C/CPS 506

Comparative Programming Languages

Prof. Alex Ufkes

Topic 8: Actions in Haskell

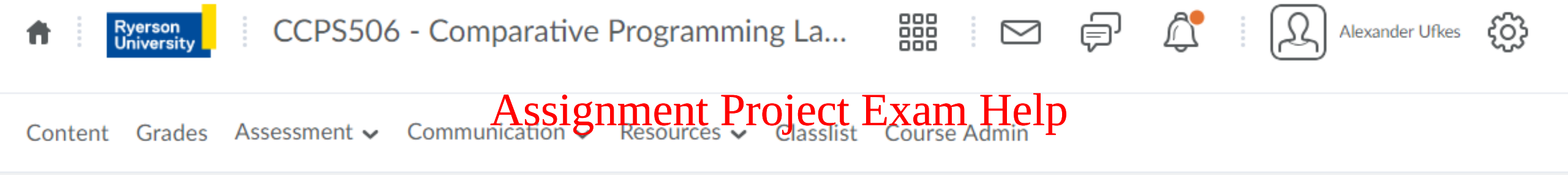
Ryerson University

Notice!

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

The copyright to this original work is held by Alex Ufkes. Students registered in course CPS 506 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



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

Let’s Get Started!

Last Week

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

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

Test.hs


```haskell
module Test where

 data Pt = Pt3 Float Float Float
         | Pt2 Float Float
        deriving (Show)


instance Eq Pt where
  (Pt2 x1 y1) == (Pt2 x2 y2) = (x1==x2 && y1==y2)
```


Haskell length : 1,623 lines : 99 Ln : 14 Col : 2 Sel : 0 | 0 Windows (CR LF) UTF-8 INS

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

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

Last Week

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 Floa
4           | Pt2 Float Floa
5           deriving (Show)
6
7   instance Num Pt where
8    (Pt2 x1 y1) + (Pt2 x2 y
9
10
11
12
```

Hasl length : 1,765 lines : 104 Ln : 11 Col : 3 Sel : 0 | 0

WinGHCi

File Edit Actions Tools Help

```
[2 of 2] Compiling Test             ( Test.hs, in
terpreted )

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

WinGHCi

File Edit Actions Tools Help

```
*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 Pt -- Defined at Test.hs:7:11
```

Last Week

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

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

Test.hs | HaskellList.hs

```haskell
module Test where

 data Pt = Pt3 Float Float Float
         | Pt2 Float Float
         --deriving (Show)

 instance Show Pt where
  show (Pt2 x y) =
   "< " ++ (show x) ++ ", " ++ (show y) ++ " >"
```


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

- 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

Every function is pure

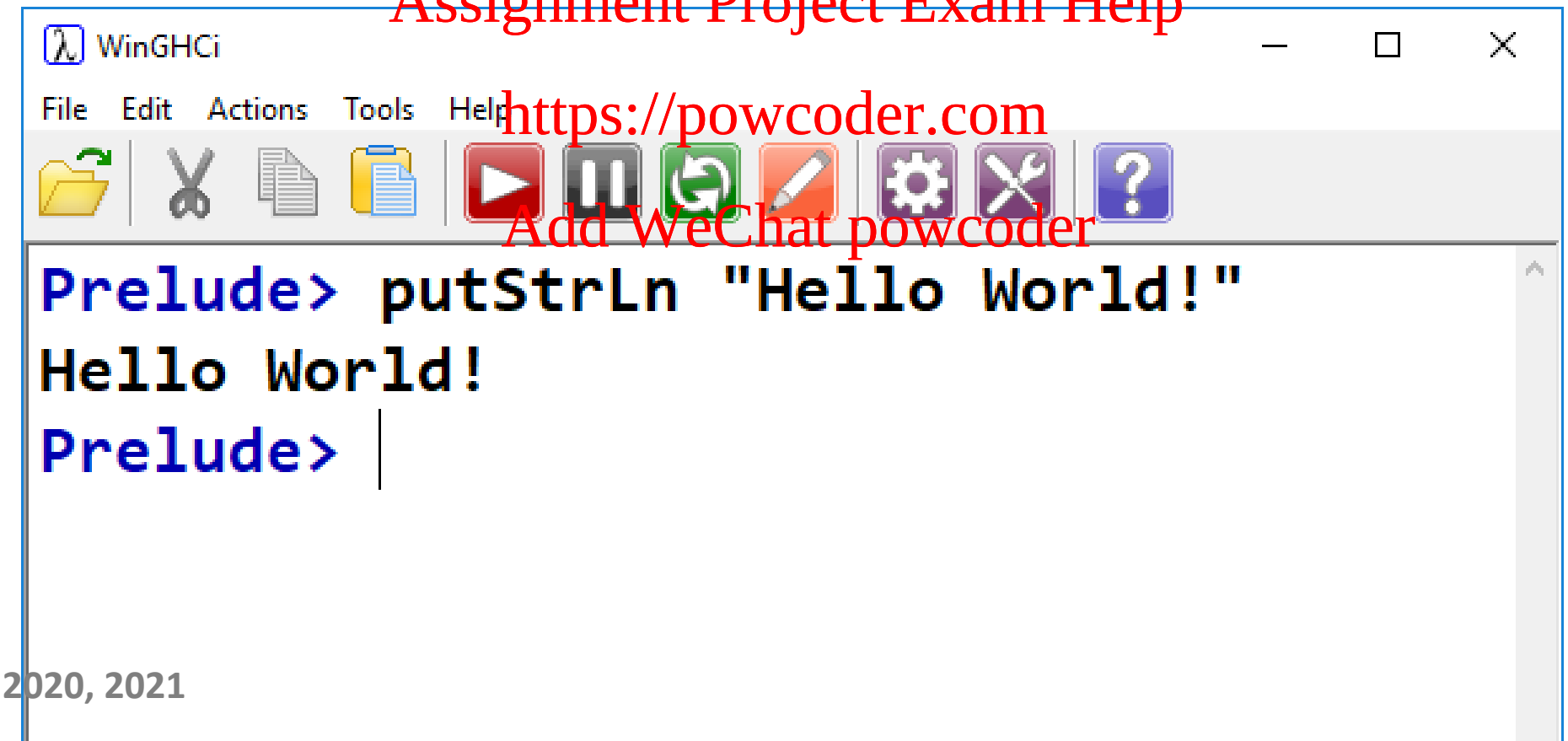
<u>Pure Functions:</u> <u>Functions that have no side effects.</u>

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.

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

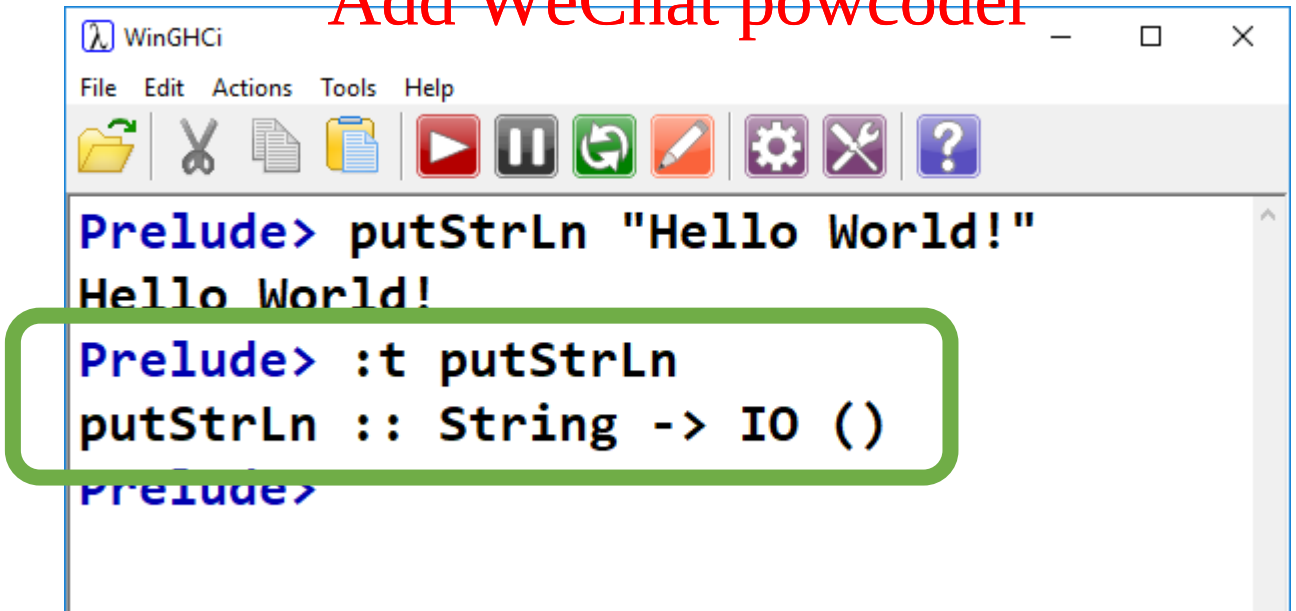
Write to Display

This was the very first thing we saw!

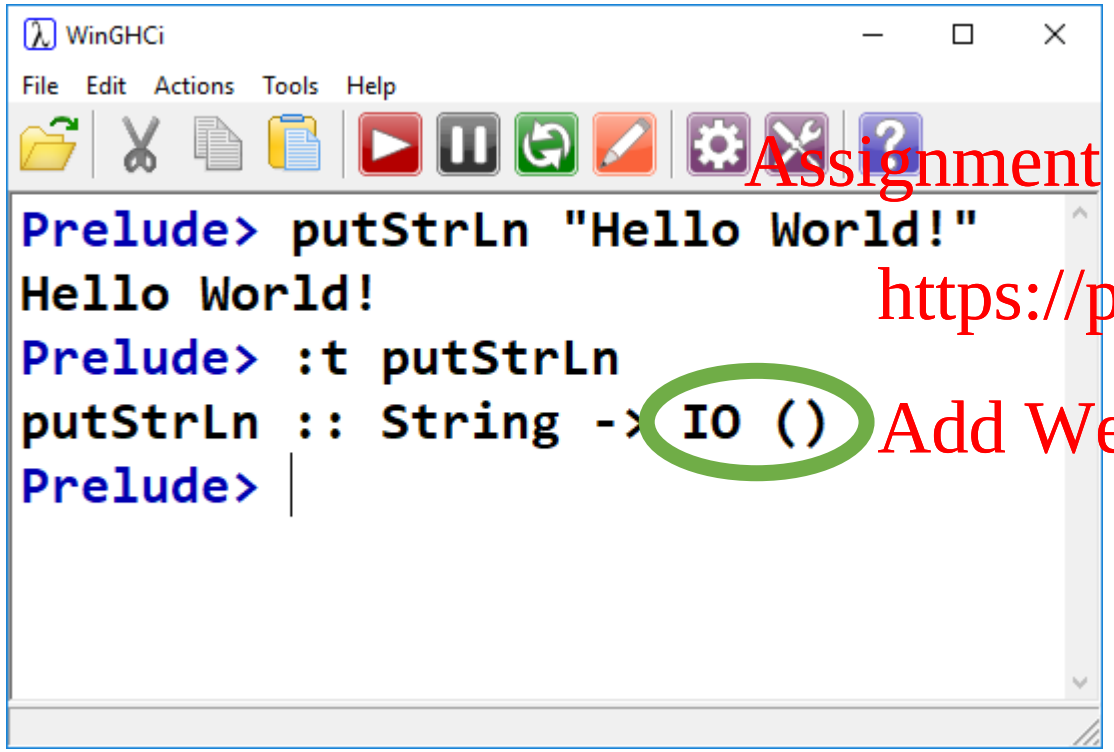


Haskell and I/O

- 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:

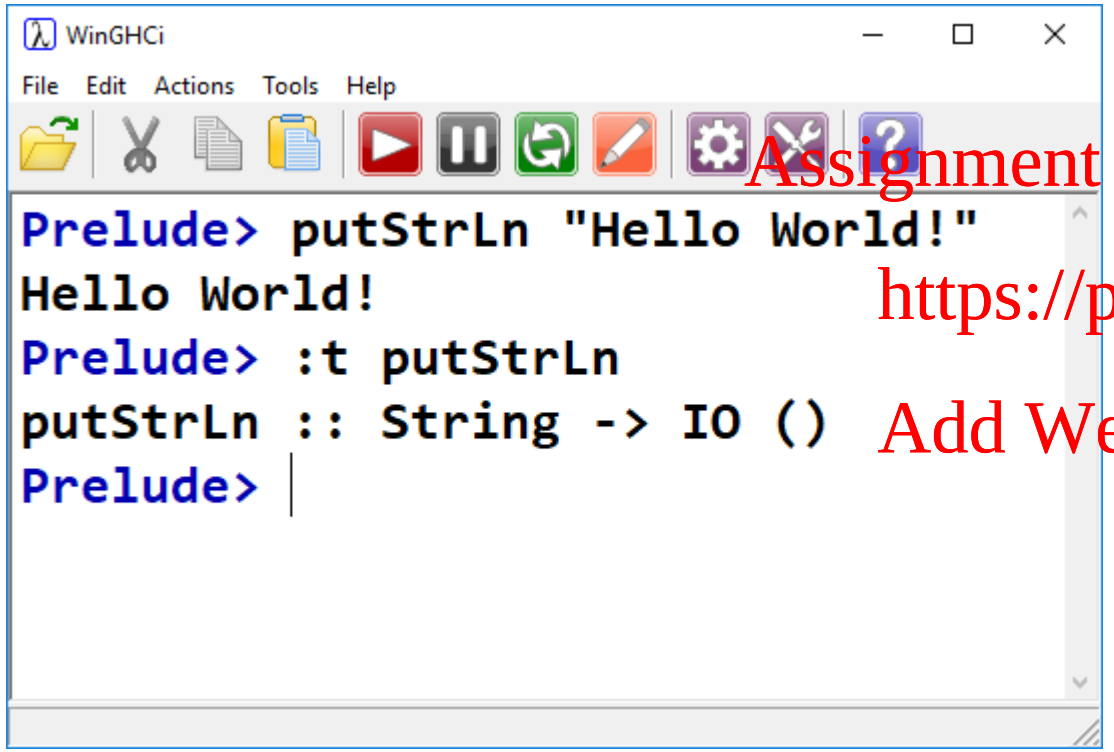


Haskell and I/O



- 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()`**

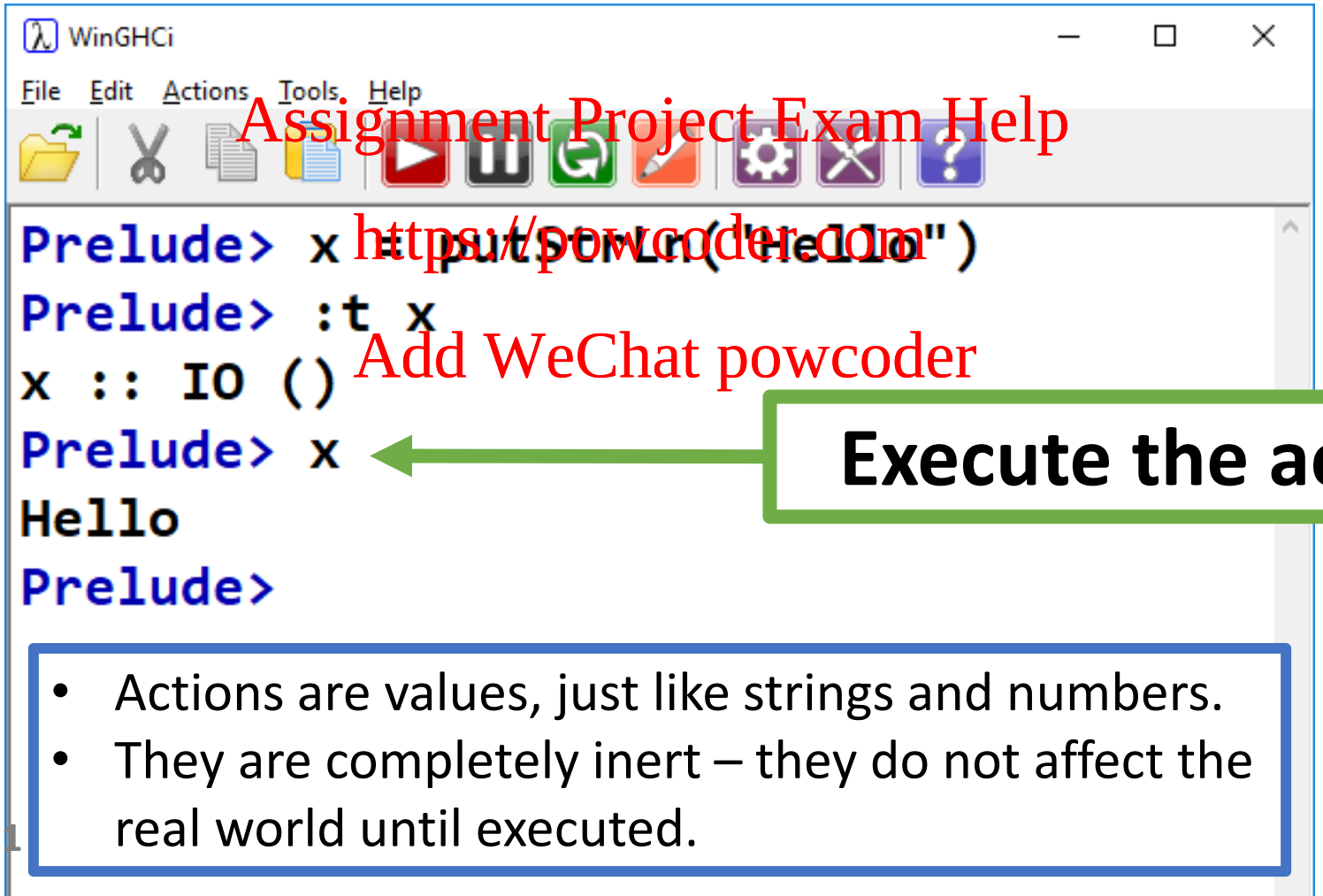
Haskell and I/O



<u>Speaking precisely:</u>

- **`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 ***<u>executed</u>***, it returns `()`.
- This can be read as an empty tuple.
- The *<u>action</u>*, when executed, produces a side effect.
- The **`putStrLn`** *<u>function</u>*, strictly speaking, does **not.**

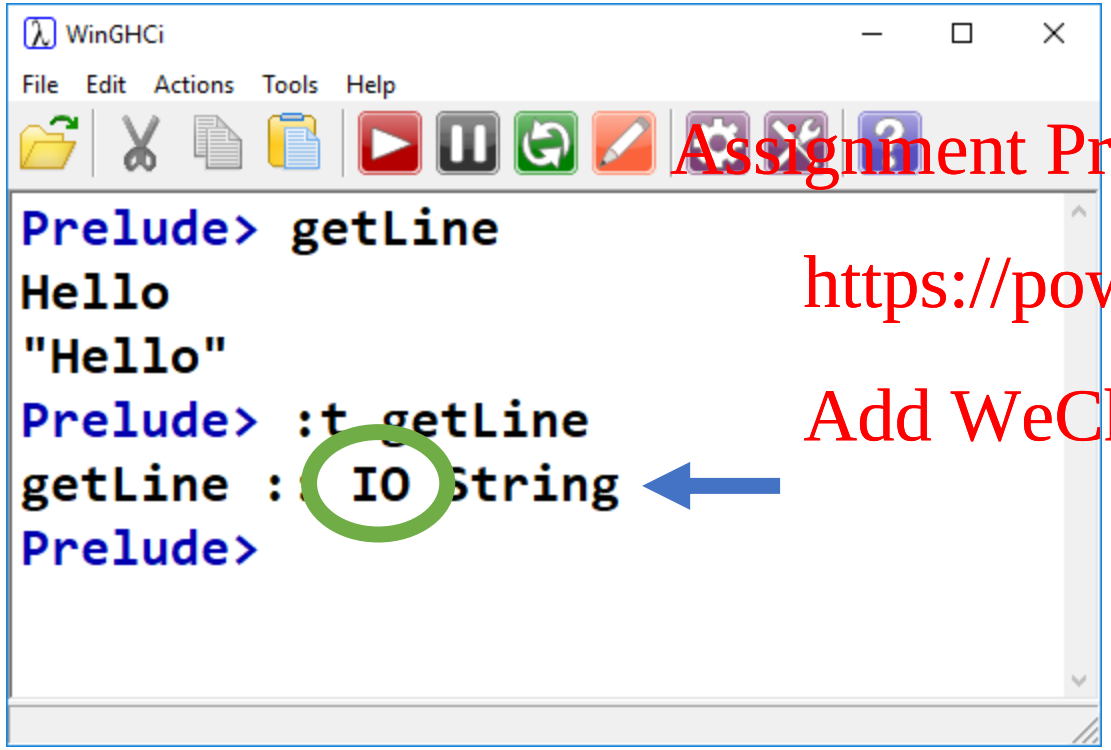
Haskell and I/O



Execute the action

- 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



- 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.

Functions are *evaluated*, actions are *executed* or *run*

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

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

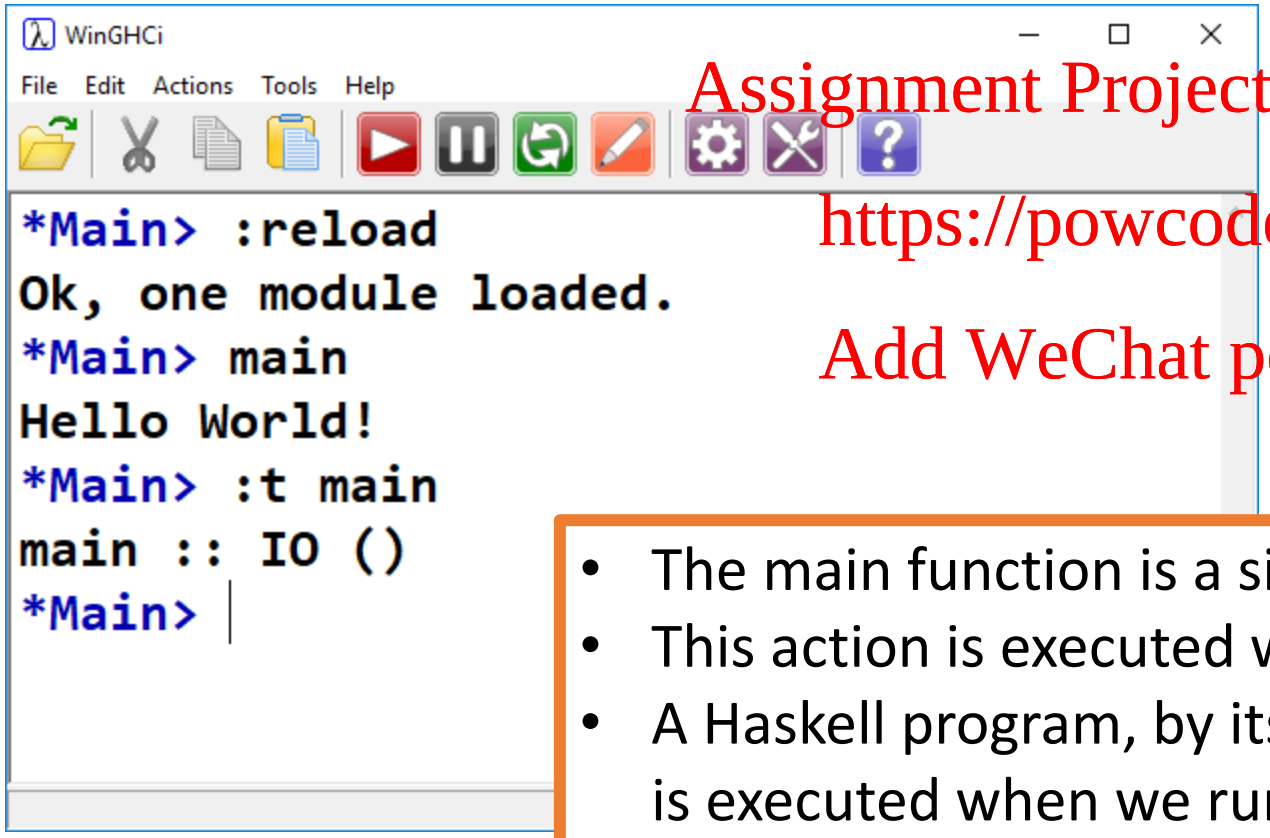
Actions can only be executed from within other actions.

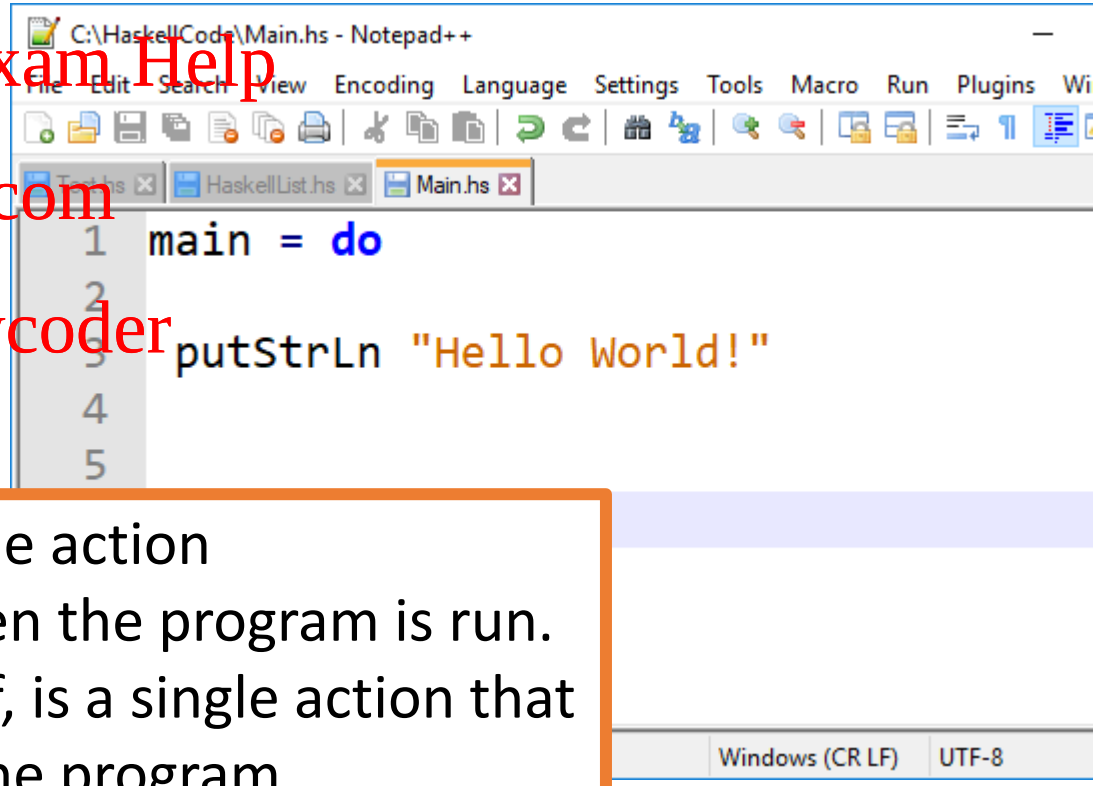
A compiled Haskell program begins by executing a single action – **`main::IO()`**

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

main::IO()

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





- 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

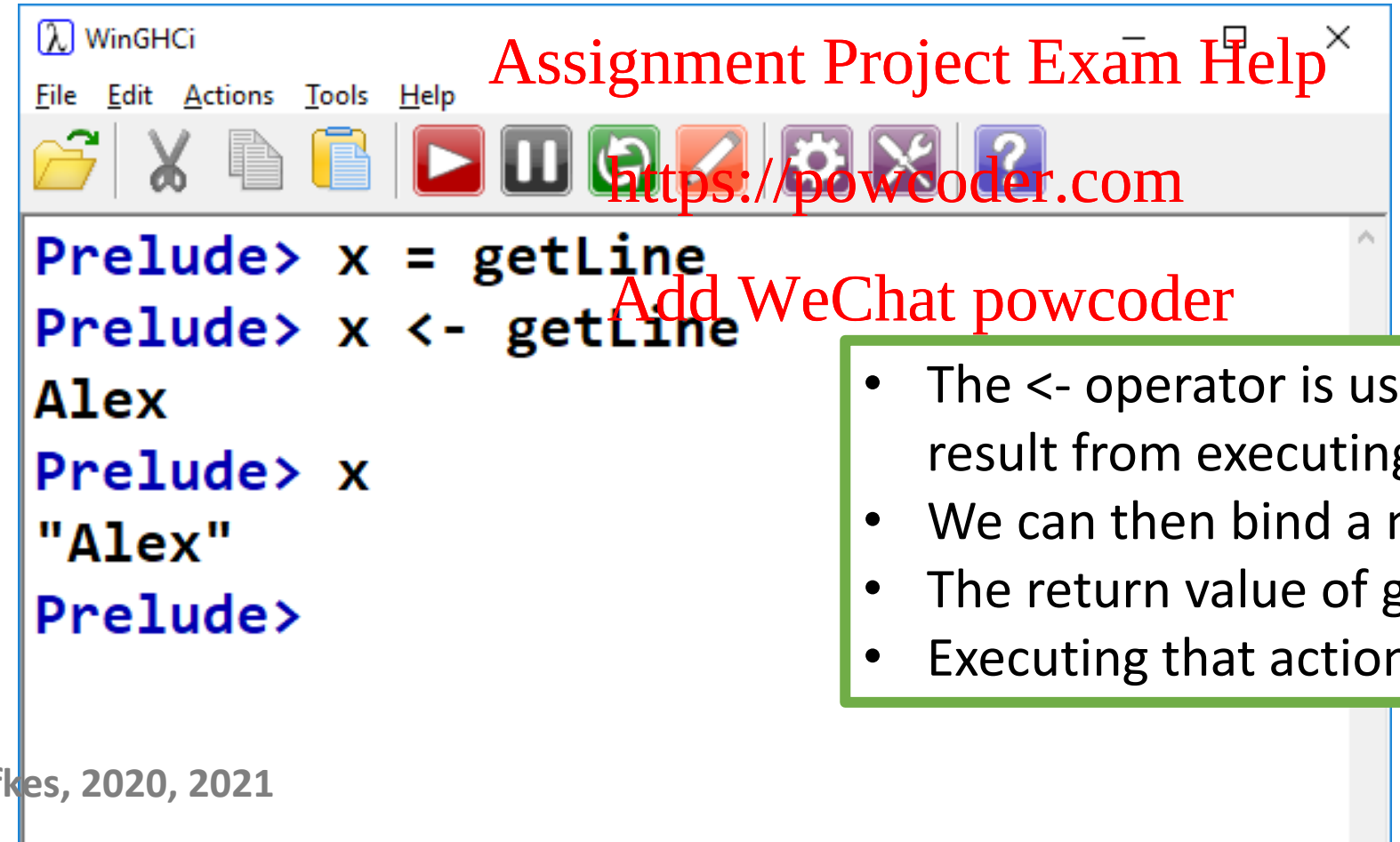
- A Haskell program begins with the execution of a single action (`main::IO()`)
  - Functions that *return* actions are often incorrectly referred to as actions.
- From within this action, any number of additional actions can be executed
- Pure functions can also be called/evaluated from within actions!
- 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 <- 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

WinGHCi

File Edit Actions Tools Help


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

Combining Actions

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

WinGHCi

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

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

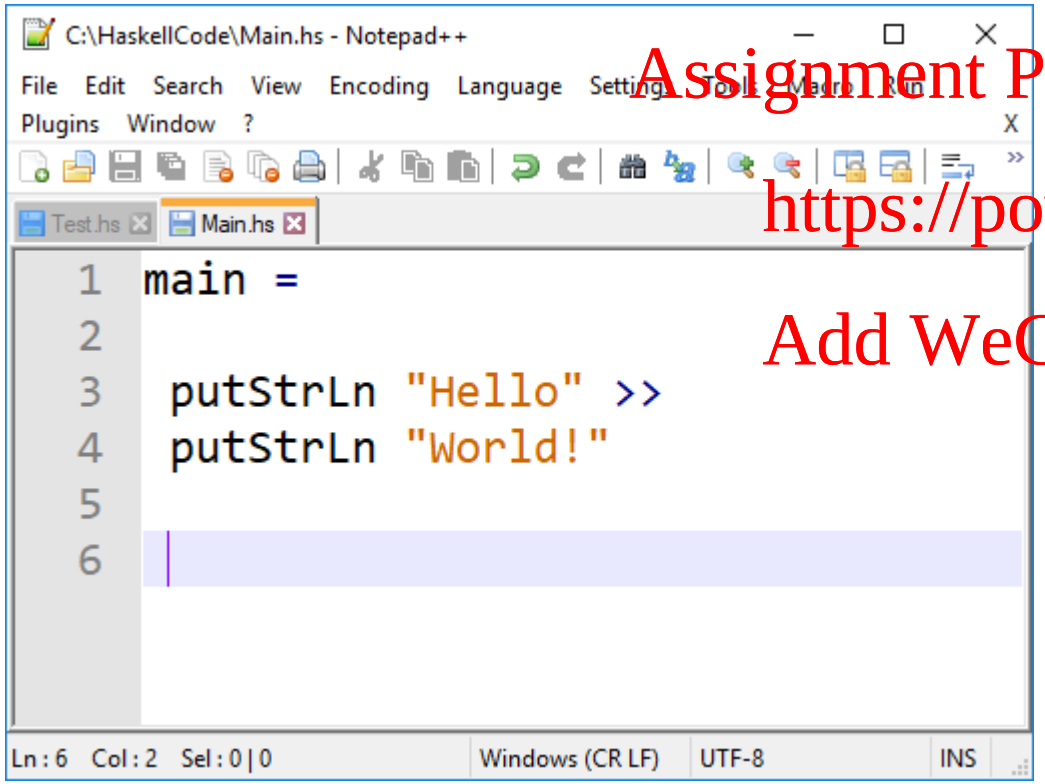
```
1 main = do
2
3   putStrLn "Hello"
4   putStrLn "World!"
5
```

length : 57 line Ln : 6 Col : 2 Sel : 0 | 0 Windows (CR LF) UTF-8

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

Combining Actions

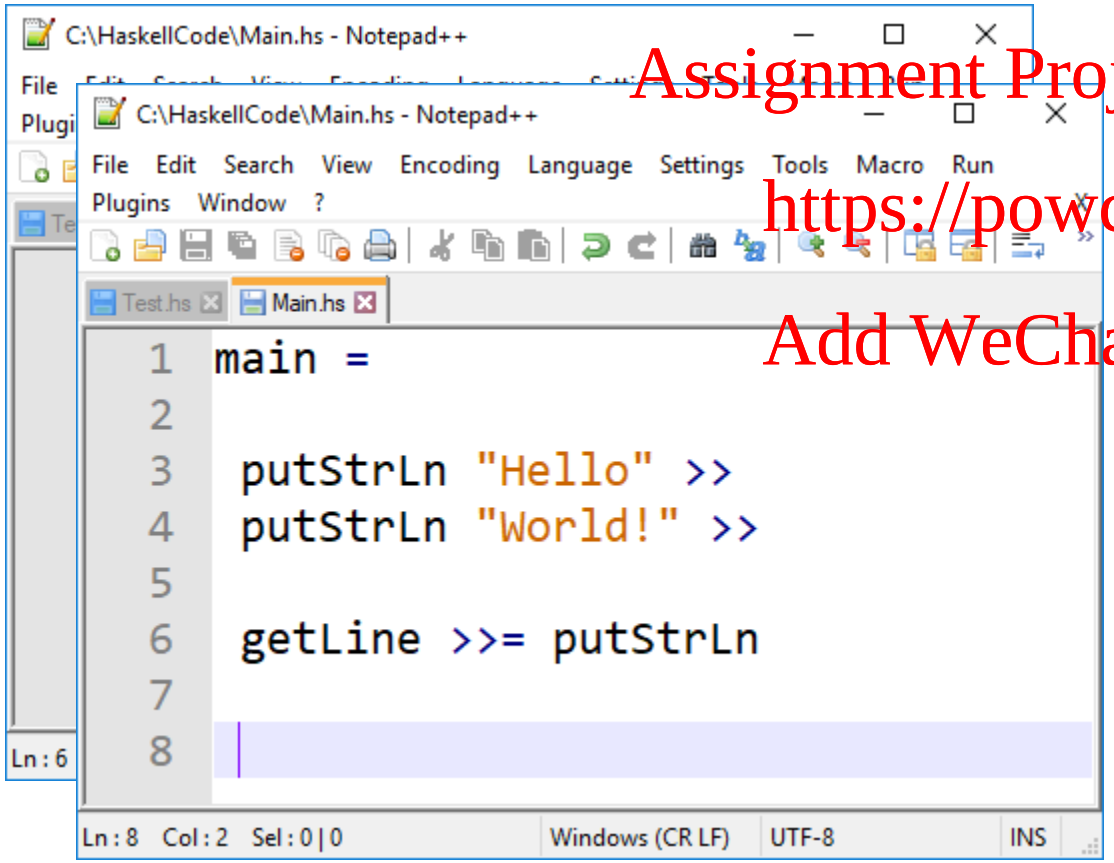
do is syntactic sugar for **>>**



- >> 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 **>>**



- >> 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 | Main.hs

```
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>
```

More Complicated

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

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

Test.hs | Main.hs


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


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

Haskel UTF-8

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.

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 | Main.hs

```haskell
main =
 putStrLn "What is your name?"
 >> getLine
 >>= \name -> putStrLn ("Hello, " ++ name ++ "!")
```

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

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

File Edit Search View Encoding Language Settings Tools Macro Run Plugins

Test.hs | Main.hs

```haskell
main = do
 putStrLn "Enter a number:"
 numStr <- getLine
 let num = (read numStr::Double)
 putStrLn (show (sqrt num))
```

length : 132 lines : 9

Command Prompt

```
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?

D:\GoogleDrive\Teaching - Ryerson\(C)CPS 506\Resources\Code\Haskell\main.hs - ...

File Edit Search View Encoding Language Settings Tools Macro Run Plugins

main.hs

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

length : 138 lir Ln : 9 Col : 2 Sel : 0 | 0 Windows (CR LF) UTF-8

- 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

WinGHCi

File Edit Actions Tools Help

```
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 ()

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

File Edit Search View Encoding Language Settings Tools Macro Run Plugins

Test.hs | Main.hs

```
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
```

length : 141 lines Ln : 9 Col : 2 Sel : 0 | 0 Windows (CR LF) UTF-8 INS

WinGHCi

File Edit Actions Tools Help

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


- 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 ()

WinGHCi

File Edit Actions Tools Help

```
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

WinGHCi

File Edit Actions Tools Help

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

WinGHCi

File Edit Actions Tools Help

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


```
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
instance Monad [] -- Defined in 'GHC.Base'
```


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”	xxx is an instance of type class Monad. xxx implements `>>` `>>=` and `return`
“action”	Another name for a monadic value

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.


```
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

```haskell
module Test where

 positive = do
  putStr "Enter a number: "
  num <- getLine
  (read num::Double) < 0
```

- 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*

WinGHCi

File Edit Actions Tools Help

```
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++

```haskell
module Test where

 positive = do
  putStr "Enter a number: "
  num <- getLine
  return ((read num::Double) < 0)
```

WinGHCi

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

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

```haskell
findBigger x y = if x > y then x else y

main = do
 putStrLn "Enter first number:"
 nStr <- getLine
 let num1 = (read nStr::Double)
 putStrLn "Enter second number:"
 nStr <- getLine
 let num2 = (read nStr::Double)
 let big = findBigger num1 num2
 putStrLn ("Larger: " ++ (show big))
```

Length : 304 lines Ln : 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             ( 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:

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

```haskell
module Test where

 testPos numString = do
  let x = read numString::Double
  if x < 0 then False else True

 positive = do
  putStr "Enter a number: "
  num <- getLine
  return (testPos num)
```

Pure!

!Pure

WinGHCi

```
*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.

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

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

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

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**"*

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

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

Computations that might "go wrong"

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

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

Maybe Monad

WinGHCi

File Edit Actions Tools Help

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

We've seen this before…

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

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

Test.hs | HaskellList.hs

```haskell
module Test where

data Pt = Pt3 Float Float Float
        | Pt2 Float Float

ptX (Pt3 x y z) = x
ptY (Pt3 x y z) = y
ptZ (Pt3 x y z) = z
```

length : 1,358 lines : 78 Ln : 12 Col : 2 Sel : 0 | 0

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

Maybe can take the value Nothing or Just a

WinGHCi

File Edit Actions Tools Help

```
*Test> :i Maybe
data Maybe a = Nothing | Just a    -- Defined in 'GHC.Ba
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

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

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

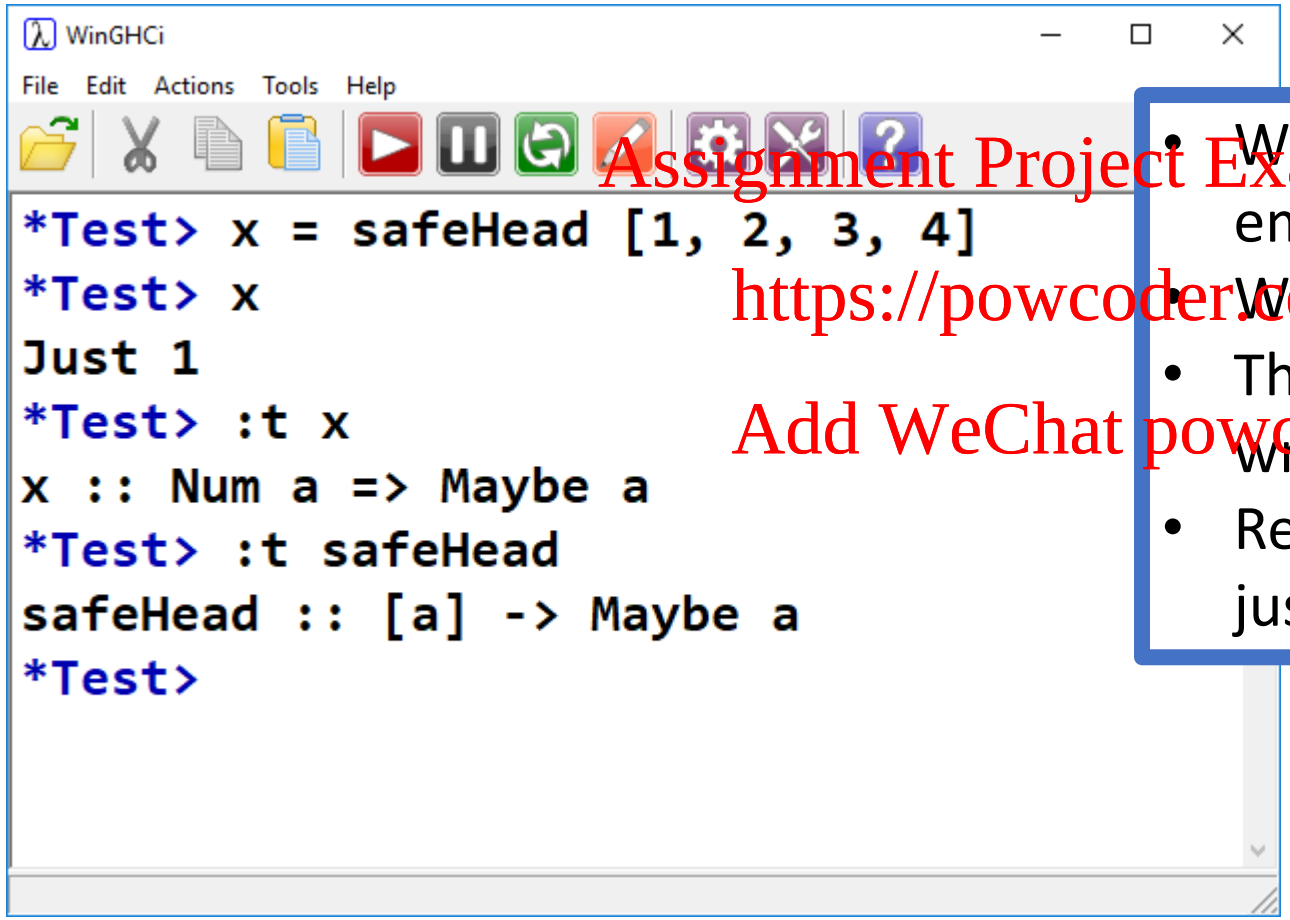
Main.hs | Test.hs

```haskell
module Test where

 safeTail x
  | (length x > 0) = Just (tail x)
  | otherwise = Nothing

 safeHead x
  | (length x > 0) = Just (head x)
  | otherwise = Nothing
```

length : 2,585 lin Ln : 11 Col : 2 Sel : 0 | 0 Windows (CR LF) UTF-8 INS

- 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?`

Maybe Monad



- 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

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

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

Main.hs | Test.hs

```haskell
module Test where

 safeTail x
  | (length x > 0) = Just (tail x)
  | otherwise = Nothing

 safeHead x
  | (length x > 0) = Just (head x)
  | otherwise = Nothing
```

length : 2,585 li Ln : 11 Col : 2 Sel : 0 | 0 Windows (CR LF) UTF-8 INS

WinGHCi

File Edit Actions Tools Help

Run text editor

```
*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++

Test.hs | HaskellList.hs

```
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

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

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

Test.hs | cond1.c

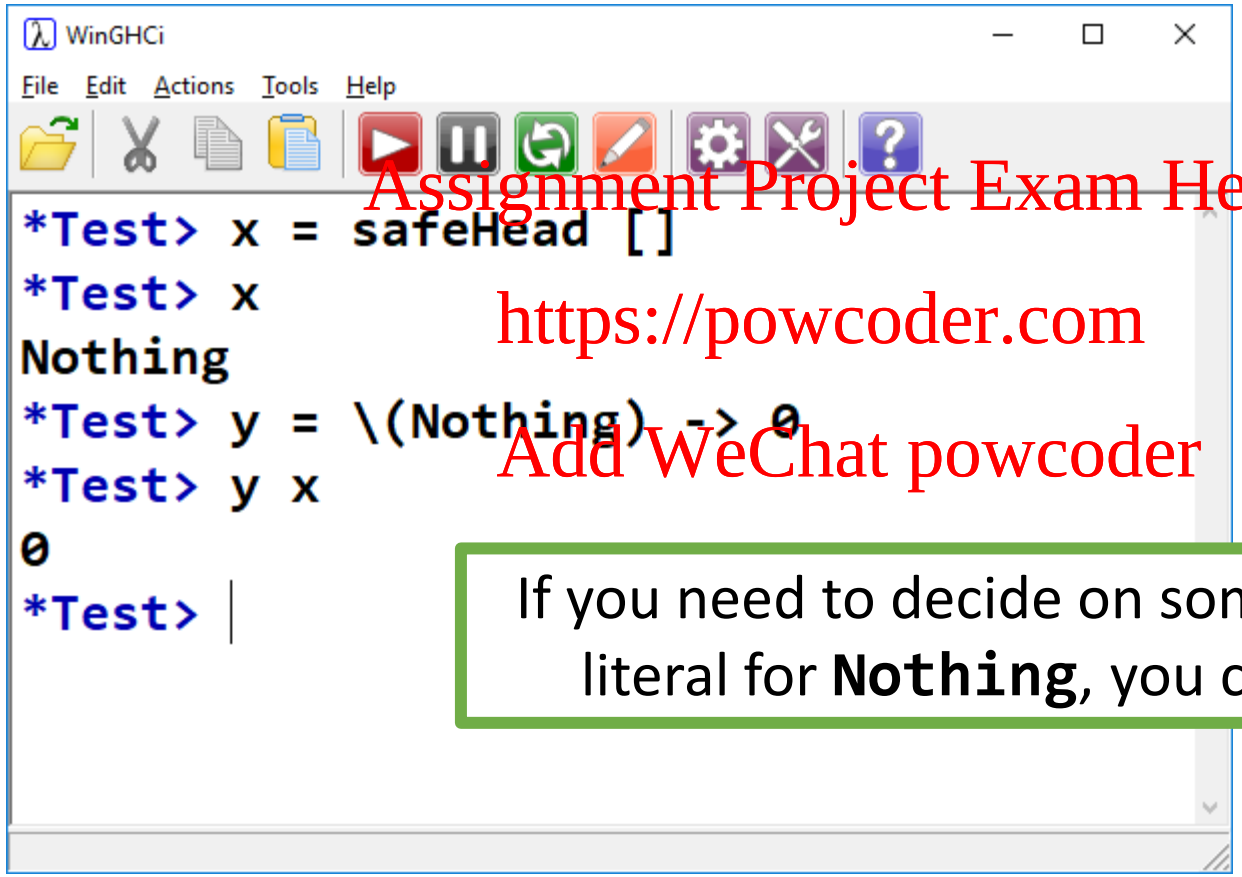
```haskell
module Test where

 safeTail x
  | (length x > 0) = Just (tail x)
  | otherwise = Nothing

 safeHead x
  | (length x > 0) = Just (head x)
  | otherwise = Nothing

 getMaybeVal (Just a) = a
```

WinGHCi

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>
```

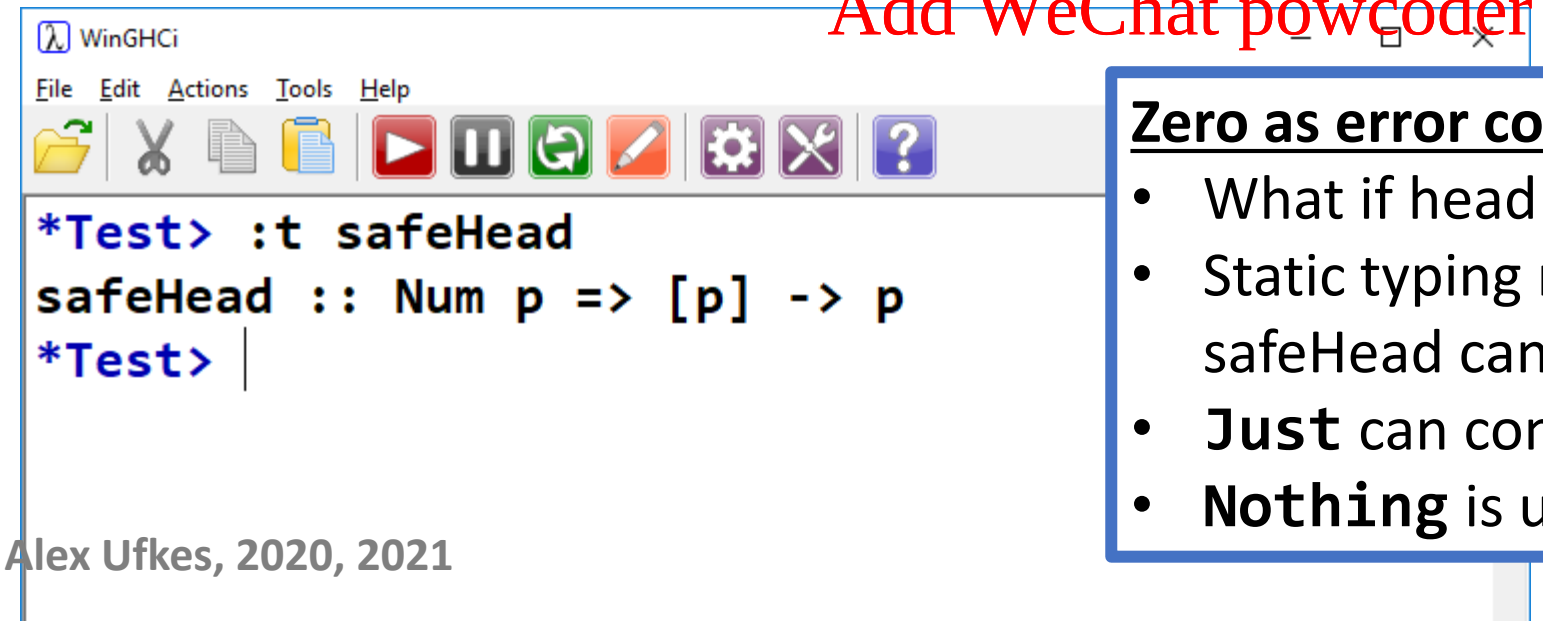
Unwrap *Nothing*?



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
```

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.

Should we use Maybe for everything?

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)  ]
```


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

Use lookup

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

File Edit Search View Encoding Language Settings Tools Macro Run

Main.hs | Test.hs

```haskell
module Test where

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

length : 2,729 lines Ln : 9 Col : 2 Sel

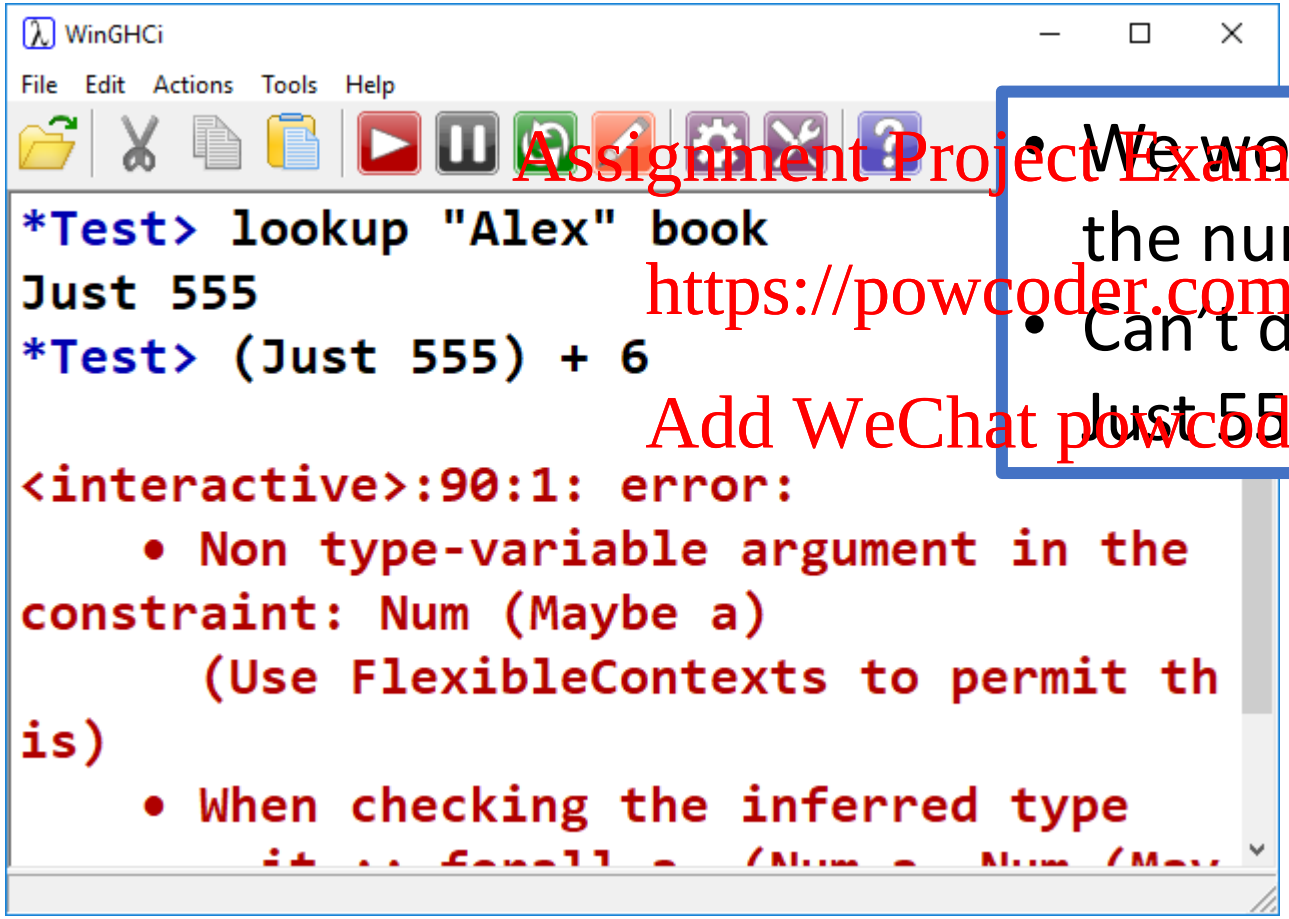
WinGHCi

File Edit Actions Tools Help

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


- 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



- 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

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   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) ]
12
```


WinGHCi

File Edit Actions Tools Help


```
*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 Wind

Test.hs | Main.hs

```
10   Nothing -> 0
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
22
23
24
```


- 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

```haskell
module Test where

 getPlace :: String -> Maybe String
 getPlace name = do
  code <- lookup name book1
  num <- lookup code book2
  lookup num book3

 fm m = case m of
  Nothing -> ""
  Just x -> x

 book1 = [ ("Alex", 555), ("John", 444),
           ("Tim",  333), ("Mark", 222) ]

 book2 = [ (555, 1), (444, 2),
           (333, 3), (111, 4) ]

 book3 = [ (1, "First"), (2, "Second"),
           (5, "Third"), (4, "Fourth") ]
```

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**?

WinGHCi

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

```haskell
module Test where

 getPlace :: String -> Maybe String
 getPlace name = do
  code <- lookup name book1
  num <- lookup code book2
  lookup num book3

 fm m = case m of
  Nothing -> ""
  Just x -> x

 book1 = [ ("Alex", 555), ("John", 444),
           ("Tim",  333), ("Mark", 222) ]
```


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

```haskell
module Test where

 getPlace :: String -> Maybe String
 getPlace name =
  lookup name book1 >>=
   (\code -> lookup code book2) >>=
   (\num  -> lookup num book3)

 fm m = case m of
  Nothing -> ""
  Just x -> x

 book1 = [ ("Alex", 555), ("John", 444),
           ("Tim",  333), ("Mark", 222) ]
```

Cascading Failure

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

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

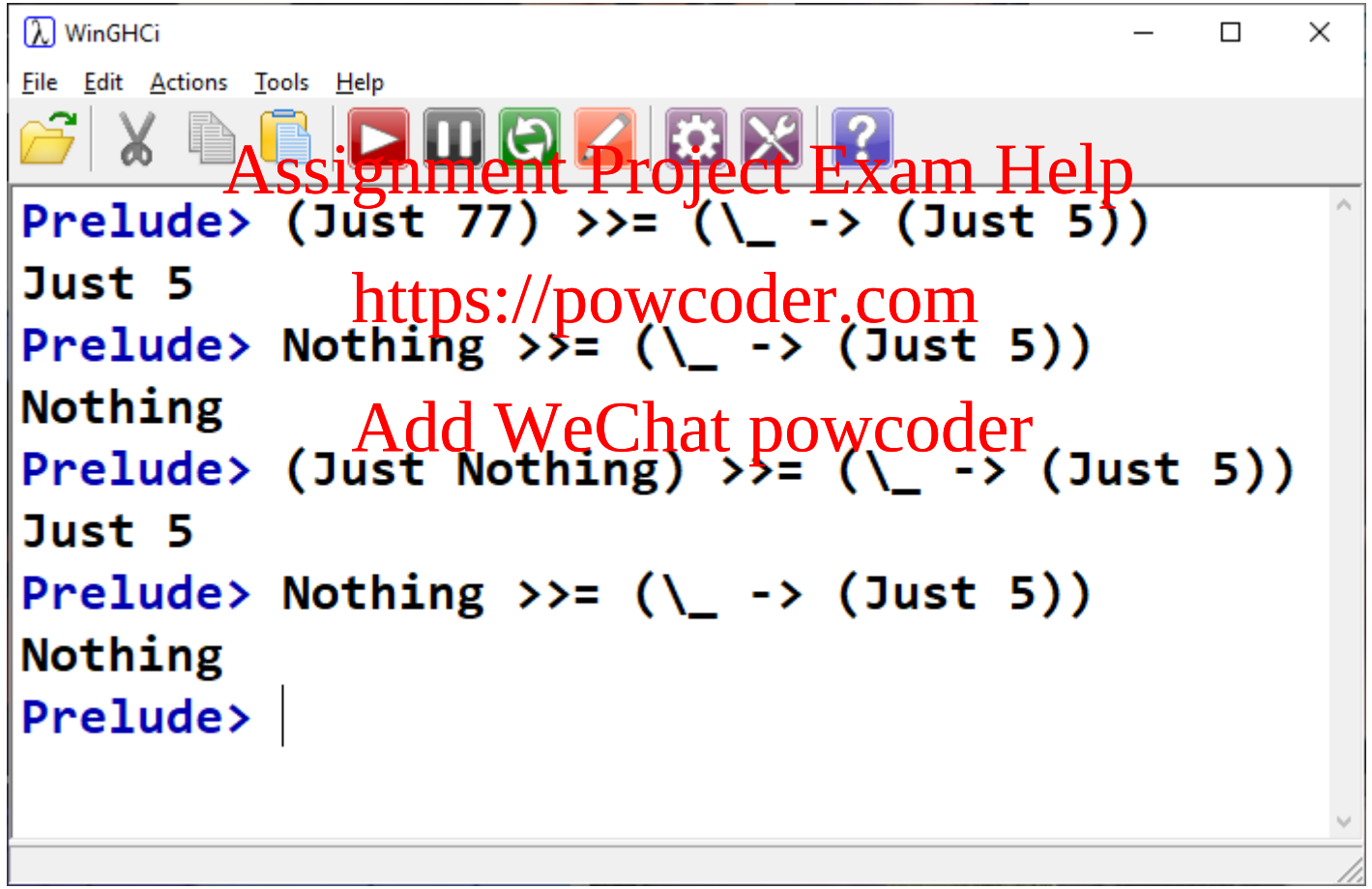
Test.hs | Main.hs

```
1  module Test where
2
3   getPlace :: String -> Maybe String
4   getPlace name =
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
```

Nothing

- When the first argument to (>>=) 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 >>=.
- The failure then cascades into the third >>=, and is returned.
- After the first Nothing, subsequent >>= pass that Nothing to each other

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



Haskell Tutorials/References:

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

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

Moving on…

…to imperative.

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