

Introduction To Haskell Programming

Prof. S. P. Suresh

Chennai Mathematical Institute

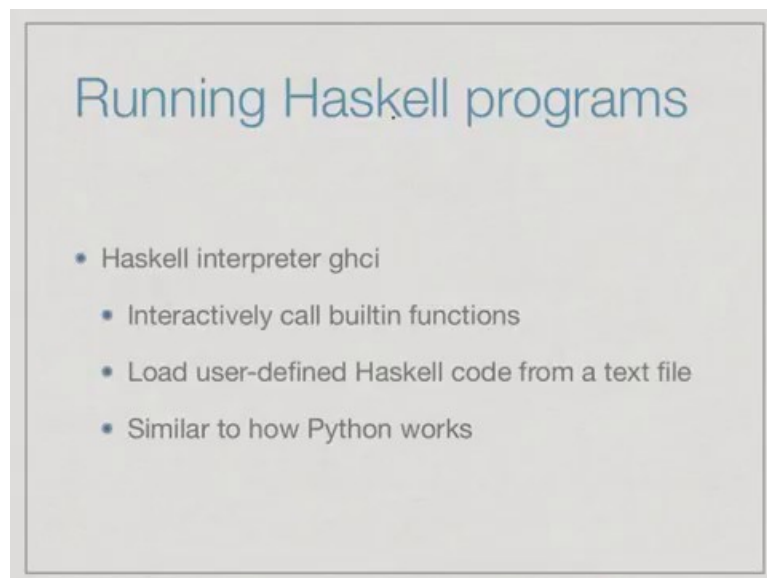
Module # 01

Lecture - 04

Running Haskell Programs

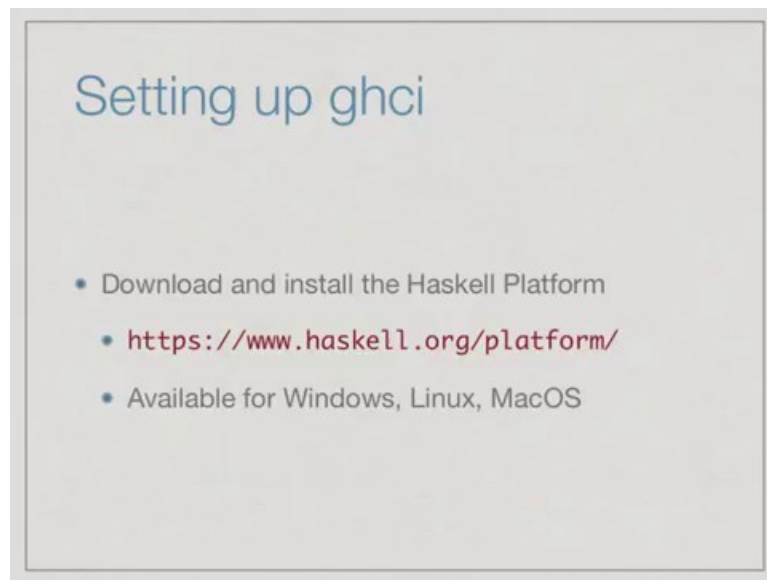
Having seen how to write Haskell programs, let us now look at how to run them on the computer.

(Refer Slide Time: 00:08)



The easiest way to run a Haskell program is to use the Haskell interpreter ghci. Using ghci will be very familiar to people who have used languages like python. So, ghci is an interpreter, you load ghci and then you can interact with it. You can directly call built in functions or you can load user defined Haskell code which you have previously entered into a text file.

(Refer Slide Time: 00:35)



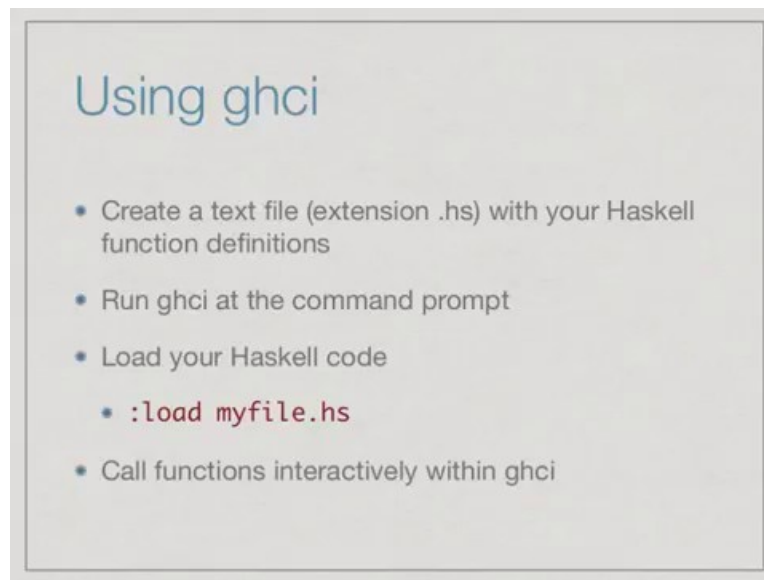
ghci is freely available on the internet; the easiest way to get ghci working on your system is to download what is called the Haskell platform from the web URL given here.

(Refer Slide Time: 00:51)



So, here is a screenshot of what the Haskell platform website looks like, as you can see you can download versions of the Haskell platform for any operating system for windows, for Mac OS and for Linux.

(Refer Slide Time: 01:10)



So, how do we actually use ghci? We first create a text file with the extension ‘.hs’ to indicate that it is the Haskell file containing your Haskell function definitions exactly as we wrote them in our previous lecture, then we run ghci from the command line. Inside ghci you can then feed it further commands and the most basic command you need to know is a command to load Haskell code, this command is written as `:load` followed by the file name, note the use of colon. So, colon is an indicator to ghci that it has to perform some internal action, anything without a colon is interpreted as a Haskell function that it must evaluate. So, in this way you can interactively call function from within ghci, let us look at some examples.

(Refer Slide Time: 02:01)

```
boolean.hs      factorial-fail.hs    factorial-good.hs
factorial-positive.hs  factorial.hs      intreverse.hs
madhavan@dolphinair:...o-lectures/week1/code$ vi factorial.hs
madhavan@dolphinair:...o-lectures/week1/code$ ghci
GHCi, version 7.8.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> :load factorial.hs
[1 of 1] Compiling Main                ( factorial.hs, interpreted )
Ok, modules loaded: Main.
*Main> factorial 7
5040
*Main> factorial 5
120
*Main> 4 + 5
9
*Main> True || False
True
*Main> div 7 3
2
*Main> sqr x = x * x
<interactive>:8:7: parse error on input '='
*Main> |
```

So, here is the command window in a Unix like system. So, in this we have already created some text files, but let us start from scratch, so supposing we want to create a version of factorial program. So, we open an editor like vi or Linux. So, let us say vi factorial.hs

(Refer Slide Time: 02:22)



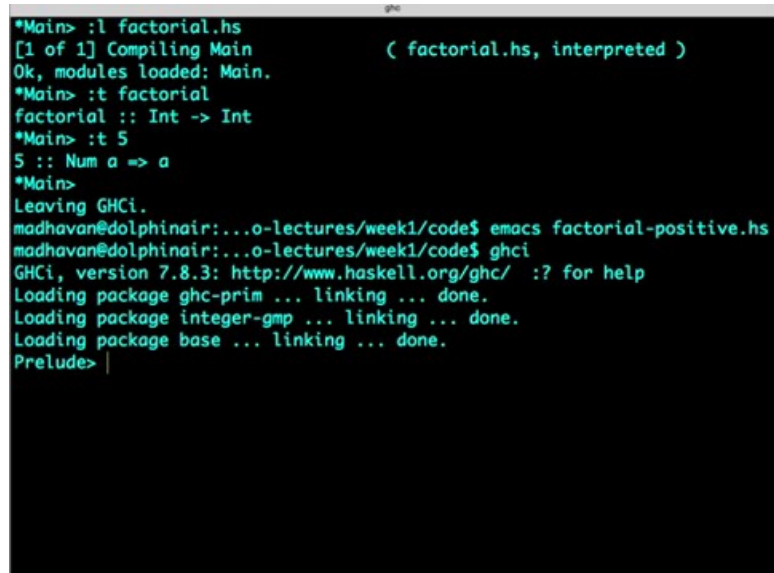
```
vim
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * factorial (n-1)
```

So, this is the first factorial program that we wrote, so there is an error, so let us fix that. So, it says that the type of factorial is Int -> Int, factorial 0 = 1 and factorial n = n * factorial (n-1). So, now, we can exit from the editor ((Refer Time: 02:37)), save the file and load ghci. So, ghci will load up and give us a command prompt of its own, which in this case is with the prefix Prelude. Now, we can load the file that we just created using the :load command that we have discussed and now if all is well you will get message like this saying it is okay. Having done this, we can now ask it to evaluate versions of factorial like factorial of 7 is 5040, factorial of 5 if you want to check a familiar number is 120 and so on.

You can also evaluate built in functions, for instance you can ask 4+5. So, you can use it as a calculator, this is pretty much how you will use it in things like python, you can say true or false or you can say div 7 3. So, it is very similar to the python interpreter if you use that, in that you can invoke any built in function or any function which you have defined in the file that has been loaded. As we will see one of the things that you cannot do in this, which you can do in a python interpreter is to define functions here. So, I cannot write a new function here which says something like `sqr x = x*x`. So, this kind of a definition which is allowed in

python, is not allowed in the interpreter, you have to write this in a separate file and then load.

(Refer Slide Time: 03:56)



```
*Main> :l factorial.hs
[1 of 1] Compiling Main                ( factorial.hs, interpreted )
Ok, modules loaded: Main.
*Main> :t factorial
factorial :: Int -> Int
*Main> :t 5
5 :: Num a => a
*Main>
Leaving GHCi.
madhavan@dolphinair:...o-lectures/week1/code$ emacs factorial-positive.hs
madhavan@dolphinair:...o-lectures/week1/code$ ghci
GHCi, version 7.8.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> |
```

So, as a shortcut for instance you can use `:l` instead of `load`, first of all if you say `:h` you get a list of all the possible commands that you can do in `ghci` in case you want to investigate, but `:l` is a short form. So, I can say `:l factorial.hs` and now it will just replace the old `factorial` by a new `factorial`. The other thing is that you can query types, for instance I can now ask what is the type of `factorial` and it tells me that the type of `factorial` is `Int->Int`.

So, if you have a function whose type you are unsure of you can ask this, but beware that the types reported here can be more complicated than the types we have seen. For instance, you might ask what is the type of a number 5, the number 5 you would expect as type `Int`, but actually it gives you this complicated thing which says `Num a => a`. We will see later what these things mean.

But, for functions that you define you can verify that the types are indeed what is say they are. So, let us get out of the interpreter for a moment and now let us look at this `factorial` that we wrote. So, I have got the same code in a file called `factorial-positive` and this is to indicate that this `factorial` handles only the positive case. So, this is the same `factorial` we just used except I added now a comment. A comment in Haskell starts with two hyphens on the first two columns of line. So, the first line is a comment which says this is a `factorial` that does not handle negative inputs, which we already know, but lets see what it does.

(Refer Slide Time: 05:39)

```
ghc
Prelude> :l factorial-positive.hs
[1 of 1] Compiling Main                ( factorial-positive.hs, interpreted )
Ok, modules loaded: Main.
*Main> factorial (-1)
^C
^D
^C
^A^Z
[1]+  Stopped                  ghci
madhavan@dolphinair:...o-lectures/week1/code$
madhavan@dolphinair:...o-lectures/week1/code$ kill %

[1]+  Stopped                  ghci
madhavan@dolphinair:...o-lectures/week1/code$
madhavan@dolphinair:...o-lectures/week1/code$ vi factorial-good.hs
madhavan@dolphinair:...o-lectures/week1/code$ ghci
GHCi, version 7.8.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> :l factorial-good.hs
[1 of 1] Compiling Main                ( factorial-good.hs, interpreted )
Ok, modules loaded: Main.
*Main> |
```

So, if I go back to ghci and I load this factorial-positive.hs and I ask it to do factorial say -1, then as we would expect it goes into an infinite computation of factorial -2 , -3 and so on. Because, we have not handled the negative case and the only way to get out of this is to physically break the program from running, and one way to do this is use a control C or control D or nothing works unfortunately.

So, we have to figure out a way to abort the program. So, now, instead we have written a better version which we had done already in the class, which said if $n < 0$ then you negate the factorial. So, now this is a factorial with negative inputs.

(Refer Slide Time: 06:45)

```
ghc
*Main> factorial (-4)
24
*Main>
Leaving GHCi.
madhavan@dolphinair:...o-lectures/week1/code$ ls
boolean.hs          factorial-fail.hs    factorial-good.hs
factorial-positive.hs  factorial.hs        intreverse.hs
madhavan@dolphinair:...o-lectures/week1/code$ more factorial-fail.hs
-- Factorial with pattern match failure

factorial :: Int -> Int
factorial 0 = 1
factorial n
  | n < 0 = factorial (-n)
  | n > 1 = n * (factorial (n-1))
madhavan@dolphinair:...o-lectures/week1/code$ ghci
GHCi, version 7.8.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude>
```

So, now, if I load this version and now they say factorial of -4 for instance then it will be converted to +4 and give me factorial of 4. Now, we also saw that we could have this kind of factorial function, where we have made a mistake. So, instead of $n > 0$ we have written $n > 1$ and this says that there will need not be any case when factorial 1 works. So, let us see what happens when we load this.

(Refer Slide Time: 07:10)

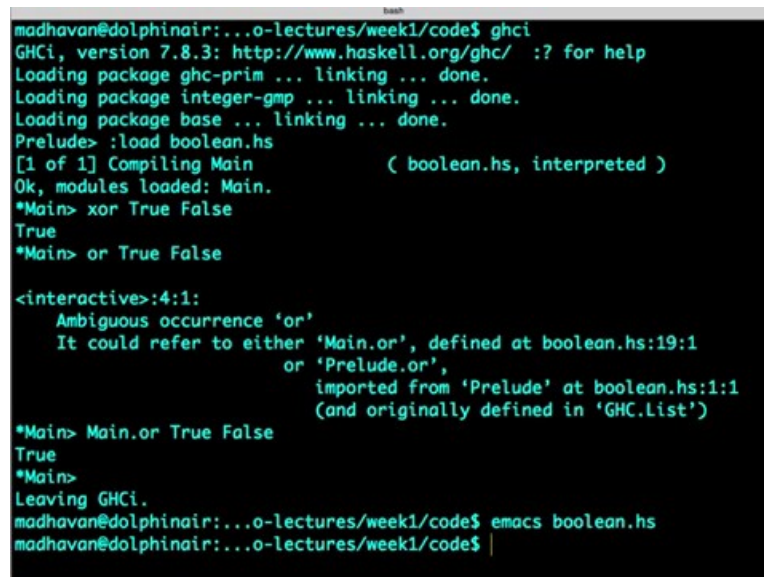
```
bash
Prelude> :l factorial-fail.hs
[1 of 1] Compiling Main                ( factorial-fail.hs, interpreted )
Ok, modules loaded: Main.
*Main> factorial 0
1
*Main> factorial 1
*** Exception: factorial-fail.hs:(4,1)-(7,34): Non-exhaustive patterns in
function factorial

*Main>
Leaving GHCi.
madhavan@dolphinair:...o-lectures/week1/code$ ls
boolean.hs          factorial-fail.hs    factorial-good.hs
factorial-positive.hs  factorial.hs        intreverse.hs
madhavan@dolphinair:...o-lectures/week1/code$ more boolean.hs |
```

So, now, if I say factorial of 0 it does not go into the problematic case, so it is fine. But, if I say factorial of 1 then I get some kind of pattern match failure. So, it says it is an exception

factorial fail and so on. So, the actual error message depends on the version of ghci, the error message written on the slides is slightly less expressive than this. So, you get some kind of error message and now finally, we have also written a version of the XOR and OR that we did. So, we have xor and or.

(Refer Slide Time: 07:54)



```
madhavan@dolphinair:...o-lectures/week1/code$ ghci
GHCi, version 7.8.3: http://www.haskell.org/ghc/ :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> :load boolean.hs
[1 of 1] Compiling Main                ( boolean.hs, interpreted )
Ok, modules loaded: Main.
*Main> xor True False
True
*Main> or True False

<interactive>:4:1:
  Ambiguous occurrence 'or'
    It could refer to either 'Main.or', defined at boolean.hs:19:1
                          or 'Prelude.or',
                          imported from 'Prelude' at boolean.hs:1:1
                          (and originally defined in 'GHC.List')

*Main> Main.or True False
True
*Main>
Leaving GHCi.
madhavan@dolphinair:...o-lectures/week1/code$ emacs boolean.hs
madhavan@dolphinair:...o-lectures/week1/code$
```

Now, here we have another problem which comes up and which you should be aware of. So, if I say load boolean.hs now if I say 'xor True False' there is no problem. But if I say 'or True False' then I get an error message saying that is confused by the word 'or', because the built in symbol for OR is the two vertical lines '||', but 'or' as an English word can also be used in a Haskell program.

So, 'or' is a built in function and it says that the built in function 'or' is in conflict with the 'or' which is defined in this file. Now of course, one could say Main.or and this Main is the version that we have just loaded or even better than this is to actually go back and edit the file and use a new name. So, it is conventional to use the prefix my, so whenever you have a function whose name looks like a familiar function it is best to rename it with something which distinguishes it from the familiar function.

(Refer Slide Time: 09:03)

```
madhavan@dolphinair:~$ ghci
GHCi, version 7.8.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> :l boolean.hs
[1 of 1] Compiling Main                ( boolean.hs, interpreted )
Ok, modules loaded: Main.
*Main> myor True False
True
*Main> |
```

So that now if I load this and say ‘myor True False’ it will work fine. So, we have seen how to load files into the interpreter, some kinds of error messages that you see. You can also query types. So, it is very easy to use this interpreter and since Haskell program tend to be small, it is not very difficult to debug them as we will see. So, please get familiar . Do download ghci onto your system and start playing around it.

(Refer Slide Time: 09:37)

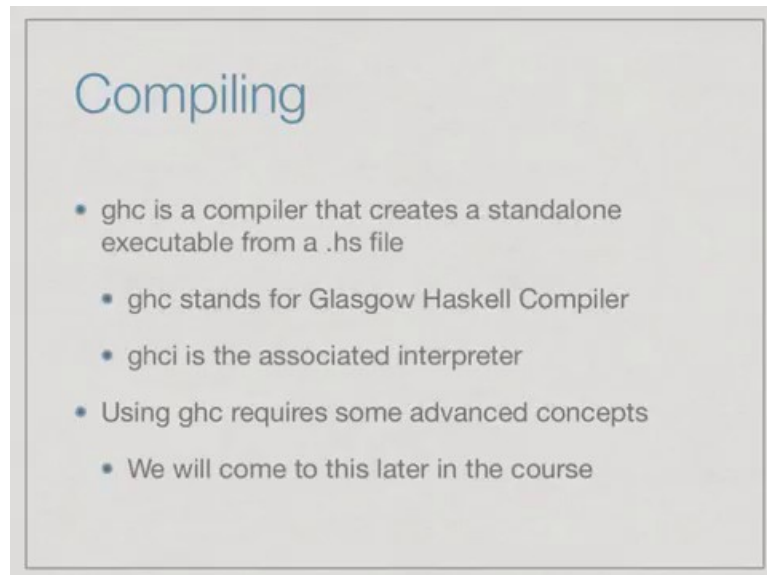
Caveats

- Cannot define new functions directly in ghci
 - Unlike Python
- Must create a separate .hs file and load it

So, as we mentioned when we were showing the use of ghci, one of the differences between ghci and python is that you cannot create function definitions on the fly. So, within the

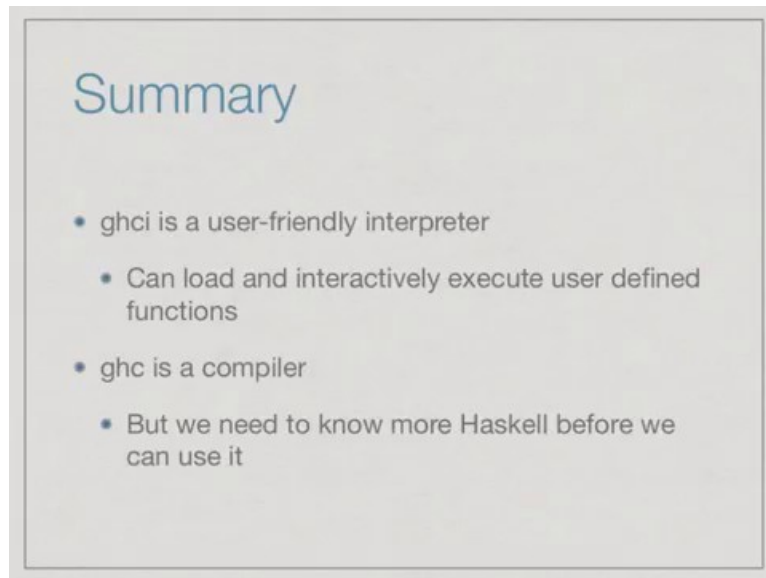
interpreter you cannot define a new function, you must load it as a separate function from an outside file. So, you must first create a .hs file and then load it into the interpreter.

(Refer Slide Time: 10:00)



So, this is an interpreter, but Haskell also comes with a full fledged compiler. So, ghc is a compiler that creates an executable file from a .hs file. So, ghc stands for the Glasgow Haskell compiler and ghci is the associated interpreter which is the one that we have seen. Now, unfortunately we cannot use ghc directly on the Haskell function that we have been writing. We need some more advanced concepts which we will see as we go along. So, at some point we will start compiling our code, but for the moment we will just stick to ghci the interpreter.

(Refer Slide Time: 10:33)



Summary

- ghci is a user-friendly interpreter
 - Can load and interactively execute user defined functions
- ghc is a compiler
 - But we need to know more Haskell before we can use it

So, to summarize the easiest way to run Haskell code is to use the interpreter ghci, which is quite user friendly and in which we can load and interactively execute user defined functions, there is an associated compiler called ghc but we need to know more Haskell before we can use it.