Functional Programming (H)
Lecture 7: Introduction to IO

Simon Fowler & Jeremy Singer

Semester 1, 2024/2025
21st October 2024

https://sli.do, code 4098 384

University of Glasgow

https://sli.do, code 4098 384

Today

- Last time:
  - Recap lecture for Section 1 of the course
  - Class test
  - Parametric polymorphism / evaluation strategies
- Today:
  - (Finally) – how do we write Haskell programs that interact with the outside world?
  - How do we integrate pure and impure code?
  - The IO type and do-notation

https://sli.do, code 4098 384

Overall Course Structure

1. Haskell Fundamentals
2. Monads **<- you are here** (starting this section today)
   (Note: Jeremy will start lecturing next week – but I'll be back)
3. Haskell in the Large
4. Advanced Concepts

https://sli.do, code 4098 384

Programming Assignment

- At the beginning of Week 8 we will release the **programming assignment**, worth 25% of the course grade
- This will be a larger program (last year was Connect 4) – this year's will be similar
- Released: Monday 11th November 2024, deadline: Friday 6th December 2024
- The lecture on the 11th November will introduce the coursework in more depth
- Feel free to work on this in the labs; we are happy to discuss it

https://sli.do, code 4098 384

Class Test

- Thanks to everyone for completing the class test on Friday
- It's worth **10%** of the final course grade
  - If you think you did well – great, marks in the bag!
  - If you had trouble – don't worry, you can make up the marks elsewhere (and it hopefully highlighted where you need more practice)
- If you missed it, submit a Good Cause claim if appropriate
- Jeremy will be going through and marking these
  - You should hopefully have the results back within the usual timeframe (3 weeks from submission)

https://sli.do, code 4098 384

W4 Reflections

- Overall the feeling was “OK I guess” on the class test – several people thought they did OK, several struggled with not knowing the precise format
- **If you are struggling please get in touch with me: I am happy to meet**
- Type classes are still causing a few issues: I have put a video out going into a bit more detail and can distribute a bit more reading material.
- People seemed to enjoy the lectures (esp. revision lecture). This is good to know since I was considering axing that one next year!

https://sli.do, code 4098 384

Interesting Facts* (1)

***Not fact-checked!**

- People are born with only 2 fears. The fear of falling and loud noises.
- Most sharks don’t know how to file taxes
- Plato was a nick name meaning "Broad Shouldered" likely because he was muscular
- Fishing crates in Terraria are actually locked to the difficulty you fished them in. If you stockpile a bunch of crates & defeat wall of flesh, thus putting you into hardmode, your crates will still drop pre-hardmode loot. Shimmer updates their loot drops.
- Ants can survive in the microwave cos they can see microwave radiation like we can see light.

https://sli.do, code 4098 384

Interesting Facts* (2)

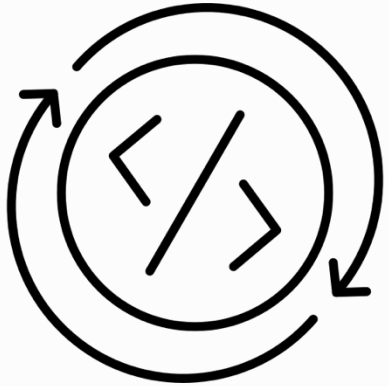
***Not fact-checked!**

- The inventor of the frisbee was cremated and had their ashes made into a frisbee.
- Humanity is doomed by overpopulation, climate change, resource depletion, and uncontrolled technology, leading to ecosystem collapse, uninhabitable environments, and societal breakdown.

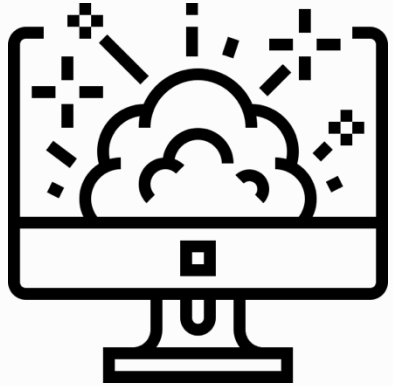
https://sli.do, code 4098 384

Purity

- So far, all the Haskell we have seen has been *pure code*
- But what does **functional purity** mean?



Stateless: the function always evaluates to the same result, given the same arguments



Side-effect free: the function does nothing apart from simple expression evaluation - no interaction with the 'outside world'

(ideally) **Total**: defined for all inputs

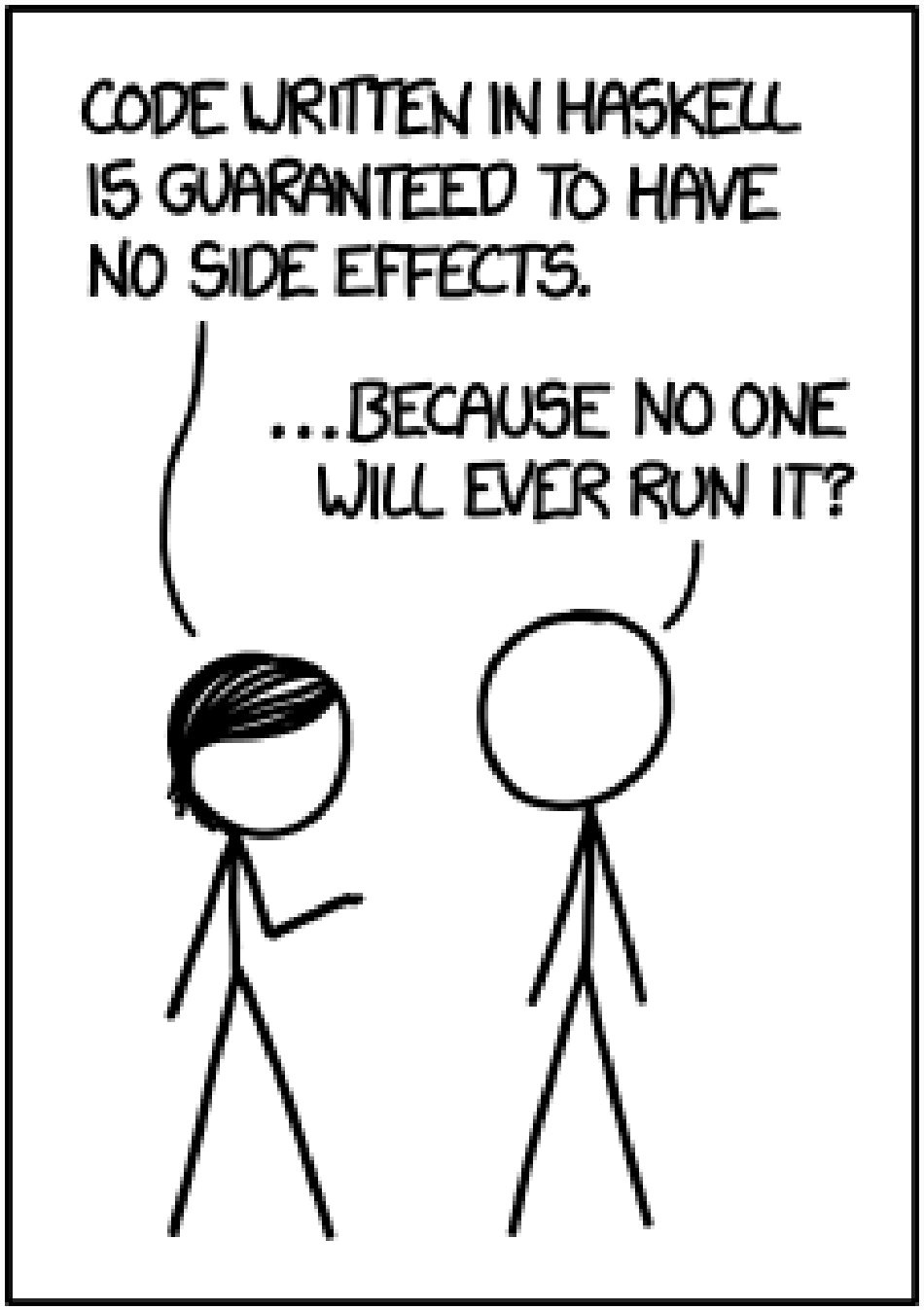
https://sli.do, code 4098 384

What's the problem with purity?

- Our code needs to do I/O
  - to get input from peripheral devices
  - to send output to the user
  - to read/write files
  - ...any others?

- But these are externally visible operations - side-effecting code
  - How do we manage this in a pure functional language like Haskell?
  - How do we integrate pure and impure code?

https://sli.do, code 4098 384

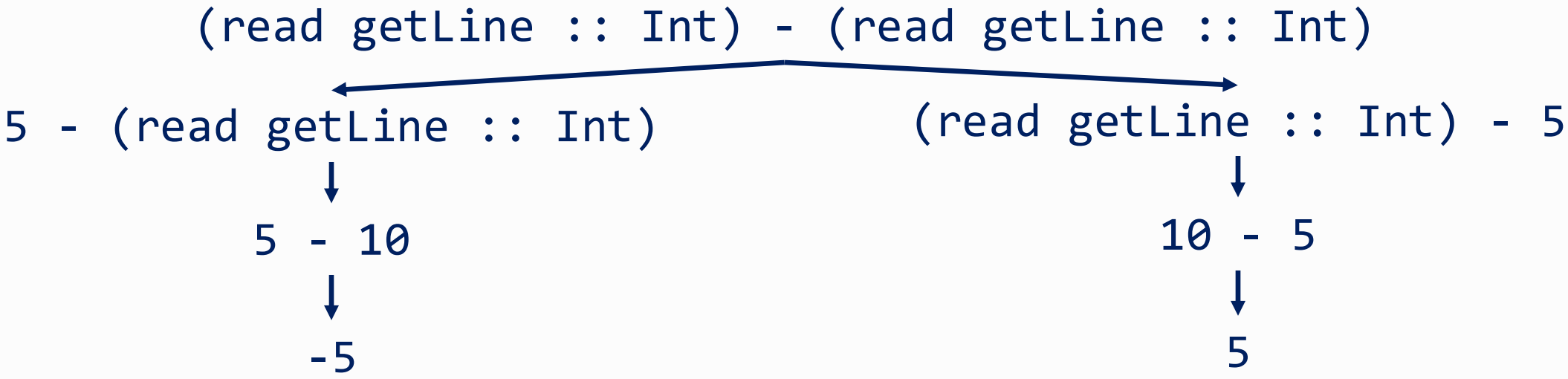


https://xkcd.com/1312/

Randall Munroe

https://sli.do, code 4098 384

What if we did the naïve thing?

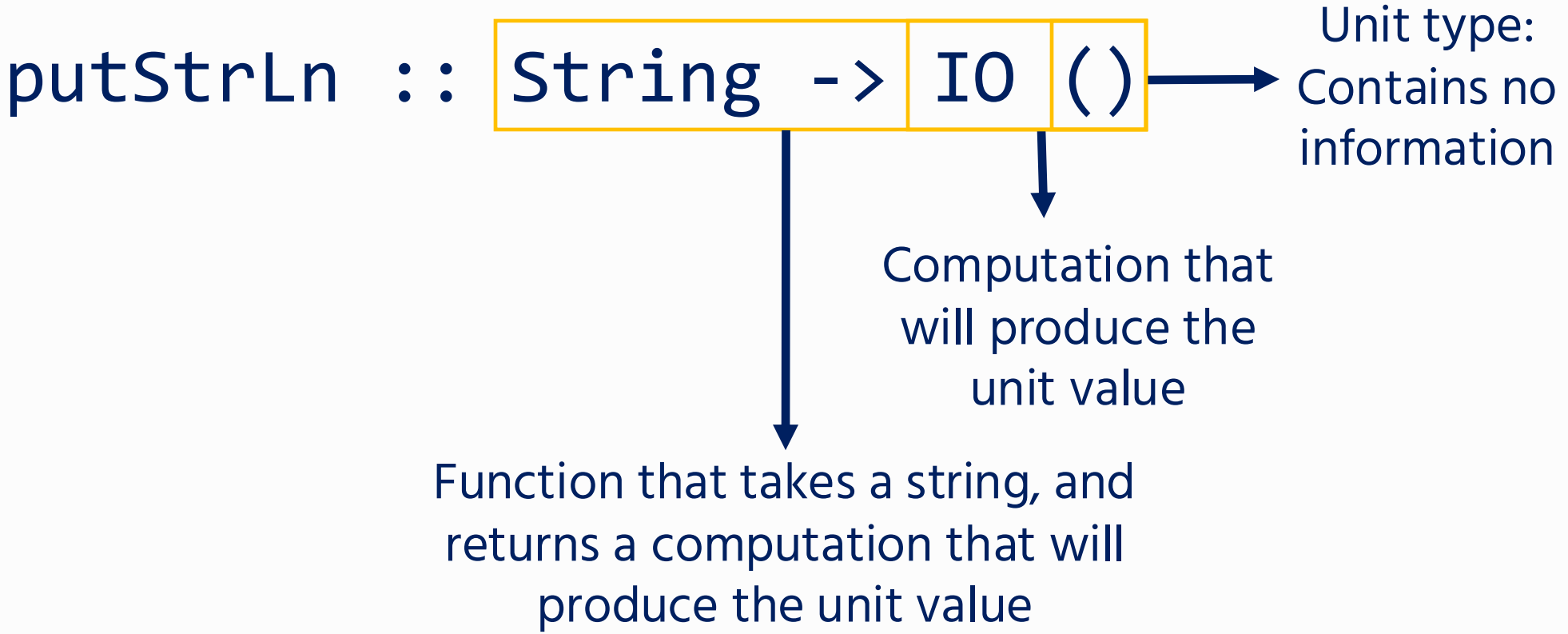


- We lose **Church-Rosser**
- We also lose **referential transparency** (the property that we can replace a piece of code with the value it produces)
- It also wreaks havoc with lazy evaluation
- …and therefore we lose all our reasoning power

https://sli.do, code 4098 384

IO Types

- **Key idea**: Each side-effecting operation is marked with a *type constructor*, IO



https://sli.do, code 4098 384

Mixing pure functions and IO computations

```
reverse getLine
```

```
reverse :: String -> String
getLine :: IO String
```

A `String` is not the same as an `IO String`!

A `String` is **data.**
An `IO String` is a **computation** which **produces** a `String`.

https://sli.do, code 4098 384

Side-effecting computations are **always** marked in their types.

https://sli.do, code 4098 384

Mixing pure functions and IO computations

```
getAndPrintReverse :: IO ()
getAndPrintReverse = do
  str <- getLine
  let revStr = reverse str
  putStrLn revStr
```

Mark that we're putting together a computation

Within a 'do' block, the `<-` operator allows us to give a name (`str`) to the result of an IO operation. Here `str` has type `String`

Print reversed string to console

Bind result of (**pure**) `reverse` function to `revStr` using **let**
(you don't need '**in**' in a do block)

https://sli.do, code 4098 384

IO computations that return values

```
getAndReverse :: IO String
getAndReverse = do
  str <- getLine
  let revStr = reverse str
  return revStr
```

Here we are defining an `IO` computation that returns a `String`

Note that we have to use the **return** function

```
return :: a -> IO a   -- roughly
```

(since `revStr` is of type `String`, and we need an `IO String`)

https://sli.do, code 4098 384

The Bigger Picture

- Every Haskell program has an entry point, `main`

```
main :: IO ()
main = do
  line <- getLine
  putStrLn (makeUpper line)
```

```
makeUpper :: String -> String
```

- `main` function is evaluated when program is run
  - It can then make use of other functions with IO type
- GHCI runs everything as an IO computation – which is why we can print things out
- Good practice: keep as much of the program as **pure as possible**
  - Pure functions are much easier to test and reason about

https://sli.do, code 4098 384

Escaping IO?

- There is no* way to "project" a value out of an IO computation
  - While you might want a function with type `IO String -> String`, this doesn't make sense: we would run into the same issues with Church-Rosser as before
- Instead, think of IO as though you are using do-notation to build a bigger computation by stringing together smaller IO computations, with `main` as your entry point

*I don't like lying to you.
If you import `System.IO.Unsafe`, there is a function

```
unsafePerformIO :: IO a -> a
```

This is not idiomatic Haskell and should only be used when you **really** know what you're doing. **You won't need it in this course.**

Sli.do Break

https://sli.do, code 4098 384

Trace Debugging

- That said, it is sometimes useful to do ‘print debugging’ where we wish to print some program state to the console
- We can do this using the **trace** function

```
import Debug.Trace
trace :: String -> a -> a
```

```
trace “returning 42” 42
```

- This uses unsafePerformIO under the hood, and should **only** be used for debugging

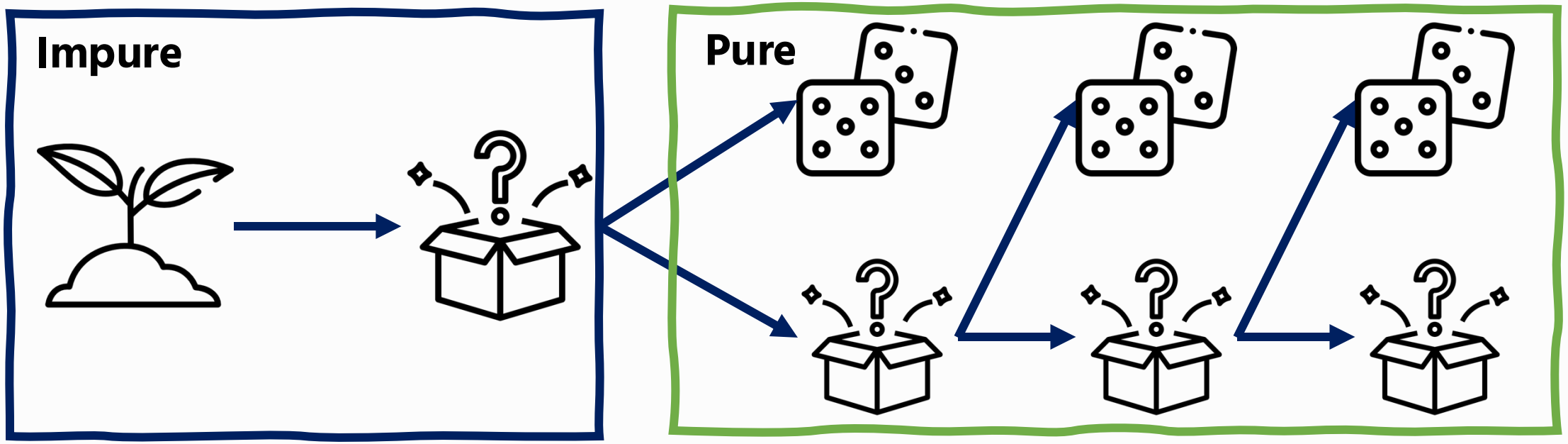
https://sli.do, code 4098 384

Beyond Console IO

- IO encompasses a large number of impure operations beyond just reading / writing to a console
- Getting current time:
  - `getCurrentTime :: IO UTCTime`
- File operations:
  - `readFile :: FilePath -> IO String`
  - `writeFile :: FilePath -> String -> IO ()`
- Also sockets, graphics, printing, spawning processes…
- Anyone think of any others?

https://sli.do, code 4098 384

Pseudo-Random Number Generation



- Random number generation might **seem** to be an impure operation
- In fact, a pseudo-random number generator generates a random value, and a new generator
- Only **seeding** the PRNG is impure, generation is pure

https://sli.do, code 4098 384

Reference Cells

- With IO we can make use of **mutable reference cells** that store some data, and its contents can be changed

`newIORef :: a -> IO (IORef a)`	Creates a new IO reference with an initial value of type a
`readIORef :: IORef a -> IO a`	Reads the content of an `IORef`
`writeIORef :: IORef a -> a -> IO ()`	Writes to an `IORef`

https://sli.do, code 4098 384

Sli.do Questions (1)

- Multiple choice in class test?
  - This was my (Simon)'s bad for suggesting it might contain MCQs – I'd extrapolated from previous Moodle quizzes / class tests.
    Nevertheless hopefully people still found it a fair assessment (this was the sense I got from the reflections).
- That fact about the fears seems made up
  - I am not fact checking these ☺ Added a disclaimer.
- Can Slide 10 be fixed as the text doesn't make sense without animations?
  - Yes – fixed (although I had to look up the meaning of "tgt")
- Can Monads be explained in detail?
  - Certainly! This is the subject of Thursday's lecture – and next week.

https://sli.do, code 4098 384

Sli.do Questions (2)

- Reference cells were a little rushed
  - They're not particularly important, but I'm happy to go over them again at the start of Thursday's lecture.
  - In short, though, a reference cell is like a type-safe pointer.
    We can create an `IORef Int` that is a reference to an integer (for example).
    We can then read the current value of that reference, or update its contents to a new integer.
- How can you have reference cells in Haskell -- I thought everything was immutable?
  - So the IORef *itself* is still immutable – it's a pointer. But its *contents* are mutable using the readIORef and writeIORef operators. Since working with IORefs is impure, all creation/read/write operations must be in IO computations.
- So reference cell = mutable reference cell, there's no such thing as immutable reference cell so immutable data just has no reference cell?
  - Yes. Data by default is immutable so doesn't need a reference cell. We use a reference cell when we want (controlled) mutability. Whenever I say "mutable reference cell" I mean "reference cell that contains mutable data".

https://sli.do, code 4098 384

Sli.do Questions (3)

- What's the difference between `putStrLn`, `show`, and `print`?
  - `putStrLn :: String -> IO ()` – this is an IO computation that prints the given string and a newline character to the console
  - `show :: Show a => a -> String` – this is a pure function that, for a member of the Show typeclass, returns its string representation
  - `print :: Show a => a -> IO ()` – this is an IO computation that (using the show instance) prints the string representation of the given value to the console
- Are `IO ()` and `IO a` the same?
  - No. `IO ()` is the type of a computation that produces the unit value (but may perform some side effects).
  - `IO a` itself is a strange type: the only way something could have type `IO a` is if it did not return (e.g. by erroring or looping forever).
  - However, you may be thinking of the return function `return :: a -> IO a`. The `return` function takes a value of type a, and gives back a trivial IO computation that produces that result.

https://sli.do, code 4098 384

Sli.do Questions (4)

- Why is it getLine and then putStrLn and not putStrLine? It annoys me.
  - Good point! (Not my choice, but sensible critique).
  - That said this is a great teachable moment about the difference between '**let**' and '<-' in do-blocks. Let's instead go with the convention that we want putStrLn and getLn…

```
getAndPrintReverse :: IO ()
getAndPrintReverse = do
  let getLn = getLine
  str <- getLn
  let revStr = reverse str
  putStrLn revStr
```

Note that
`let getLn = getLine`
is **renaming** the `getLine` computation rather than running it and binding the result. We can then use the `<-` operator to invoke our new `getLn` computation.

https://sli.do, code 4098 384

Sli.do Questions (5)

- Could you use the <- operator to "bind" the result of `reverse str` on `revStr` instead of the keyword `let`?
  - No – it's strictly "let" is for pure computations, "<-" is for running an IO computation and naming its result
- So to keep program as pure as possible you use the minimum number of IO operations?
  - It doesn't matter so much the number of IO operations you use. What you want to do is write as many pure functions as you can and not unnecessarily 'pollute' otherwise pure functions with IO.
  - It's better to do the IO in 'main' and then pass the results of user interactions as parameters, for example.

https://sli.do, code 4098 384

Sli.do Questions (6)

- Can we use do and the `<-` operator to split `Just` from a the same way we can split IO under a do operation?
  - Indeed we can! IO is what's known as a monad, and we can use do-notation and `<-` for all monads. We'll see this on Thursday.
- What do you mean by "escaping" from IO - do you mean converting it to string and returning string without the IO?
  - I meant taking an IO computation (e.g. `getLine :: IO String`) and using it in a pure function (using some imaginary function with type `IO String -> String` – which would allow us to 'pretend' that an IO computation is actually pure).

https://sli.do, code 4098 384

Sli.do Questions (7)

- On slide 17, `revStr` has type `String` and we only return `revStr` so in the type signature why do we write `IO String` instead of just `String`?
  - This is a great question. Remember that when we use a 'do' block we are describing how to build up a computation. That computation (since it uses IO operations) must be an IO computation, so the final type must be `IO String` (hence why we need to use `return`).
- What do you mean, `return` is 'pure'?
  - 'return' creates a 'trivial' computation that returns the given value (without performing any side effects).
  - My point was that '`return`' is a bad name as it confuses people who are familiar with '`return`' in Java and C. So you can also use the function called '`pure`' in Haskell instead of '`return`' (indeed this is what's done in Idris).
- So IO has no referential transparency?
  - Indeed – IO computations are not referentially transparent.

https://sli.do, code 4098 384

Sli.do Questions (8)

- What is an "IOU"?
  - Typically a note acknowledging debt. But I think you may have misheard me say "IO Unit", which is the type `IO ()` that describes an IO computation that returns the unit value ()
- So the unit type is kind of like futures?
  - No. The unit type on its own is just an empty record. In a pure function it's not much use, but an IO computation of type `IO ()` typically indicates that the computation doesn't return a result, but performs a side effect (like `putStrLn`: we print to the console, but don't return a sensible result).
  - The main comparison to `IO ()` in imperative languages is a void method in Java. But since Haskell is expression based, we need to return *something*, so we return `()`.
  - On the other hand you can think of a future as a placeholder value. You make a request, and get back a `Future a`. You can then block waiting for the result.
  - There are implementations of futures in Haskell (see e.g. https://hackage.haskell.org/package/futures-0.1/docs/Futures.html, and the Concurrent Haskell MVar structure: https://hackage.haskell.org/package/base-4.20.0.1/docs/Control-Concurrent-MVar.html#t:MVar).
    Since futures make use of concurrency, they are necessarily IO computations.

https://sli.do, code 4098 384

Sli.do Questions (9)

- In strict evaluation it's always left one evaluated first then right, but in lazy evaluation you never know the order?
  E.g. in `(read getLine :: Int) - (read getLine :: Int)`
  - It depends on the language semantics. The main thing with strict evaluation is that all arguments are evaluated prior to making a function call – the order is implementation-dependent (for example I believe OCaml has a right-to-left evaluation order).
  - With lazy evaluation, the arguments are only evaluated on-demand. If there are multiple possible reduction paths then (assuming pure computations) you can evaluate them in any order and get the same result.
  - The point with this example is that sometimes ordering really matters when doing side-effecting computations, so we need to separate (at the type level) pure and impure computations.

https://sli.do, code 4098 384

Sli.do Questions (10)

- Functor vs map?
  - Jeremy will go into this in more detail later in the course – so he will explain it much more there.
  - But a 'functor' is a generalisation of the 'map' function we have seen for lists: it allows us to apply a function to the contents of a data structure without changing its structure. For example…

```
map     :: (a -> b) -> [a]    -> [b]
treeMap :: (a -> b) -> Tree a -> Tree b
```

  - We are able to generalise this using the Functor typeclass, which we separately introduce for trees and lists, then we can use the fmap function:
  - `fmap :: Functor f => (a -> b) -> f a -> f b`

https://sli.do, code 4098 384

Sli.do Questions (11)

- So 'type class' is the only class in Haskell - there is no other 'class'?
  - Yes. The object-oriented notion of a 'class' is not present in Haskell (since Haskell doesn't have objects). Think of them as separate things.
  - A class in Java is a template for creating objects (strictly speaking).
  - A typeclass in Haskell is more like a Java interface. It specifies a list of operations that a data type must implement in order to be a member of a class (e.g. to be a member of the Leggy typeclass, a type a must implement `numLegs :: a -> Int`).
- Parametric polymorphism vs ad-hoc polymorphism? So parametric = fn can accept val of any type, adhoc = fn behaves differently depending on each different type?
  - Yes, that's exactly it. Parametric polymorphism means that we can't make any assumptions about the precise types, but then the function works for any type. For example:
    rotate :: (a,b,c) -> (c, a, b) is parametrically polymorphic because we just shift around the elements of a tuple without doing any concrete operations on them.
  - Then as you say, ad-hoc polymorphism allows us to implement a function with the same name differently on each type.

https://sli.do, code 4098 384

Sli.do Questions (12)

- So is it that all adhoc polymorphic functions must be parametric polymorphic (so it can accept different types in the first place) but not the other way around?
  - Mostly correct. The idea is that we write a type signature, say `numLegs`...
  - `numLegs :: Insect -> Int`
  - Then we think – could this be useful for other data types? And we can generalise it as a typeclass (like our Leggy typeclass). And then the type of numLegs becomes...
  - `numLegs :: Leggy a => a -> Int`
  - Which can be read: numLegs is defined for any 'a', *as long as* that 'a' is a member of the `Leggy` typeclass. So unlike parametric polymorphism we're allowed to make *some* assumptions about the type we have.

https://sli.do, code 4098 384

Sli.do Questions (13)

- IO is like a generator then?
  - I would say a generator is more specific to lists / arrays, whereas an IO computation is a description of a side-effecting operation.
- Why don't you need 'in' in `main` when you use let?
  - When we're using 'do' notation we can drop the 'in' (it's syntactic sugar). We'll go more into specifics about do-notation on Thursday.
- Is do not a while loop if in main?
  - No, please don't confuse 'do'-notation and the imperative 'do-while' looping construct; there are no similarities between the two.
- Why does getLine assume input is a string? What if you want to perform int operations? Do you need to convert?
  - Yes – you need to convert manually. Otherwise Haskell would be guessing the meaning of your input. It's better to return it as a string and leave you to be explicit. The lab sheet has a question on this.

https://sli.do, code 4098 384

Sli.do Questions (14)

- How did you make random number generator generate the same set of values each time? Did it not have a seed at all in this case?
  - There were two times that I made the RNG give the same set of values. The first was when I explicitly seeded it with '123'.
  - The second was when I seeded it with the number of seconds since the Unix epoch. Invoking the function twice in the same second would give the same seed and generate the same numbers.
- When you say only seeding is impure is it because just the seeding dynamically chooses the probability distribution?
  - The main thing is that we need some entropy (randomness) to pick the initial seed. This can be things like the current time, mouse position, contents of memory – but picking any of these things is impure.

https://sli.do, code 4098 384

Sli.do Questions (15)

- In Haskell aren't functions defined as a chain of single-argument functions by default? How is monad type class special when it comes to 'chaining'?
  - So the chaining you are thinking of is currying. Here we're building up an n-argument function by nesting n single-argument functions. The chaining here is based on having access to all parameters.
  - Monadic 'chaining' is different: we're building up a description of a side-effecting computation. The 'chaining' allows us to use the results of previous side-effecting computations in the remainder.
  - This should hopefully become clear when I give the concrete typeclass definitions next time. The main thing about this lecture was to build up the intuition.

https://sli.do, code 4098 384

Sli.do Questions (16)

- Type constructor is different from type signature?
  - Yes. A type signature tells us what type a function has, e.g.
    `add :: Int -> Int -> Int`
  - A type constructor is something like `Maybe`, which takes other types as parameters (allowing us to have Maybe Int, Maybe Bool, etc.)
- Would ' x <- [1..]' break lazy evaluation?
  - This has a nuanced answer and should become a little clearer next lecture. In essence we couldn't write x <- [1..] in an IO computation, since [1..] doesn't have an IO type. We can build a "list" computation though, which you'll see next time. Lazy evaluation still works as expected.

https://sli.do, code 4098 384

Sli.do Questions (17)

- Can you explain the trace function a little more?
  - trace :: String -> a -> a
  - Here “String” is the message to print to the console. The “a” parameter is what we want to return.
    So trace “returning 42” 42 will print “returning 42” to the console, and return 42.
  - Here is an example for tracing a factorial function.

```
fact :: Int -> Int
fact 0 = trace (show 1) 1
fact n = trace ((show n) ++ " * ") (n * fact (n-1))
```

```
ghci> fact 5
5 *
4 *
3 *
2 *
1 *
1
120
```

https://sli.do, code 4098 384

Sli.do Questions (18, phew!)

- How does it wreak havoc with lazy evaluation? With strict evaluation it still accepts user input twice?
  - This is a good question. "Wreak havoc" probably was a bit hyperbolic, but it does mean that we need to be careful with IO.
  - The point is – laziness means that we only evaluate an expression when it is needed (and indeed this means we can have *multiple reduction paths,* which in turn causes problems with Church-Rosser).
  - do-notation and IO computations ensure that the IO operations have a strict sequencing, which gives us some predictability about when they are evaluated.

https://sli.do, code 4098 384

Wrapping Up

- **Today**
  - The need for impurity
  - The IO type
  - do-notation
- **Next up**
  - Building up other types of computation: for example, Maybe computations, nondeterministic computations
  - The ‘M’ word…
- **Questions?**

do, code 4098 384