

C/CPS 506

Assignment Project Exam Help

Comparative Programming Languages

<https://powcoder.com>

Prof. Alex Ufkes

Add WeChat powcoder

Topic 8: Actions in Haskell

Notice!

**Obligatory copyright notice in the age of digital
delivery and online classrooms:**

Assignment Project Exam Help

<https://powcoder.com>

The copyright to this original work is held by Alex Ufkes. Students registered in course CCPS 505 can use this material for the purposes of this course but no other use is permitted, and there can be no sale or transfer or use of the work for any other purpose without explicit permission of Alex Ufkes.

Course Administration



Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

- Getting closer! Three more weeks.
- Don't forget about the assignments!



Let's Get Started!

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

Last Week

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

Declare **Pt** to be
an instance of **Eq**

Define what it means for two Pt2
variables to be considered equal

```
C:\HaskellCode\Test.hs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
Test.hs HaskellList
1 module Test where
2
3 data Pt = Pt3 Float Float Float
4         | Pt2 Float Float
5         deriving (Show)
6
7
8
9 instance Eq Pt where
10     (Pt2 x1 y1) == (Pt2 x2 y2) = (x1==x2 && y1==y2)
11
12
Haskell length: 1,623 lines: 99 Ln: 14 Col: 2 Sel: 0|0 Windows (CR LF) UTF-8 INS
```

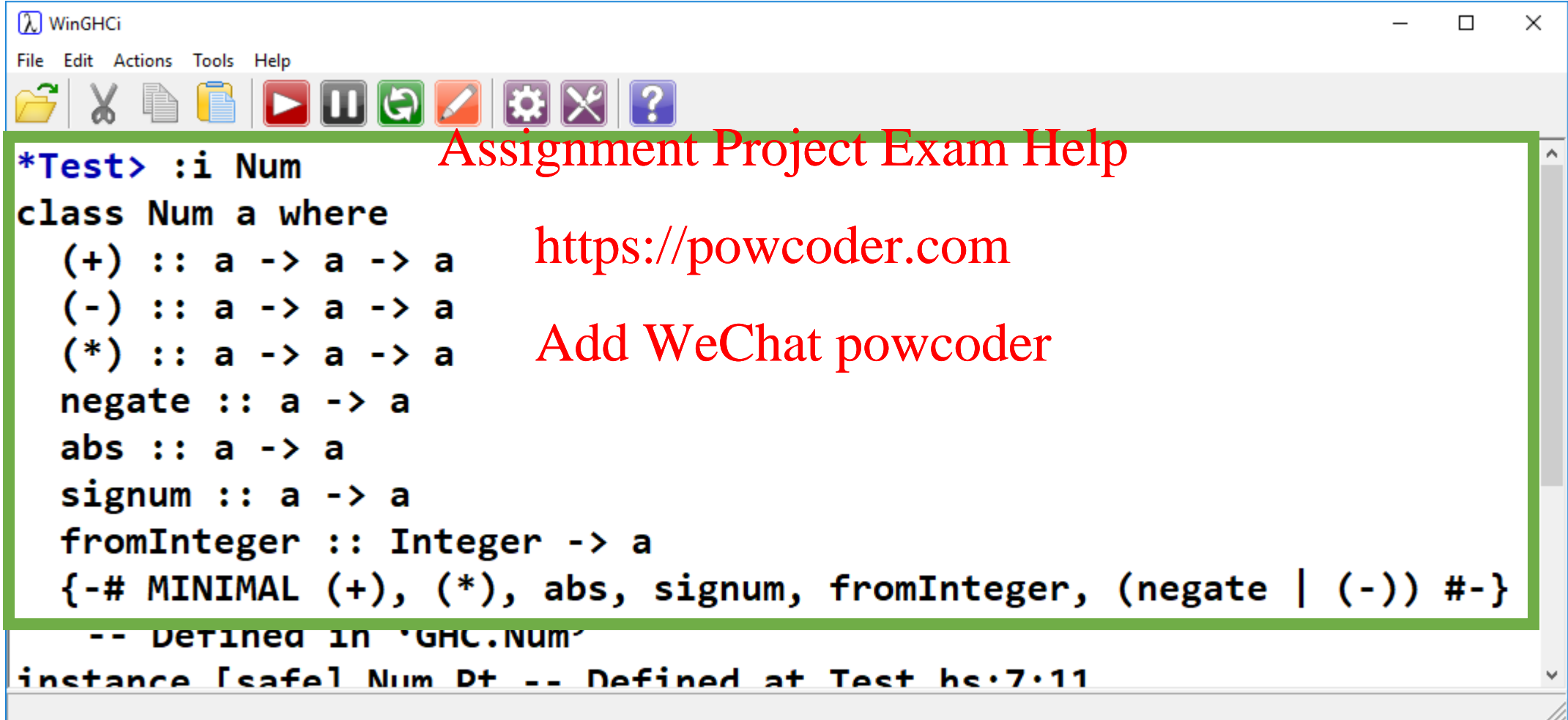
Last Week

Assignment Project Exam Help
<https://powcoder.com>
Add WeChat powcoder

```
C:\HaskellCode\Test.hs - Notepad++
File Edit Search View Encoding Language Settings Tools
Test.hs HaskellList.hs
1 module Test where
2
3 data Pt = Pt3 Float Float
4         | Pt2 Float Float
5         deriving (Show)
6
7 instance Num Pt where
8     (Pt2 x1 y1) + (Pt2 x2 y2) = Pt2 (x1 + x2) (y1 + y2)
9
10
11
12
Hasl length: 1,765 lines: 104 Ln: 11 Col: 3 Sel: 0|0
```

WinGHCi
File Edit Actions Tools Help
[1 of 1] Compiling Test (Test.hs, interpreted)
Test.hs:7:11: warning: [-Wmissing-methods]
• No explicit implementation for
 ‘*’, ‘abs’, ‘signum’, ‘fromInteger’, and
(either ‘negate’ or ‘-’)
• In the instance declaration for ‘Num Pt’
7 | instance Num Pt where
Ok, one module loaded.
*Test>

Last Week

A screenshot of a WinGHCi window. The window has a title bar with the WinGHCi logo and standard window controls. Below the title bar is a menu bar with 'File', 'Edit', 'Actions', 'Tools', and 'Help'. Underneath the menu bar is a toolbar with icons for file operations (open, save, copy, paste), execution (run, pause, refresh), and editing (undo, redo, settings, help). The main text area contains Haskell code for defining a Num instance. The code is as follows:

```
*Test> :i Num
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
  {-# MINIMAL (+), (*), abs, signum, fromInteger, (negate | (-)) #-}
  -- Defined in 'GHC.Num'
instance [safe] Num Dt -- Defined at Test.hs:7:11
```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

Last Week

```
C:\HaskellCode\Test.hs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
Test.hs HaskellList.hs
1 module Test where
2
3 data Pt = Pt3 Float Float Float
4         | Pt2 Float Float
5         --deriving (Show)
6
7 instance Show Pt where
8     show (Pt2 x y) =
9         "< " ++ (show x) ++ ", " ++ (show y) ++ " >"
10
11
12
13
```

Assignment Project Exam Help

<https://powcoder.com>

No longer need to derive Show, we've made our own

Add WeChat powcoder

- Use string concatenation to create a pleasing visual output for Pt2.
- In doing so, we make use of show as defined for Floats

Pure Code, Monads, Actions

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder



Every function is pure



Pure Functions: Functions that have no side effects.

Assignment Project Exam Help



<https://powcoder.com>

A function can be said to have a side effect if it has an observable interaction with the outside world aside from returning a value.

Add WeChat powcoder



- Modify global variable
- Raise an exception
- **Write data to display** or file

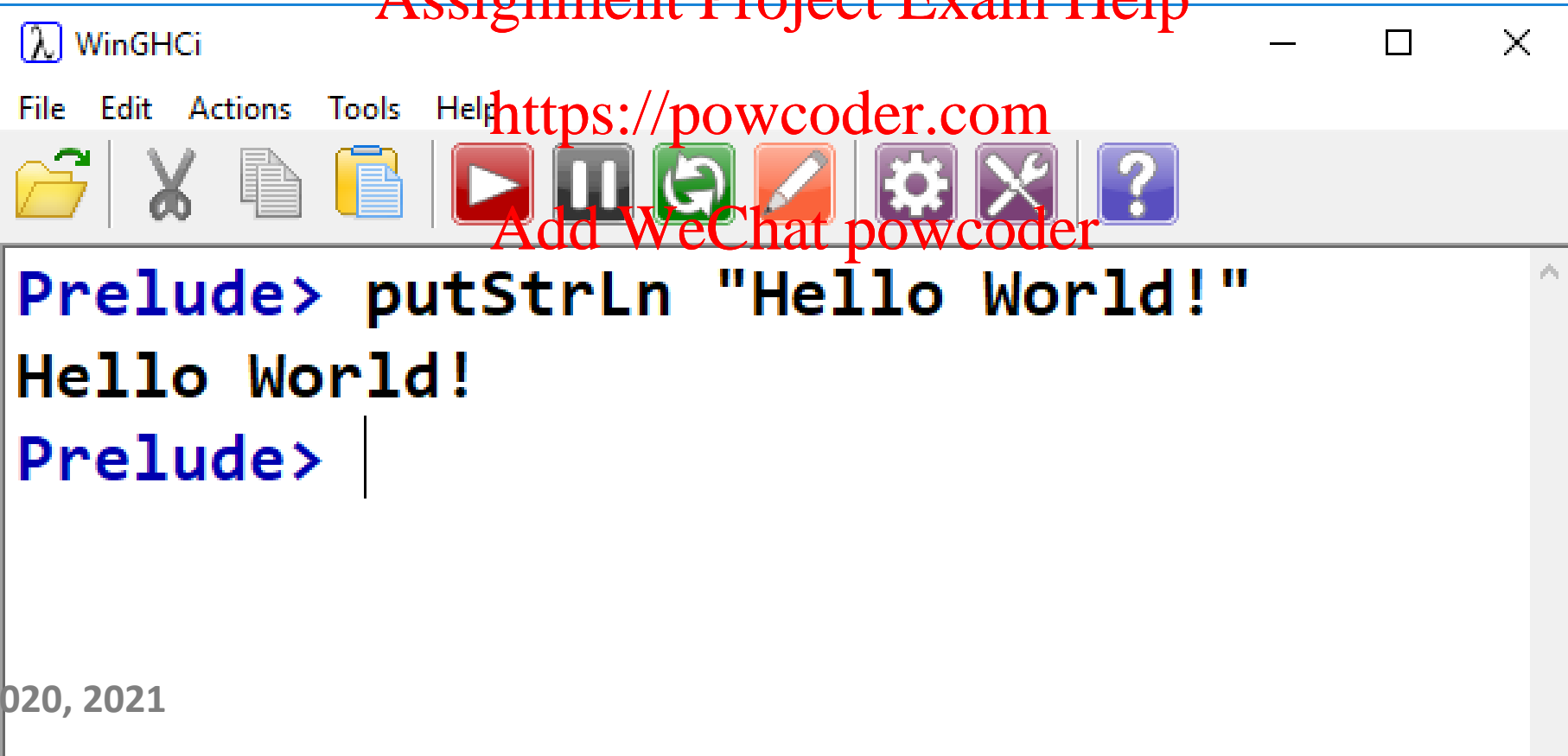
Write to Display

This was the very first thing we saw!

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

A screenshot of the WinGHCi window. The window has a title bar with the icon and text 'WinGHCi'. Below the title bar is a menu bar with 'File', 'Edit', 'Actions', 'Tools', and 'Help'. Under the menu bar is a toolbar with icons for file operations (folder, copy, paste, save), execution (play, stop, refresh), and settings (gear, wrench, help). The main area of the window shows a command prompt where the command 'Prelude> putStrLn "Hello World!'" has been entered and executed, resulting in the output 'Hello World!'. The prompt 'Prelude>' is followed by a vertical cursor bar.

```
Prelude> putStrLn "Hello World!"
Hello World!
Prelude> |
```

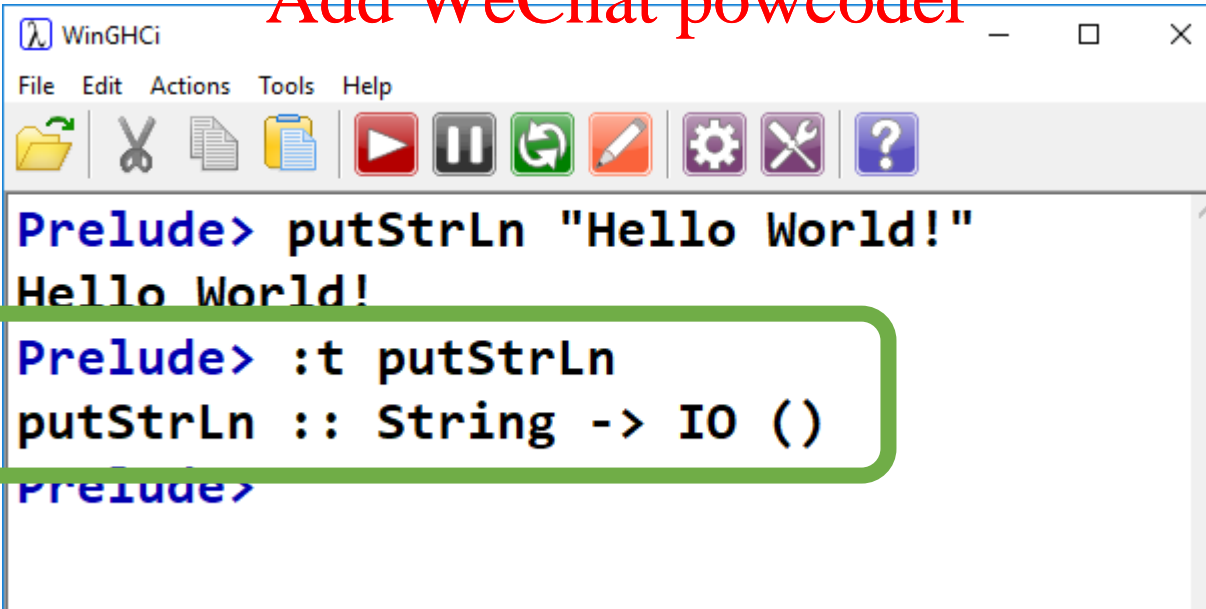
Haskell and I/O

- Haskell separates pure functions from computations where side effects must be considered
- Encodes side effect-producing functions with a specific type.
- We've already seen an example of this:

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder



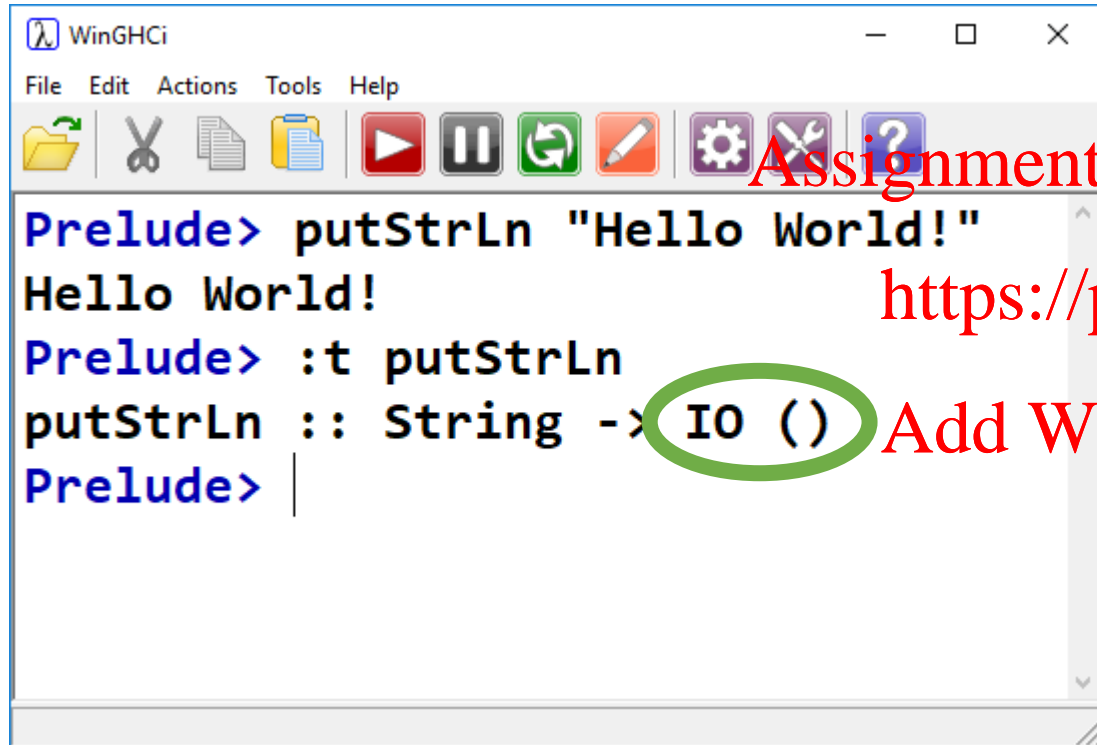
```
WinGHCi
File Edit Actions Tools Help
[Icons: Folder, Scissors, Document, Copy, Run, Pause, Refresh, Edit, Settings, Wrench, Help]

Prelude> putStrLn "Hello World!"
Hello World!

Prelude> :t putStrLn
putStrLn :: String -> IO ()

Prelude>
```

Haskell and I/O



```
Prelude> putStrLn "Hello World!"
Hello World!
Prelude> :t putStrLn
putStrLn :: String -> IO ()
Prelude> |
```

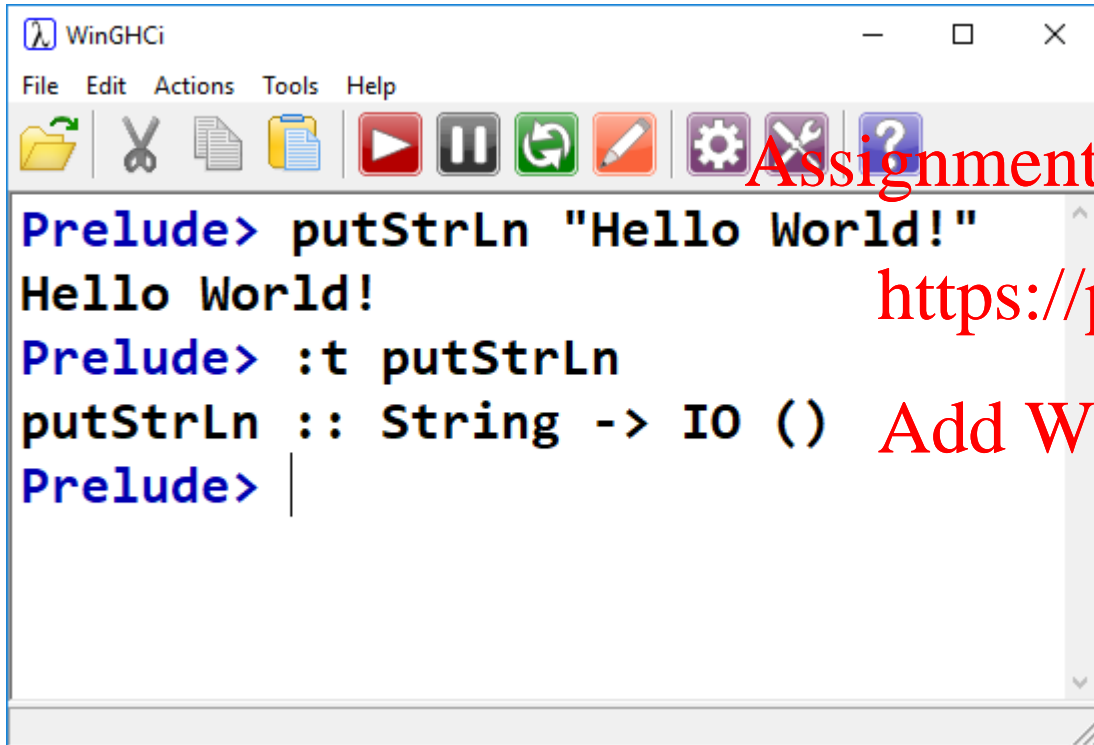
Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

- The actual **act** of printing to the screen does not occur as a result of a function call.
- Printing to the screen is an **action**.
- Actions are **values**, they have a type!
- `putStrLn` accepts a **String** argument.
- What it returns is an action of type **IO()**

Haskell and I/O

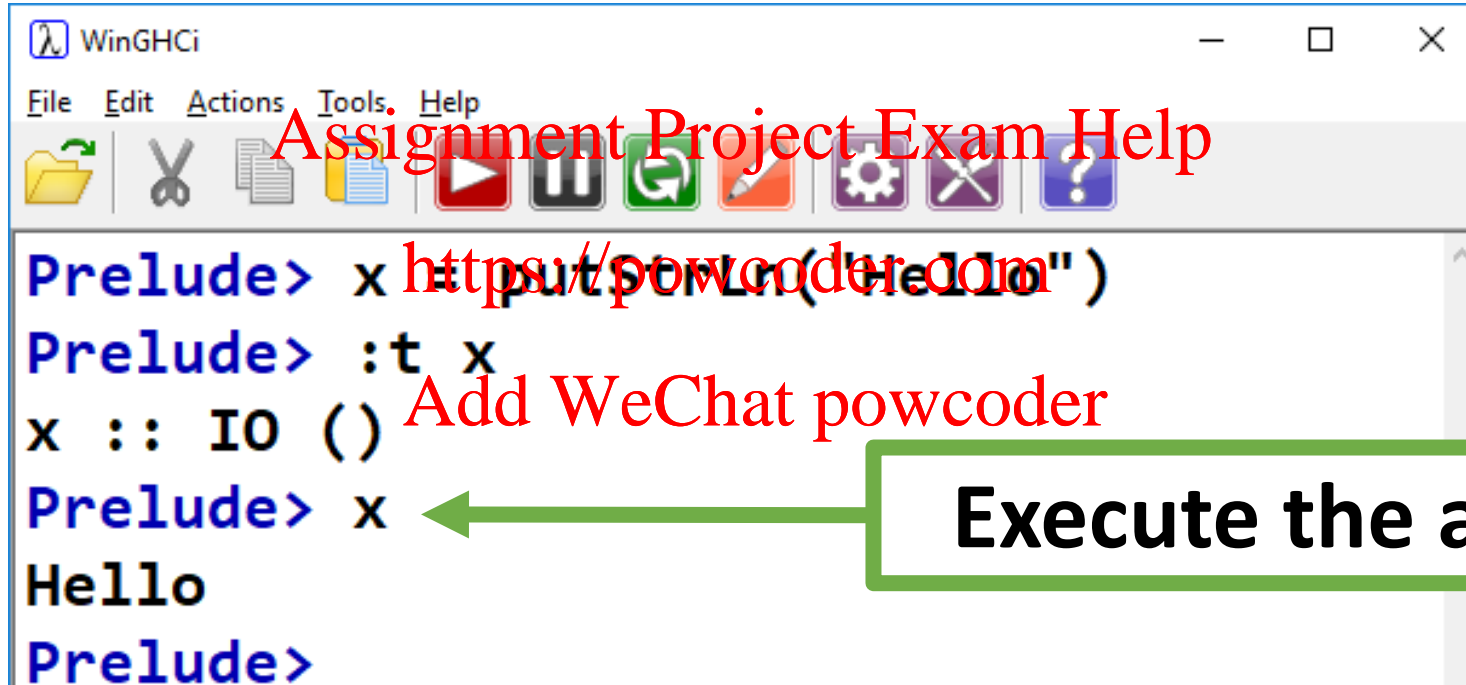


```
WinGHCi
File Edit Actions Tools Help
[Icons]
Prelude> putStrLn "Hello World!"
Hello World!
Prelude> :t putStrLn
putStrLn :: String -> IO ()
Prelude> |
```

Speaking precisely:

- **putStrLn** is a *function* (no side effects!)
 - Takes a `String` as an input argument
 - Returns an action, whose type is `IO ()`
- When the `IO ()` action is executed, it returns `()`.
- This can be read as an empty tuple.
- The action, when executed, produces a side effect.
- The **putStrLn** function, strictly speaking, does **not**.

Haskell and I/O



The screenshot shows the WinGHCi window with a menu bar (File, Edit, Actions, Tools, Help) and a toolbar. The command prompt shows the following sequence of commands and output:

```
Prelude> x = putStrLn("Hello")
Prelude> :t x
x :: IO ()
Prelude> x
Hello
Prelude>
```

Red text overlays the image: "Assignment Project Exam Help" and "https://powcoder.com" are at the top, and "Add WeChat powcoder" is in the middle. A green arrow points from a box labeled "Execute the action" to the 'x' in the command 'Prelude> x'.

- Actions are values, just like strings and numbers.
- They are completely inert – they do not affect the real world until executed.

Haskell and I/O

```
Prelude> getLine
Hello
"Hello"
Prelude> :t getLine
getLine :: IO String
Prelude>
```

- We can also look at `getLine`
- `getLine` returns an IO action also
- It returns a `String` (`IO String` vs `IO ()`)
- Ordinary Haskell *evaluation* doesn't cause actions to be executed.
- GHCi will execute actions for us, as seen previously.

Just remember: *actions* are not *functions*.

Functions are pure. Actions (specifically IO actions),
when executed are not.

Assignment Project Exam Help

Functions are *evaluated*, actions are *executed* or *run*
<https://powcoder.com>

Actions are values. Actions can be returned by functions
or passed as arguments.

Actions have a type. We've seen one so far, **IO**

Actions can only be executed from within other actions.

Assignment Project Exam Help

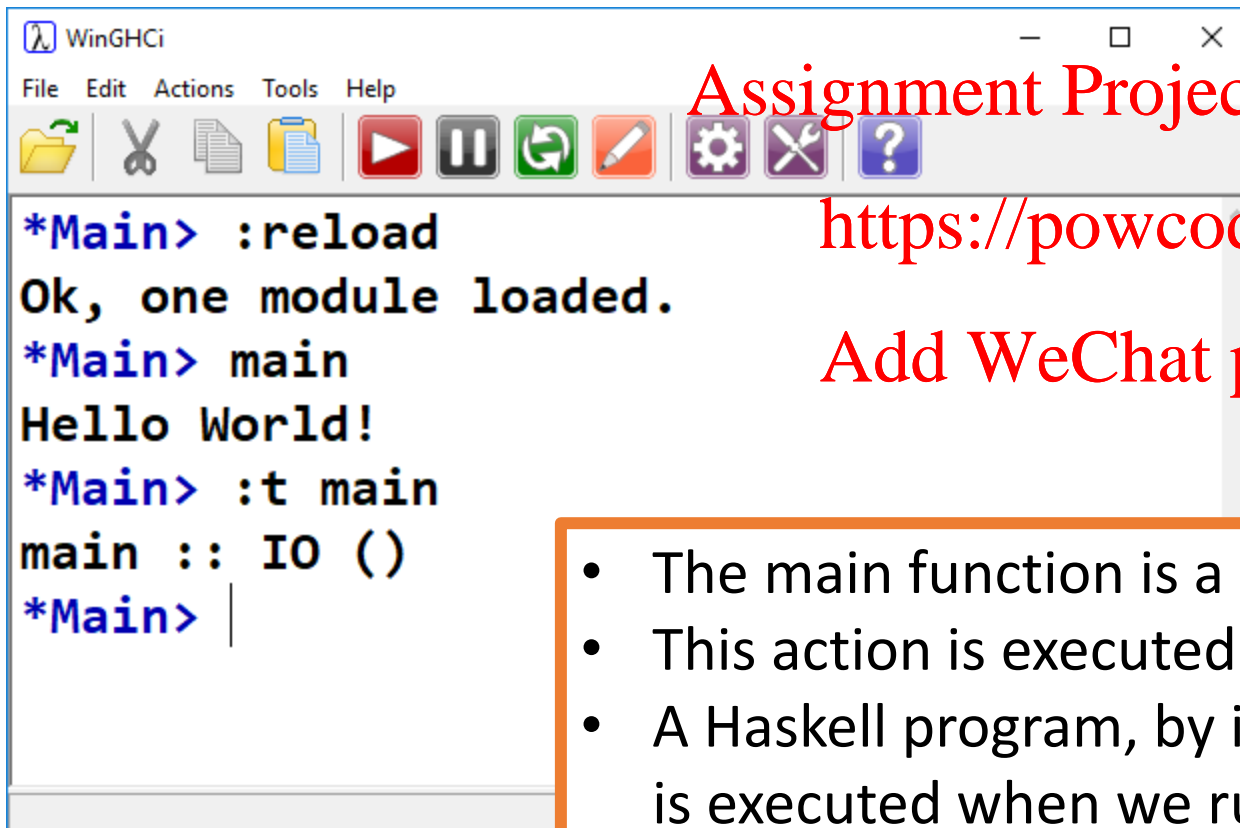
A compiled Haskell program begins by executing a
single action – `main :: IO()`

<https://powcoder.com>
Add WeChat powcoder

https://wiki.haskell.org/Introduction_to_Haskell_IO/Actions

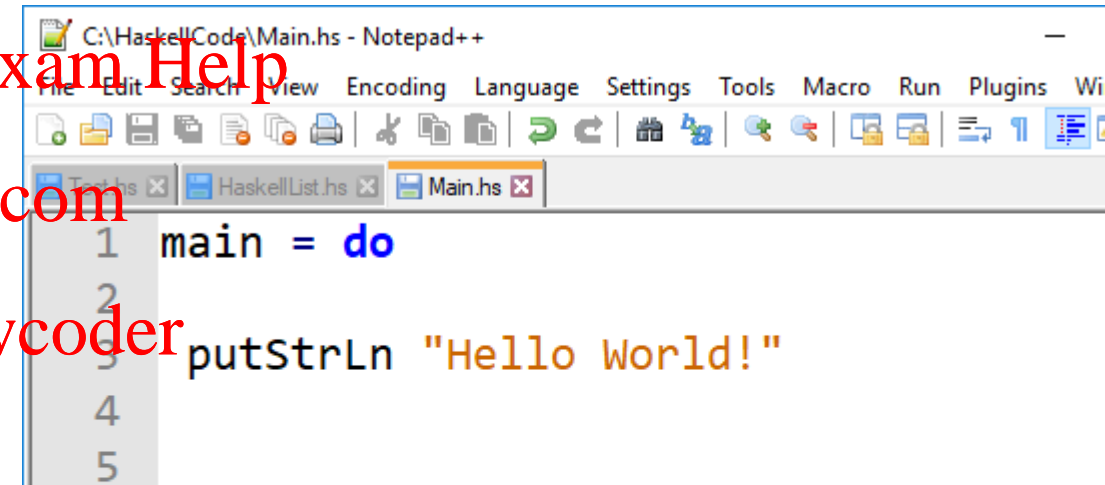
main :: IO()

Recall: Every compiled Haskell program must have a main function:



A screenshot of the WinGHCi terminal window. The window has a menu bar with 'File', 'Edit', 'Actions', 'Tools', and 'Help'. Below the menu is a toolbar with icons for file operations and execution. The terminal text shows the following sequence of commands and output:

```
*Main> :reload
Ok, one module loaded.
*Main> main
Hello World!
*Main> :t main
main :: IO ()
*Main> |
```



A screenshot of a Notepad++ editor window titled 'C:\HaskellCode\Main.hs - Notepad++'. The window has a menu bar with 'File', 'Edit', 'Search', 'View', 'Encoding', 'Language', 'Settings', 'Tools', 'Macro', 'Run', 'Plugins', and 'Window'. The editor shows the following Haskell code:

```
1 main = do
2
3   putStrLn "Hello World!"
4
5
```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

- The main function is a single action
- This action is executed when the program is run.
- A Haskell program, by itself, is a single action that is executed when we run the program.

Staying Grounded

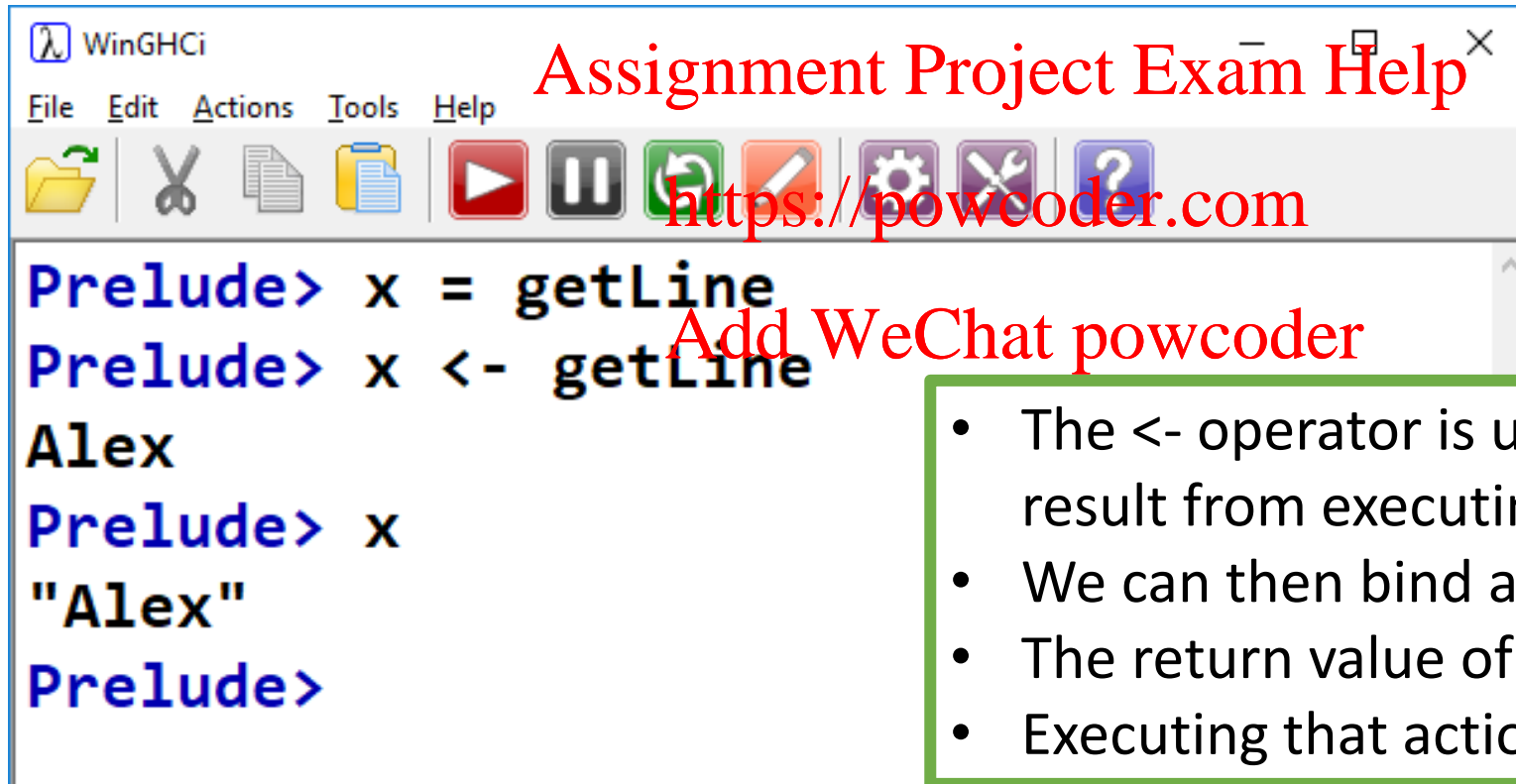
- A Haskell program begins with the execution of a single action (`main :: IO()`)
 - Functions that return actions are often incorrectly referred to as actions.
- From within this action, any number of additional actions can be executed
- Pure functions can also be called/evaluated from within actions!
- However – actions cannot be executed from within pure functions.
- If we try, Haskell will infer the type of the no-longer-pure function as an action.

Staying Grounded

- An action can be thought of as a *recipe*
- This recipe (in the case of IO) is a list of instructions that would produce side effects.
- *The act of creating this recipe does not have side effects.*
- The recipe can be the output of a pure function.
- Same inputs to the function, same recipe.

IO Actions

We can use the <- operator to execute:



The screenshot shows the WinGHCi window with the following text:

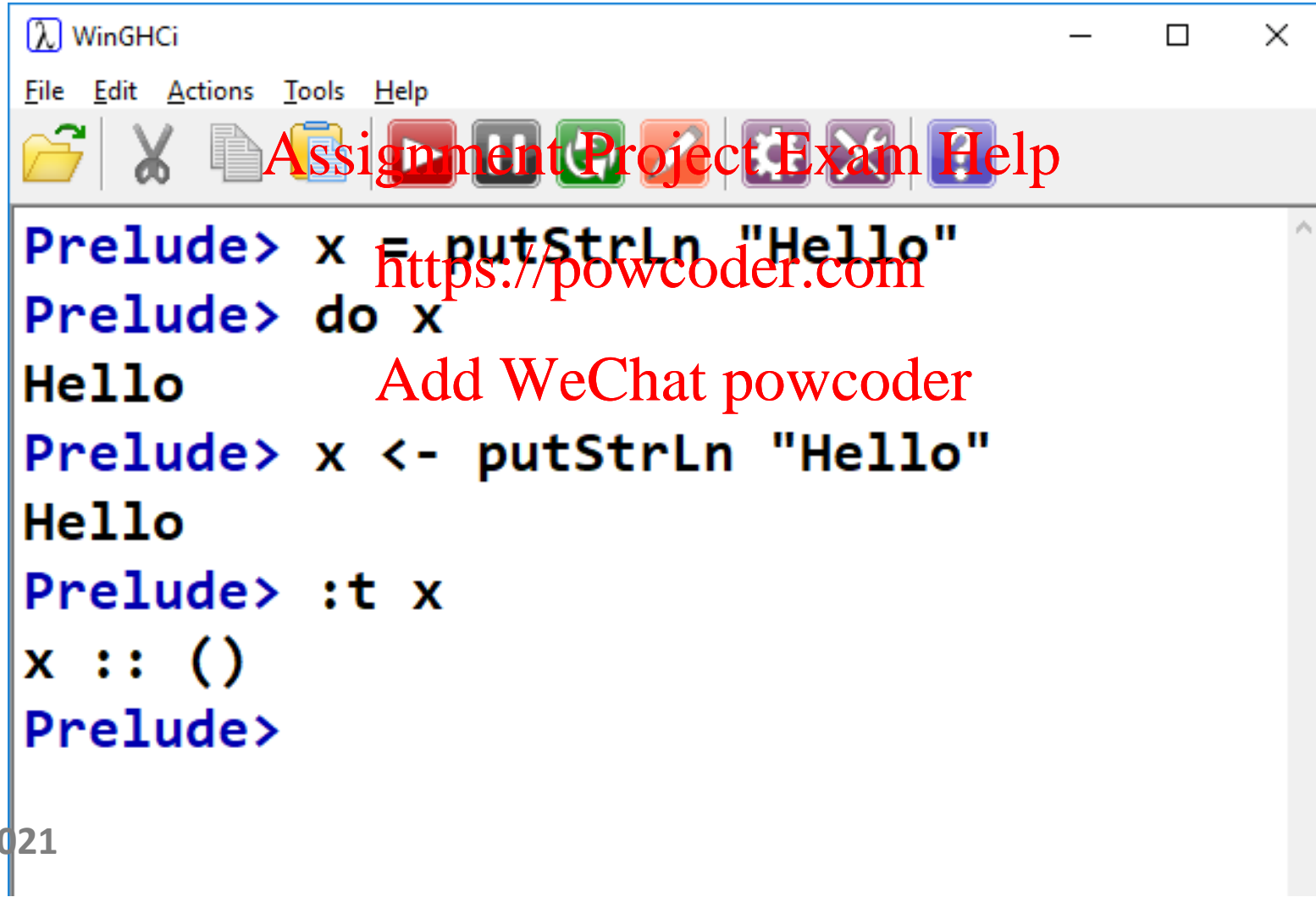
```
WinGHCi
File Edit Actions Tools Help
[Icons: Folder, Scissors, Document, Clipboard, Play, Pause, Refresh, Erase, Settings, Wrench, Question Mark]

Prelude> x = getLine
Prelude> x <- getLine
Alex
Prelude> x
"Alex"
Prelude>
```

Assignment Project Exam Help
<https://powcoder.com>
Add WeChat powcoder

- The <- operator is used to pull out the result from executing an IO action.
- We can then bind a name to it.
- The return value of getLine is an action.
- Executing that action returns a String.

IO Actions



A screenshot of the WinGHCi window. The title bar says 'WinGHCi'. The menu bar includes 'File', 'Edit', 'Actions', 'Tools', and 'Help'. The toolbar contains icons for file operations (folder, copy, paste, delete) and development tools (run, debug, settings, etc.). The main text area shows the following Haskell code:

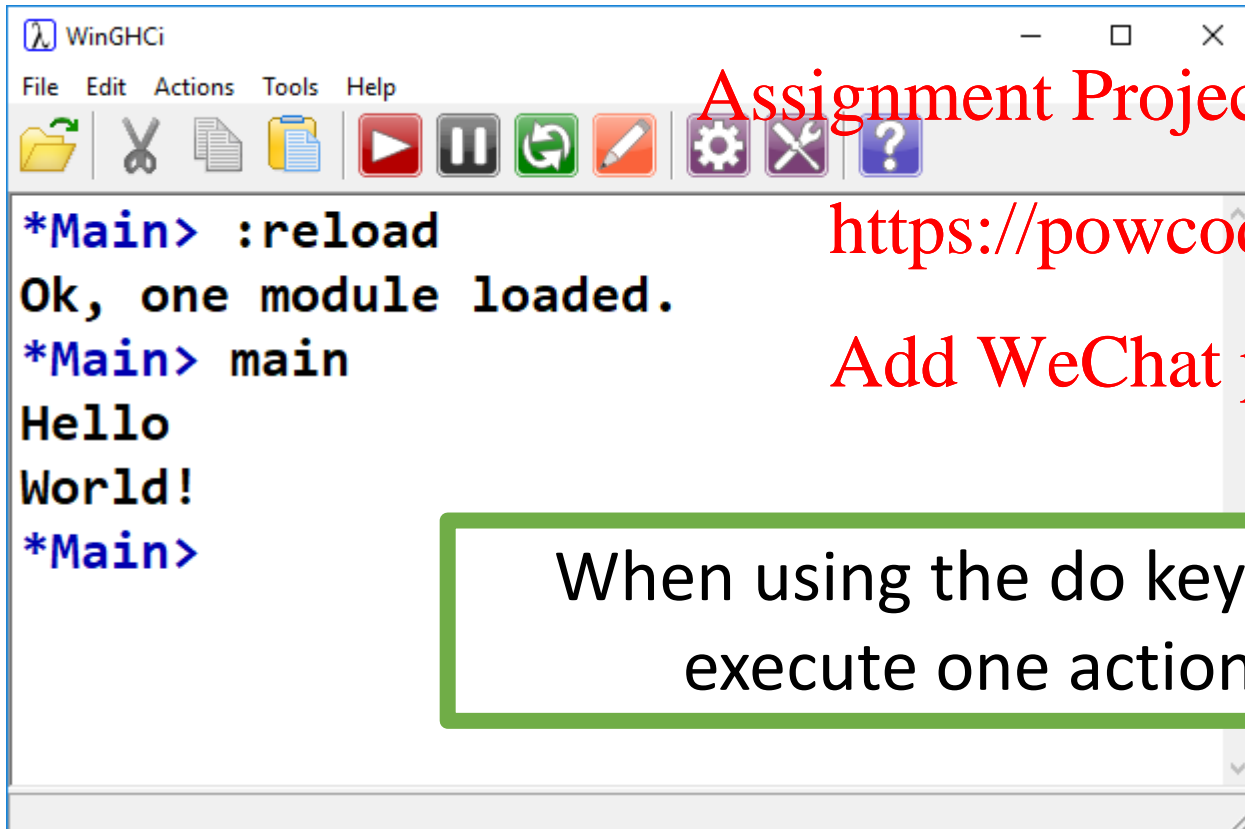
```
Prelude> x = putStrLn "Hello"
Prelude> do x
Hello
Prelude> x <- putStrLn "Hello"
Hello
Prelude> :t x
x :: ()
Prelude>
```

Red text overlays are present on the image:

- Assignment Project Exam Help
- <https://powcoder.com>
- Add WeChat powcoder

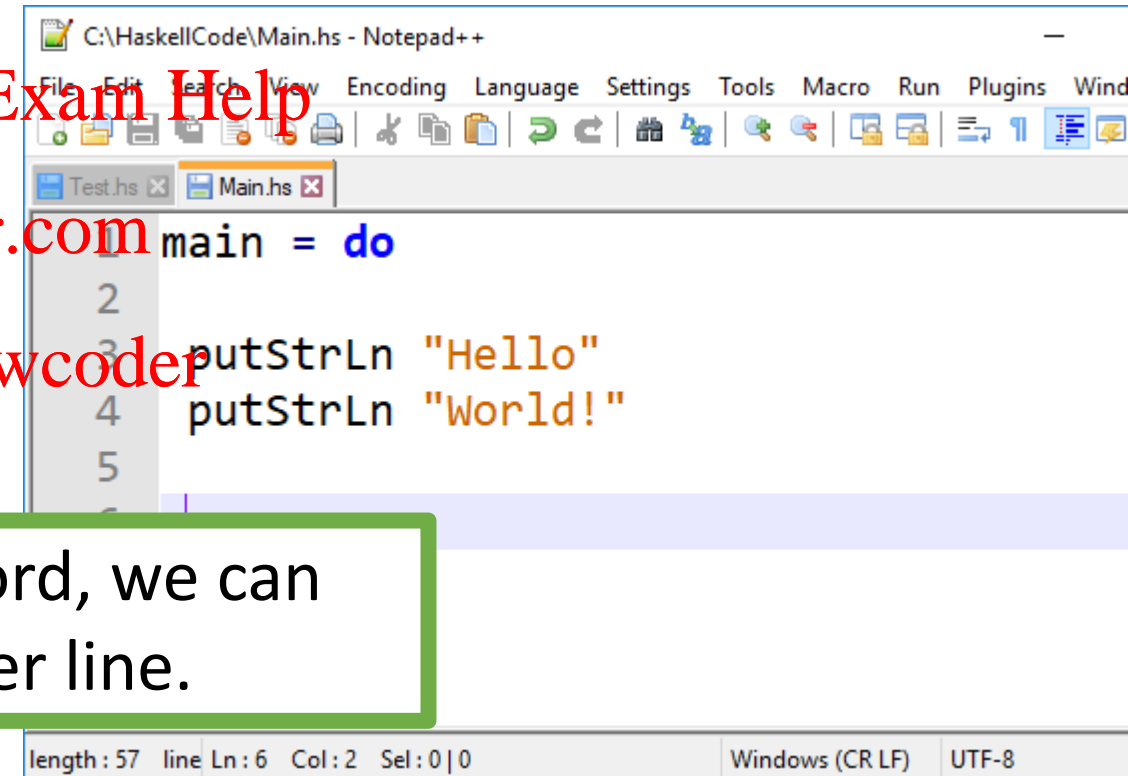
Combining Actions

We can do this using the **do** keyword:



A screenshot of the WinGHCi terminal window. The window has a menu bar with 'File', 'Edit', 'Actions', 'Tools', and 'Help'. Below the menu is a toolbar with icons for file operations and execution. The terminal text shows the following sequence of commands and outputs:

```
*Main> :reload
Ok, one module loaded.
*Main> main
Hello
World!
*Main>
```



A screenshot of a Notepad++ editor window titled 'C:\HaskellCode\Main.hs - Notepad++'. The window has a menu bar with 'File', 'Edit', 'Search', 'View', 'Encoding', 'Language', 'Settings', 'Tools', 'Macro', 'Run', 'Plugins', and 'Window'. The editor shows the following Haskell code:

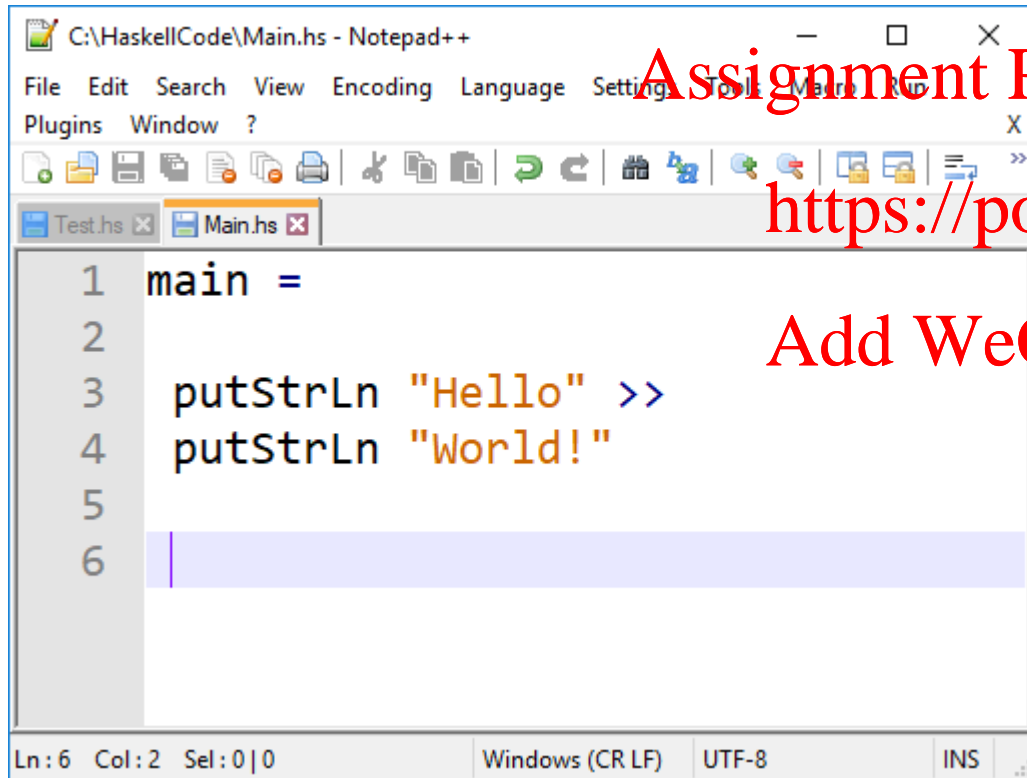
```
1 main = do
2
3   putStrLn "Hello"
4   putStrLn "World!"
5
```

The status bar at the bottom indicates 'length: 57 line Ln: 6 Col: 2 Sel: 0|0' and 'Windows (CR LF) UTF-8'.

When using the **do** keyword, we can execute one action per line.

Combining Actions

do is syntactic sugar for **>>**



```
1 main =
2
3   putStrLn "Hello" >>
4   putStrLn "World!"
5
6
```

Assignment Project Exam Help

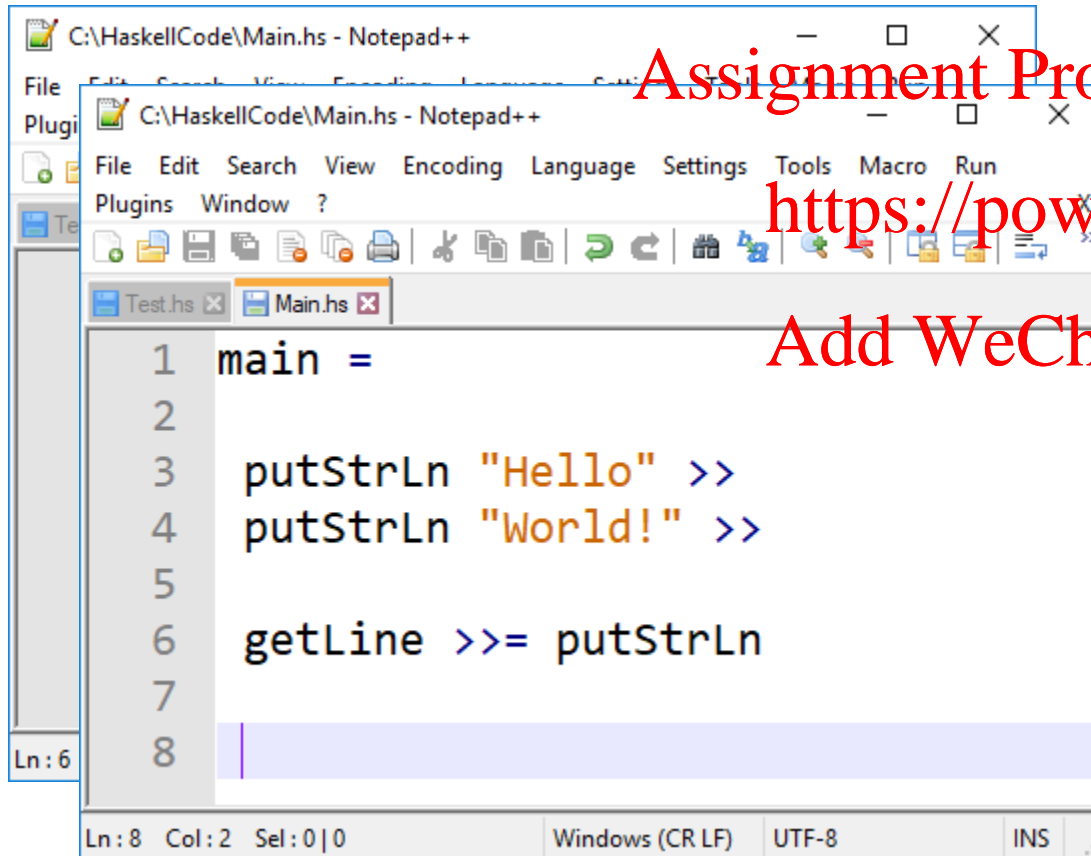
<https://powcoder.com>

Add WeChat powcoder

- **>>** says execute this, then this.
- If the first action produces a result, it is discarded.
- What if we want to use the result?
- Use the **>>=** operator to pipe the result into the next action.

Combining Actions

do is syntactic sugar for **>>**



```
1 main =  
2  
3   putStrLn "Hello" >>  
4   putStrLn "World!" >>  
5  
6   getLine >=> putStrLn  
7  
8
```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

- **>>** says execute this, then this.
- If the first action produces a result, it is discarded.
- What if we want to use the result?
- Use the **>=>** operator to pipe the result into the next action.
- Here, we grab a string using `getLine`, and display it using `putStrLn`
- `getLine` returns an action that produces a string
- `putStrLn` takes string as an argument.

C:\HaskellCode\Main.hs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run
Plugins Window ?

Test.hs x Main.hs x

```
1 main =  
2  
3   putStrLn "Hello" >>  
4   putStrLn "World!" >>  
5  
6   getLine >>= putStrLn  
7  
8
```

Ln: 8 Col: 2 Sel: 0 | 0 Windows (CR LF) UTF-8

WinGHCi

File Edit Actions Tools Help

```
*Main> :reload  
[1 of 1] Compiling Main  
 ( Main.hs, interpreted )  
Ok, one module loaded.  
*Main> main  
Hello  
World!  
Alex  
Alex  
*Main> |
```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

More Complicated

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

- Lambda function accepting 1 arg, name
- Received directly from the getLine above

```
WinGHCi
File Edit Actions Tools Help
*Main> :reload
Ok, one module loaded.
*Main> main
What is your name?
Alex
Hello, Alex!
*Main>
```

Up until now, we've only really seen how to evaluate expressions (and execute actions, though we didn't know that's what we were doing) in GHCi.

<https://powcoder.com>

Now we're seeing how to write, compile, and execute a complete Haskell program containing *actions*.

C:\HaskellCode\Main.hs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

Test.hs x Main.hs x

```
1 main =
2   putStrLn "What is your name?"
3   >> getLine
4   >>= \name -> putStrLn ("Hello, " ++ name ++ "!")
5
6
```

Haskell length: 110 lines: 6

Command Prompt

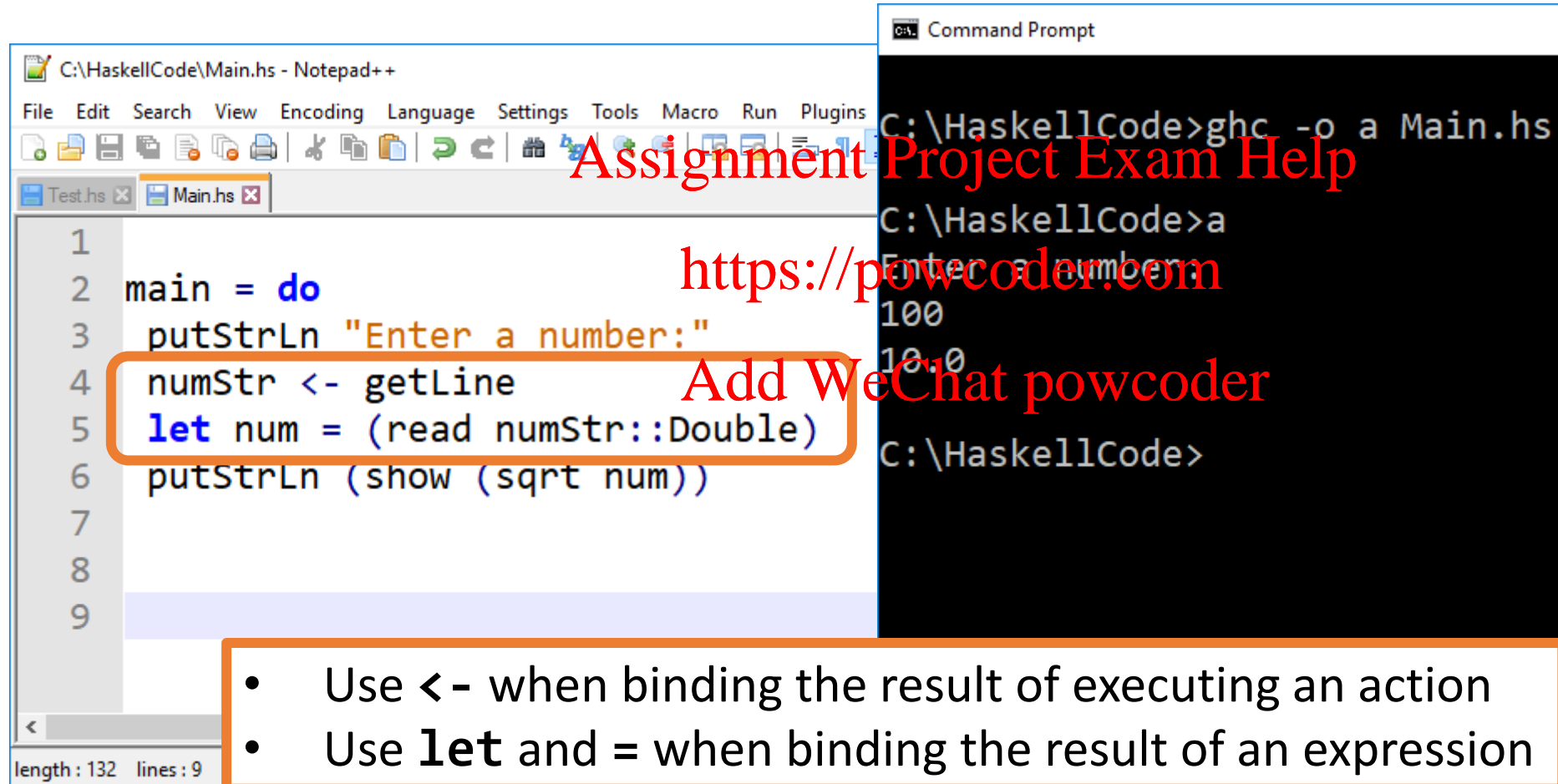
```
C:\HaskellCode>ghc -o a Main.hs
[1 of 1] Compiling Main                ( Main.hs, Main.o )
Linking a.exe ...

C:\HaskellCode>a
What is your name?
Alex
Hello, Alex!

C:\HaskellCode>
```

Actions & Functions

Assignment Project Exam Help
<https://powcoder.com>
Add WeChat powcoder



The image shows a Notepad++ window with a Haskell file named Main.hs. The code defines a main function that prompts the user for a number, reads it, and prints its square root. The code is as follows:

```
1
2 main = do
3   putStrLn "Enter a number:"
4   numStr <- getLine
5   let num = (read numStr :: Double)
6   putStrLn (show (sqrt num))
7
8
9
```

The Command Prompt window shows the execution of the program:

```
C:\HaskellCode>ghc -o a Main.hs
C:\HaskellCode>a
Enter a number:
100
10.0
C:\HaskellCode>
```

- Use `<-` when binding the result of executing an action
- Use `let` and `=` when binding the result of an expression

Problem?

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

- We are executing actions in **main**
- Its return type must be an action.
- The value of a “do” block is the value of the last expression evaluated

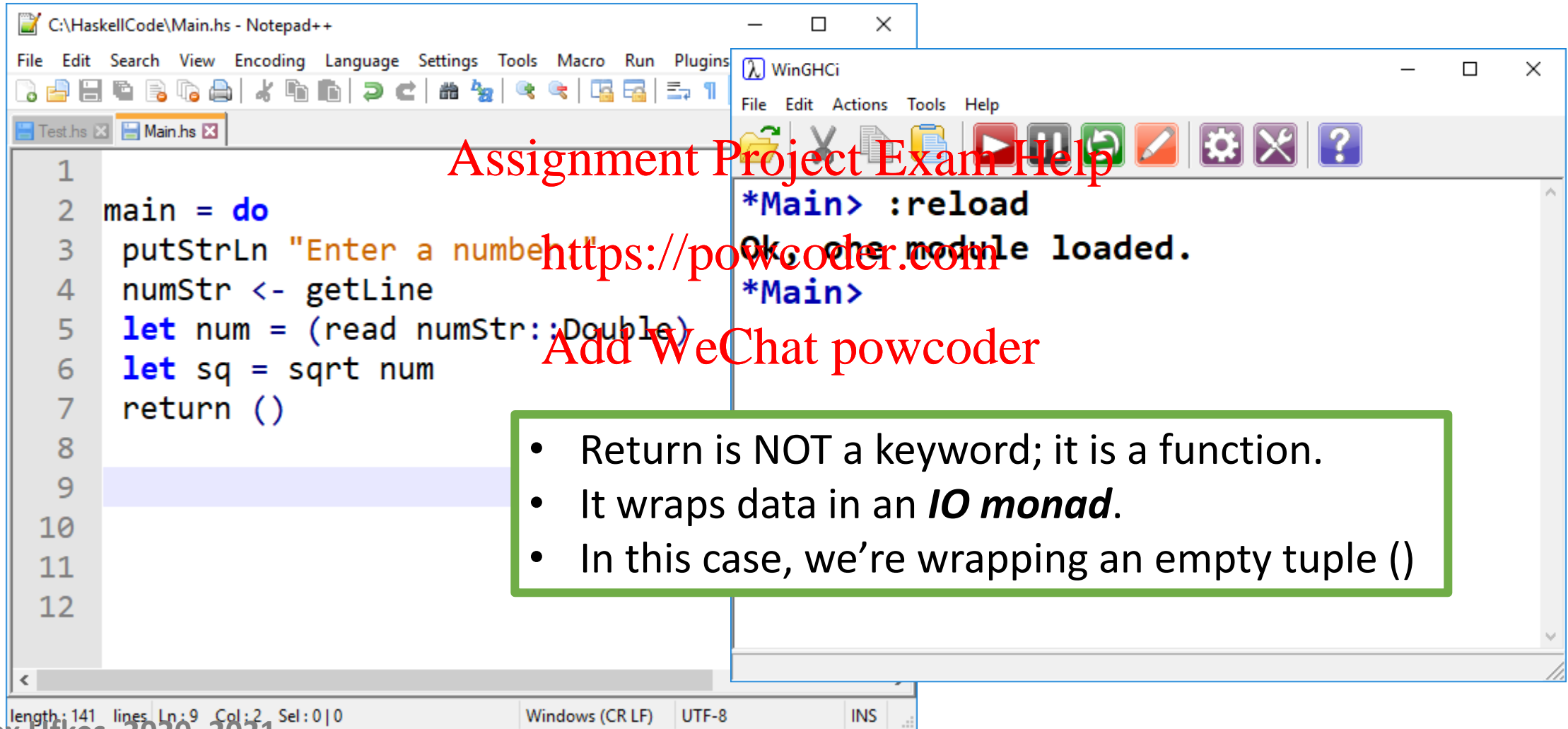
```
Prelude> :reload
```

```
main.hs:6:2: error:
```

- Couldn't match expected type 'IO b' with actual type 'Double'
- In a stmt of a 'do' block: num
- In the expression:

```
do {putStrLn "Enter a number:"
```

return ()



The screenshot shows two windows. The Notepad++ window displays a Haskell file named Main.hs with the following code:

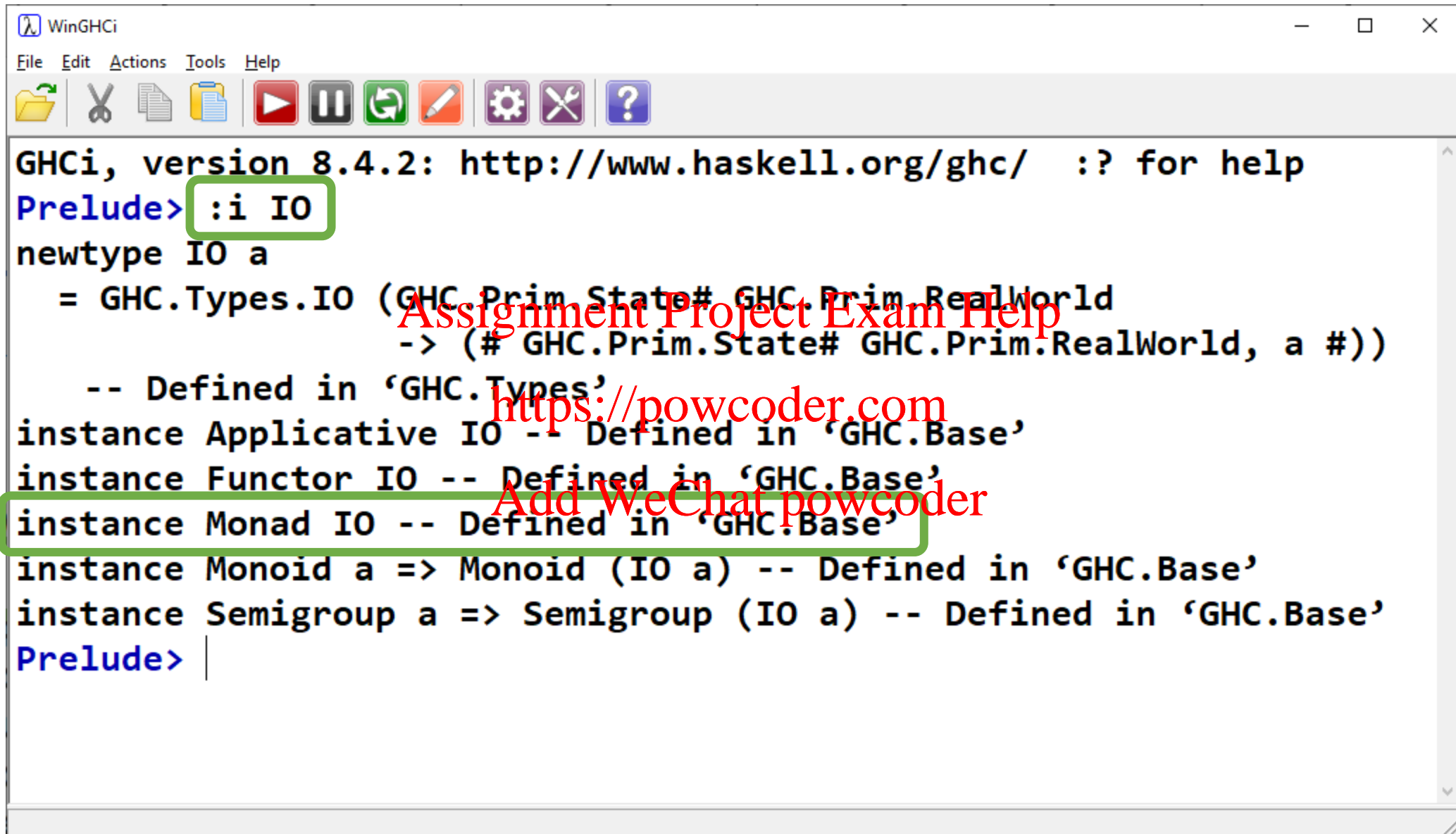
```
1  
2 main = do  
3   putStrLn "Enter a number"  
4   numStr <- getLine  
5   let num = (read numStr :: Double)  
6   let sq = sqrt num  
7   return ()  
8  
9  
10  
11  
12
```

The WinGHC window shows the terminal output:

```
*Main> :reload  
Ok, one module loaded.  
*Main>
```

Overlaid on the image are several red text annotations: "Assignment Project Exam Help", "https://powcoder.com", and "Add WeChat powcoder". A green-bordered box contains a list of bullet points explaining the return statement.

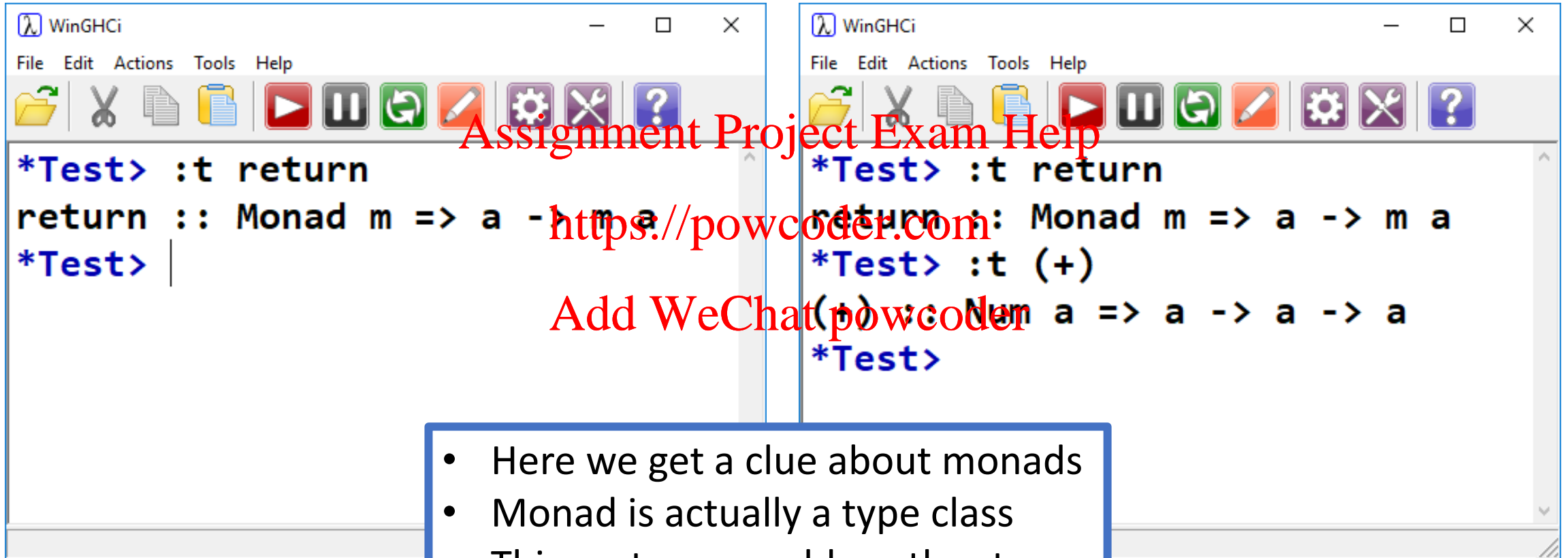
- Return is NOT a keyword; it is a function.
- It wraps data in an ***IO monad***.
- In this case, we're wrapping an empty tuple ()

A screenshot of a WinGHCi window. The title bar says 'WinGHCi'. The menu bar has 'File', 'Edit', 'Actions', 'Tools', and 'Help'. The toolbar contains icons for file operations (folder, copy, paste, save), execution (play, pause, refresh), editing (eraser), settings (gear, wrench), and help (question mark). The main text area shows the GHCi prompt and the output of the ':i IO' command. The output defines the IO monad as a newtype, lists its instances (Applicative, Functor, Monad, Monoid, Semigroup), and defines its type signature. A green box highlights the ':i IO' command. A green box highlights the 'instance Monad IO' line. A green box highlights the 'instance Monoid a' line. A green box highlights the 'instance Semigroup a' line. A red watermark 'Assignment Project Exam Help' is overlaid on the text. A red watermark 'https://powcoder.com' is overlaid on the text. A red watermark 'Add WeChat powcoder' is overlaid on the text.

```
WinGHCi
File Edit Actions Tools Help
[Icons]

GHCi, version 8.4.2: http://www.haskell.org/ghc/  :? for help
Prelude> :i IO
newtype IO a
  = GHC.Types.IO (GHC.Prim.State# GHC.Prim.RealWorld
                  -> (# GHC.Prim.State# GHC.Prim.RealWorld, a #))
    -- Defined in 'GHC.Types'
instance Applicative IO -- Defined in 'GHC.Base'
instance Functor IO -- Defined in 'GHC.Base'
instance Monad IO -- Defined in 'GHC.Base'
instance Monoid a => Monoid (IO a) -- Defined in 'GHC.Base'
instance Semigroup a => Semigroup (IO a) -- Defined in 'GHC.Base'
Prelude>
```

Monads



The image shows two side-by-side screenshots of the WinGHCi Haskell interpreter. The left window shows the prompt `*Test> :t return` followed by the type signature `return :: Monad m => a -> m a`. The right window shows the prompt `*Test> :t (+)` followed by the type signature `(+) :: Num a => a -> a -> a`. Both windows have a menu bar (File, Edit, Actions, Tools, Help) and a toolbar with icons for file operations and execution. A large red watermark is overlaid across the center of the image.

```
*Test> :t return
return :: Monad m => a -> m a
*Test>

*Test> :t (+)
(+) :: Num a => a -> a -> a
*Test>
```

- Here we get a clue about monads
- Monad is actually a type class
- This syntax resembles other type classes we've seen.

Monads

Monad is a typeclass:

```
WinGHCi
File Edit Actions Tools Help
[Icons]

Prelude> :i Monad
class Applicative m => Monad (m :: * -> *) where
  (>>=) :: m a -> (a -> m b) -> m b
  (>>)  :: m a -> m b -> m b
  return :: a -> m a
  fail   :: String -> m a
  {-# MINIMAL (>>=) #-}
  -- Defined in 'GHC.Base'
instance Monad (Either e) -- Defined in 'GHC.Base'
instance Monad [] -- Defined in 'GHC.Base'
instance Monad Maybe -- Defined in 'GHC.Base'
```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

We've seen these:

- `>>=` passes the result on the left into the function on the right.
- `>>` Ignores the result on the left
- `return` wraps data in a monad

Monad Jargon

“Monadic” Pertaining to monads. A monadic type is an instance of type class `Monad` (IO, for example)

“type xxx is a Monad” **Assignment Project Exam Help**
xxx is an instance of type class `Monad`. xxx implements `>>`, `>>=`, and `return`
<https://powcoder.com>

“action” Another name for a monadic value
Add WeChat powcoder

By the way:

- It turns out that Monads are good for things other than side effect-producing IO.
- We’ll see an example coming up.

>>= VS >>

Where the
magic happens

>>=

Chains actions together. Result of left side is given as input to the right side.

>>

Chains actions together. Ignore result of left side.

<https://powcoder.com>
Add WeChat powcoder

a >> b VS a >>= _ -> b

>> can be defined in terms of >>=

Non-main Example

```
C:\HaskellCode\Test.hs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ? X
Test.hs Main.hs
1 module Test where
2
3 positive = do
4   putStr "Enter a number: "
5   num <- getLine
6   (read num :: Double) < 0
7
```

- Function that reads in a number
- Returns true if > zero, false otherwise
- **Problem:** We're executing IO actions
- The return type cannot be Boolean
- It must be IO something

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```
WinGHCi
File Edit Actions Tools Help
length
1
1
1
1
Test.hs:7:3: error:
• Couldn't match expected type 'IO b' with actual type 'Bool'
• In a stmt of a 'do' block: (read num :: Double) < 0
In the expression:
do putStr "Enter a number: "
```


Non-main Example

What if we still want to get a Boolean back?

```
C:\HaskellCode\Test.hs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins
Test.hs Main.hs
1 module Test where
2
3 positive = do
4   putStr "Enter a number: "
5   num <- getLine
6   return ((read num::Double) < 0)
7
8
9
10
11
```

```
WinGHCi
File Edit Actions Tools Help
*Test> :t positive
positive :: IO Bool
*Test> x <- positive
Enter a number: 8
*Test> x
False
*Test> :t x
x :: Bool
*Test> |
```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

Extract the value from the action using <-

- The return type of positive is an IO action.
- When executed, that action produces a Bool

Calling Pure Code

We can still call pure functions from actions:

```
C:\HaskellCode\Main.hs - Notepad+
File Edit Search View Encoding
Test.hs Main.hs
1 findBigger x y = if x > y then x else y
2
3 main = do
4   putStrLn "Enter first number:"
5   nStr <- getLine
6   let num1 = (read nStr::Double)
7   putStrLn "Enter second number:"
8   nStr <- getLine
9   let num2 = (read nStr::Double)
10  let big = findBigger num1 num2
11  putStrLn ("Larger: " ++ (show big))
12
13
length: 304 lines Lm: 13 Col: 3 Sel: 0 | 0
Windows (CR LF) UTF-8 INS
```

```
Command Prompt
C:\HaskellCode>ghc -o a Main.hs
[1 of 1] Compiling Main
Linking a.exe ...

C:\HaskellCode>a
Enter first number:
4.5
Enter second number:
7.9
Larger: 7.9

C:\HaskellCode>
```

Best Practice

Separate pure code into its own functions:

The image shows two windows side-by-side. The left window is Notepad++ editing 'Test.hs'. It contains Haskell code for a module 'Test'. A green box highlights the 'testPos' function, which is annotated with a green 'Pure!' label. A red box highlights the 'positive' function, which is annotated with a red '!Pure' label. The right window is WinGHCi, showing the execution of the code. It displays the type signatures for 'positive' and 'testPos', and the result of running 'testPos' with the input '-8'.

```
1 module Test where
2
3 testPos numString = do
4   let x = read numString :: Double
5   if x < 0 then False else True
6
7 positive = do
8   putStr "Enter a number: "
9   num <- getLine
10  return (testPos num)
11
```

Pure!

!Pure

```
*Test> x <- positive
Enter a number: -8
*Test> x
False
*Test> :t positive
positive :: IO Bool
*Test> :t testPos
testPos :: String -> Bool
*Test> |
```

When looking at `main`, Haskell looks rather imperative...

Even at this point, however, Haskell sets itself apart from imperative languages.

Assignment Project Exam Help

It creates a separate type of programming construct for operations that produce side effects

<https://powcoder.com>

Add WeChat powcoder

We can always be sure of which parts of the code will alter the state of the world, and which parts won't.

Imperative languages do no such thing, and make no guarantees whatsoever regarding function purity

Assignment Project Exam Help

https://wiki.haskell.org/Introduction_to_Haskell_IO/Actions

<https://powcoder.com>

Add WeChat powcoder

<https://wiki.haskell.org/Monad>

“The essence of monad is thus separation of composition timeline from the composed computation's execution timeline, as well as the ability of computation to implicitly carry extra data”

Assignment Project Exam Help

<https://powcoder.com>

“This lends monads to supplementing pure calculations with features like I/O, common environment, updatable state, etc.”

Add WeChat powcoder

Not just for I/O! Not just for side effects!

Maybe Monad

Monads were originally introduced for IO operations

It turns out, as a construct, they are useful for modelling other things as well!

For example: exception handling, non-determinism, etc.

Maybe Monad

Represents a computation that might not produce a result

Assignment Project Exam Help

Computations that might “go wrong”

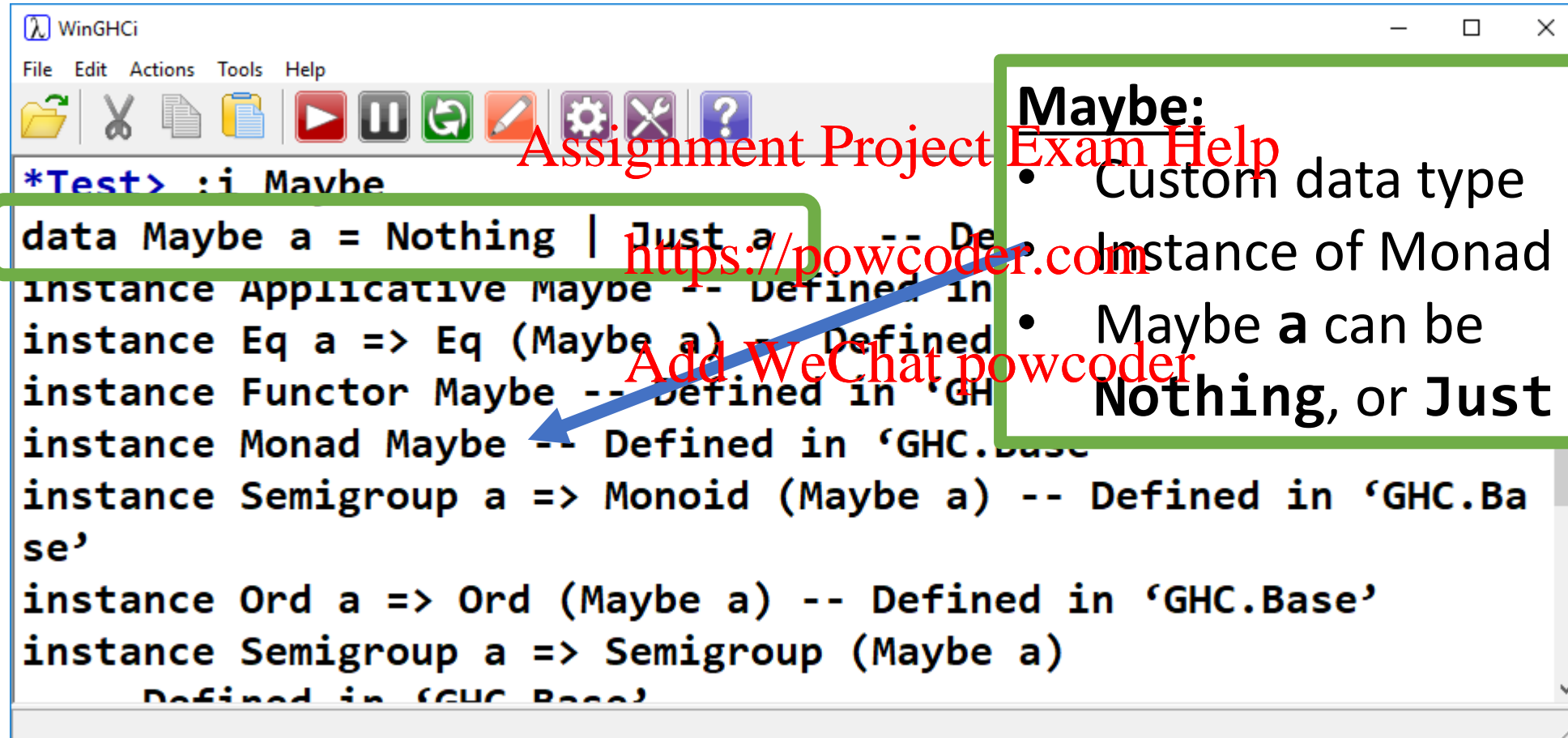
<https://powcoder.com>

For example – calling tail with a list that might be empty

Add WeChat powcoder

We can use Maybe to create a safety wrapper for functions that might fail, depending on input.

Maybe Monad



```
*Test> :i Maybe
data Maybe a = Nothing | Just a -- Defined in 'GHC.Base'
instance Applicative Maybe -- Defined in 'GHC.Base'
instance Eq a => Eq (Maybe a) -- Defined in 'GHC.Base'
instance Functor Maybe -- Defined in 'GHC.Base'
instance Monad Maybe -- Defined in 'GHC.Base'
instance Semigroup a => Monoid (Maybe a) -- Defined in 'GHC.Base'
instance Ord a => Ord (Maybe a) -- Defined in 'GHC.Base'
instance Semigroup a => Semigroup (Maybe a) -- Defined in 'GHC.Base'
```

Maybe:

- Custom data type
- Instance of Monad
- Maybe a can be **Nothing**, or **Just a**

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

We've seen this before...

Assignment Project Exam Help

https://powcoder.com

Add WeChat powcoder

Pt can take the value Pt3 Float
Float Float, or Pt2 Float Float

Maybe can take the value
Nothing or Just a

```
*Test> :i Maybe
```

```
data Maybe a = Nothing | Just a      -- Defined in 'GHC.Base'
```

```
instance Applicative Maybe -- Defined in 'GHC.Base'
```

```
instance Eq a => Eq (Maybe a) -- Defined in 'GHC.Base'
```

```
instance Functor Maybe -- Defined in 'GHC.Base'
```

```
instance Monad Maybe -- Defined in 'GHC.Base'
```

Maybe Monad

Assignment Project Exam Help

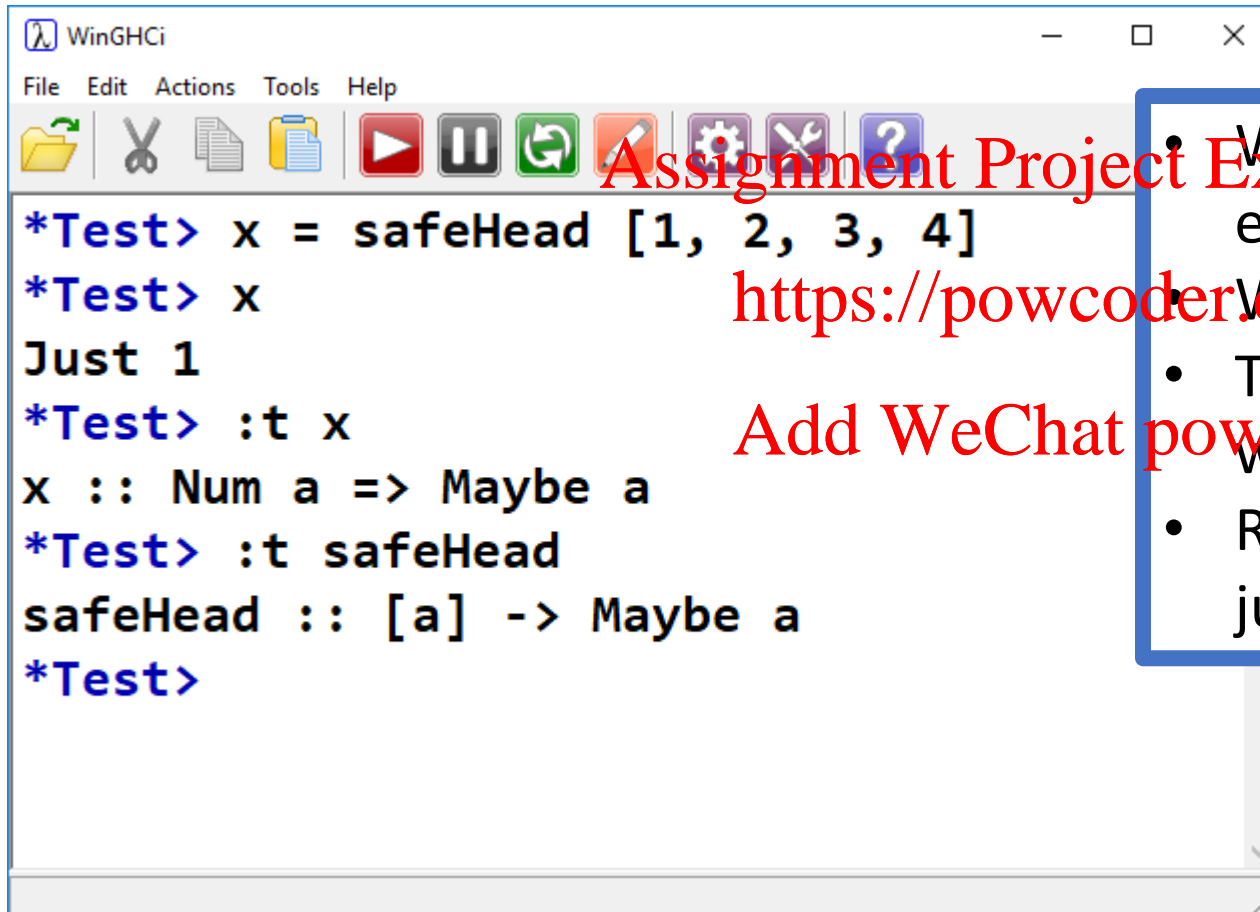
<https://powcoder.com>

Add WeChat powcoder

- Define safe functions for head and tail.
 - Using guards - |
- Instead of failing on empty lists, evaluate to Nothing.
- If a tail or head can be found, evaluate to Just head x, or Just tail x
- Just head? Just tail?

```
C:\HaskellCode\Test.hs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ? X
Main.hs Test.hs
1 module Test where
2
3 safeTail x
4   | (length x > 0) = Just (tail x)
5   | otherwise = Nothing
6
7 safeHead x
8   | (length x > 0) = Just (head x)
9   | otherwise = Nothing
10
11
12
13
length: 2,585 |Ln: 11 |Col: 2 |Sel: 0 |0 Windows (CR LF) UTF-8 INS
```

Maybe Monad



```
WinGHCi
File Edit Actions Tools Help
[Icons]
*Test> x = safeHead [1, 2, 3, 4]
*Test> x
Just 1
*Test> :t x
x :: Num a => Maybe a
*Test> :t safeHead
safeHead :: [a] -> Maybe a
*Test>
```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

- When we call `safeHead` on a non-empty list, we don't get the head. We get *Just head*
- This is the head of the list wrapped in a `Maybe` monad.
- Remember that `Maybe` is a type, just like our custom `Pt` type

Maybe Monad

Assignment Project Exam Help

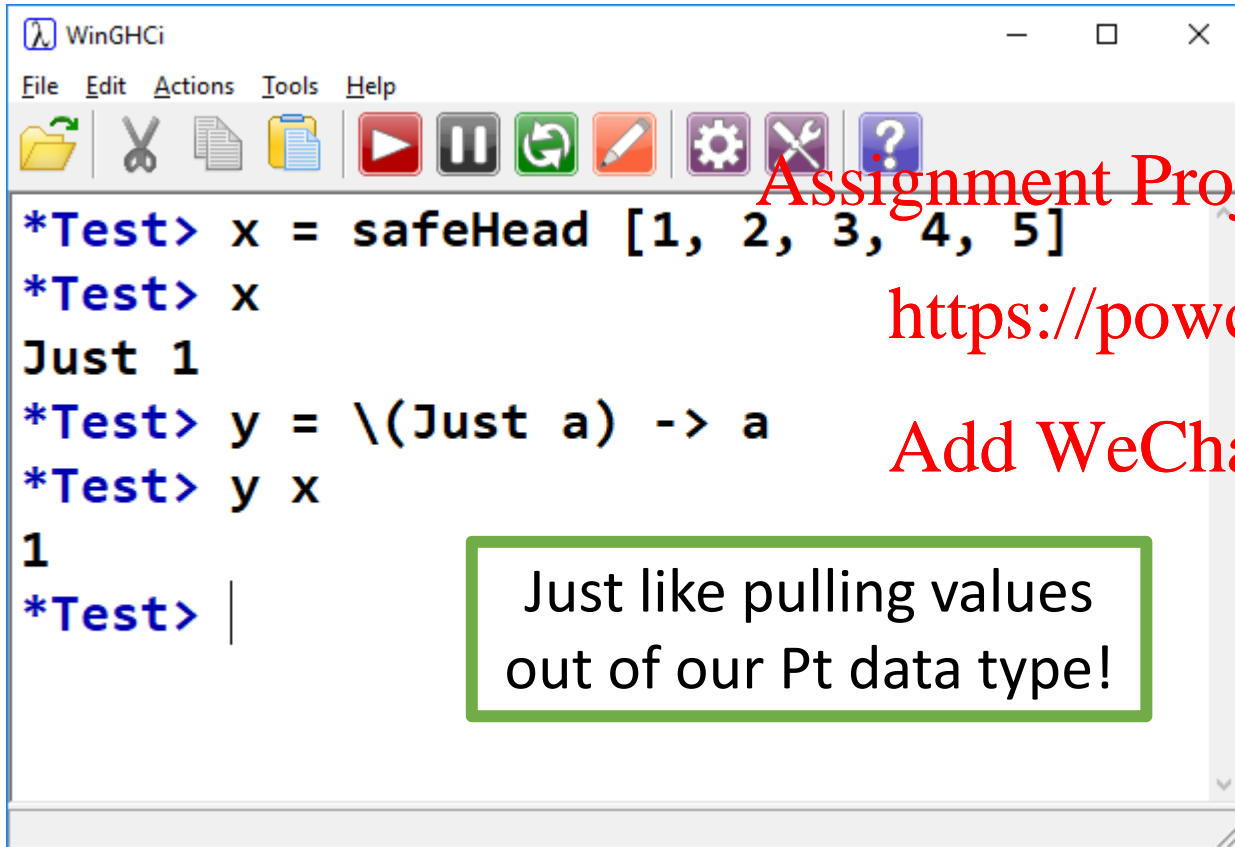
<https://powcoder.com>

Add WhatsApp powcoder

```
C:\HaskellCode\Test.hs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Win
Main.hs Test.hs
1 module Test where
2
3 safeTail x
4   | (length x > 0) = Just (tail x)
5   | otherwise = Nothing
6
7 safeHead x
8   | (length x > 0) = Just (head x)
9   | otherwise = Nothing
10
11
12
13
length: 2,585 |Ln: 11 Col: 2 Sel: 0|0 Windows (CR LF) UTF-8 INS

WinGHCi
File Edit Actions Tools Help
*Test> safeTail []
Nothing
*Test> safeHead []
Nothing
*Test> tail []
*** Exception: Prelude.tail: empty list
*Test> head []
*** Exception: Prelude.head: empty list
*Test> |
```

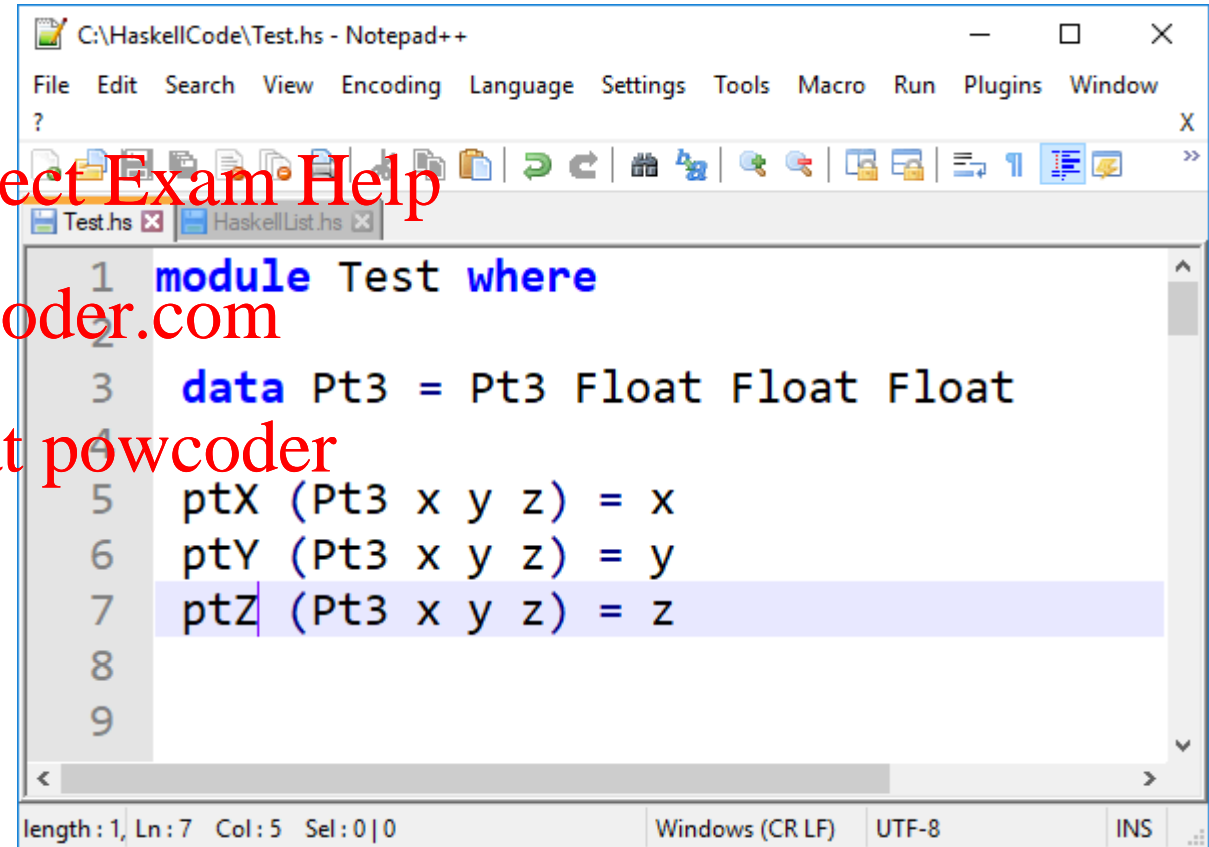
Unwrap Just a?



WinGHCi

```
*Test> x = safeHead [1, 2, 3, 4, 5]
*Test> x
Just 1
*Test> y = \(Just a) -> a
*Test> y x
1
*Test> |
```

Just like pulling values out of our Pt data type!



C:\HaskellCode\Test.hs - Notepad++

```
1 module Test where
2
3 data Pt3 = Pt3 Float Float Float
4
5 ptX (Pt3 x y z) = x
6 ptY (Pt3 x y z) = y
7 ptZ (Pt3 x y z) = z
8
9
```

length: 1, Ln: 7 Col: 5 Sel: 0 | 0 Windows (CR LF) UTF-8 INS

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

*C:_cps506\haskell\Test.hs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Win

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Win

Test.hs x cond1.c x

```
1 module Test where
2
3 safeTail x
4   | (length x > 0) = Just (tail x)
5   | otherwise = Nothing
6
7 safeHead x
8   | (length x > 0) = Just (head x)
9   | otherwise = Nothing
10
11 getMaybeVal (Just a) = a
12
13
14
15
```

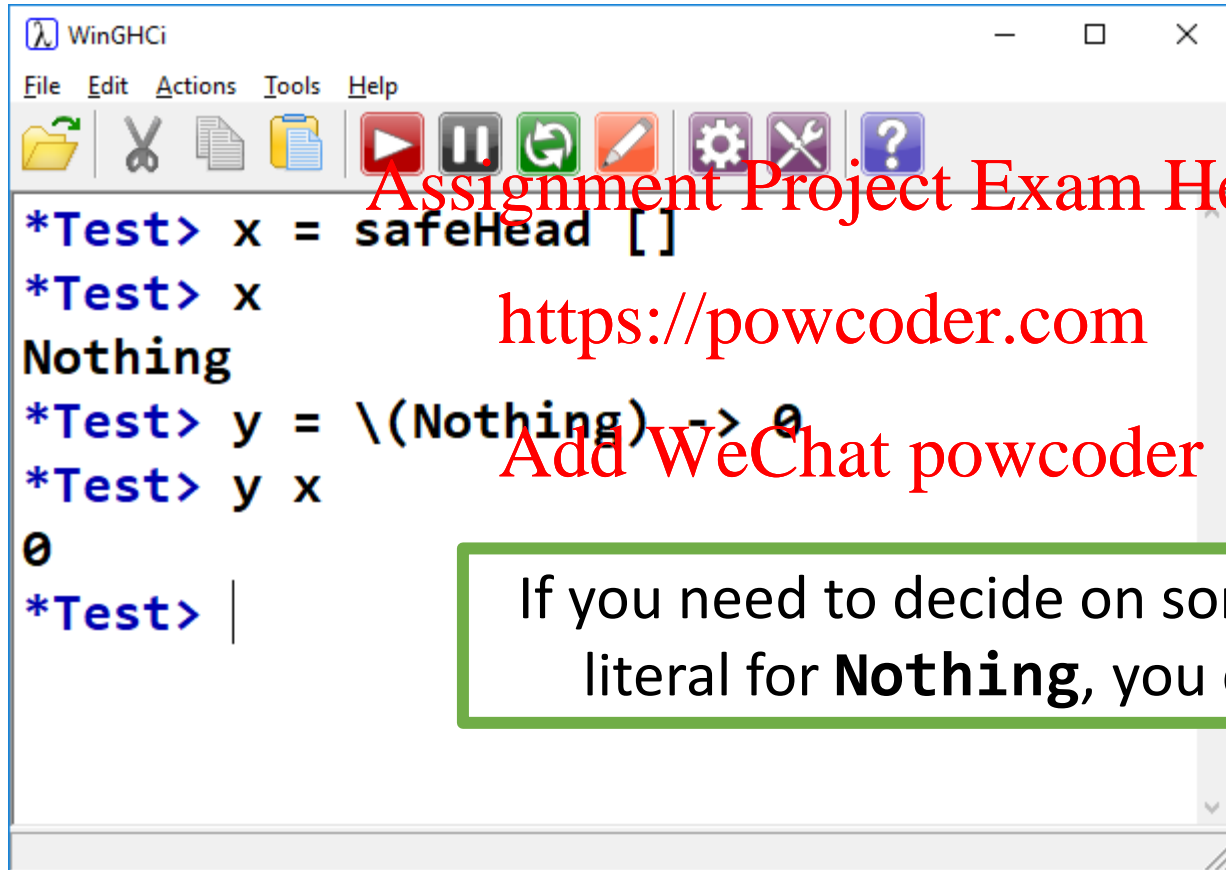
WinGHCi

File Edit Actions Tools Help

File Edit Actions Tools Help

```
*Test> x = safeHead [8, 6, 4]
*Test> y = safeTail [8, 6, 4]
*Test> getMaybeVal x
8
*Test> getMaybeVal y
[6, 4]
*Test> :t getMaybeVal
getMaybeVal :: Maybe a -> a
*Test>
```

Unwrap *Nothing*?



A screenshot of a WinGHCi terminal window. The window has a title bar 'WinGHCi' and a menu bar with 'File', 'Edit', 'Actions', 'Tools', and 'Help'. Below the menu bar is a toolbar with icons for file operations and execution. The terminal shows the following interaction:

```
*Test> x = safeHead []
*Test> x
Nothing
*Test> y = \(Nothing) -> 0
*Test> y x
0
*Test> |
```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

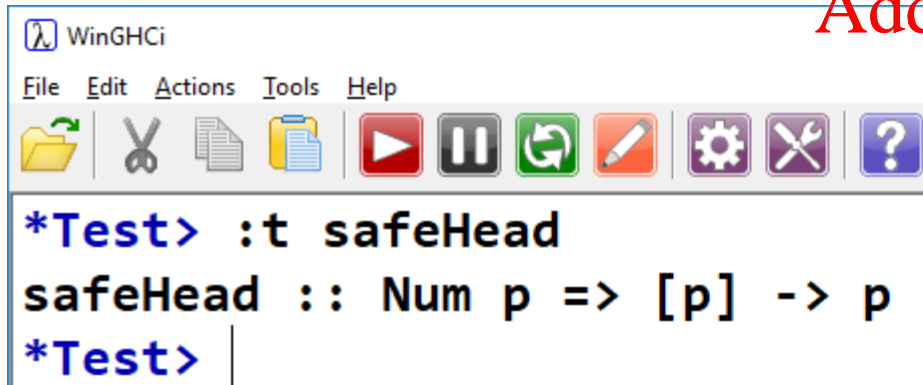
If you need to decide on some numeric literal for **Nothing**, you can do so

Why Not This?

```
safeHead x
| (length x > 0) = head x
| otherwise = 0
```

Assignment Project Exam Help
<https://powcoder.com>

Add WeChat powcoder



```
WinGHCi
File Edit Actions Tools Help
[Icons: Folder, Scissors, Document, Copy, Run, Pause, Refresh, Erase, Settings, Wrench, Help]

*Test> :t safeHead
safeHead :: Num p => [p] -> p
*Test> |
```

Zero as error code

- What if head of list is *actually* 0?
- Static typing means list passed to safeHead can only be instance of Num!
- **Just** can contain anything
- **Nothing** is useful as an “error” value

Using Maybe

Maybe can make code safer by gracefully dealing with failure.

Assignment Project Exam Help

Should we use Maybe for everything?

Add WeChat powcoder

No. Not everything has a chance to fail. Wrapping the return type of $(x > y)$ in Maybe only serves to obfuscate your code.

Consider a Lookup Table

We have a list of tuple pairs:

```
book = [ ("Alex", 555),  
         ("John", 444),  
         ("Tim", 333),  
         ("Mark", 222),  
         ("Bill", 111) ]
```

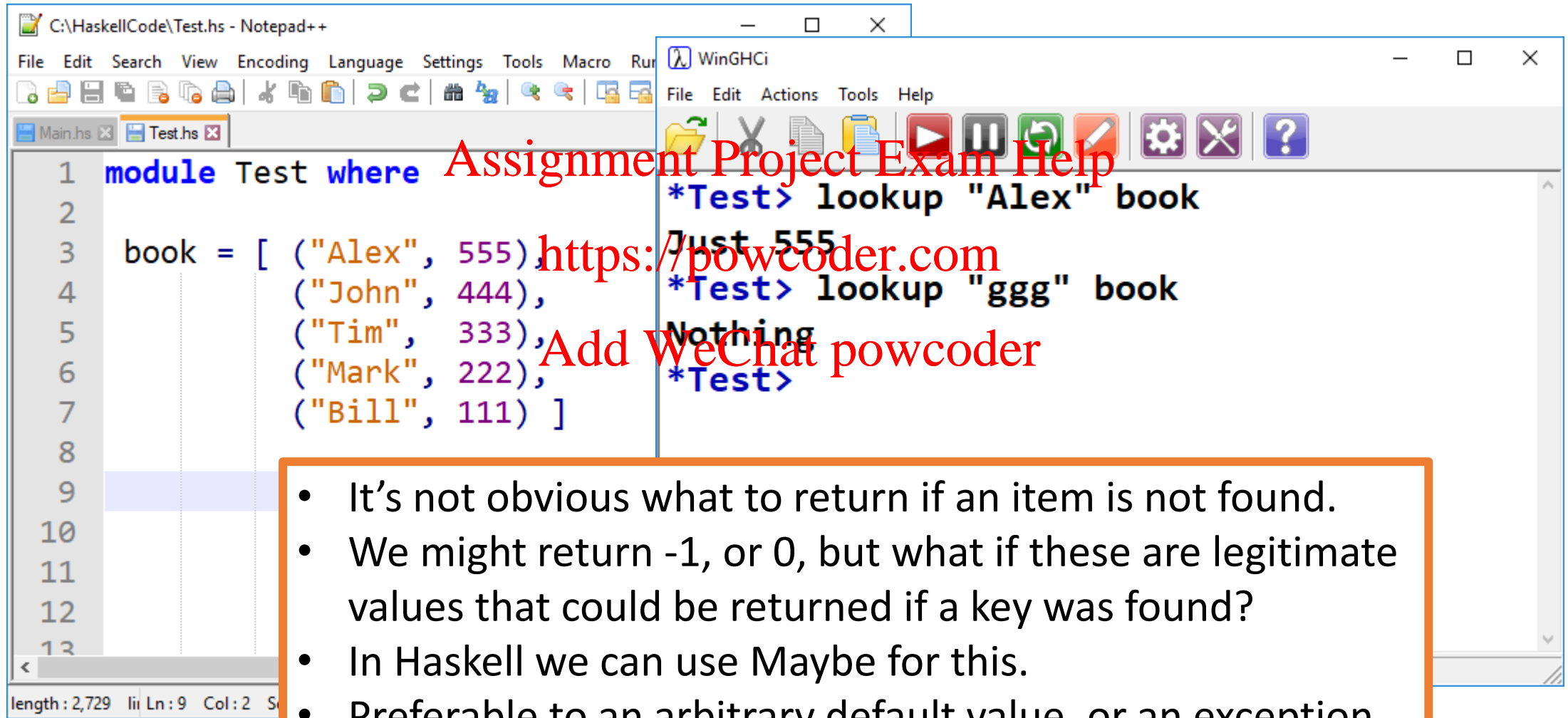
Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

- We want to search the table for a name
- If found, return its number
- If not found, return.... ?

Use lookup



The screenshot shows a Haskell code editor (Notepad++) and a WinGHCi terminal window. The code defines a list of names and ages, and the terminal shows the results of using the `lookup` function to find values for "Alex" and "ggg".

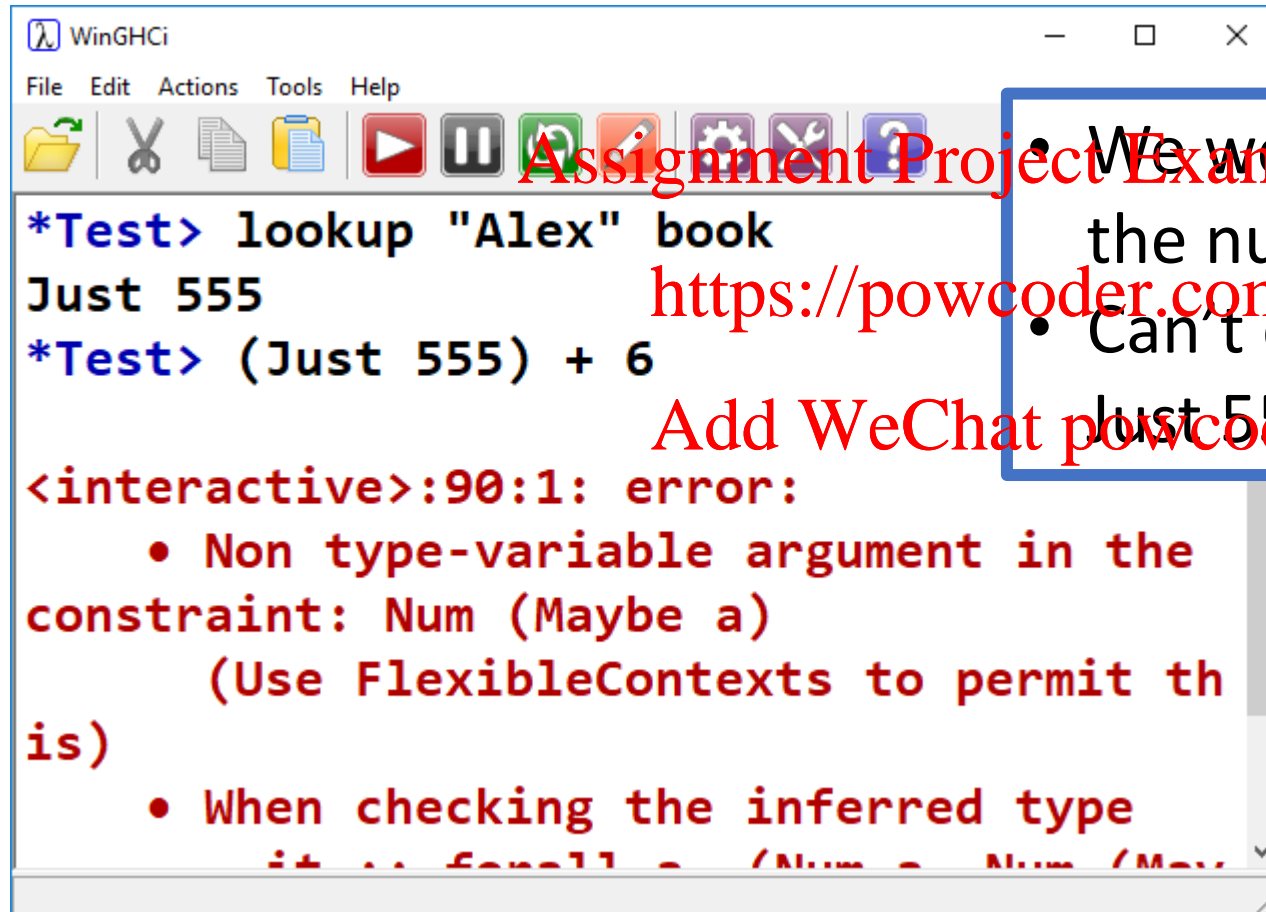
```
1 module Test where
2
3 book = [ ("Alex", 555),
4          ("John", 444),
5          ("Tim", 333),
6          ("Mark", 222),
7          ("Bill", 111) ]
8
9
10
11
12
13
```

```
*Test> lookup "Alex" book
Just 555
*Test> lookup "ggg" book
Nothing
*Test>
```

Assignment Project Exam Help
<https://powcoder.com>
Add WeChat powcoder

- It's not obvious what to return if an item is not found.
- We might return -1, or 0, but what if these are legitimate values that could be returned if a key was found?
- In Haskell we can use Maybe for this.
- Preferable to an arbitrary default value, or an exception.

Just 555 VS 555



The screenshot shows the WinGHCi window with the following content:

```
WinGHCi
File Edit Actions Tools Help
[Icons]
*Test> lookup "Alex" book
Just 555
*Test> (Just 555) + 6
<interactive>:90:1: error:
  • Non type-variable argument in the
  constraint: Num (Maybe a)
    (Use FlexibleContexts to permit this)
  • When checking the inferred type
    it is forall a. (Num a, Num (Maybe a)) => Maybe a
```

- We would like to extract the numeric value 555
- Can't do arithmetic on Just 555 for example.

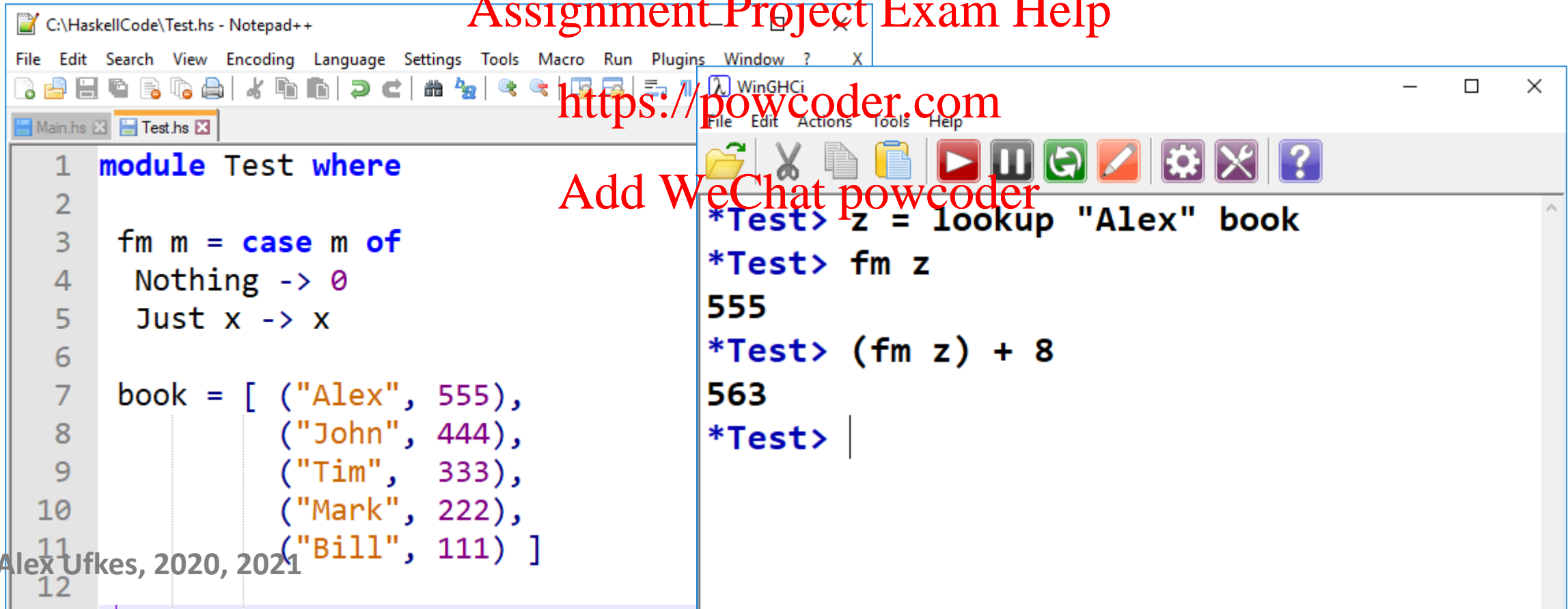
Just 555 VS 555

If we have a **Just** value, we can see its contents and extract through pattern matching

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder



The image shows a screenshot of a Haskell development environment. On the left, a Notepad++ window titled 'C:\HaskellCode\Test.hs - Notepad++' displays the following code:

```
1 module Test where
2
3 fm m = case m of
4   Nothing -> 0
5   Just x -> x
6
7 book = [ ("Alex", 555),
8          ("John", 444),
9          ("Tim", 333),
10         ("Mark", 222),
11         ("Bill", 111) ]
```

On the right, the WinGHCi REPL window shows the following interactions:

```
*Test> z = lookup "Alex" book
*Test> fm z
555
*Test> (fm z) + 8
563
*Test> |
```

Use lookup

```
C:\HaskellCode\Test.hs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Windows
Test.hs Main.hs
10 Nothing -> 0
11 Just x -> x
12
13 book1 = [ ("Alex", 555), ("John", 444),
14           ("Tim", 333), ("Mary", 222) ]
15
16 book2 = [ (555, 1), (444, 2),
17           (333, 3), (111, 4) ]
18
19 book3 = [ (1, "First"), (2, "Second"),
20           (5, "Third"), (4, "Fourth") ]
21
22
23
```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

- Value from book1 is the key to book2
- Value of book2 is the key to book3
- We want the value from book3

- Not every value in book1 corresponds to a key in book2.
- Not every value in book2 corresponds to a key in book3
- There are several ways a lookup could fail

```
C:\HaskellCode\Test.hs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
Test.hs Main.hs
1 module Test where
2
3 getPlace :: String -> Maybe String
4 getPlace name = do
5   code <- lookup name book1
6   num <- lookup code book2
7   lookup num book3
8
9 fm m = case m of
10   Nothing -> ""
11   Just x -> x
12
13 book1 = [ ("Alex", 555), ("John", 444),
14           ("Tim", 333), ("Mark", 222) ]
15
16 book2 = [ (555, 1), (444, 2),
17           (333, 3), (111, 4) ]
18
19 book3 = [ (1, "First"), (2, "Second"),
20           (5, "Third"), (4, "Fourth") ]
21
© Alex Ufkes, 2020, 2021
length: 3,024 lines: 1 Ln: 23 Col: 2 Sel: 0 | 0 Windows (CR LF) UTF-8 INS
```

- What happens if lookup fails to find a match?
- We saw that it returns **Nothing**
- What happens if we try to lookup **Nothing**?

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```
WinGHC
File Edit Actions Tools Help
*Test> :t getPlace
getPlace :: String -> Maybe String
*Test> getPlace "Alex"
Just "First"
*Test> getPlace "Tim"
Nothing
*Test> getPlace "Mark"
Nothing
*Test> fm (getPlace "Alex")
"First"
*Test>
```


Cascading Failure

```
C:\HaskellCode\Test.hs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ? X
Test.hs Main.hs
1 module Test where
2
3 getPlace :: String -> Maybe String
4 getPlace name = do
5   code <- lookup name book1
6   num <- lookup code book2
7   lookup num book3
8
9 fm m = case m of
10   Nothing -> ""
11   Just x -> x
12
13 book1 = [ ("Alex", 555), ("John", 444),
14           ("Tim", 333), ("Mark", 222) ]
15
16 © Alex Ufkes, 2020, 2021
```

Assignment Project Exam Help

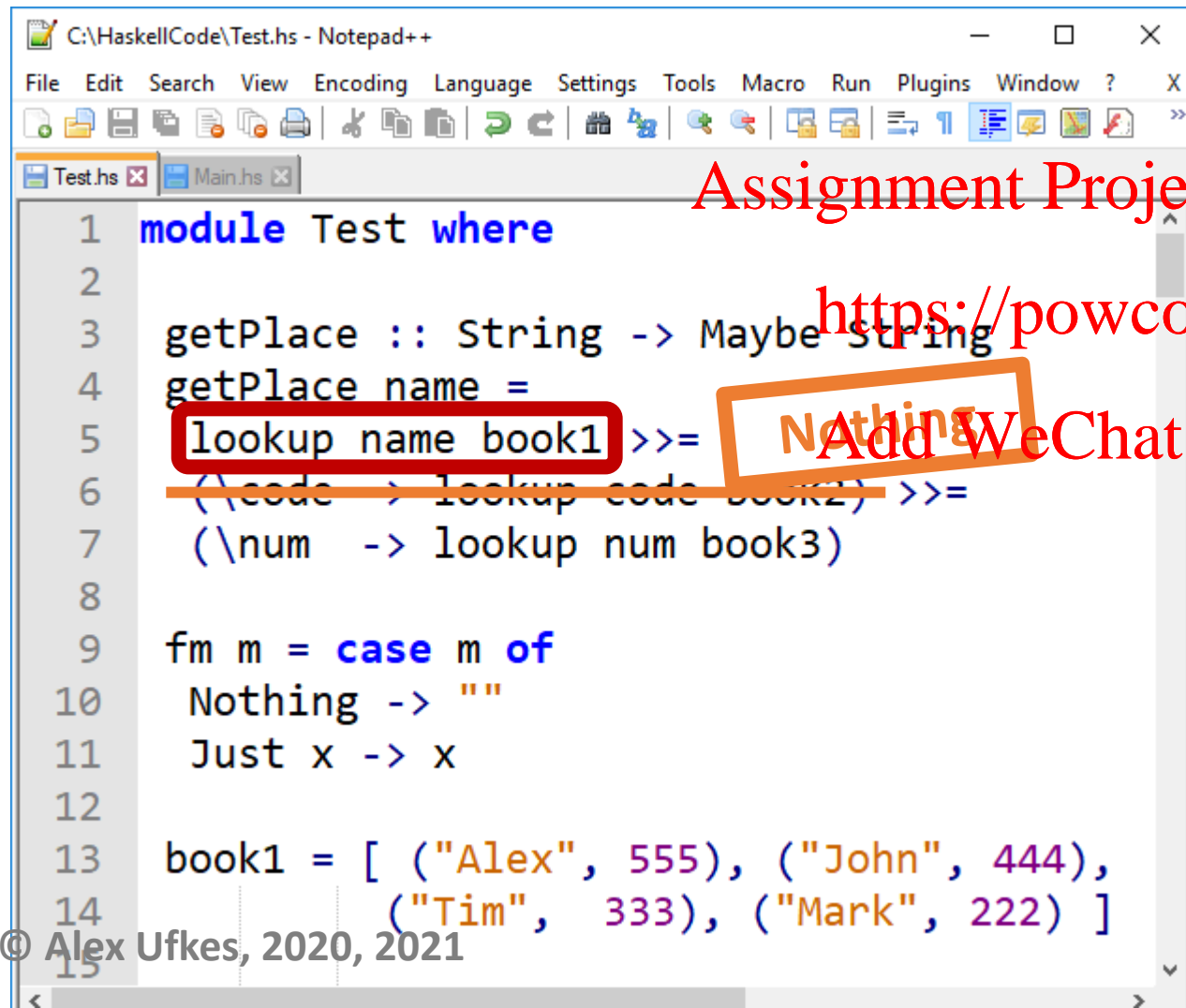
<https://powcoder.com>

Add WeChat powcoder

Is the
same as:

```
C:\HaskellCode\Test.hs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ? X
Test.hs Main.hs
1 module Test where
2
3 getPlace :: String -> Maybe String
4 getPlace name =
5   lookup name book1 >>=
6     (\code -> lookup code book2) >>=
7       (\num -> lookup num book3)
8
9 fm m = case m of
10   Nothing -> ""
11   Just x -> x
12
13 book1 = [ ("Alex", 555), ("John", 444),
14           ("Tim", 333), ("Mark", 222) ]
15
16 65
```

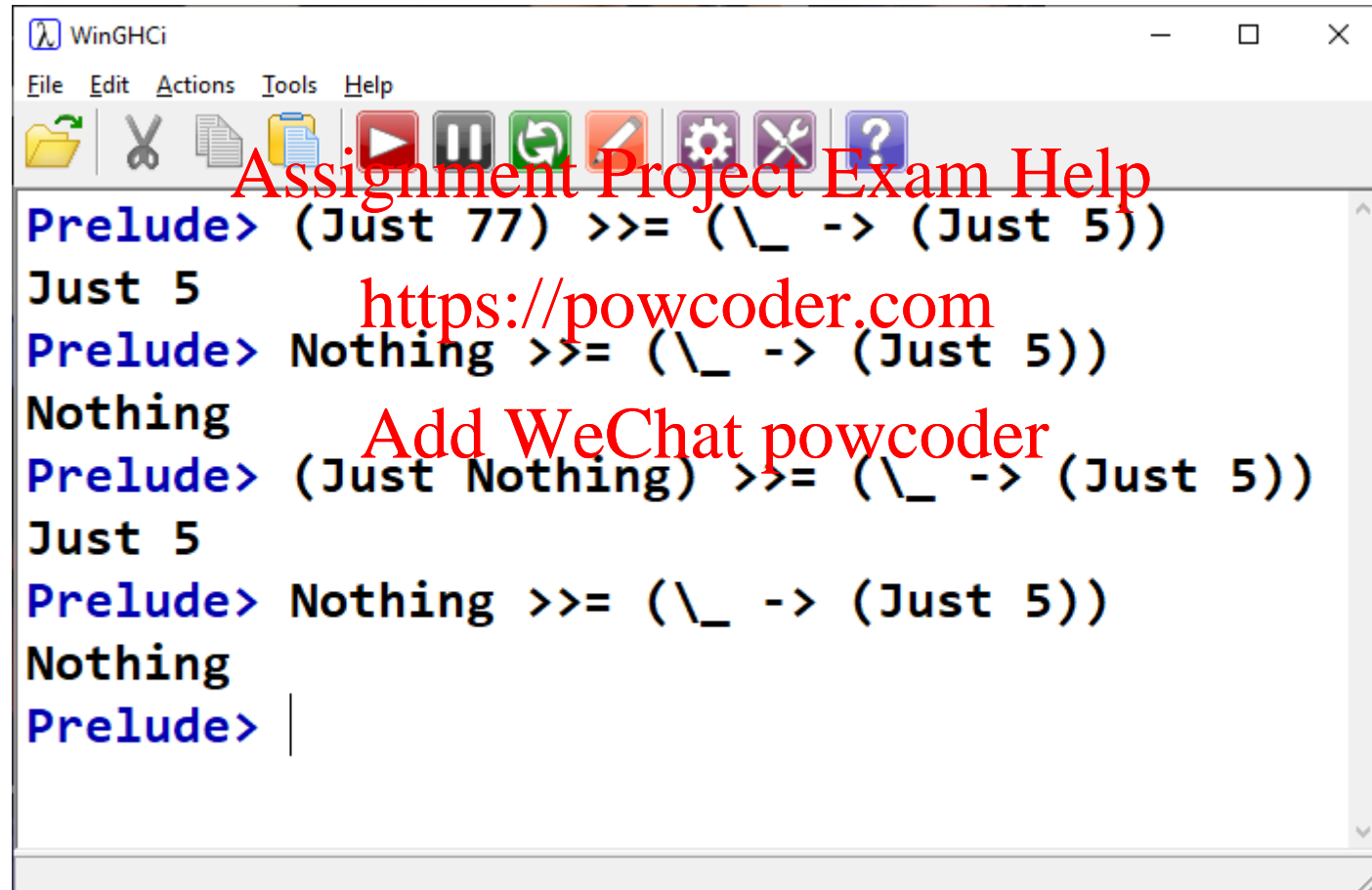
Cascading Failure



```
1 module Test where
2
3 getPlace :: String -> Maybe String
4 getPlace name =
5     lookup name book1 >>=
6     (\code -> lookup code book2) >>=
7     (\num -> lookup num book3)
8
9 fm m = case m of
10     Nothing -> ""
11     Just x -> x
12
13 book1 = [ ("Alex", 555), ("John", 444),
14           ("Tim", 333), ("Mark", 222) ]
15
```

- When the first argument to ($\gg=$) is **Nothing**, it just returns **Nothing** while ignoring the given function
- This causes failure to cascade
 - If the first lookup fails, **Nothing** is passed into the second $\gg=$.
- The failure then cascades into the third $\gg=$, and is returned.
- After the first **Nothing**, subsequent $\gg=$ pass that **Nothing** to each other

*When the first argument to ($>>=$) is **Nothing**, it just returns **Nothing** while ignoring the given function*



The screenshot shows the WinGHCi window with a menu bar (File, Edit, Actions, Tools, Help) and a toolbar. The main text area contains the following Haskell code and its output:

```
Prelude> (Just 77) >>= (\_ -> (Just 5))
Just 5
Prelude> Nothing >>= (\_ -> (Just 5))
Nothing
Prelude> (Just Nothing) >>= (\_ -> (Just 5))
Just 5
Prelude> Nothing >>= (\_ -> (Just 5))
Nothing
Prelude> |
```

Red text overlays are present on the image:

- Assignment Project Exam Help
- <https://powcoder.com>
- Add WeChat powcoder

Haskell Tutorials/References:

https://en.wikibooks.org/wiki/Yet_Another_Haskell_Tutorial

Assignment Project Exam Help

<https://powcoder.com>

<http://cheatsheet.codeslower.com/CheatSheet.pdf>

Add WeChat powcoder

Moving on...

Assignment Project Exam Help
...to imperative.
<https://powcoder.com>

Add WeChat powcoder

Rust is an imperative language. However, we'll see many cool features that remind us of the functional languages we've seen.



Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder