

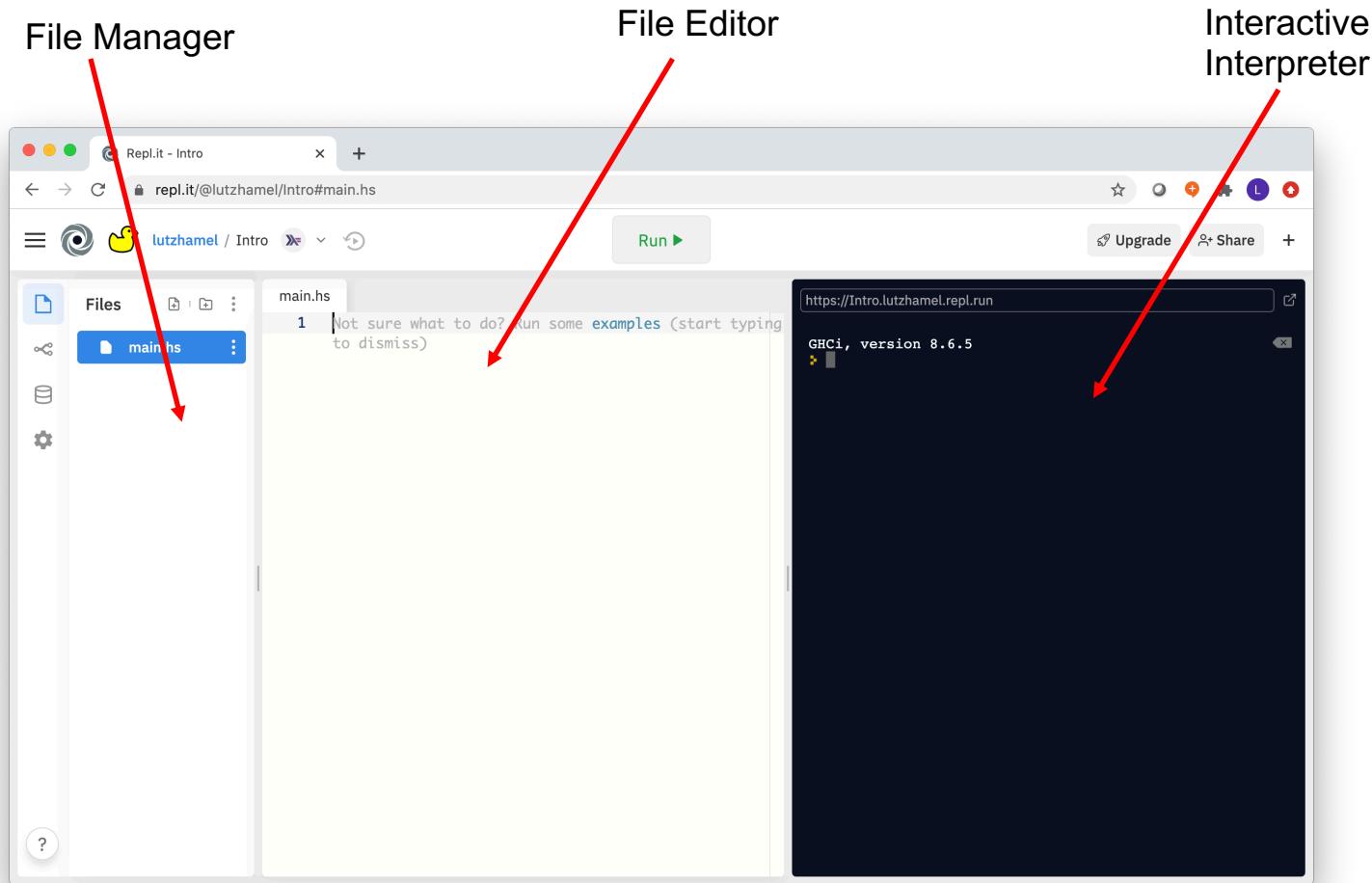
Haskell

- Haskell is a general-purpose, statically typed, purely functional programming language with type inference and lazy evaluation.
- We will be using the repl.it interactive environment
 - Please sign up for a free account
- At the prompt the system expects a valid Haskell expression
- In Haskell **everything** is an expression – that is, everything has a **value** and a **type**

Reading

- Please read Haskell Basics
 - <https://github.com/lutzhamel/CSC301>

Haskell – repl.it



Live Coding

- For this lecture and the remaining lectures on Haskell please have your browser window open to your repl.it account
- You should be pointing to a Haskell repl so you can follow along with the coding.

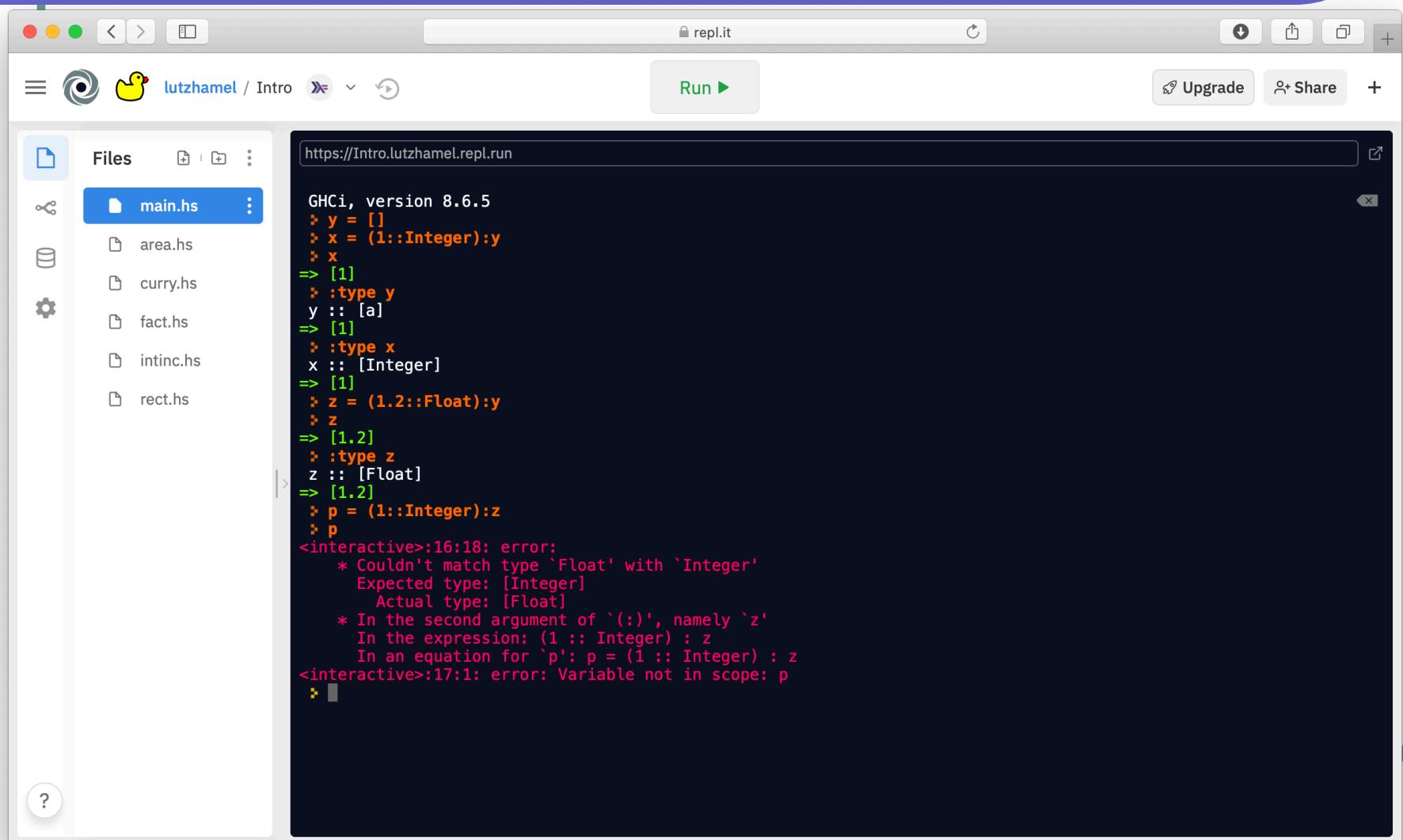
Why Functional Programming?

- The interesting part is that it is a programming language modeled after the *mathematics of functions* and *type theory*.
- In particular, the type systems in functional programming languages really shines because of the solid mathematical foundation and their expressiveness.
- Parametric polymorphism, that is types with type variables, occur very naturally within the types of functional programming and do not need the heavy handed $\langle T \rangle$ notation of non-functional programming languages.

Why Functional Programming?

- Consider the empty list []
- By itself it is impossible for Haskell to figure out what element type this list should have.
- The way Haskell's type system solves this dilemma is by using a *type variable* for the element type
- so the type of an empty list is: [a]
 - Here a is a type variable which will be instantiated to a type as soon as we put the first element in the list

Example Session



The screenshot shows a web-based Haskell development environment on repl.it. The interface includes a file browser on the left, a code editor in the center displaying a GHCi session, and various navigation and sharing buttons at the top.

File Browser: Shows files main.hs, area.hs, curry.hs, fact.hs, intinc.hs, and rect.hs.

Title Bar: Shows the URL https://Intro.lutzhamel.repl.run and the repl.it logo.

Toolbar: Includes a Run button, Upgrade, Share, and a plus sign for creating new files.

Code Editor (GHCi Session):

```
GHCi, version 8.6.5
> y = []
> x = (1::Integer):y
> x
=> [1]
> :type y
y :: [a]
=> [1]
> :type x
x :: [Integer]
=> [1]
> z = (1.2::Float):y
> z
=> [1.2]
> :type z
z :: [Float]
=> [1.2]
> p = (1::Integer):z
> p
<interactive>:16:18: error:
  * Couldn't match type `Float' with `Integer'
    Expected type: [Integer]
      Actual type: [Float]
  * In the second argument of `(:)', namely `z'
    In the expression: (1 :: Integer) : z
    In an equation for `p': p = (1 :: Integer) : z
<interactive>:17:1: error: Variable not in scope: p
>
```

Why Functional Programming?

- Higher-order programming is the rule rather than the exception.
- This gives rise to library frameworks that allow for partially evaluated functions enabling for a completely different path of code reuse.
 - Curried functions

Haskell – Constant Expressions

The screenshot shows a Haskell REPL session in a web-based environment. On the left, there's a sidebar with file management icons and a dropdown menu. The main area has tabs for 'main.hs' and 'https://intro.lutzhamel.repl.run'. A 'Run ▶' button is at the top right. The repl window shows the following interaction:

```
GHCI, version 8.6.5
*:1234
=> 1234
*:type 1234
1234 :: Num p => p
=> 1234
*:1.2
=> 1.2
*:type 1.2
1.2 :: Fractional p => p
=> 1.2
*:
```

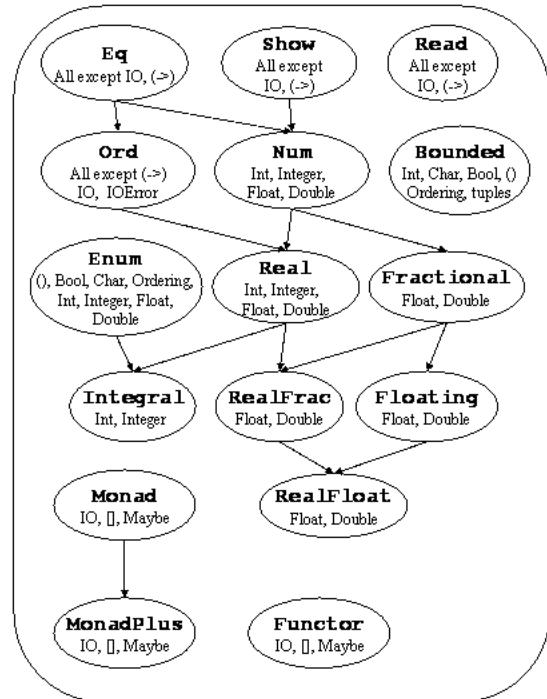
In the center, a red box contains the text: "In Haskell we can ask about the type of any quantity with the ':type' command!"

To the right, a red-bordered box lists "Constants:" with examples:

Constants:	
Integer	1234
Float	1.2
Bool	true/false
String	“Susan”
Char	‘x’

The simplest expression in the Haskell is a constant expression

Haskell Type Classes



- You can think of these type classes in terms of bounds on the type in Rust
 - E.g. the type class **Num** expresses the constraint that a type has the notion of being a number...
 - $\text{Num } p \Rightarrow p$ expresses that constraint

Haskell Operators & Expressions

The screenshot shows a web-based Haskell repl interface on Repl.it. On the left, the file structure shows a main.hs file containing the following code:

```
main.hs
main.hs
GHCi, version 8.6.5
> (3+2)*2
=> 10
> :type (3+2)*2
(3+2)*2 :: Num a => a
=> 10
> 1.2+1
=> 2.2
> :type 1.2+1
1.2+1 :: Fractional a => a
=> 2.2
> "Hello" ++ "World!"
=> "Hello World!"
> :type "Hello" ++ "World!"
"Hello" ++ "World!" :: [Char]
=> "Hello World!"
> 2+1==3
=> True
> :type 2+1==3
2+1==3 :: Bool
=> True
>
```

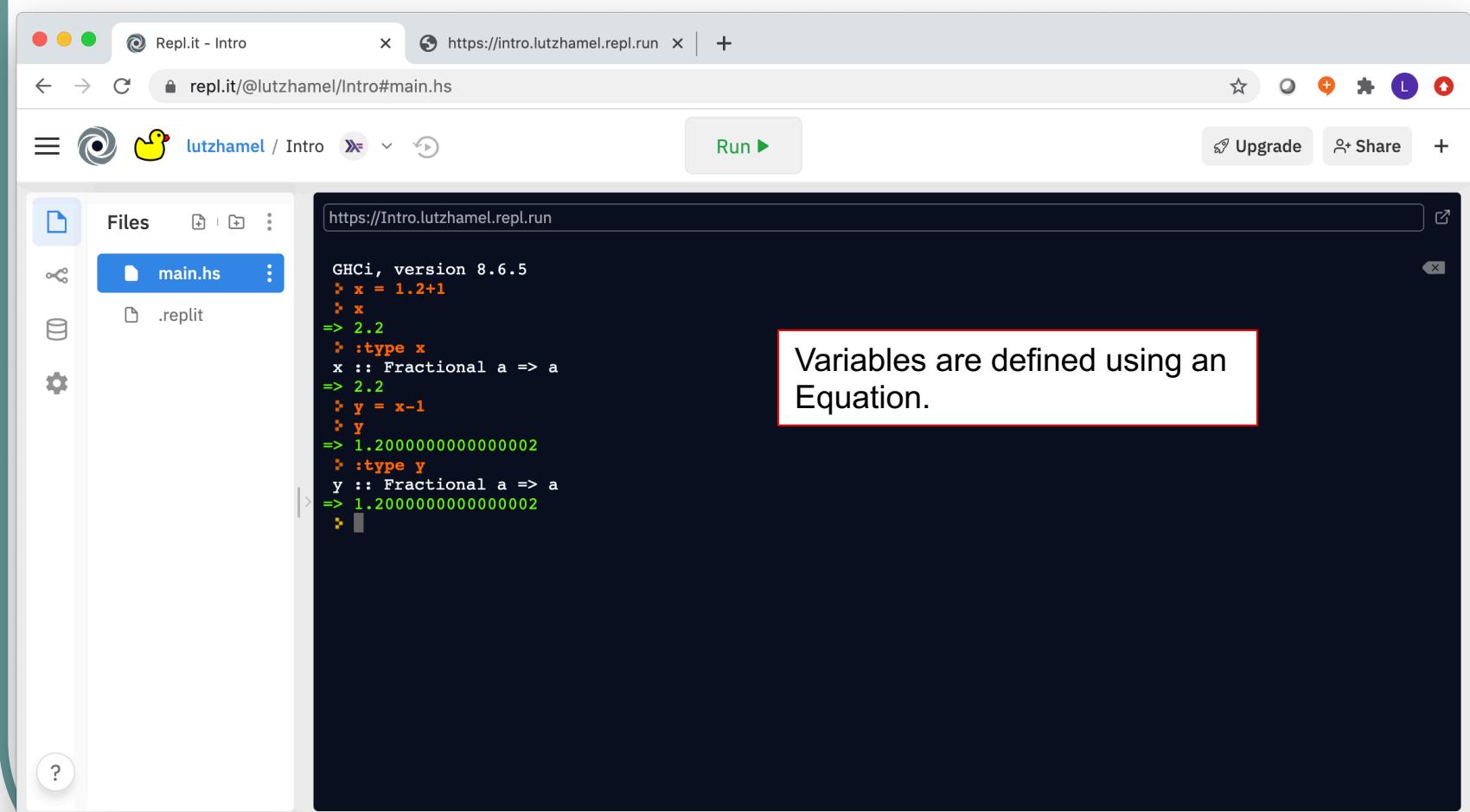
The repl bar at the top has tabs for "Repl.it - Intro" and "https://intro.lutzhamel.repl.run". The URL in the address bar is "repl.it/@lutzhamel/Intro#main.hs". There are buttons for "Run" and "Upgrade". A sidebar on the left includes "Files", ".replit", and settings.

A large table titled "Precedence Level" is overlaid on the right side of the repl area, listing operators by precedence level from 0 to 9. The table structure is as follows:

Precedence Level	Left associative operators	Non-associative Operators	Right associative operators
0			\$, \$!, `seq`
1	>>, >>=		
2			
3			&&
4		==, /=, <, <=, >, >=, 'elem', 'notElem'	
5			: , ++
6	+ , -		
7	*	/, `div`, 'mod', 'rem', 'quot'	
8			^ , ^^ , **
9	!!		.

Operators and expressions work as expected

Variable Definition



The screenshot shows a web-based Haskell repl interface on Repl.it. The left sidebar shows files: main.hs and .replit. The main area shows the REPL history:

```
GHCI, version 8.6.5
> x = 1.2+1
> x
=> 2.2
> :type x
x :: Fractional a => a
=> 2.2
> y = x-1
> y
=> 1.2000000000000002
> :type y
y :: Fractional a => a
=> 1.2000000000000002
>
```

A red box highlights the first two lines of the REPL output:

Variables are defined using an Equation.

Variable Definition

- When we define variables in a program file we can attach type definitions.

The screenshot shows a web-based Haskell development environment on repl.it. On the left, the code editor displays a file named `main.hs` with the following contents:

```
1 x :: Num a => a
2 x = 1
```

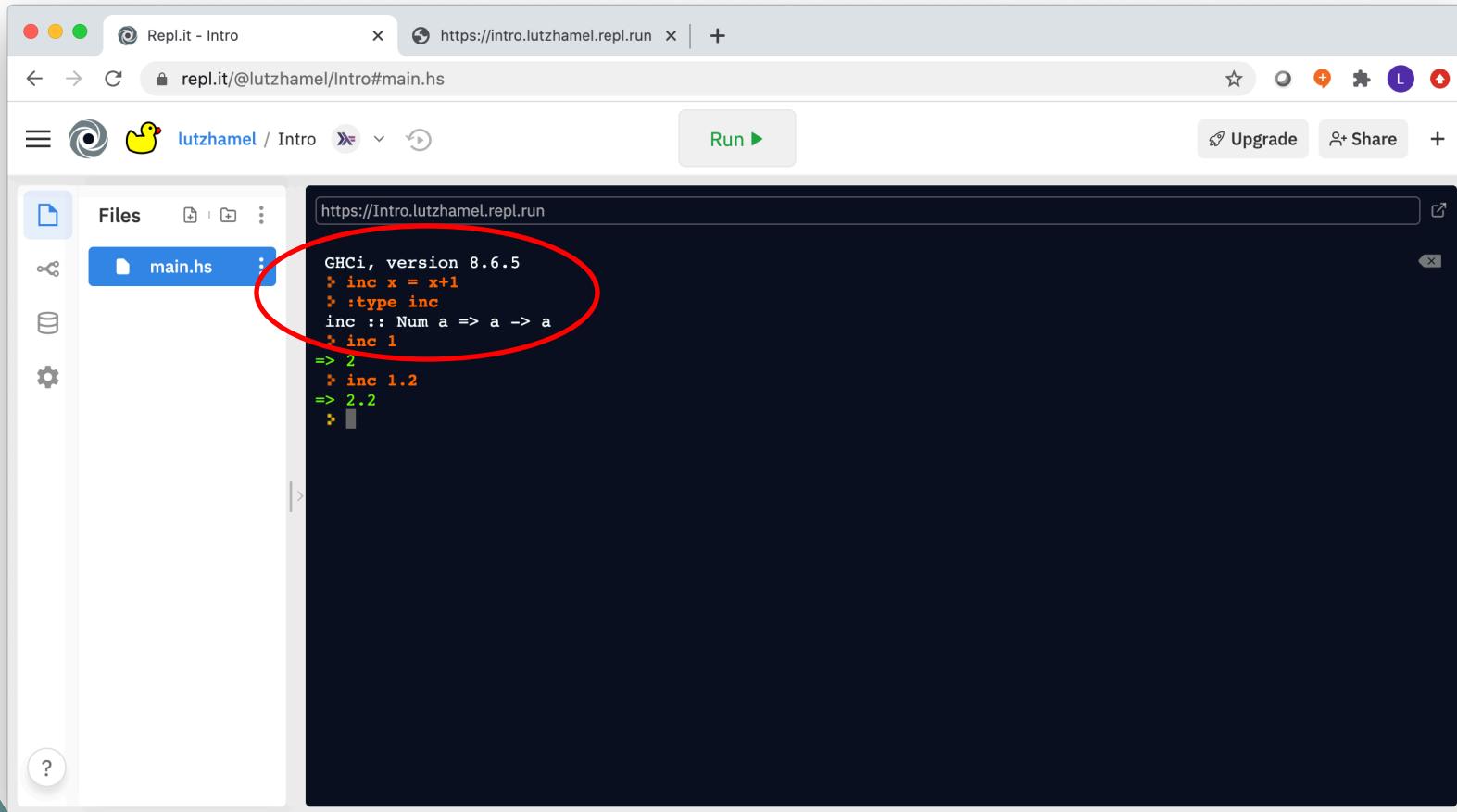
A red oval highlights the first line of code. On the right, the GHCi interpreter window shows the following session:

```
GHCI, version 8.6.5
* :load main
[1 of 1] Compiling Main
interpreted
Ok, one module loaded.
* x
=> 1
* :type x
x :: Num a => a
=> 1
* 
```

A red box contains the explanatory text:

The ':load' command allows you to load program files into the interpreter.

Function Definition



A screenshot of the Repl.it web-based Haskell environment. The interface includes a top navigation bar with tabs for 'Repl.it - Intro' and the current URL 'https://intro.lutzhamel.repl.run'. Below the bar is a header with a profile icon (a yellow duck), the username 'lutzhamel / Intro', a 'Run' button, and upgrade/share/share buttons.

The main area is divided into two panes. The left pane, titled 'Files', shows a single file named 'main.hs' containing the code for the 'inc' function. The right pane is a Haskell REPL window with the URL 'https://Intro.lutzhamel.repl.run' at the top. The REPL session starts with 'GHCi, version 8.6.5' and then defines the 'inc' function:

```
GHCi, version 8.6.5
> inc x = x+1
> :type inc
inc :: Num a => a -> a
> inc 1
=> 2
> inc 1.2
=> 2.2
>
```

A red oval highlights the first four lines of the REPL session, specifically the definition of the 'inc' function and its type signature.

Function Definition

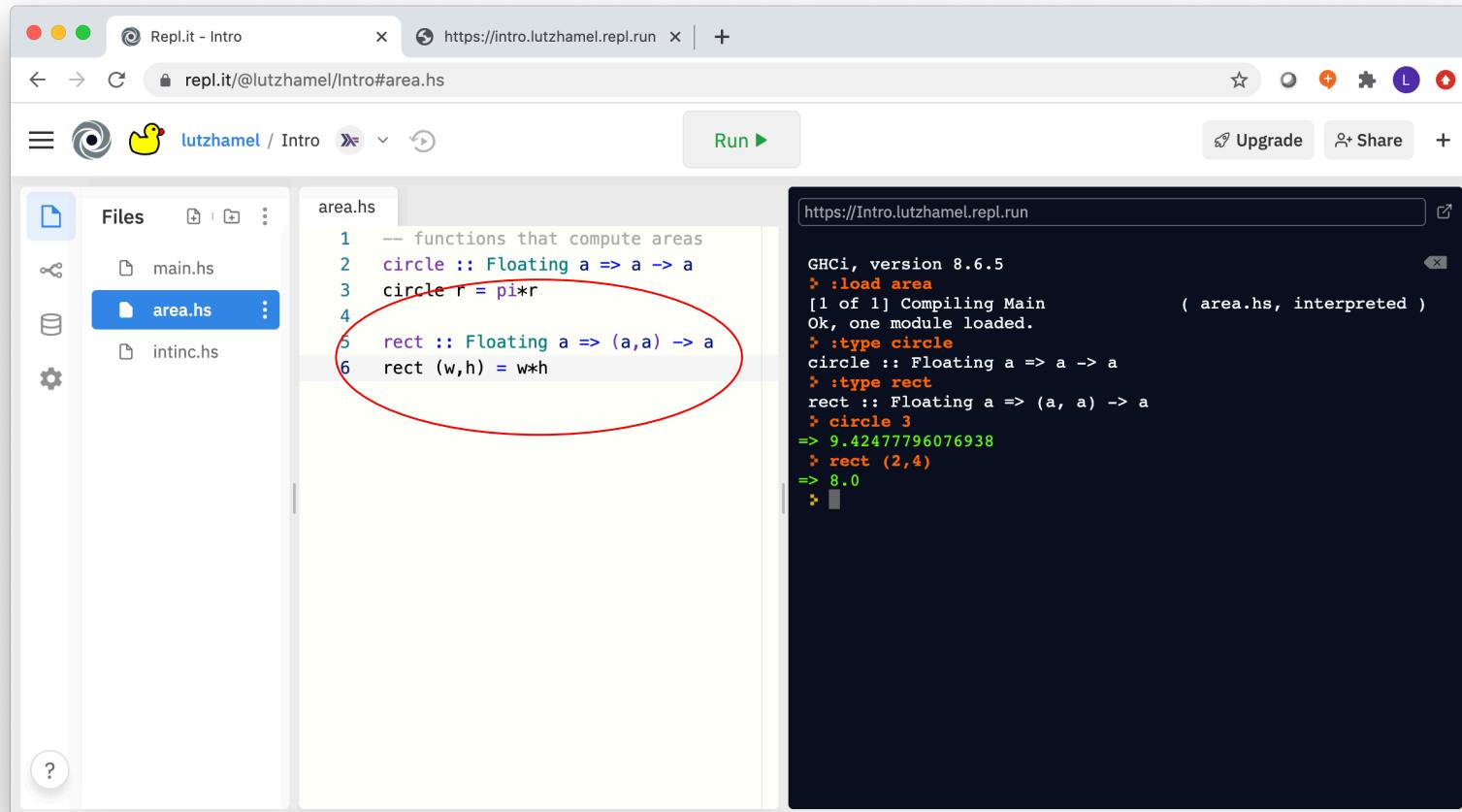
The screenshot shows a web-based Haskell development environment on Repl.it. On the left, the file `main.hs` contains the following code:

```
1 intinc :: Integer -> Integer
2 intinc x = x+1
```

A red oval highlights the first two lines of this code. To the right, the GHCi interpreter window shows the following session:

```
GHCi, version 8.6.5
> :load main
[1 of 1] Compiling Main             ( main.hs, interpreted )
Ok, one module loaded.
> :type intinc
intinc :: Integer -> Integer
> intinc 1
=> 2
> :intinc 1.2
<interactive>:9:8: error:
    * No instance for (Fractional Integer)
      arising from the literal `1.2'
    * In the first argument of `intinc', namely `1.2'
      In the expression: intinc 1.2
      In an equation for `it': it = intinc 1.2
>
```

Function Definition – Multiple Args



The screenshot shows a Repl.it interface with a purple header. The main area displays a Haskell code editor and a GHCi terminal.

Code Editor (area.hs):

```
1  --- functions that compute areas
2  circle :: Floating a => a -> a
3  circle r = pi*r
4
5  rect :: Floating a => (a,a) -> a
6  rect (w,h) = w*h
```

A red oval highlights the definition of the `rect` function.

GHCi Terminal:

```
https://Intro.lutzhamel.repl.run
GHCi, version 8.6.5
> :load area
[1 of 1] Compiling Main           ( area.hs, interpreted )
Ok, one module loaded.
> :type circle
circle :: Floating a => a -> a
> :type rect
rect :: Floating a => (a, a) -> a
> circle 3
=> 9.42477796076938
> rect (2,4)
=> 8.0
>
```

Conditionals & Guards

The screenshot shows a web-based Haskell development environment on Repl.it. On the left, the file tree shows files like main.hs, area.hs, fact.hs, and intinc.hs. A red arrow points from the text "Conditional" to line 8 of the fact.hs code, which contains a traditional if-then-else construct. Another red arrow points from the text "Guard" to line 18, which shows pattern guards for the fact2 function.

fact.hs

```
1 -- functions that compute the factorial
2
3 -- traditional
4 fact1 :: Integral a => a -> a
5 fact1 i =
6   | if i < 0 then
7   |   error("illegal value")
8   | else if i == 0 then
9   |   1
10  | else
11    |   i*fact1(i-1)
12
13 -- using guards:
14 -- f (pattern) | predicate1 = x
15 --                 | predicate2 = y
16 fact2 :: Integral a => a -> a
17 fact2 i
18   | i < 0      = error("illegal value")
19   | i == 0     = 1
20   | otherwise  = i*fact2(i-1)
```

Conditional

Guard

Run ▶

https://Intro.lutzhamel.repl.run

```
GHCi, version 8.6.5
> :load fact
[1 of 1] Compiling Main
( fact.hs,
interpreted )
Ok, one module loaded.
> fact1 3
=> 6
> fact2 3
=> 6
>
```

Lambda Functions

The screenshot shows a web-based Haskell REPL interface on repl.it. The sidebar on the left lists files: main.hs, area.hs, fact.hs (which is selected), and intinc.hs. The main pane displays a GHCi session:

```
GHCi, version 8.6.5
> :type (\x -> x+1)
(\x -> x+1) :: Num a => a -> a
> (\x -> x+1) 1
=> 2
> f = (\y -> y-1)
> :type f
f :: Num a => a -> a
=> 2
> f 1
=> 0
>
```

Annotations with red arrows point to the code: one arrow points from the text "Function definition" to the line starting with "`> :type`", and another arrow points from the text "Function application" to the line starting with "`> (\x -> x+1)`".

Curried Functions

- A curried function is a particular way to write multi argument functions using lambda functions that allows for **partial evaluation**.
- Named after the American logician Haskell Curry.



Curried Functions

- Let's play a trick: assume we have a function that has two arguments like our 'rect' function that computes the area of a rectangle:

$$\text{rect } (w, h) = w * h$$

- This function takes a pair of values and computes the area as the product of those two values.

Curried Functions

- Now, the trick is that instead of handing the function a pair of values we hand it one value at a time.
- This means we have to rewrite the function like so:

```
rectch w = (\h -> w*h)
```

- Note that rectch given the first argument will return a function that accepts the second argument and then finishes the computation
- Haskell allows you to write this kind of function as:

```
rectc w h = w*h
```

Curried Functions

The screenshot shows a web-based Haskell development environment on Repl.it. On the left, the file tree shows files like main.hs, area.hs, fact.hs, intinc.hs, and rect.hs (which is selected). The code editor contains the following Haskell code:

```
rect (w,h) = w*h
rectch w = (\h -> w*h)
rectc w h = w*h
```

To the right is the GHCi terminal window, which has been circled in red. It shows the execution of the code and some type queries:

```
GHCi, version 8.6.5
* :load rect
[1 of 1] Compiling Main
Ok, one module loaded.
* :type rect
rect :: Num a => (a, a) -> a
* :type rectch
rectch :: Num a => a -> a -> a
* :type rectc
rectc :: Num a => a -> a -> a
* rect (3,2)
=> 6
* f = rectch 3
* f 2
=> 6
* g = rectc 3
* g 2
=> 6
* :type f
f :: Num a => a -> a
=> 6
* :type g
g :: Num a => a -> a
=> 6
*
```

Curried Functions

- Notice the drastic type difference between curried and non-curried functions
- All functions and operators in Haskell are curried
- This gives us an interesting way to use built-in operators

Curried Functions

The screenshot shows a web-based Haskell development environment on Repl.it. The interface includes a file browser on the left and a GHCi terminal on the right.

File Browser:

- Files:
 - main.hs
 - area.hs
 - fact.hs
 - intinc.hs
 - rect.hs (selected)
- ...

GHCi Terminal:

```
https://Intro.lutzhamel.repl.run
GHCi, version 8.6.5
* :type (*)
(*) :: Num a => a -> a -> a
* unit = (*)
* :type unit
unit :: Num a => a -> a
* unit 3
=> 3
* dbl = (*)
* dbl 3
=> 6
* :type dbl
dbl :: Num a => a -> a
=> 6
* 
```

Curried Functions

The screenshot shows a web-based Haskell development environment on Repl.it. On the left, a sidebar displays a file tree with files like main.hs, area.hs, fact.hs, intinc.hs, rect.hs, and curry.hs. The curry.hs file is currently selected and shown in the main code editor area. The code defines three curried functions: dbl, inc, and thresh.

```
curry.hs
1 -- Here we show how you can use the curried
2 -- library function to build your own
3
4 -- define a doubling function using a
5 -- partially evaluated mult operator
6 dbl = (*) 2
7
8 -- define an inc function using a partially
9 -- evaluated addition operator
10 inc = (+) 1
11
12 -- define a threshold function at 2 using a
13 -- partially evaluated min operator
14 thresh = min 2
```

To the right of the code editor is a terminal window showing interactions with the GHCi interpreter. The user loads the curry.hs module and then defines three functions: dbl, inc, and thresh. The terminal also shows the types of these functions and some additional GHCi commands.

```
https://Intro.lutzhamel.repl.run
GHCi, version 8.6.5
> :load curry
[1 of 1] Compiling Main
interpreted
Ok, one module loaded.
> dbl 3
=> 6
> inc 1
=> 2
> thresh 1
=> 1
> thresh 3
=> 2
> thresh 4
=> 2
> :type (*)
(*) :: Num a => a -> a -> a
=> 2
> :type (+)
(+) :: Num a => a -> a -> a
=> 2
> :type min
min :: Ord a => a -> a -> a
=> 2
>
```