO a`

Technically an object of type IO a is not usually referred to as a function, but of the IO action this is the important distinction. An IO action produces as a side effect when its values is extracted any function that produces as side effect will have return type IO a.

(Refer Slide Time: 09:32)

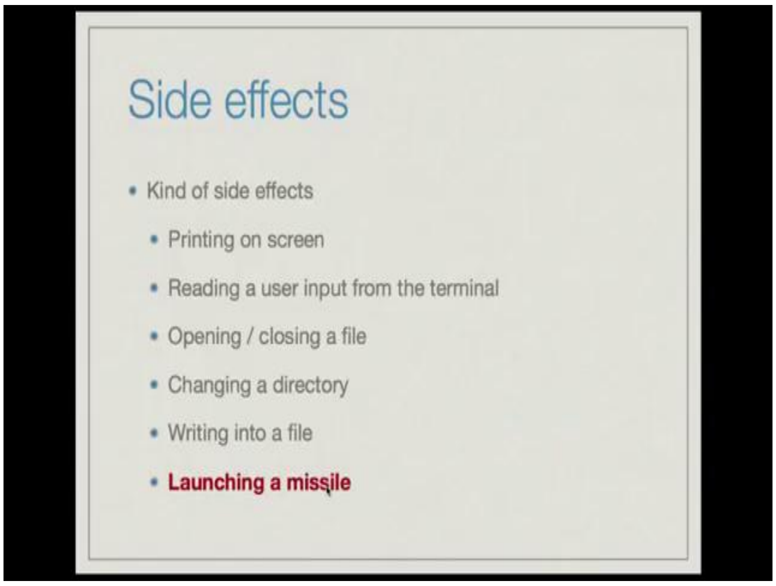


`putStr` and `main`

- `putStr :: String -> IO ()`
- `putStr` takes a string as argument and returns `()`, producing a side effect when the return value is extracted
- The side effect is that of printing on screen the string provided as argument
- `main :: IO ()`
- `main` is always of type `IO a`

So, lets get back to put string putStr is a type string arrow IO empty putStr takes as string is the argument and returns in the empty tubule. And in the process of producing the empty tubule is out as output it also produces the side effect when the return value is extract. The side effect in the case of putStr is that of printing on screen the string that is provided as argument main as you can see is of type IO empty main is always of type IO a for some a.

(Refer Slide Time: 10:15)

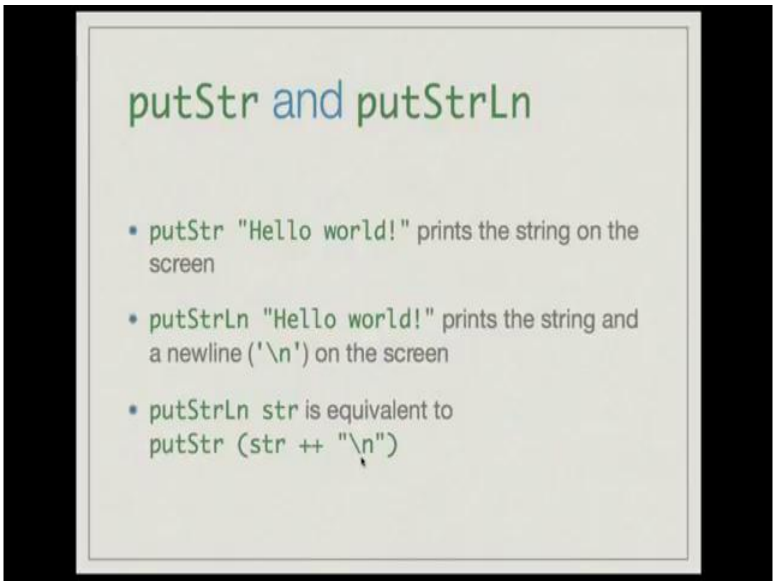


Side effects

- Kind of side effects
  - Printing on screen
  - Reading a user input from the terminal
  - Opening / closing a file
  - Changing a directory
  - Writing into a file
  - **Launching a missile**

Now, we talked about side effects in the course of executing an action, what kind of side effect can happen examples of printing on screen, reading the user input of from the terminal, opening or closing a file, changing the directing, writing in to a file etcetera are may be launching a missile driving a truck etcetera.

(Refer Slide Time: 10:45)

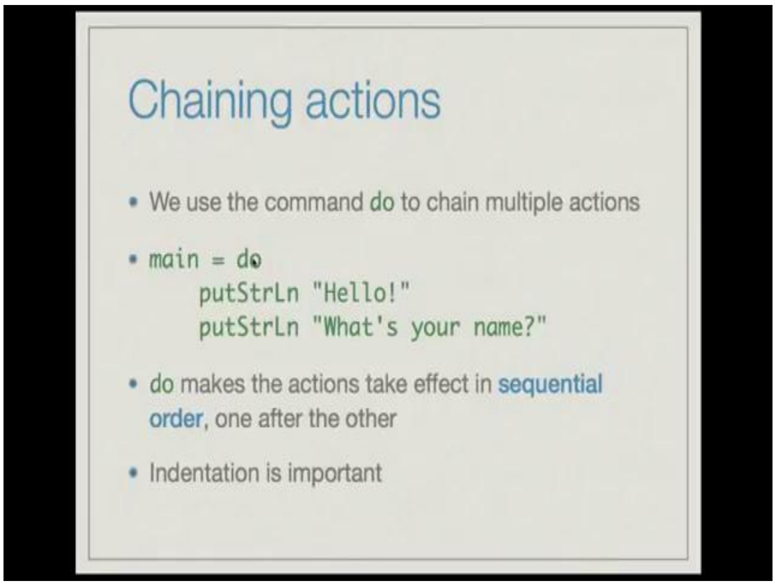


putStr and putStrLn

- `putStr "Hello world!"` prints the string on the screen
- `putStrLn "Hello world!"` prints the string and a newline (`'\n'`) on the screen
- `putStrLn str` is equivalent to `putStr (str ++ "\n")`

Now, there is a close variant of putStr namely putStrLn you can read it as putStr line putStr Hello world print the string on screen, where a putStr line Hello world prints the string and appends a new line on the screen. So, putStr line str is equivalent to putStr of str plus plus the new line character.

(Refer Slide Time: 11:17)



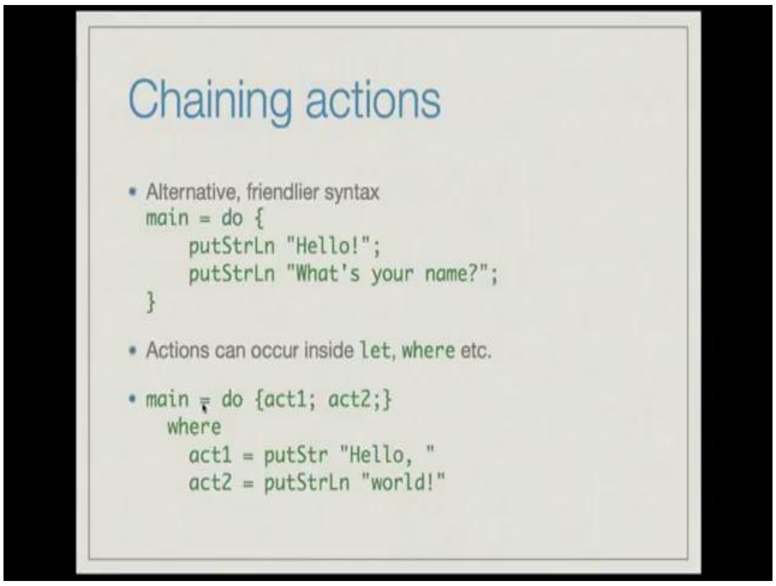
Chaining actions

- We use the command do to chain multiple actions

-
  ```
  main = do
      putStrLn "Hello!"
      putStrLn "What's your name?"
  ```
- do makes the actions take effect in sequential order, one after the other
- Indentation is important

Standard of action not of much use unless you can perform a lot of actions and a Haskell provides away to chain actions use the command do to chain multiple actions. For example, you could say main equals do putStr lien Hello putStr line, what is your name, do makes the actions take effects in sequential order one after the other in the order presented in the text in the program text. Here the intention important in the do command, but in the indentation sometimes add to keep truck of…

(Refer Slide Time: 12:00)



Chaining actions

- Alternative, friendlier syntax

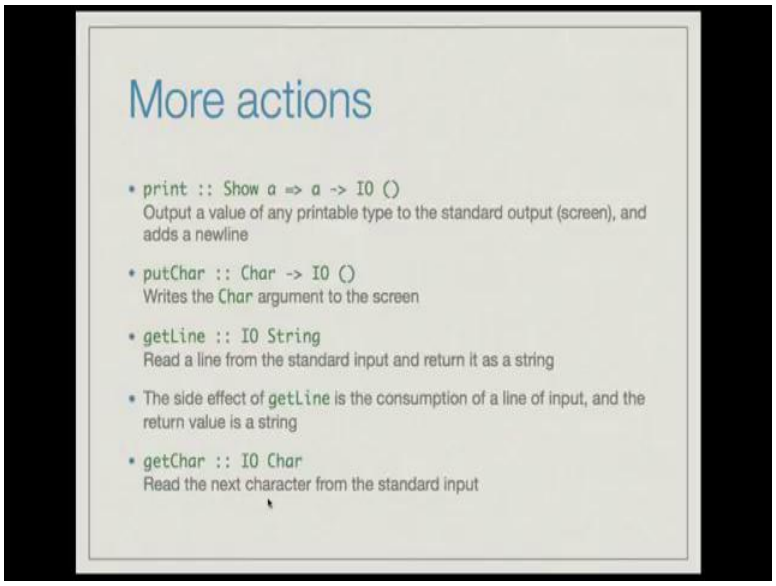
```
main = do {
    putStrLn "Hello!";
    putStrLn "What's your name?";
}
```

- Actions can occur inside `let`, `where` etc.
- `main = do {act1; act2;}`

```
    where
      act1 = putStr "Hello, "
      act2 = putStrLn "world!"
```

Haskell offers an alternative friendlier syntax, which is to use braces some semi colons main equals do open brace put string line Hello put a semi colon put string line, what is your name, with the semi colon and close the brace. And the actions can also occurrence inside, let where etcetera etcetera. For instance main equals do act one semi colon act two where act 1 is put string Hello act 2 is putStrLn world. So, you can define local actions.

(Refer Slide Time: 12:42)

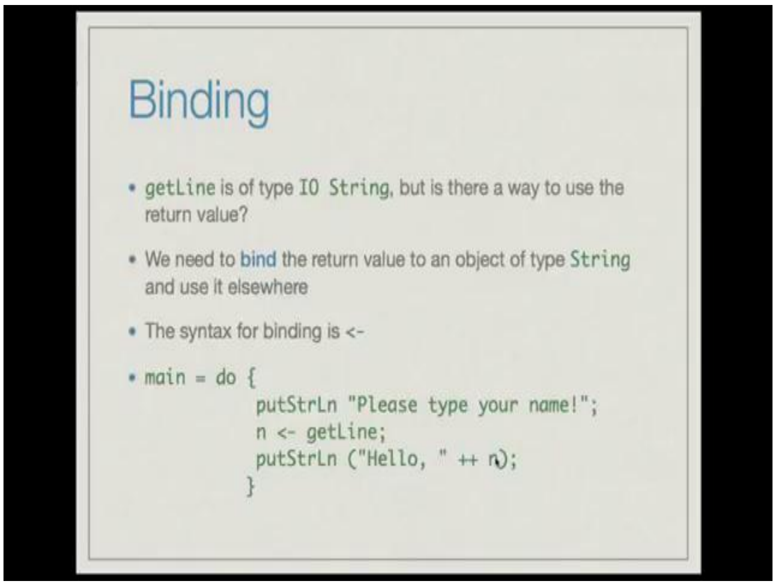


More actions

- `print :: Show a => a -> IO ()`
  Output a value of any printable type to the standard output (screen), and adds a newline
- `putChar :: Char -> IO ()`
  Writes the `Char` argument to the screen
- `getLine :: IO String`
  Read a line from the standard input and return it as a string
- The side effect of `getLine` is the consumption of a line of input, and the return value is a string
- `getChar :: IO Char`
  Read the next character from the standard input

Here are some more actions print is signature is show a implies a arrow IO empty this outputs a value of any printable type to the standard output, which is the screen and as in new line. PutChar is a functions from char into IO empty it writes the car argument that is provided to it on the screen. Get line is a type IO string if reads the lines from the standard input and return it as a string the side effect of get line is the conception of line of input rather than the production of an input and the return value of get line is a string getChar is a function that reads the next character from the standard input.

(Refer Slide Time: 13:38)



Binding

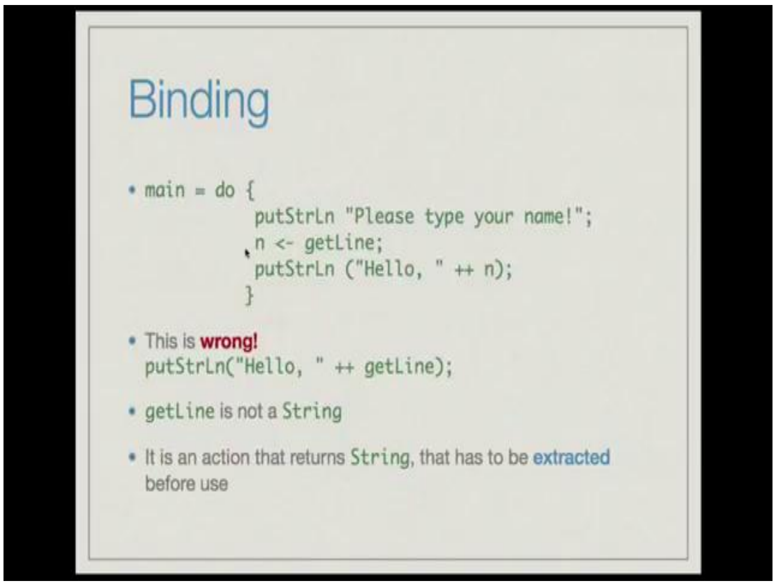
- getLine is of type IO String, but is there a way to use the return value?
- We need to bind the return value to an object of type String and use it elsewhere
- The syntax for binding is <-
- main = do {

```
            putStrLn "Please type your name!";
            n <- getLine;
            putStrLn ("Hello, " ++ n);
           }
```

We saw the get line is a type IO string, but is there a way to use the value that is return by get line. In other words we need to bind the return value of get line to an object of type string and perhaps use it elsewhere. Haskell provides the following syntax for binding and syntax is ruminant of assignment on the some other languages, which is just to say less than hyphen it is like a left arrow for instance you could do this main equals do put string line please type your name n bond by get line or another words the output of get line is bond to n.

And I can use it here, in the following action, which is put string line Hello plus, plus n this. As the effect of first asking the user for the name waiting for the user two input her name and press the enter key and then printing Hello followed by her name on the screen.

(Refer Slide Time: 14:56)

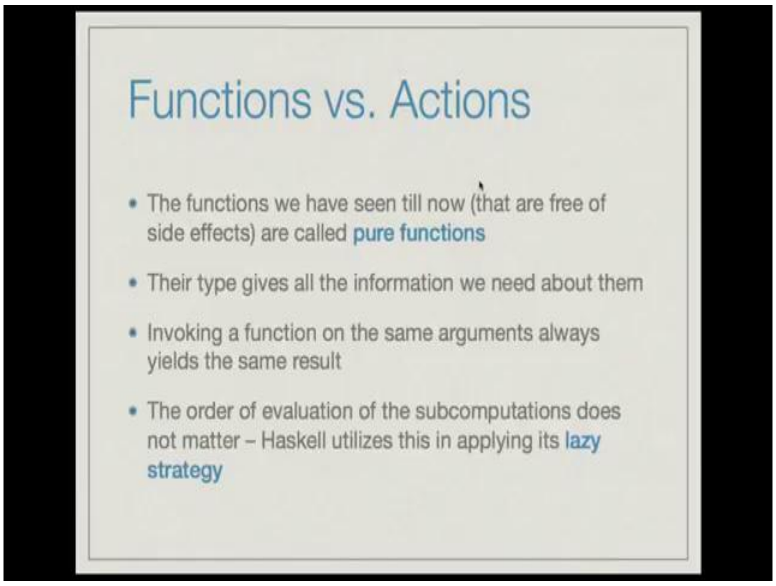


Binding

-
  ```
  main = do {
              putStrLn "Please type your name!";
              n <- getLine;
              putStrLn ("Hello, " ++ n);
             }
  ```
- This is **wrong!**
  `putStrLn("Hello, " ++ getLine);`
- `getLine` is not a `String`
- It is an action that returns `String`, that has to be extracted before use

Please note that this is wrong putStrLn of Hello plus, plus get line this is because plus plus is a. So, called pure Haskell function or operator and it is arguments are list a and list a and the output is mistake the arguments are not a type IO a. Therefore, you cannot use get line in the contest of plus, plus you should always binded to some name and then use that name get line is not a string and the action that returns from, but as to extracted before the use the extraction is the binding that is function here.

(Refer Slide Time: 15:47)



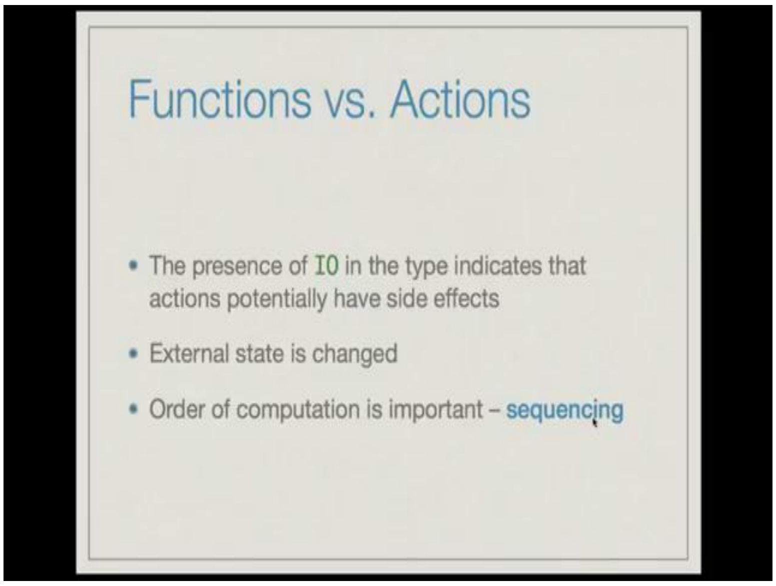
Functions vs. Actions

- The functions we have seen till now (that are free of side effects) are called pure functions
- Their type gives all the information we need about them
- Invoking a function on the same arguments always yields the same result
- The order of evaluation of the subcomputations does not matter – Haskell utilizes this in applying its lazy strategy

At this point we need to look a little closer at the distinction, but in functions on the actions. A function that takes an integer one the argument and returns an integer as a result has type Int arrow Int an action that has a side effect in addition to consumer an integer and produce an integer as type Int arrow IO Int. This distinction that Haskell maintains is an contrast two languages like c or java, where the type signature of both functions that have side effects and functions that do not have side effects are just Int arrow int.

And in general any function is assumed to potentially produce a side effect any function can produces a side effect there is nothing in the language itself that province functions from produce in side effects. Haskell enforces this distinction between pure functions and actions the functions that you seen till now that of free of side effects are called pure functions there type gives all the information we need about invoking the function on the same arguments always yields the same result. On the order of evaluation of sub computations does not matter Haskell utilizes this to great effect in applying its lazy strategy.

(Refer Slide Time: 17:20)

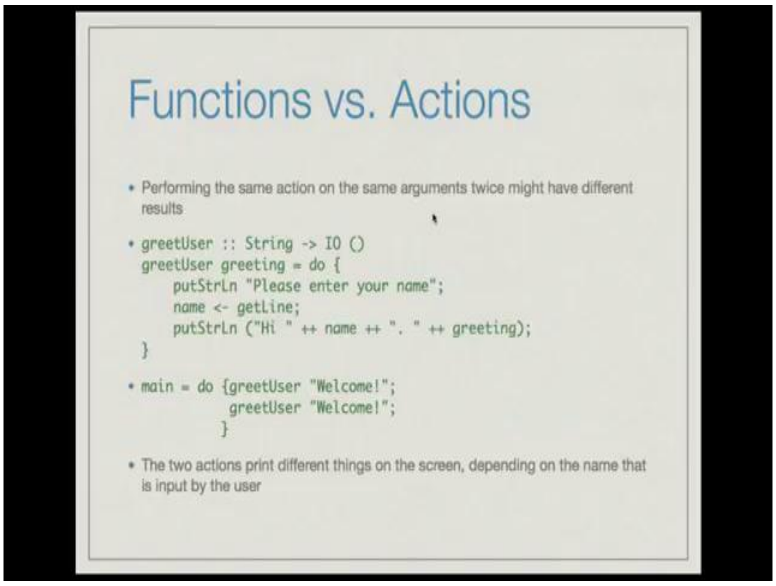


Functions vs. Actions

- The presence of IO in the type indicates that actions potentially have side effects
- External state is changed
- Order of computation is important – sequencing

In contrast functions actions usually have side effects and Haskell maintains this distinction by designating the output types with an IO. The presence of IO in the type indicates that actions potentially have side effects external state is usually changed on the order of computation important as in actions inside at do command. The action takes place in a sequence, so sequence in is something that is inherent to actions.

(Refer Slide Time: 18:03)



Functions vs. Actions

- Performing the same action on the same arguments twice might have different results

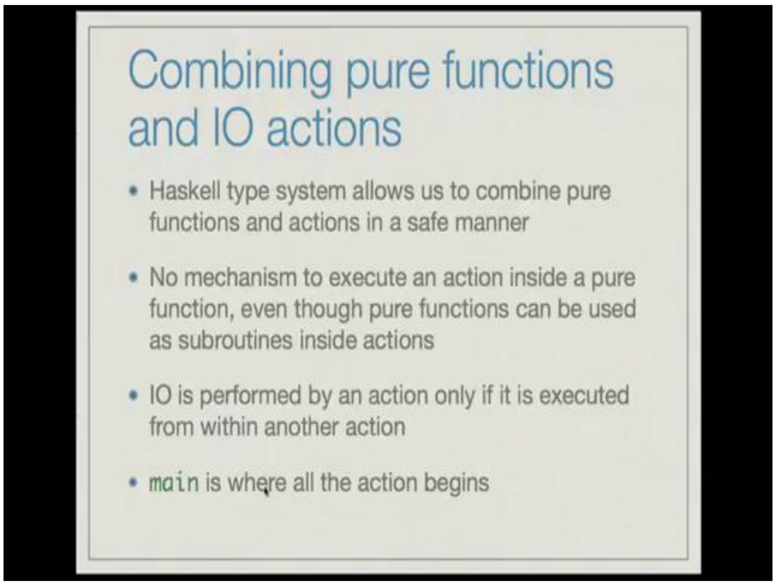
-
  ```
  greetUser :: String -> IO ()
  greetUser greeting = do {
      putStrLn "Please enter your name";
      name <- getLine;
      putStrLn ("Hi " ++ name ++ ". " ++ greeting);
  }
  ```

-
  ```
  main = do {greetUser "Welcome!";
             greetUser "Welcome!";
            }
  ```
- The two actions print different things on the screen, depending on the name that is input by the user

More over perform in the same action on the same argument twice might have a different results. For instance consider the following action greetUser, which is the type string arrow IO empty greetUser greeting equals do inside the do block we have put string line please enter your name which by hence the value of get line the value of the return value of get line ((ReferTime18:38)). Then, we put another string on the screen put string line Hi plus, plus name plus plus full stop plus plus greeting.

So, this greeting is something provided does not input in main we can do the following greet user welcome greet user welcome. You see that the greet user function is being called with the exact same argument twice, but the two actions might print different things on this screen depending on the name that is input by the user at this point. Because, the greet user action with itself has a way to get input from the user and type the input back to the user namely the use as name. So, this shows that performing the same action on the same arguments twice might have different results and this is the fundamental difference between actions and functions.

(Refer Slide Time: 19:44)

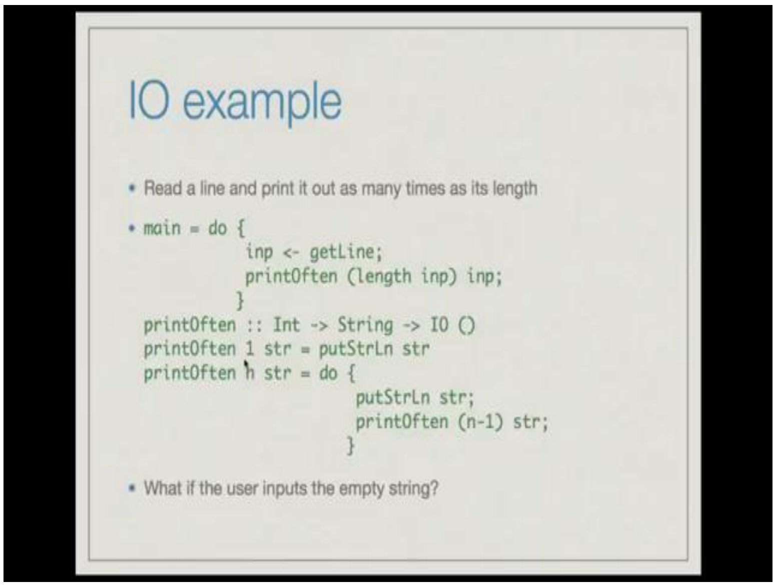


Combining pure functions and IO actions

- Haskell type system allows us to combine pure functions and actions in a safe manner
- No mechanism to execute an action inside a pure function, even though pure functions can be used as subroutines inside actions
- IO is performed by an action only if it is executed from within another action
- `main` is where all the action begins

One can combine pure functions are actions, but in the limited manner we can use pure functions as subroutines and IO actions, but not the other way round. The Haskell type system allows us to combine pure function and actions in a safe manner no mechanism exist to execute an action inside a pure function even though pure functions can be used the subroutine inside actions. IO is perform by an action only if it is a executed from within another action and main is, where all the action begins. So, main embed some actions inside it and a each of those actions might be a do block with further action inside.

(Refer Slide Time: 20:31)



IO example

- Read a line and print it out as many times as its length

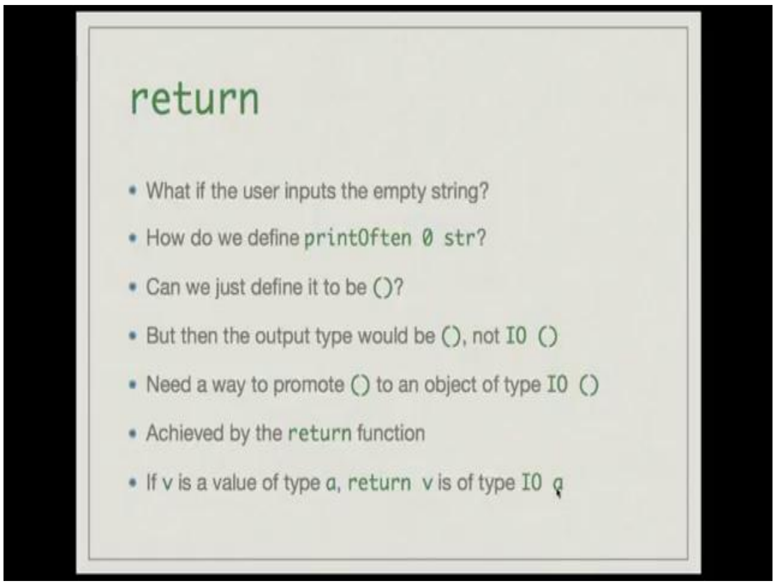
```
• main = do {
             inp <- getLine;
             printOften (length inp) inp;
            }
  printOften :: Int -> String -> IO ()
  printOften 1 str = putStrLn str
  printOften n str = do {
                          putStrLn str;
                          printOften (n-1) str;
                        }
```

- What if the user inputs the empty string?

Let us look at a few examples, here is one example, which is to read a line and print it out as many times as the length of the string that is in put on the first line. Here is the program main equals do get a line and bind it to inp it stands for input it is a variable of it is a values of types string. Call the function print often to print input length input times, if input is of length n, then print of n input will we called and it will to be print in printed n times.

Thus achieved as follows print often is the function from Int and string for IO empty print often one string is just put string lines string print often n string is recursive function inside a do block the first put string line string and then you print often n minus 1 times string. So, this is an example of a non trivial interaction with user, but what is the user input the empty string notices that, if length of the input equal 0 there is no case here the catches it.

(Refer Slide Time: 22:03)

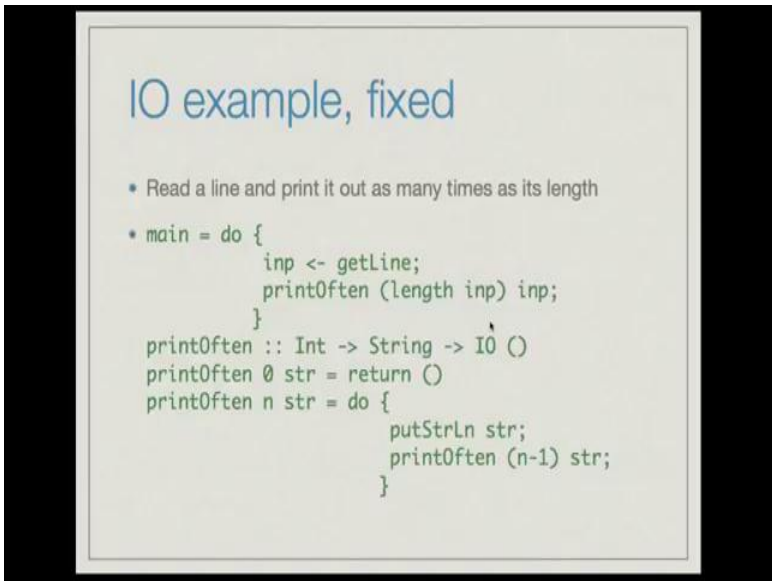


return

- What if the user inputs the empty string?
- How do we define `printOften 0 str`?
- Can we just define it to be ()?
- But then the output type would be (), not `IO ()`
- Need a way to promote () to an object of type `IO ()`
- Achieved by the `return` function
- If v is a value of type *a*, `return v` is of type `IO a`

What is the user inputs the empty string, to handle this we need to define print often 0 string recall that the return value of print often this IO empty print often 0 string the student be any anything printed on the screen. So, can we just define it to be empty, but the output type then would be empty and not IO empty. So, you get a type mismatch this means that you need of way to promote the empty tubule to an object of type IO empty this is precisely achieve by the return function. The return function of the return action takes v, which is the value of type a and a produces an action of type IO a, if v is a value of type a return v is of type IO a.

(Refer Slide Time: 23:06)



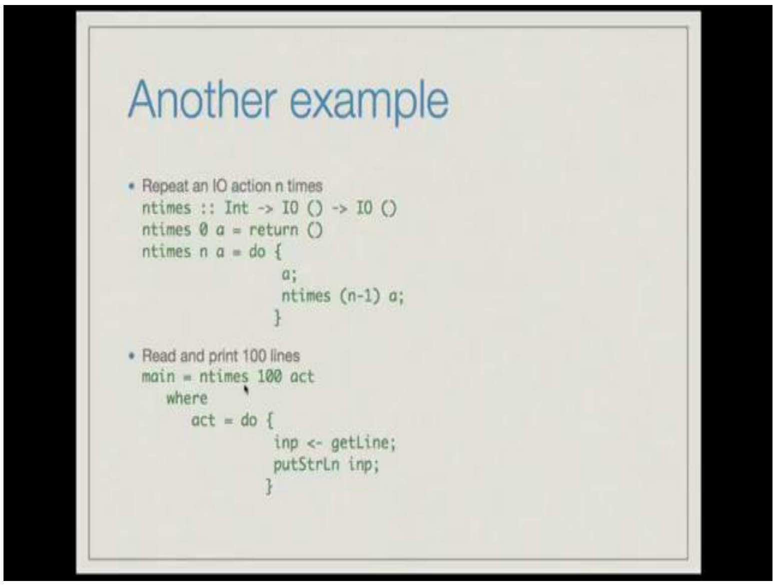
IO example, fixed

- Read a line and print it out as many times as its length

```
• main = do {
             inp <- getLine;
             printOften (length inp) inp;
            }
  printOften :: Int -> String -> IO ()
  printOften 0 str = return ()
  printOften n str = do {
                          putStrLn str;
                          printOften (n-1) str;
                         }
```

With this we can fix the earlier example as follows main equals do get a line and bind it input print often length input the crucial cases this print often 0 string equals return empty. This matches the type properly and it also handles this case in the case of not printing anything on screen. Notice, that here there is no side effect as such, but still we designate this as type IO empty. So, this illustrate the fact that if an object is an type IO a it need not necessarily produce a side effect it only indicates the potential to produces side effect where as a poor function, which is not to type is not embedded to bio can never produces side effect.

(Refer Slide Time: 24:13)



Another example

- Repeat an IO action n times

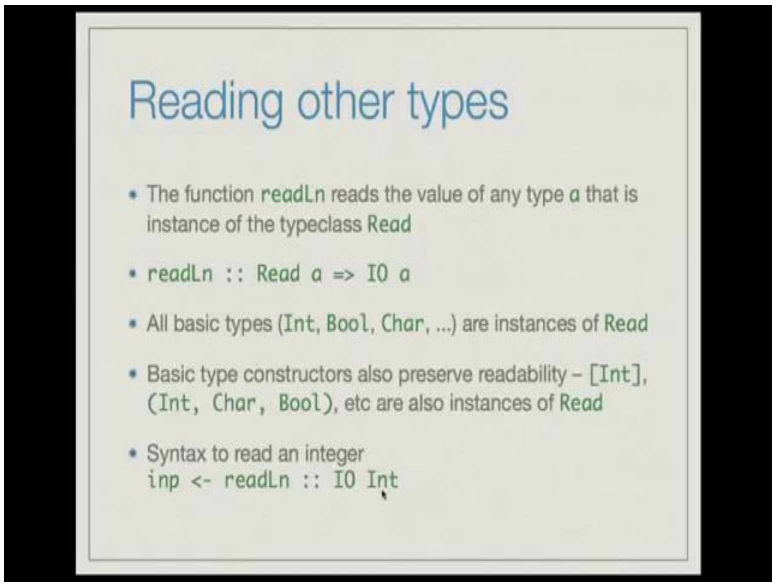
```
ntimes :: Int -> IO () -> IO ()
ntimes 0 a = return ()
ntimes n a = do {
                 a;
                 ntimes (n-1) a;
                }
```

- Read and print 100 lines

```
main = ntimes 100 act
    where
        act = do {
                  inp <- getLine;
                  putStrLn inp;
                 }
```

Here is an another example n times, which takes an integer and a action and repeat the action and n times this is the generalization of, what we did in the previous example, where we printed input out to the screen some n times, where n is the length of the input. So, n times 0 a is just return empty n times n a is do a followed by n times n minus 1 n times n minus 1 a which is two do the action a n minus 1 to x. Now, we can read and print 100 lines as follows main equals n times 100, at where act equals this block, which reads an input from the user and out and prints out that it get line binded to input putStr line input this action is repeated 100 times by main.

(Refer Slide Time: 25:15)



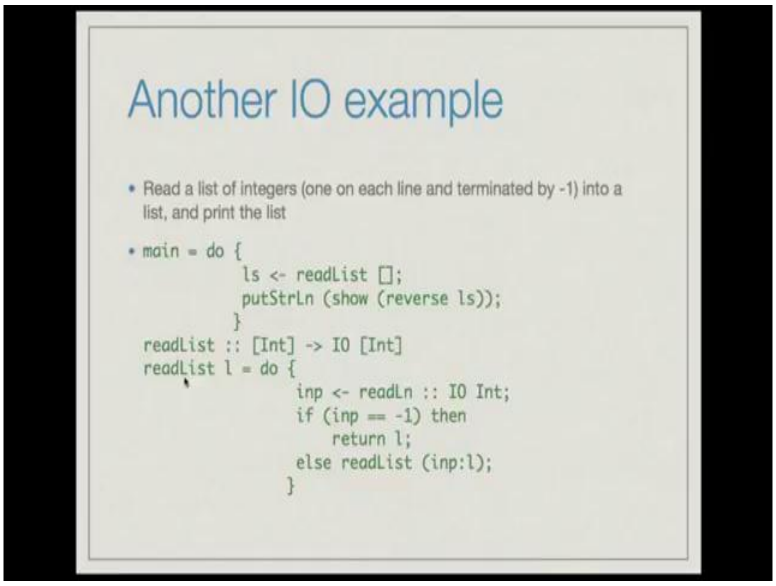
Reading other types

- The function `readLn` reads the value of any type *a* that is instance of the typeclass Read
- `readLn :: Read a => IO a`
- All basic types (`Int`, `Bool`, `Char`, ...) are instances of Read
- Basic type constructors also preserve readability – `[Int]`, `(Int, Char, Bool)`, etc are also instances of Read
- Syntax to read an integer
  `inp <- readLn :: IO Int`

Strings are not the only things that can be read, we can read other values, but these other values how to belong to a type a that is an instance that type class read. For this we use the function read line, which is denoted read Ln to type is a read a implies IO a all the basic type Int, Bool, char etc or instance of read. Therefore, you have you can use read line to read integers Booleans characters etc the basic type constructor also preserve readability.

So, for instance list Int the triple Int comma char comma Bool etc are also instance of read. Here is the syntax to ready an integer input read Ln colon colon IO Int and bind the result to. So, you since read Ln is suppose to cater the reading of any of these type, that belongs to class read you need to specify, which type you want to the input to be.

(Refer Slide Time: 26:38)



Another IO example

- Read a list of integers (one on each line and terminated by -1) into a list, and print the list

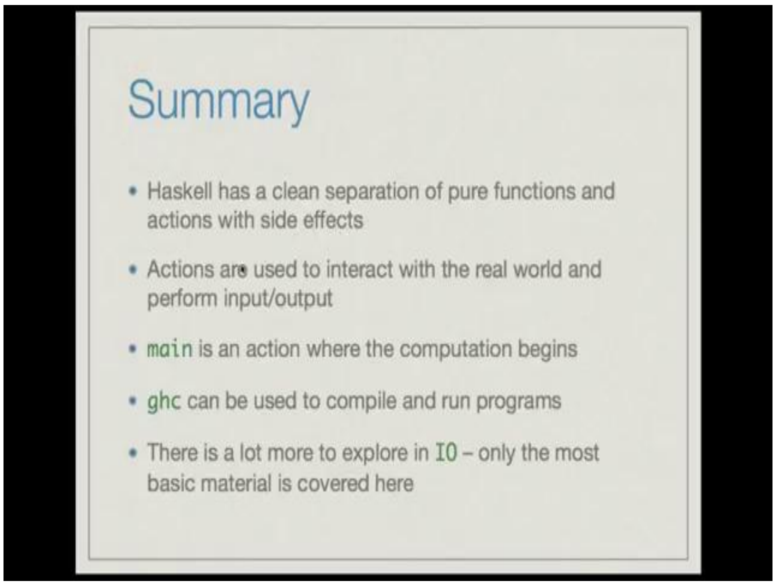
```
• main = do {
             ls <- readList [];
             putStrLn (show (reverse ls));
            }
  readList :: [Int] -> IO [Int]
  readList l = do {
                   inp <- readLn :: IO Int;
                   if (inp == -1) then
                       return l;
                   else readList (inp:l);
                  }
```

So, here is an example, in this example we read a list of integers one on each line and finally, terminated by minus 1 in to a list and print to the list main equals do l s is read list l s is the return value of read list empty list. So, read list is the function you are providing the empty list as a as an input to read list it is of type list Int arrow IO list Int. So, it produces some side effect and it is return value is a list of integers, which is what is bonded to the l s here and then it, so happens that read list read the list in reverse order.

So, we show the reverse of l s and invoke put string lines on it in this way we read a pouch of integers one on each line and output them as a list. Here is the description of read list read list l equals do read line as an integer in this denote by saying read Ln colon

colon IO Int bind the return value of input, if input is minus 1. Then, there is nothing more to do, so you just return l, which is the list that was provided as input else you read the input you add the input to the list by saying input colon l you add it to head of the list and you proceed with the read list function presumably reading more input.

(Refer Slide Time: 28:27)



Summary

- Haskell has a clean separation of pure functions and actions with side effects
- Actions are used to interact with the real world and perform input/output
- main is an action where the computation begins
- ghc can be used to compile and run programs
- There is a lot more to explore in IO – only the most basic material is covered here

In summary Haskell has a clean separation of pure functions and actions with side effects actions are used to interact to the real world and perform input output main is the action, where the computation begins ghc can be use to compile and run programs. So, there is the lot more to explore in IO put only the most basic material is covered, here with suffices for rudimentary interaction with uses.