C/CPS 506

Compara Languages

Prof. Ale

Topic 8: Actions in Haskell

Notice!

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

The copyright to this ex Ufkes. Students registered in course CCPS 506 can us al 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! Thr eeks.
- Don’t forget about the assignments!

Let’s Get Started!

Last Week

ine what it means for two Pt2 iables to be considered equal

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

Last Week

Last Week

Last Week

onger need to derive
e’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,

Mo

Ac

Every function is pure

Pure Functions: Functions that have no side effects.

A function can be sai if it has an observable interaction with the outside worl m 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

Haskell and I/O

- The actual ***act*** of printing to the screen
  cur as a result of a function call.
  he screen is an ***action***.
- A **lues**, they have a type!
- **p** cepts a **`String`** argument.
- What it returns is an action of type **`IO()`**

Haskell and I/O

Speaking precisely:

- **`putStrLn`** is a *function* (no side effects!)
  - Takes a String as an input argument
  - s an action, whose type is `IO()`
  - `IO()` action is ***executed***, it
  - r
- ead as an empty tuple.
- The *action,* when executed, produces a side effect.
- The **`putStrLn`** *function*, 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 s a String (IO String vs IO ())

  Haskel *evaluation* doesn't ns 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 *executed* or *run*

Actions are values. Actions ca ed 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 H by executing a
sin `O()`

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, actions can be executed
- Pure functions can also b within actions!
- However – actions cannot be executed f ure functions.
- If we try, Haskell will infer the type of th ure 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 prod
- *The act of cr t have side effects.*
- The recipe can be the outpu unction.
- 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

Combining Actions

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

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

Combining Actions

do is syntactic sugar for **>>**

- >> says execute this, then this.
irst action produces a result,
carded.
e want to use the result?
>>= operator to pipe the result
into the next action.

Combining Actions

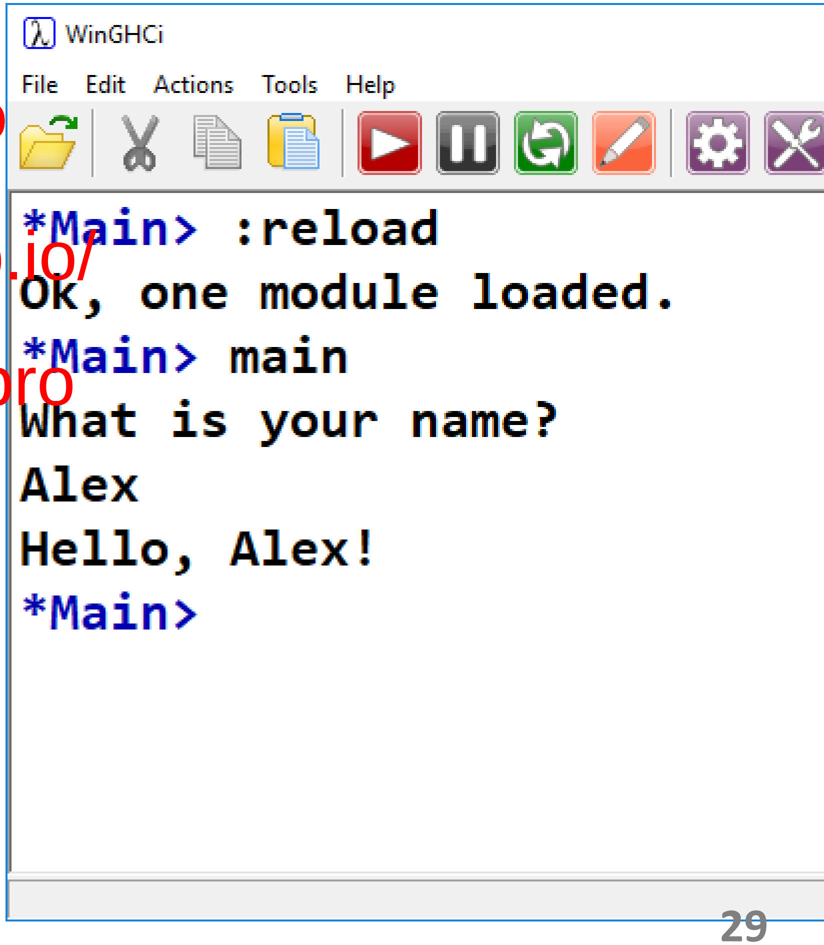
do is syntactic sugar for **>>**

- >> says execute this, then this.
  irst action produces a result,
  carded.
  e want to use the result?
  >= 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.

© Alex Ufkes, 2020, 2021

More Complicated

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



```
*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

Now we’re seeing how to wri e, and execute a complete Haskell program co *tions*.

Actions & Functions

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

Problem?

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

`return ()`

- 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 ()

Monads

- 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:

n 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

return

"action" Another name f ic 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.

>> then. Ignore result of le

`a >> b` **VS** `a >>= \_ -> b`

>> can be defined in terms of >>=

Non-main Example

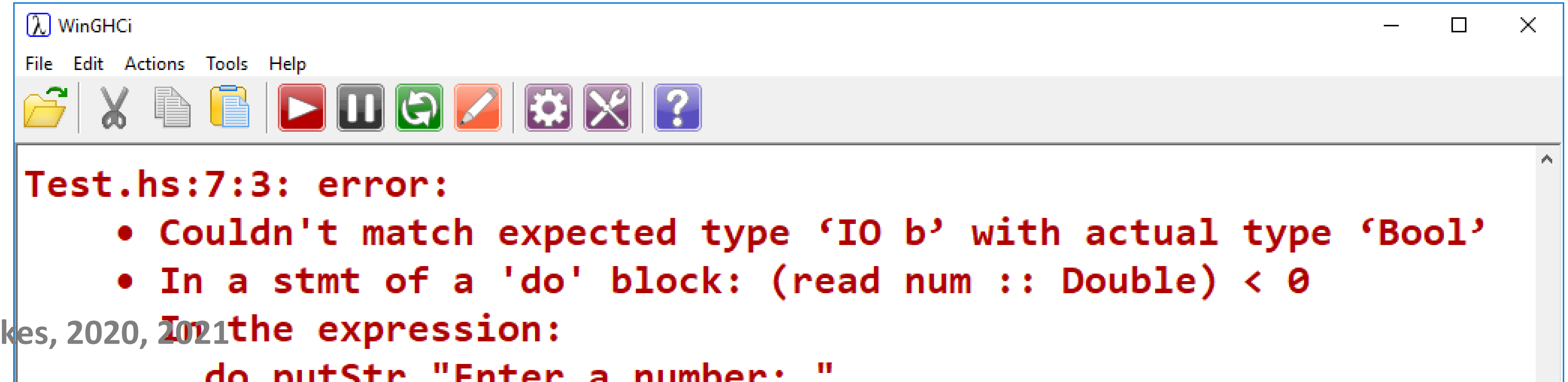
- Function that reads in a number
- Returns true if < zero, false otherwise

We’re executing IO actions

type cannot be Boolean

- I

something



```
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?

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:

Best Practice

Separate pure code into its own functions:

Pure!

!Pure

*When looking at **`main`**, Haskell looks rather imperative…*

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

It creates a separate construct for operations

cts

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. o_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 supplementin ulations with features like **I/O**, common environme ble 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 [illegible] eful for modelling other things

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

Maybe Monad

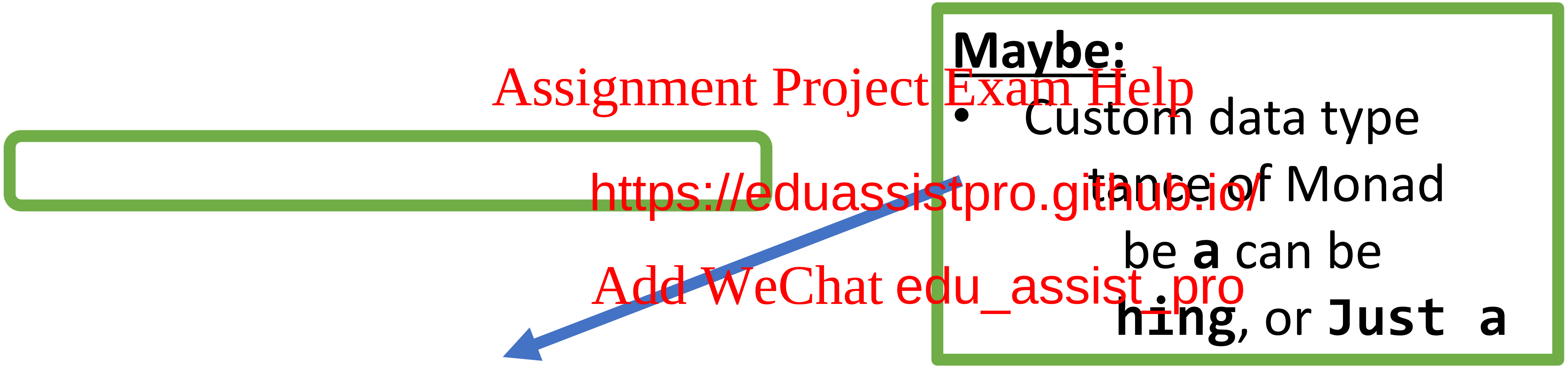
Represents a computation that might not produce a result

Comp o wrong”

For example – calling tail wit t might be empty

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

Maybe Monad



Maybe:

- Custom data type

tance of Monad

be **`a`** can be

`hing`, or **`Just a`**

We’ve seen this before…

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

take the value

g or Just a

Maybe Monad

- Define safe functions for head and tail.

  uards - |

  failing on e

  e othing.
- 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-
list, we don't get the head.
t *Just head*
e head of the list
in a Maybe monad.
- Remember that Maybe is a type, just like our custom Pt type

Maybe Monad

Unwrap `Just a`?

Just like pulling values out of our Pt data type!

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
  | oth
```

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 ything?

No. Not everything has a chance t ping 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”,
         (“John”, 444),
         (“Tim”,  333),
         (“Mark”, 222),
         (“Bill”, 111)  ]
```

want to search the

ame

nd, return its number

If not found, return…. ?

Use `lookup`

- 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
eric value 555
o arithmetic on
for example.

`Just` 555 VS 555

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

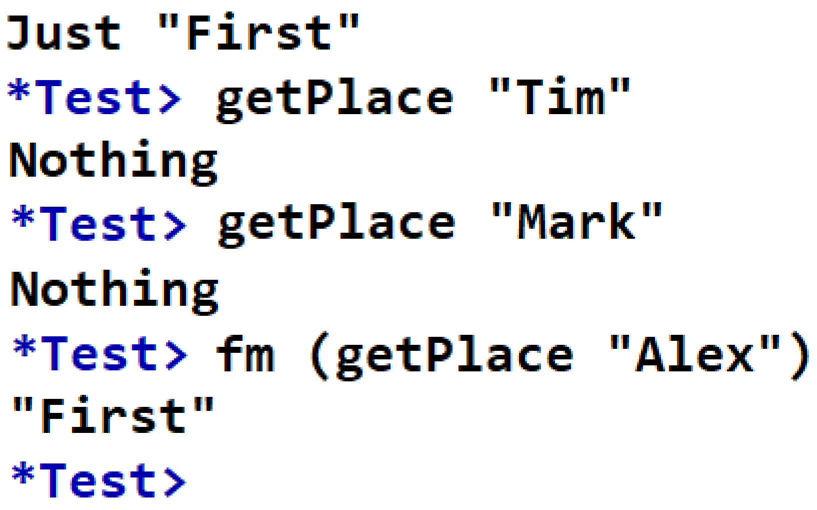
Use `lookup`

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

every value in book1 responds to a key in book2.

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

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



```
Just "First"
*Test> getPlace "Tim"
Nothing
*Test> getPlace "Mark"
Nothing
*Test> fm (getPlace "Alex")
"First"
*Test>
```

Cascading Failure

Is the same as:

Cascading Failure

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

  causes failure to cascade

  up fails, Nothing is

  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.wikiboo ther_Haskell_Tutorial

http://cheatsheet.codeslo heatSheet.pdf

Moving on…

erative.

Rust is an imperative language. How see many cool features that remind us of the functional languages we’ve seen.