Introduction To Haskell Programming

S. P. Suresh

Chennai Mathematical Institute

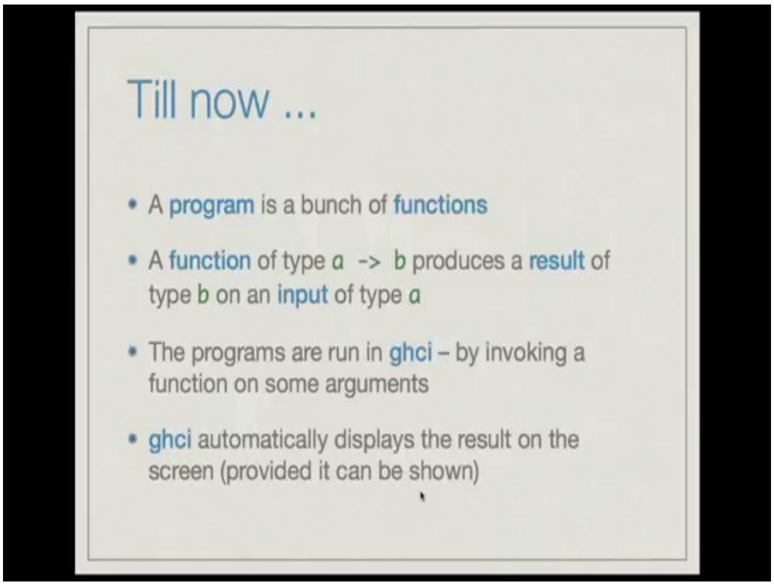
Module # 07

Lecture - 02

Input / Output

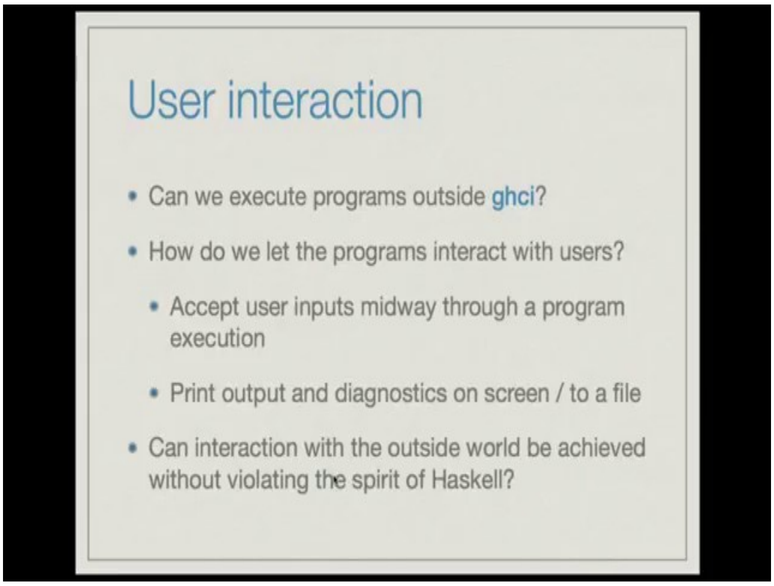
In this lecture, we shall study Input Output in Haskell.

(Refer Slide Time: 00:08)



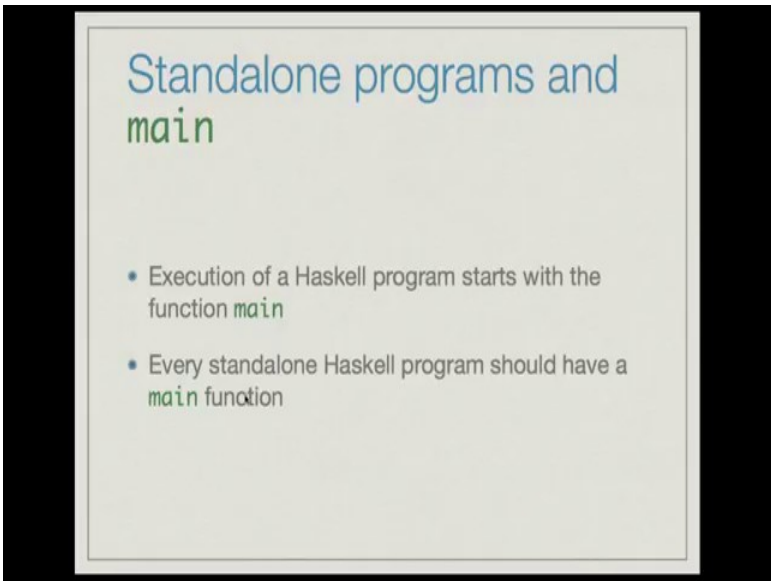
Till now, the view we have taken is that a program is a bunch of functions, a function of type a -> b produces a result of type b on an input of type a. The programs are run in ghci by invoking a function on some arguments, ghci automatically displays the result on the screen provided the result can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

(Refer Slide Time: 00:46)



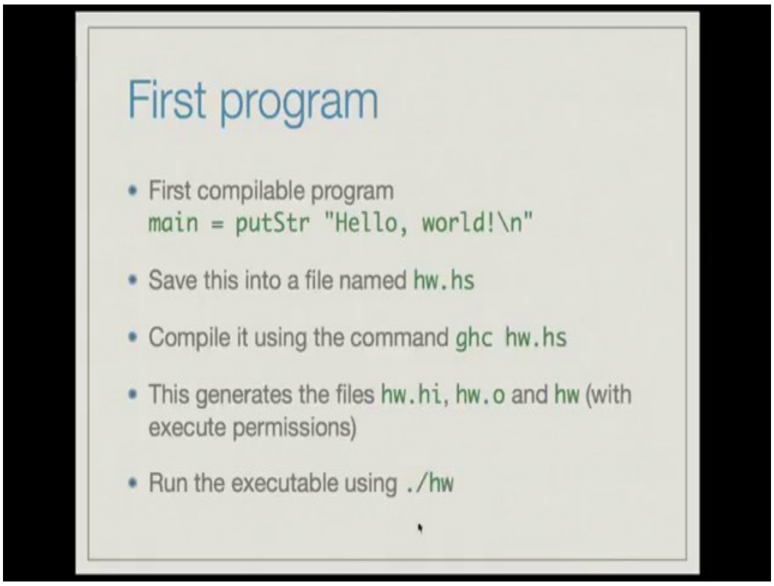
But, this is a limited form of interaction with the user, we would like a slightly better user interaction model like in other programming languages. So, the questions we have are , can we execute programs outside ghci, how do you let the programs interact with users that is accept user inputs midway 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 of the questions that we considered in this lecture.

(Refer Slide Time: 01:25)



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 just a bunch of 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 at some place and that is the function main, every standalone Haskell program should have a main function.

(Refer Slide Time: 02:03)



Here is an example program, in fact this is the simplest compilable program that we can write, main = putStr "Hello, world! \n". So, this as the name suggests puts a string on the screen, so the way we run this is to save this in to a file named hw.hs, hw standing for Hello world. We compile this file using the command ghc hw.hs. Compiling this generates the following files, hw.hi, hw.o and hw, the file of interest was the hw without any extension.

If you see this on a Unix terminal, we will see that the permissions for these two are read write, but hw has execute permissions or in other words 7 5 5. We can run the executable using ./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 resides in 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)

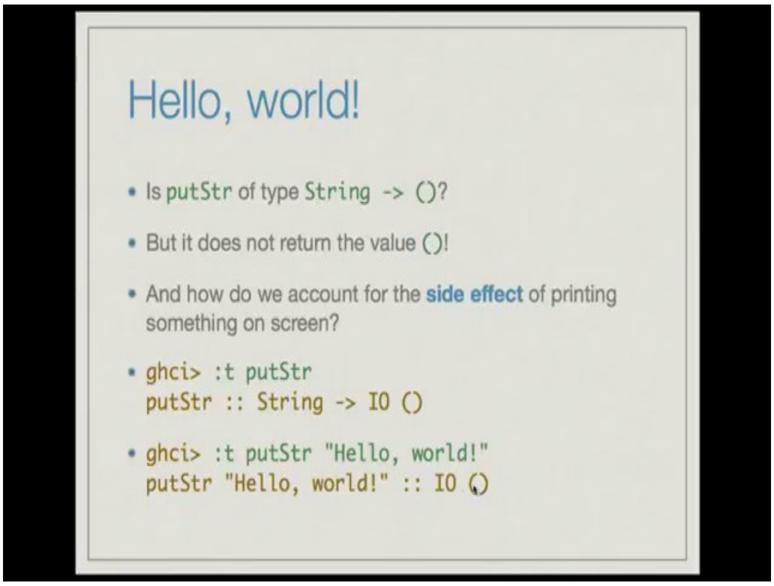
GHC is the Glasgow Haskell Compiler, ghci that we have been using till now is the interactive version of the compiler, one can view ghci as an interpreter or a play ground in which you test your programs. If the program is intended for use by others, then it is usually written as a standalone program,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

You can check the various options that ghc offers by typing ghc --help in the terminal, you can use ghc –show-options to know all the options that you can provide, all the compiler 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 stands alone and runs on its own, compile it and run it.

(Refer Slide Time: 05:05)

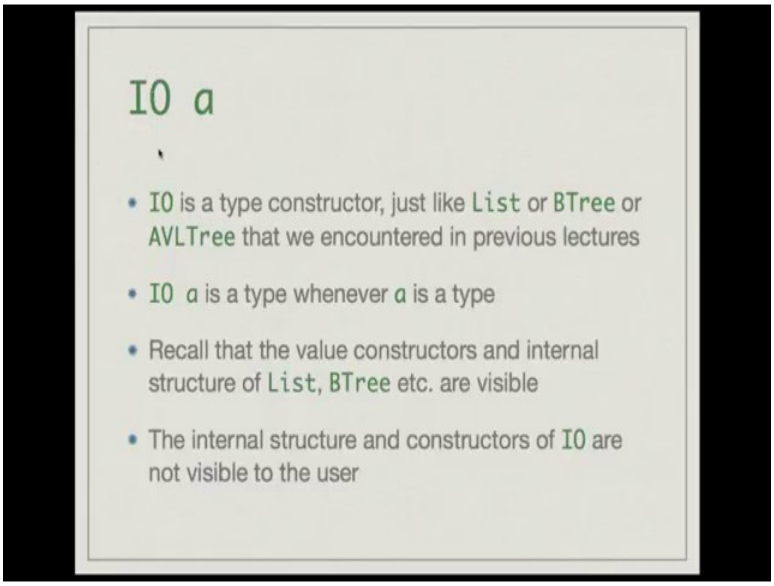
Let us study the program in more detail, main = putStr "Hello, world! \n", putStr is a function, the behavior is that putStr str prints the string str on screen. So, clearly putStr is of type string -> b for some b, because the input is a string, but you notice that in this main program the return value is not used at all. So, perhaps we can say that putStr does not return anything of significance, the type () 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 a single value, which is also denoted by an empty tuple, so the question is, is the type of putStr, String -> () ?.

(Refer Slide Time: 06:25)



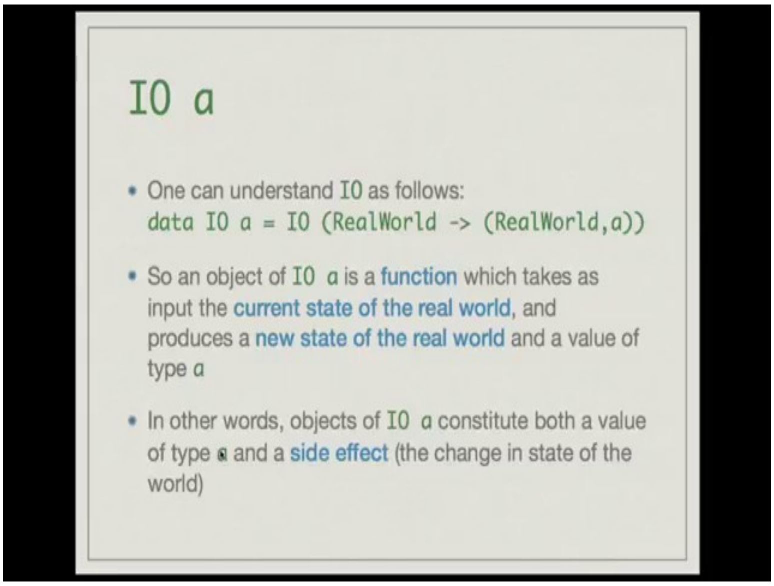
But, we notice that putStr is not an expression that returns a value and more over, it has a side effect, which is that of printing something on screen. So how do we account for the side effect? So, if you actually type :t putStr in ghci, you will see that ghci says what the type of putStr is, the type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

(Refer Slide Time: 07:13)



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, etc. So, therefore, IO a is a type whenever a is a type, but there is a distinction. Recall that the value constructors and internal structure of user defined data types like List, BTree etc are visible. But, the internal structure and constructors 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.

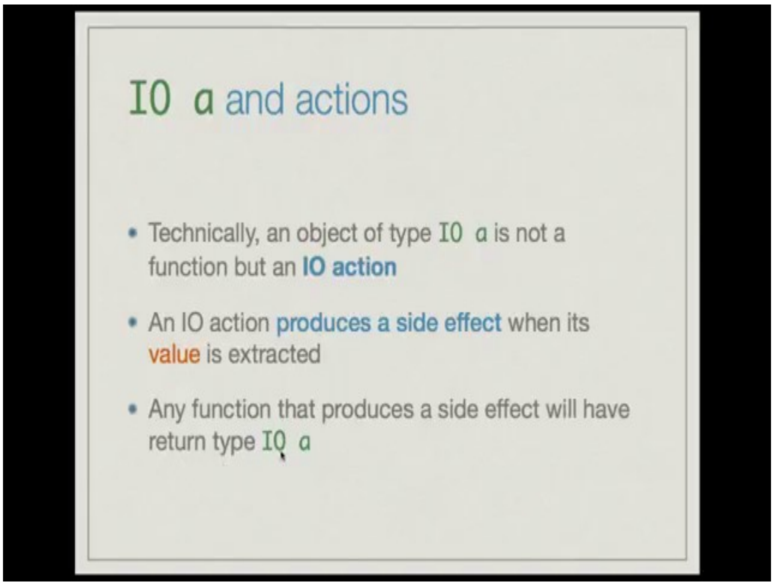
(Refer Slide Time: 07:57)



One way to understand IO is as follows ,its declaration can be thought of as data IO a = IO (RealWorld -> (RealWorld,a)) , the IO on the left is the type constructor, the IO on the right is the value constructor. And the values are of type RealWorld -> (RealWorld,a). RealWorld here is not an actual Haskell type, but it is just one way we can understand what IO means. Assume that there is a type, which represents all states of the real world.

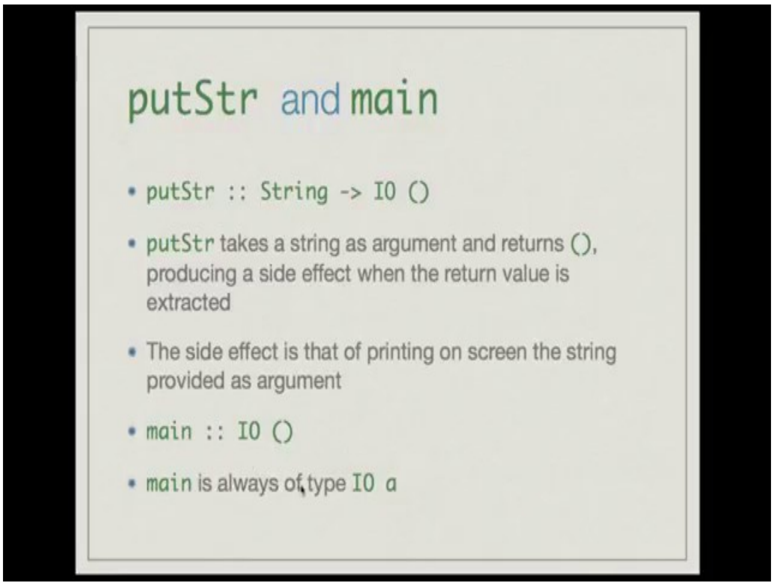
So, we can think of IO as taking as input the current state of the real world and producing a new state of real world due to side effects and also producing the value of type a. In other words object of IO a constitute both a value of type a and a side effect namely the change in state of the world.

(Refer Slide Time: 09:06)



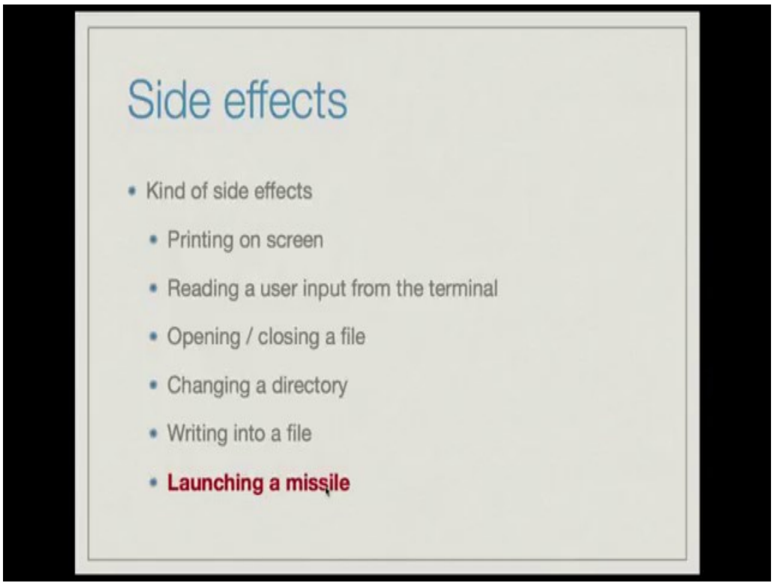
Technically an object of type IO a is not usually referred to as a function, but as an IO action. This is an important distinction. An IO action produces a side effect when its value is extracted. Any function that produces a side effect will have return type IO a.

(Refer Slide Time: 09:32)



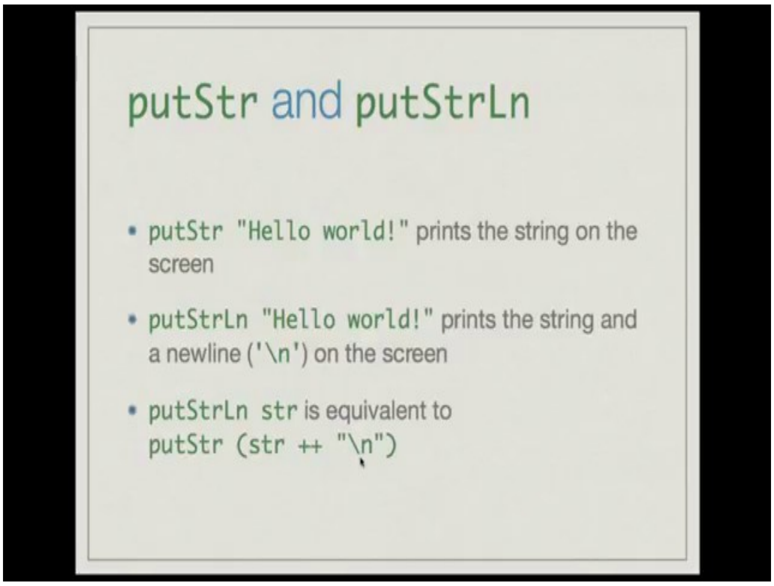
So, lets get back to put string. putStr has a type String -> IO (). putStr takes a String as the argument and returns an empty tuple (). And in the process of producing the empty tuple as output it also produces a side effect when the return value is extracted. The side effect in the case of putStr is 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)



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

(Refer Slide Time: 10:45)



Now, there is a close variant of putStr namely putStrLn, you can read it as put String line. putStr "Hello world!" prints the string on screen, whereas putStrLn "Hello world!" prints the string and appends a new line on the screen. So, putStrLn str is equivalent to putStr (str ++ "\ n")

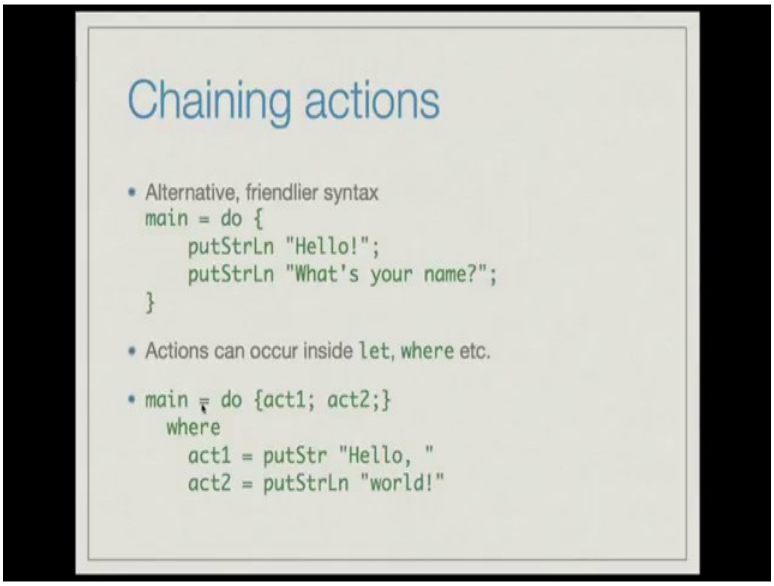
(Refer Slide Time: 11:17)

Stand alone actions are not of much use unless you can perform a lot of actions and Haskell provides a way to chain actions. We use the command do to chain multiple actions. For example, you could say

```
main = do
    putStrLn "Hello!"
    putStrLn "what's your name?"
```

do makes the actions take effect in sequential order one after the other in the order presented in the program text. Here the indentation is important in the do command, but since the indentation is sometimes hard to keep track of…

(Refer Slide Time: 12:00)



Haskell offers an alternative friendlier syntax, which is to use braces and semi colons.

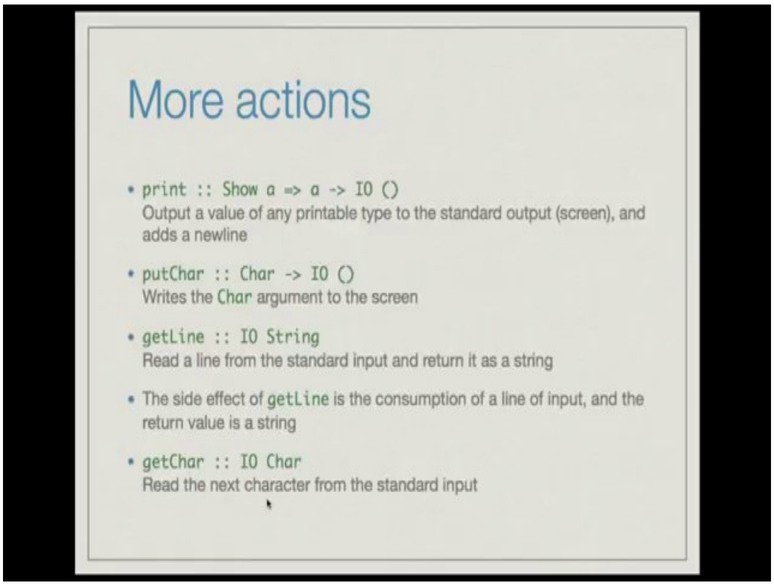
```
main = do {
    putStrLn “Hello!” ;
    putStrLn “what’s your name?”;
}
```

And these actions can also occur inside let, where etc. For instance,

```
main = do {act1; act2;}
    where
        act1 =  putStr “Hello, ” ;
        act2 = putStrlLn “world!”
```

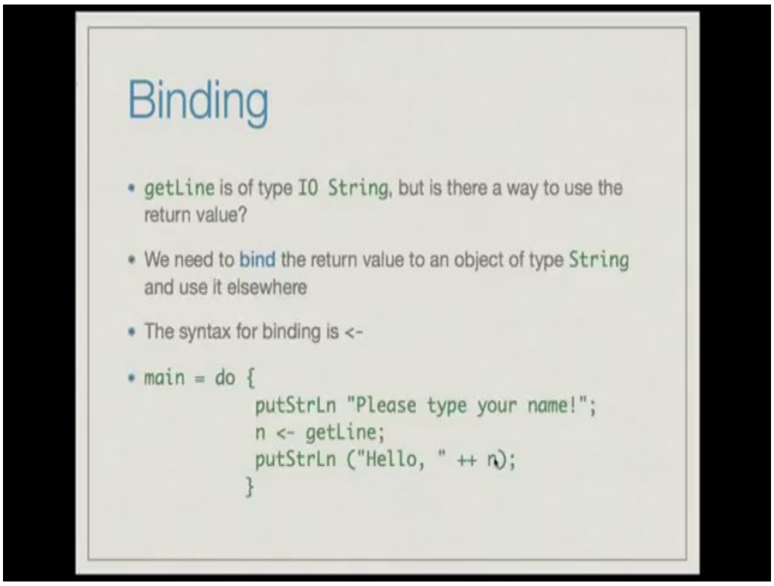
So, you can define local actions.

(Refer Slide Time: 12:42)



Here are some more actions, print, its signature is Show a => a -> IO (), this outputs a value of any printable type to the standard output, which is the screen and adds a new line. PutChar is a function from Char -> IO (). It writes the Char argument that is provided to it on the screen. getLine is of type IO string, it reads the lines from the standard input and returns it as a String, the side effect of getLine is the consumption of line of input rather than the production of a line of ouput and the return value of getLine is a String .getChar is a function that reads the next character from the standard input.

(Refer Slide Time: 13:38)



We saw that getLine is of type IO string, but is there a way to use the value that is returned by getLine.

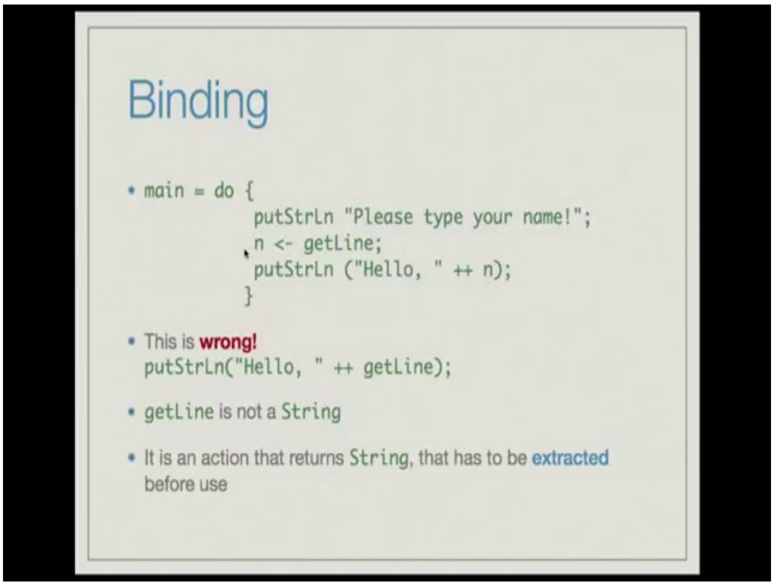
In other words we need to bind the return value of getLine to an object of type string and perhaps use it elsewhere. Haskell provides the following syntax for binding and the syntax is reminiscent of assignment in some other languages, which is just to say <- , it is like a left arrow for instance you could do this.

```
main = do {
    putStrLn "Please enter your name!" ;
    n <- getLine;
    putStrLn  ("Hello, " ++ n);
}
```

n bound by get line or in other words the output of get line is bound to n.

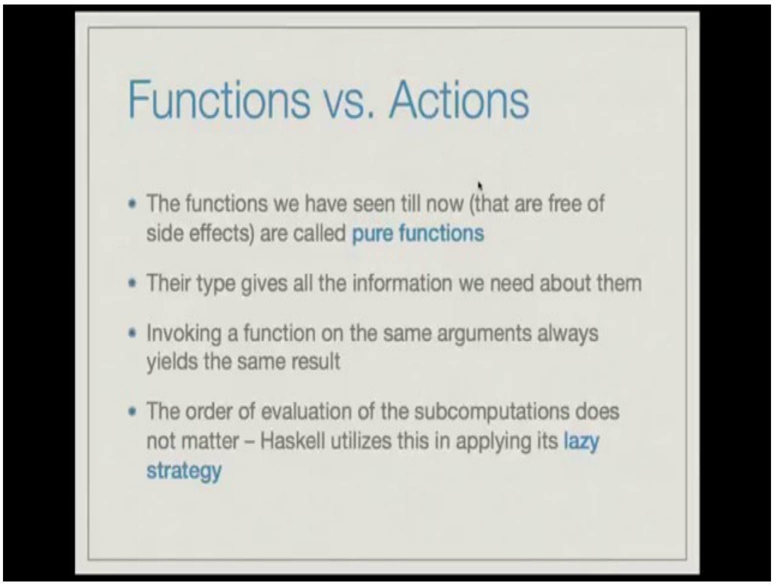
And then I can use it here, in the following action, which is putStrLn ("Hello, " ++ n); This has the effect of first asking the user for the name, waiting for the user to input her name and press the enter key and then printing Hello followed by her name on the screen.

(Refer Slide Time: 14:56)



Please note that this is wrong. putStrLn ("Hello, " ++ getLine); this is because ++ is a so called pure Haskell function or operator and its arguments are list a and list a and the output is list a, the arguments are not of type IO a. Therefore, you cannot use getLine in the context of a ++, you should always bind it to some name and then use that name. getLine is not a String. It is an action that returns String that has to be extracted before the use. The extraction is the binding that happens here through <-.

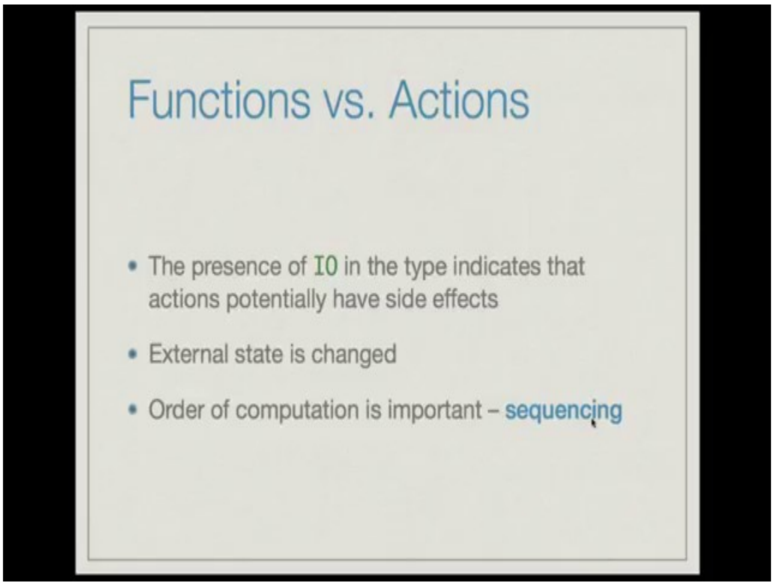
(Refer Slide Time: 15:47)



At this point we need to look a little closer at the distinction between functions and actions. A function that takes an integer as the argument and returns an integer as a result has type Int -> Int. An action that has a side effect in addition to consuming an integer and producing an integer has type Int -> IO Int. This distinction that Haskell maintains is in contrast to 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 -> Int.

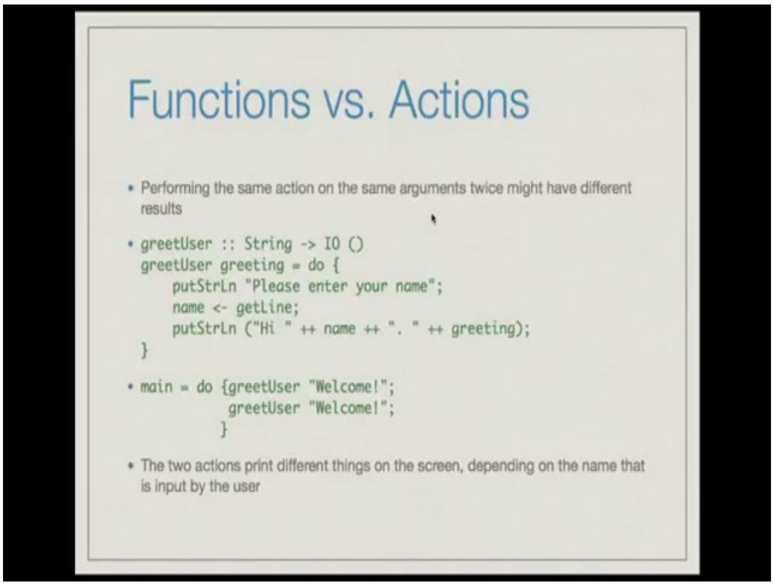
And in general any function is assumed to potentially produce a side effect , any function can produce a side effect there is nothing in the language itself that prevents functions from producing side effects. Haskell enforces this distinction between pure functions and actions. The functions that you have seen till now that are free of side effects are called pure function. Their type gives all the information we need about them. Invoking the function on the same arguments always yields the same result. And 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)



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

(Refer Slide Time: 18:03)

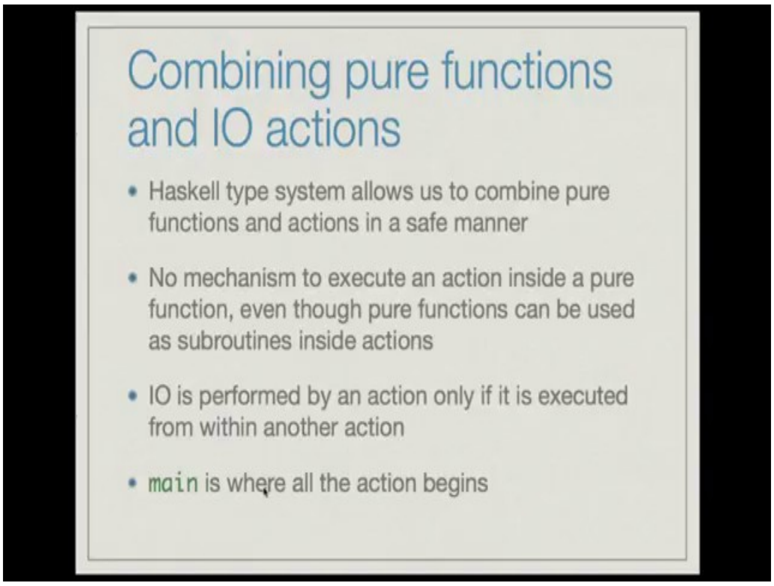


Moreover, performing the same action on the same arguments twice might have different results. For instance consider the following action greetUser, which is of type String -> IO ().

greetUser greeting equals do. Inside the do block we have putStrLn “please enter your name“; which binds the value of getLine. The return value of get line is bound to name. ((ReferTime18:38)). Then, we put another string on the screen putStrLn (“Hi” ++ name ++ ” ,” ++ greeting);

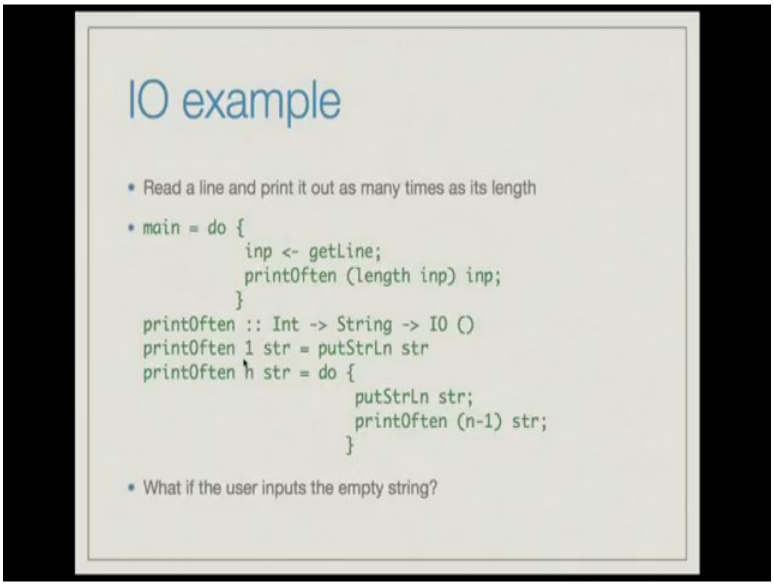
So, this greeting is something provided as an input. In main we can do the following greetUser “Welcome!”; greetUser “Welcome”;. You see that the greetUser function is being called with the exact same argument twice, but the two actions might print different things on this screen depending on the name that is input by the user at this point. Because, the greetUser action itself has a way to get input from the user and type the input back to the user namely the user's 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)



One can combine pure functions and actions, but in a 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 functions and actions in a safe manner. No mechanism exists 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 and main is, where all the action begins. So, main embed some actions inside it and each of those actions might be a do block with further actions inside.

(Refer Slide Time: 20:31)



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 input on the first line. Here is the program.

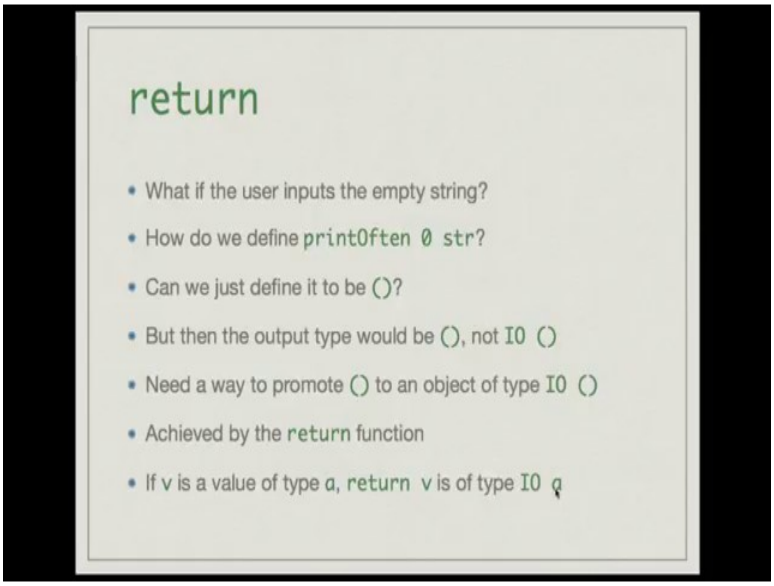
```
main = do {
    inp <- getLine;
    printOften ( length inp) inp;
   }
printOften :: Int -> String -> IO ()
printOften 1 str = putStrLn str
printOften n str = do {
                        putStrLn str;
                        printOften (n-1) str;
                       }
```

main equals do get a line and bind it to inp. inp stands for input. it is a variable that stores values of types String. Call the function printOften to print inp length inp times, if inp is of length n, then printOften input will be called and it will be printed n times.

Thus achieved as follows. printOften is a function from Int -> String -> IO ().
printOften 1 str is just putStrLn str;
printOften n str is a recursive function. Inside a do block you first put putStrLn str; and then you printOften (n-1) str. So, this is an example of a non trivial

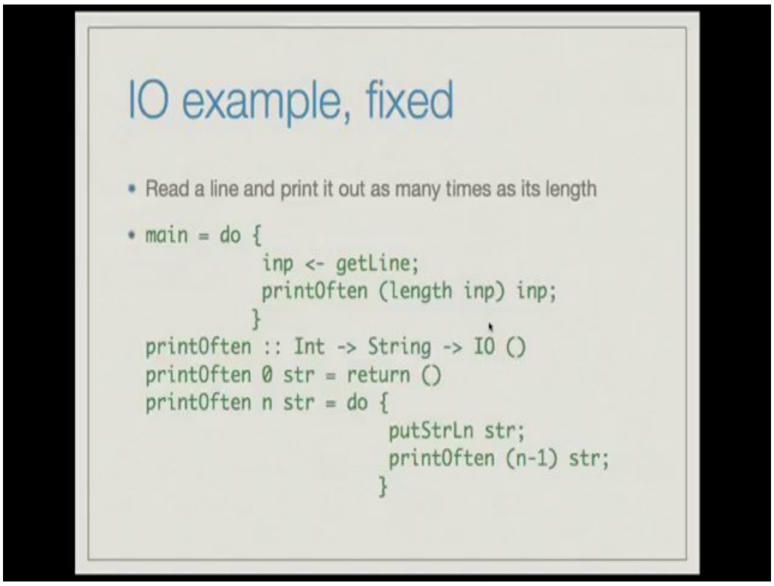
interaction with the user, but what if the user inputs the empty string. Notices that, if length of the input equals 0 there is no case here the catches it.

(Refer Slide Time: 22:03)



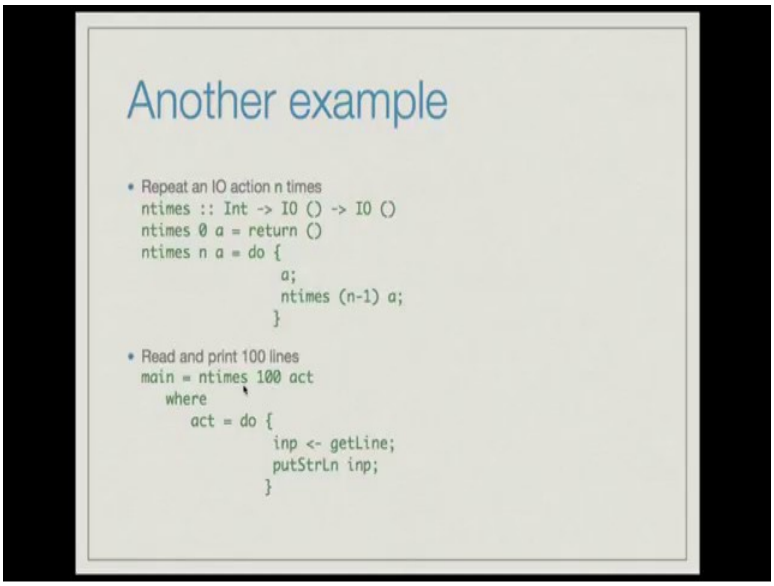
What if the user inputs the empty string, to handle this we need to define printOften 0 str. recall that the return value of printOften is IO (). printOften 0 str - there should not be anything printed on the screen. So, can we just define it to be empty, but the output type then would be empty and not IO empty. So, you would get a type mismatch this means that we need a way to promote the empty tuple to an object of type IO empty. This is precisely achieved by the return function. The return function or the return action takes v, which is the value of type a and 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)



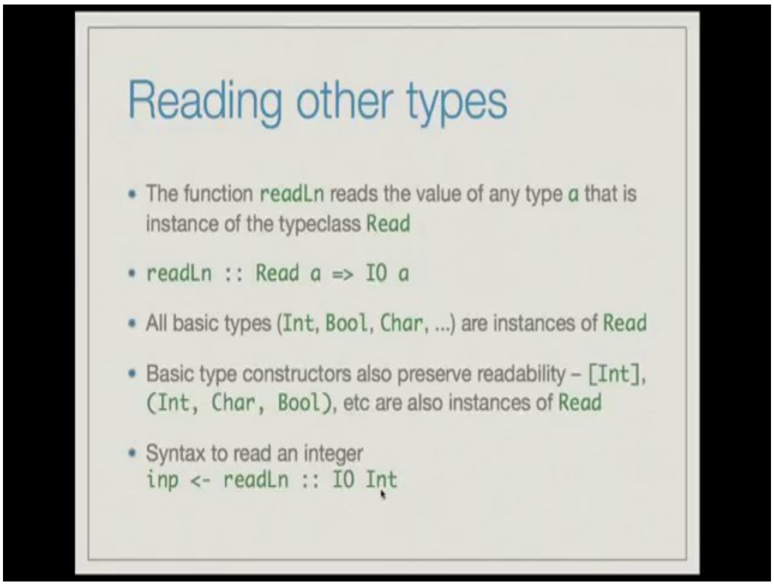
With this we can fix the earlier example as follows. main equals do get a line and bind it input print often length input input the crucial case is this printOften 0 str = return (). This matches the type properly and it also handles this case in the case of not printing anything on the screen. Notice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

(Refer Slide Time: 24:13)



Here is another example ntimes, which takes an integer and an action and repeat the action n times. This is the generalization of, what we did in the previous example, where we printed inp out to the screen some n times, where n is the length of the input. So, ntimes 0 a is just return empty ,ntimes n a is do a followed by ntimes (n-1) a. ntimes (n-1) a which is to do the action a (n-1) times. Now, we can read and print 100 lines as follows main = ntimes 100 act where act equals this block, which reads an input from the user and prints out that input. getLine binded to inp and putStrLn inp. This action is repeated 100 times by main.

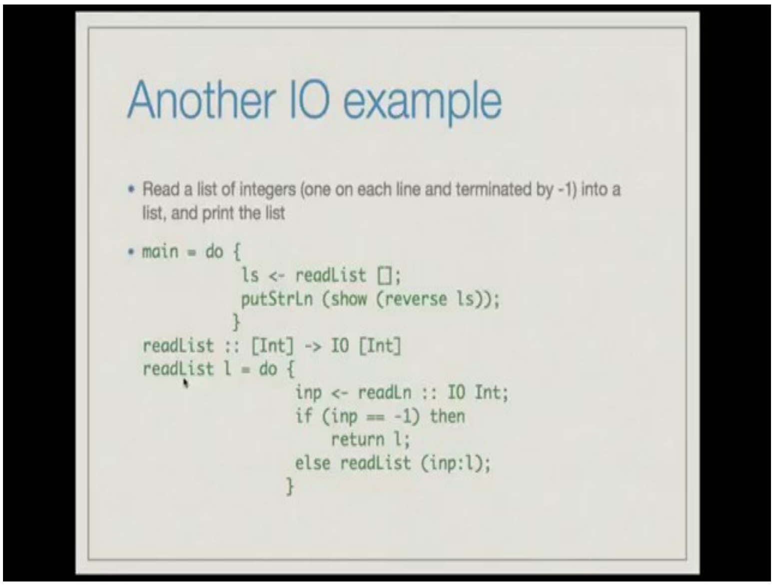
(Refer Slide Time: 25:15)



Strings are not the only things that can be read, we can read other values, but these other values have to belong to a type a that is an instance of the type class read. For this we use the function readLn, which is denoted readLn whose type is Read a=> IO a. All the basic types Int, Bool, Char etc are instances of Read. Therefore, you can use readLn to read Integers, Booleans Characters etc the basic type constructors also preserve readability.

So, for instance list [Int], the triple (Int, Char, Bool) etc are also instances of Read. Here is the syntax to read an integer: inp <- readLn :: IO Int , and bind the result to inp. So, since readLn is supposed to cater to the reading of any of these types that belongs to class Read you need to specify, which type you want the input to be.

(Refer Slide Time: 26:38)



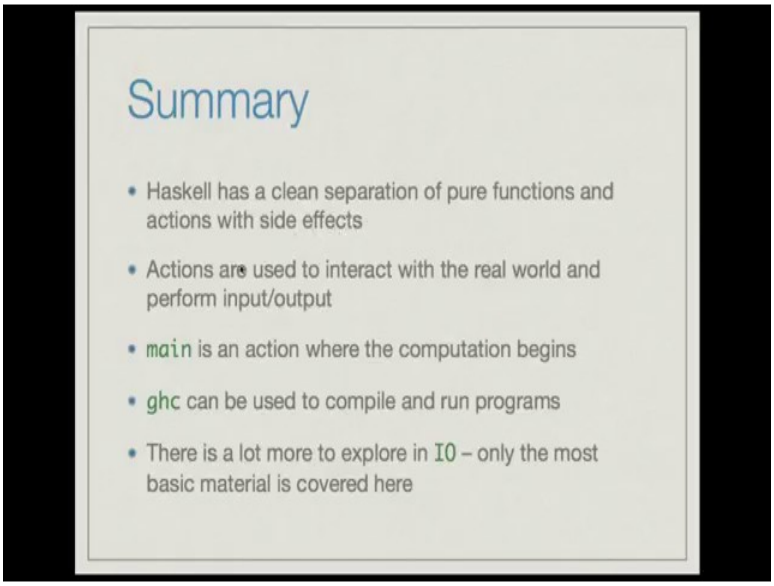
So, here is an example, in this example we read a list of integers one on each line and finally, terminated by -1 into a list and print the list.

main = do {
ls <- readList [];
putStrLn (show (reverse ls));;
}

The return value of readList is empty list. So, readList is a function you are providing the empty list as an input to read list. It is of type [Int] -> IO [Int]. So, it produces some side effects and its return value is a list of integers, which is what is bound to the ls here and then it so happens that readList reads the list in reverse order.

So, we show the reverse of ls and invoke putStrLn on it , in this way we read a bunch of integers one on each line and output them as a list. Here is the description of read list. read list l equals do readLn as an integer, this you denote by saying inp <- readLn :: IO Int and bind the return value to inp. If inp is -1, then, there is nothing more to do, so you just return l, which is the list that was provided as input else you have read the input you add the input to the list by saying inp:l you add it to head of the list and you proceed with the readList function presumably reading more input.

(Refer Slide Time: 28:27)



In 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 the action where the computation begins, ghc can be used to compile and run programs. So, there is a lot more to explore in IO but only the most basic material is covered here which suffices for rudimentary interaction with users.