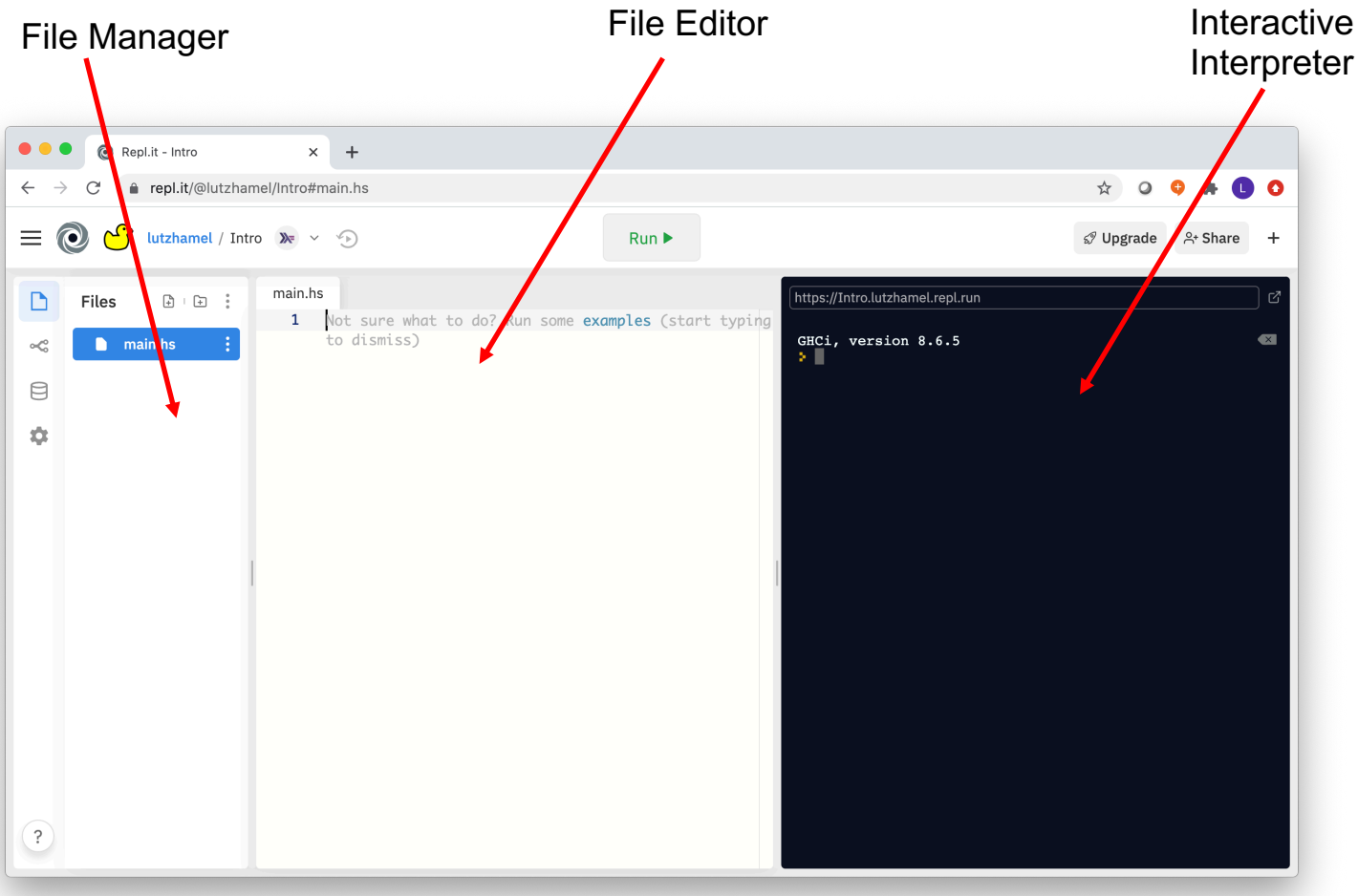
Haskell

- 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.

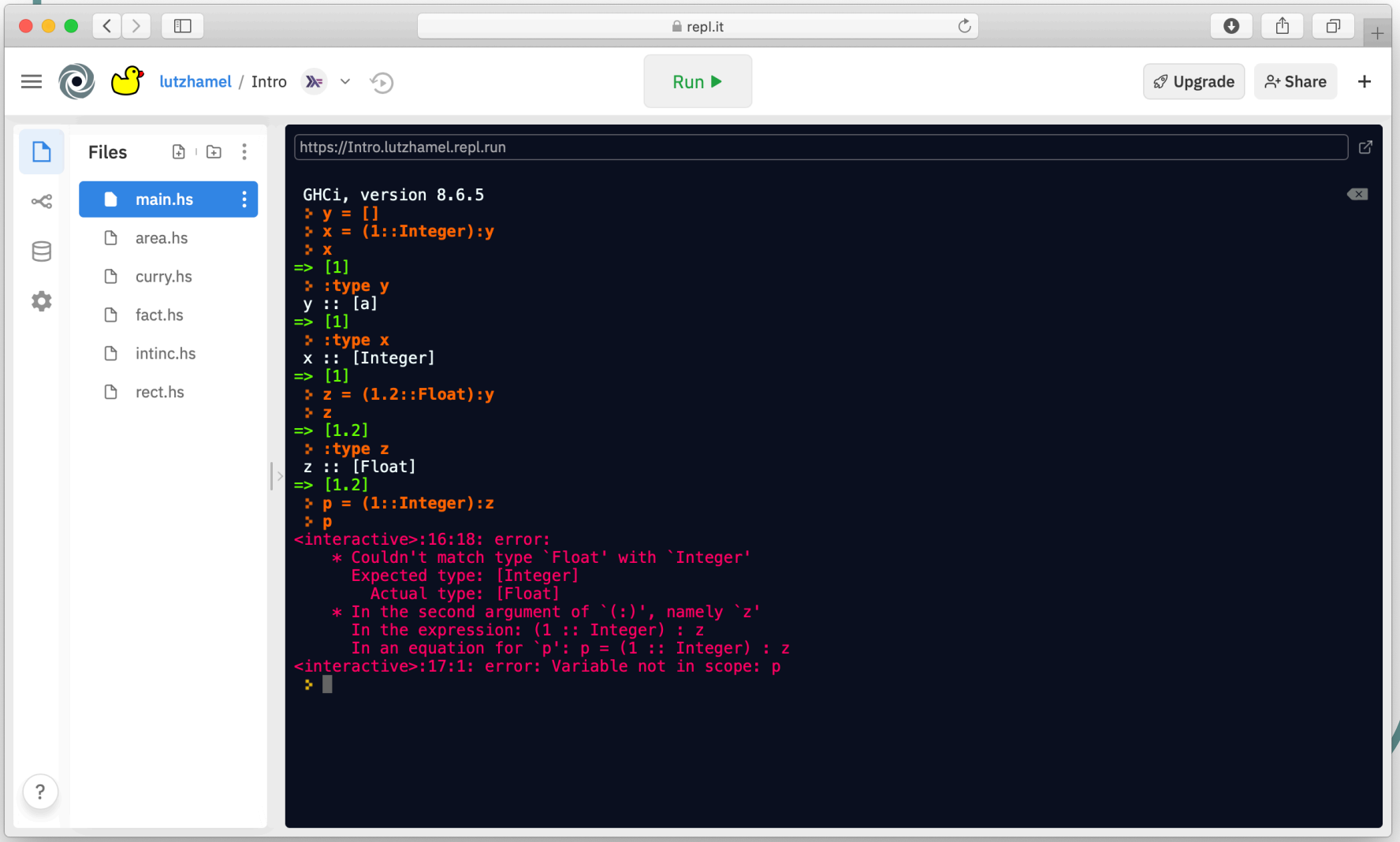
Why Functional Programming?

- 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 <T> 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

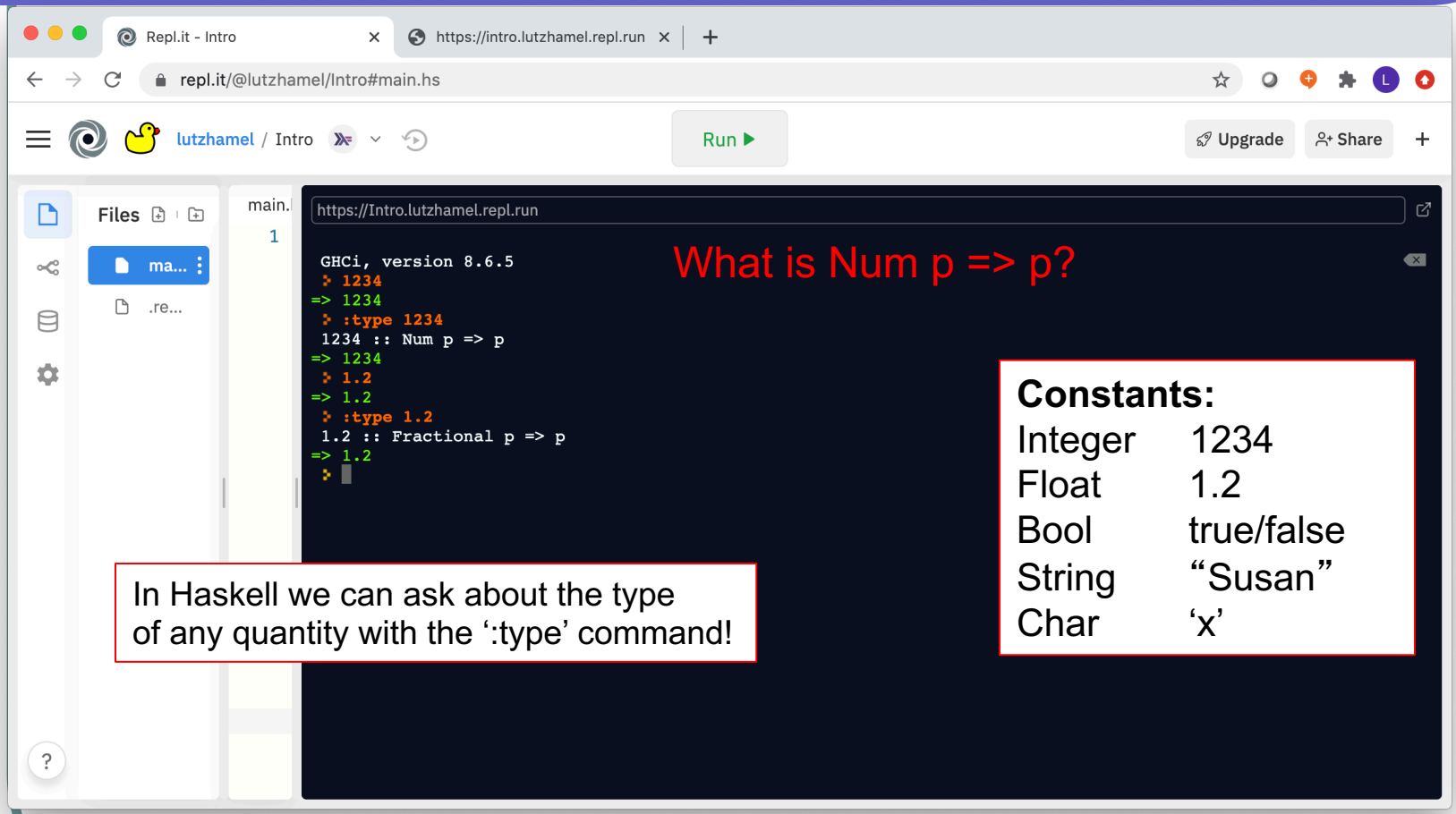
Example Session



Why Functional Programming?

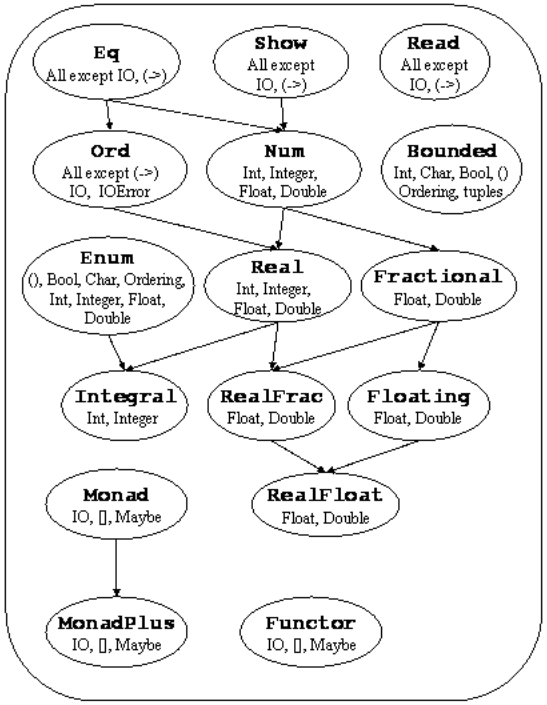
- Higher-order programming is the rule rather than the exception.
- This gives rise to library frameworks that allow for partially evaluated functions enabling for a completely different path of code reuse.
  - Curried functions

Haskell – Constant Exressions



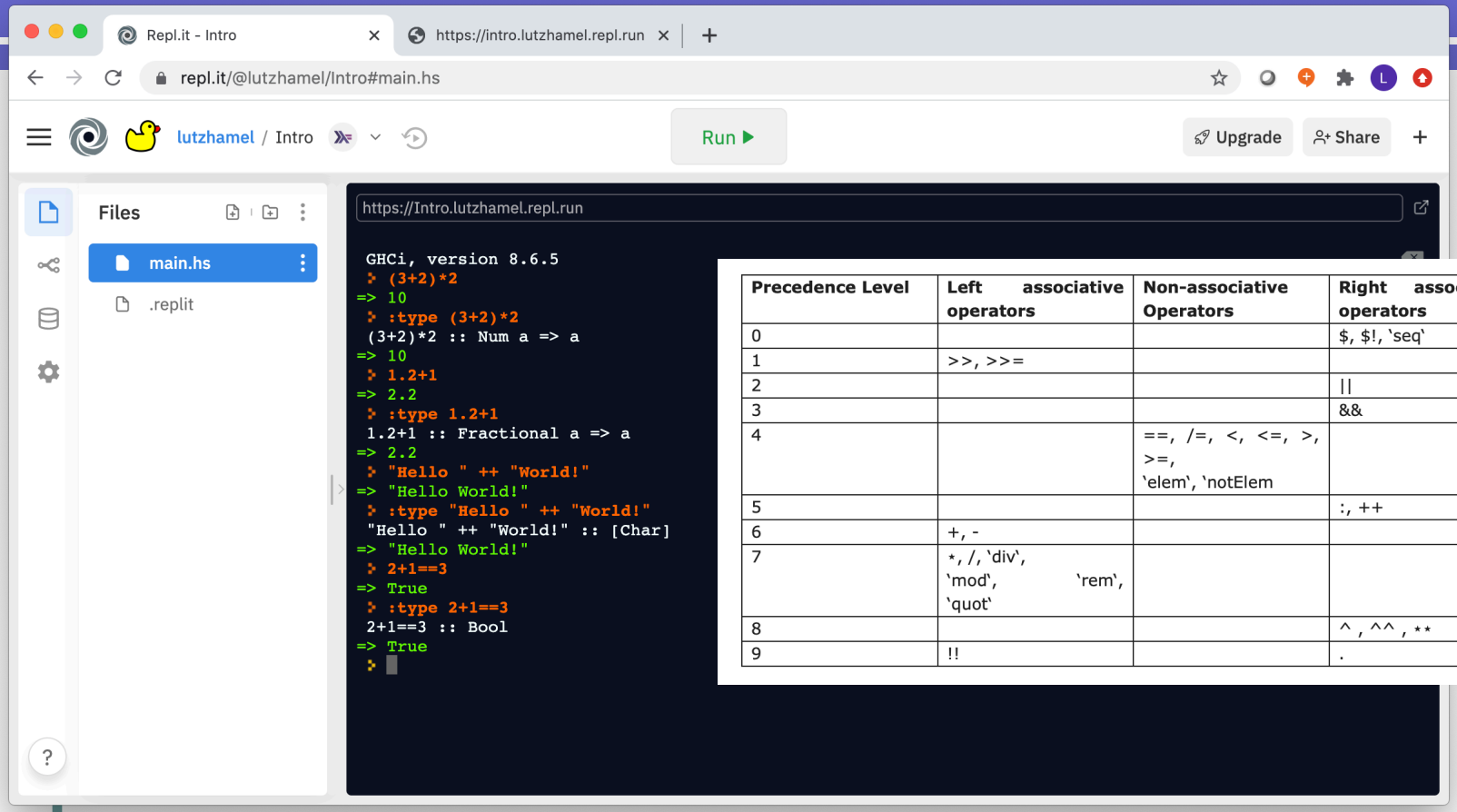
The simplest expression in the Haskell is a constant expression

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…
- `Num p => p` expresses that constraint

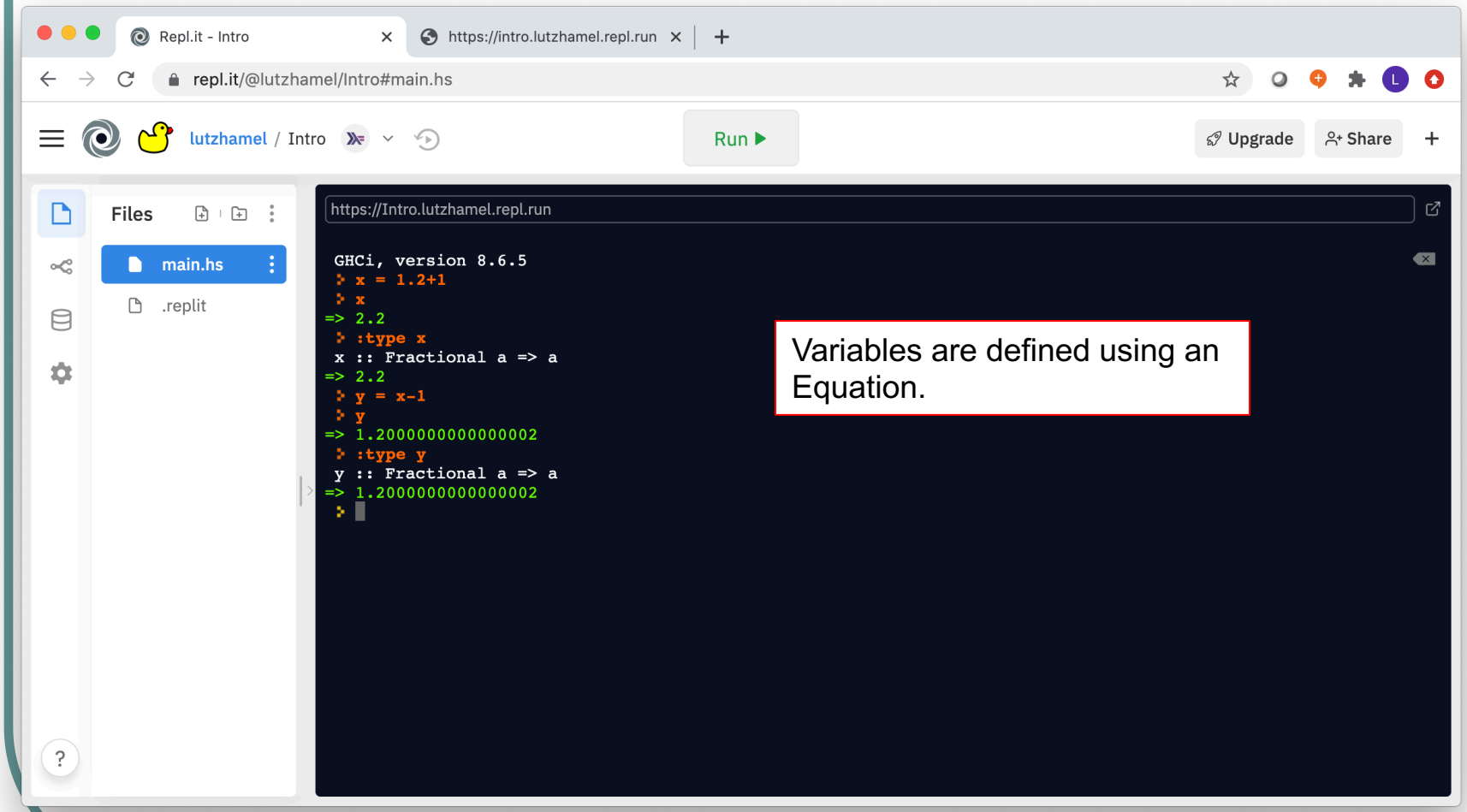
Haskell Operators & Expressions



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	!!		.

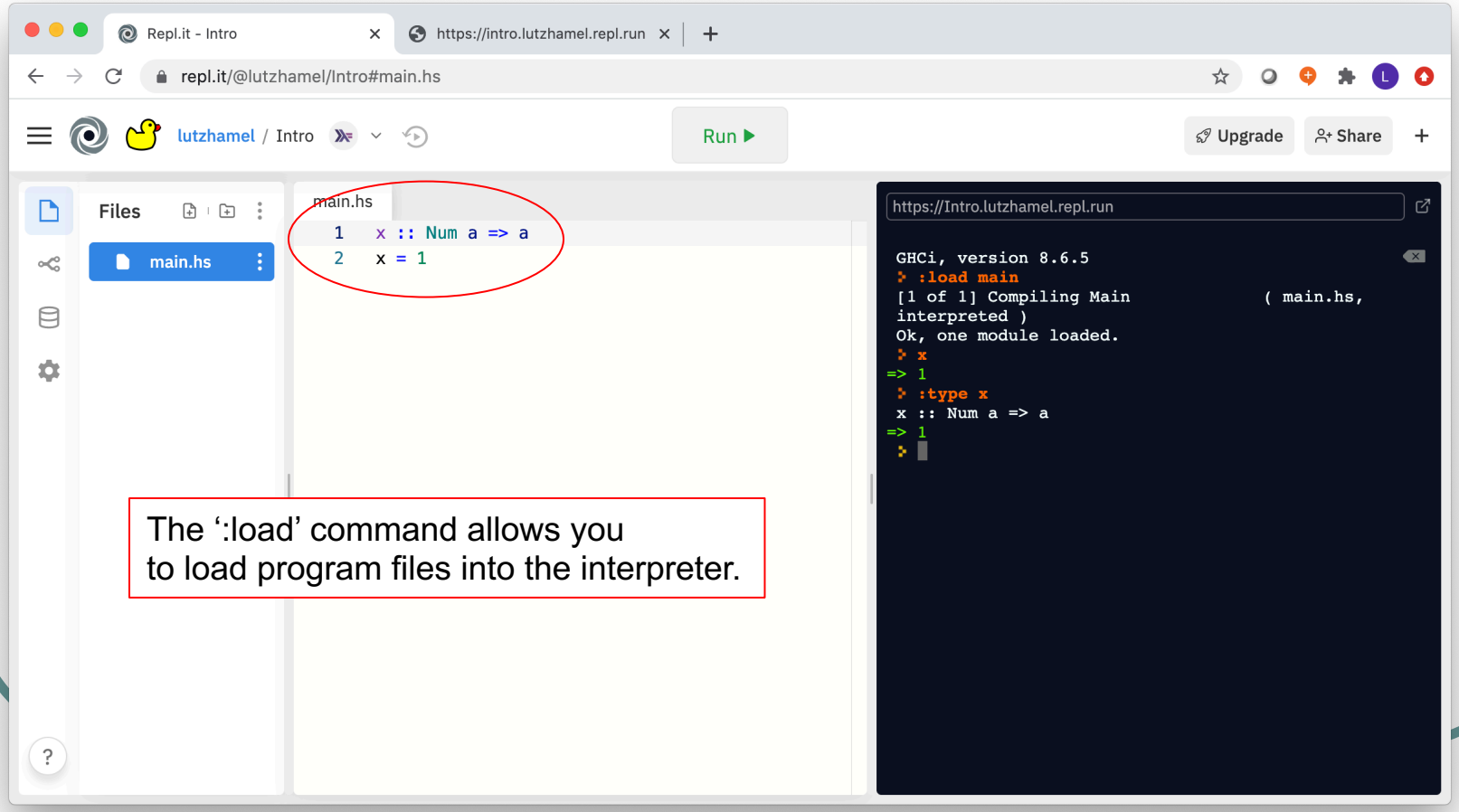
Operators and expressions work as expected

Variable Definition

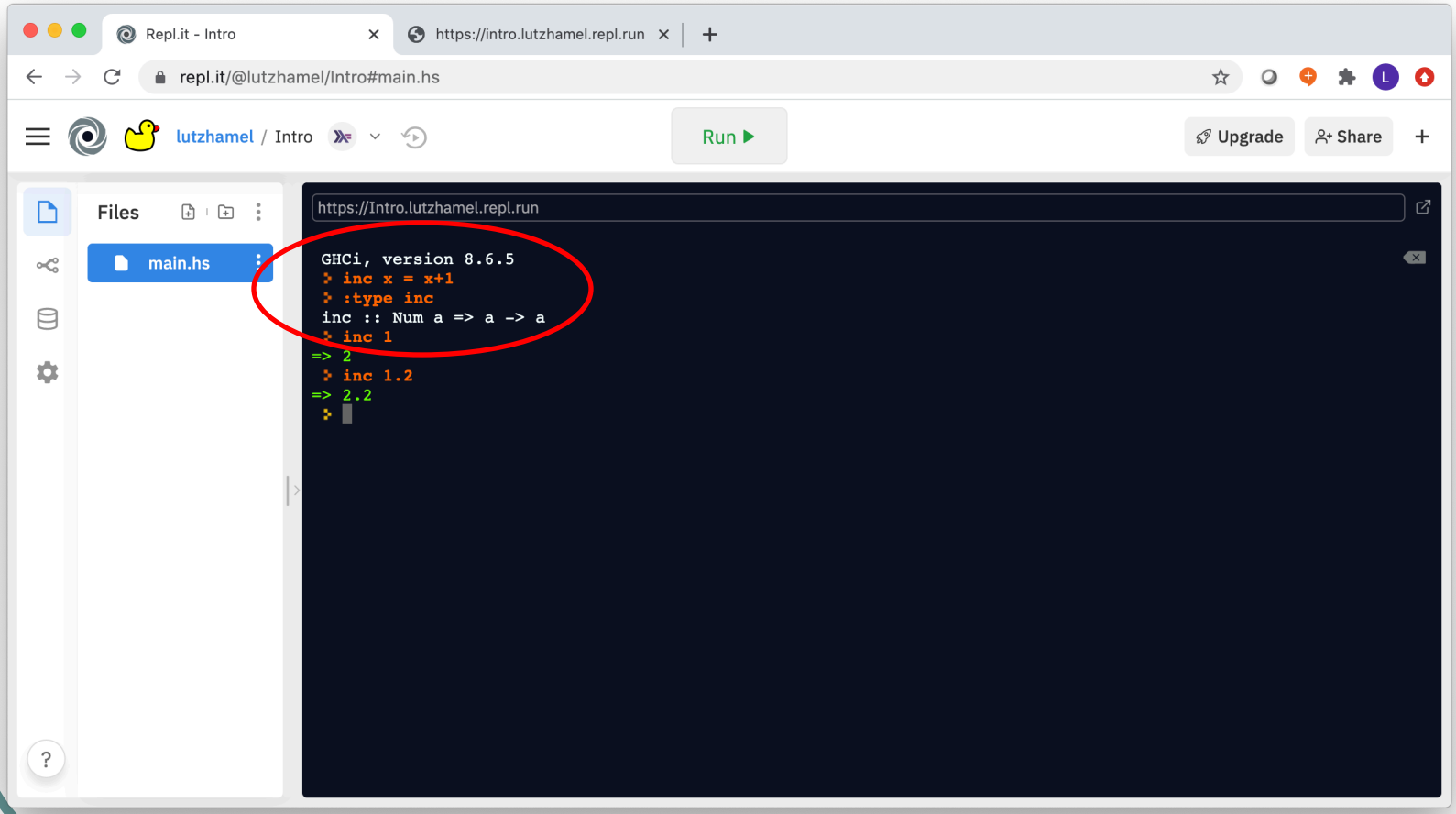


Variable Definition

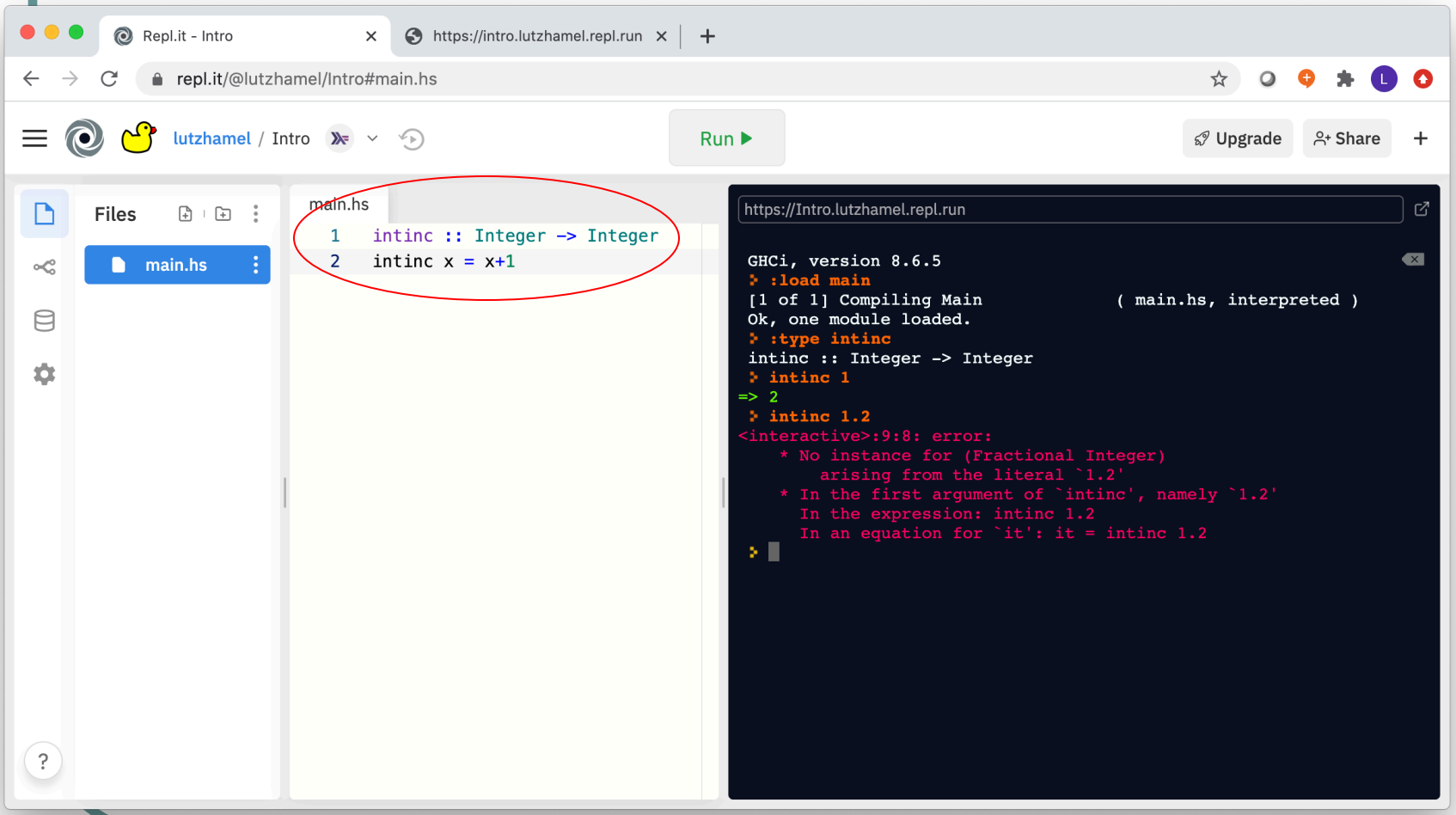
- When we define variables in a program file we can attach type definitions.



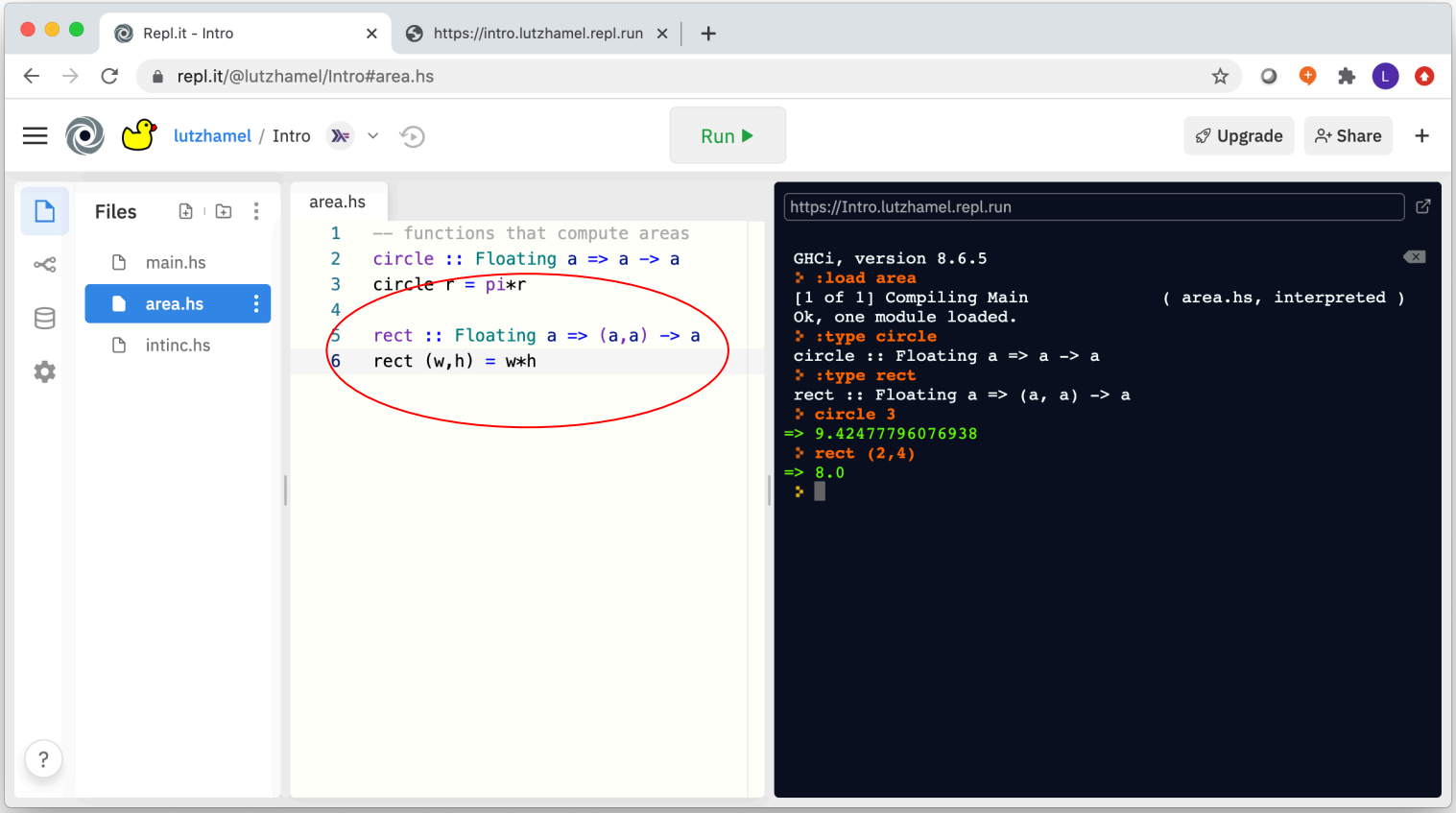
Function Definition



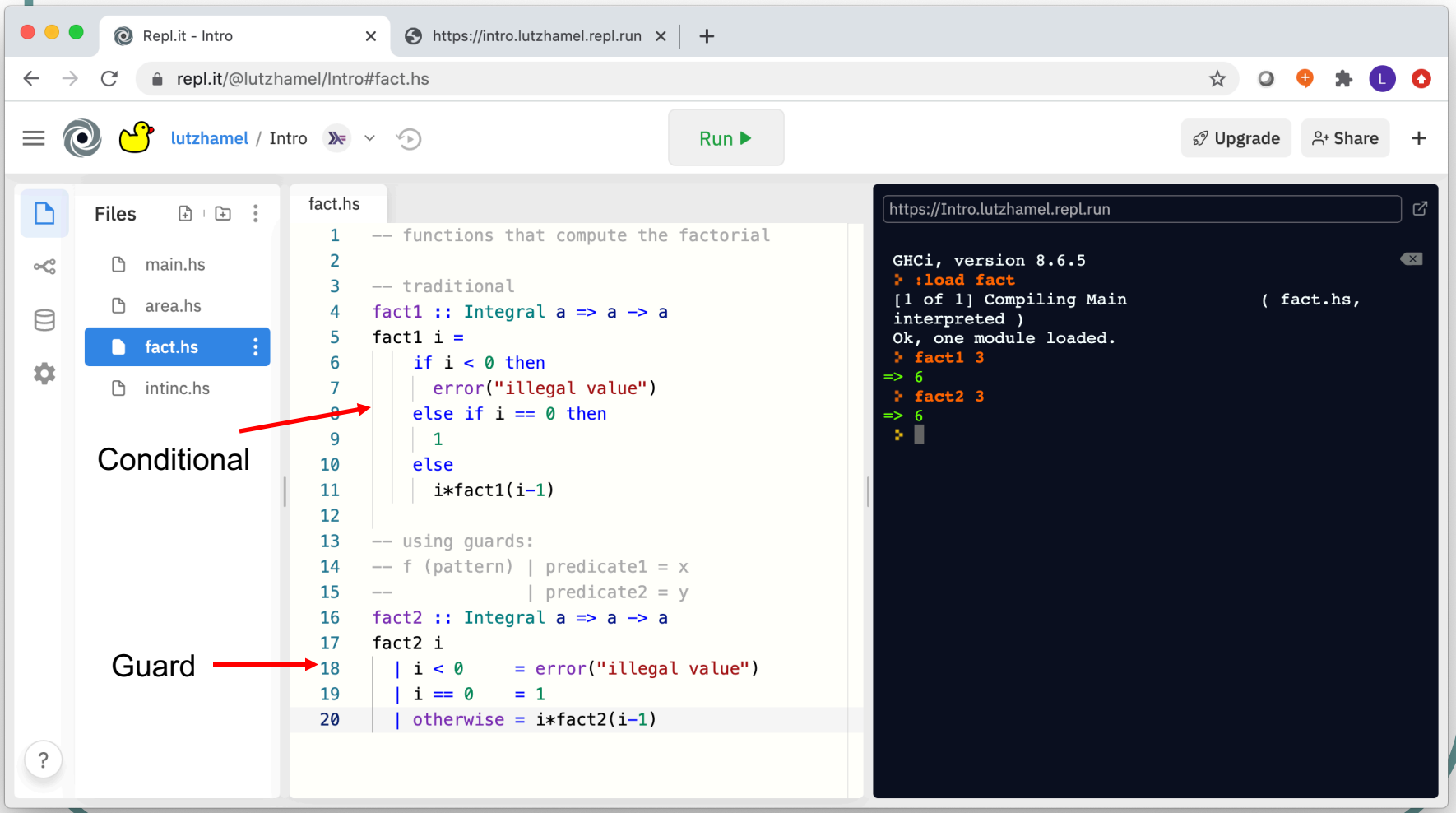
Function Definition



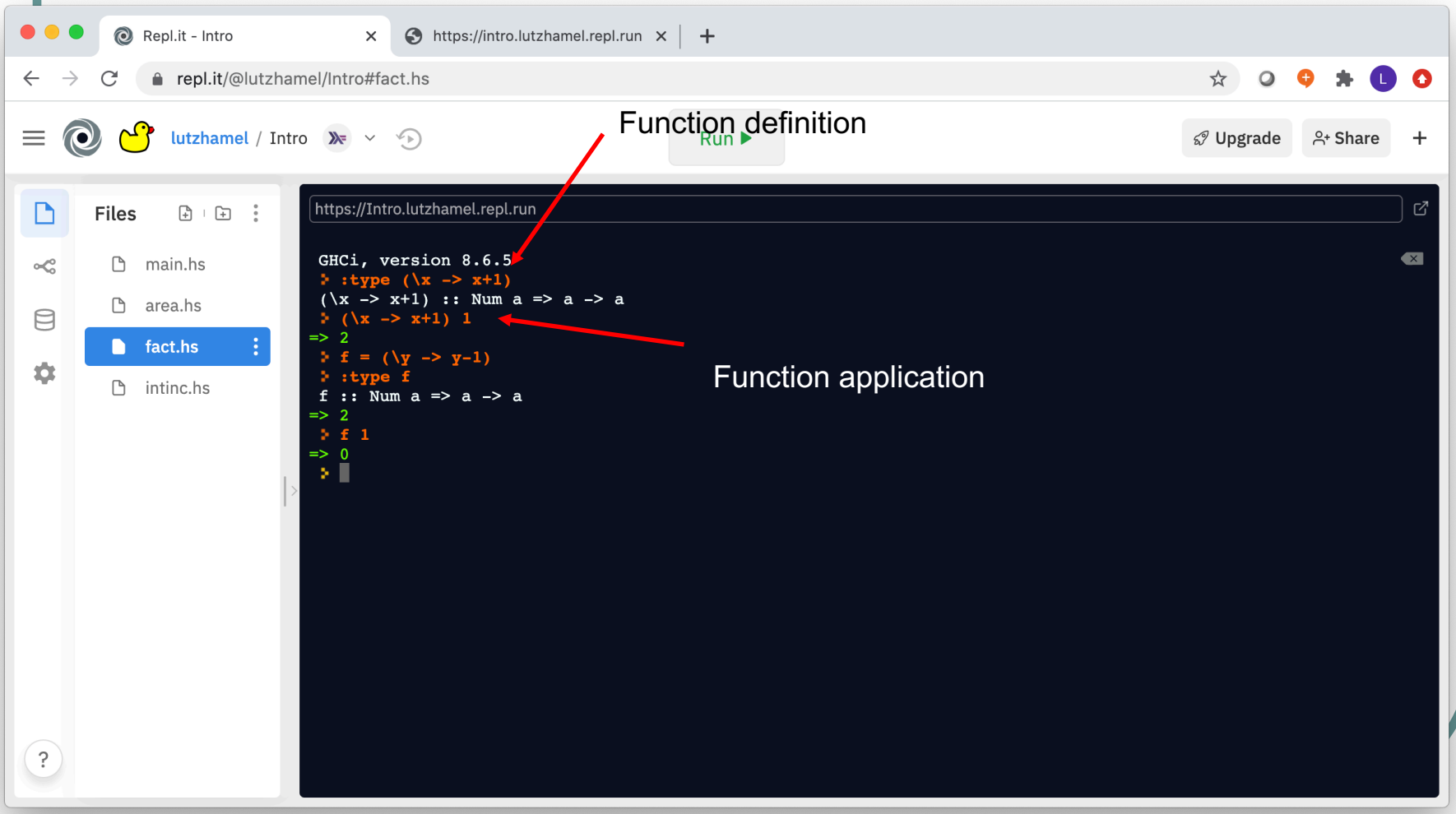
Function Definition – Multiple Args



Conditionals & Guards



Lambda Functions



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:

```
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

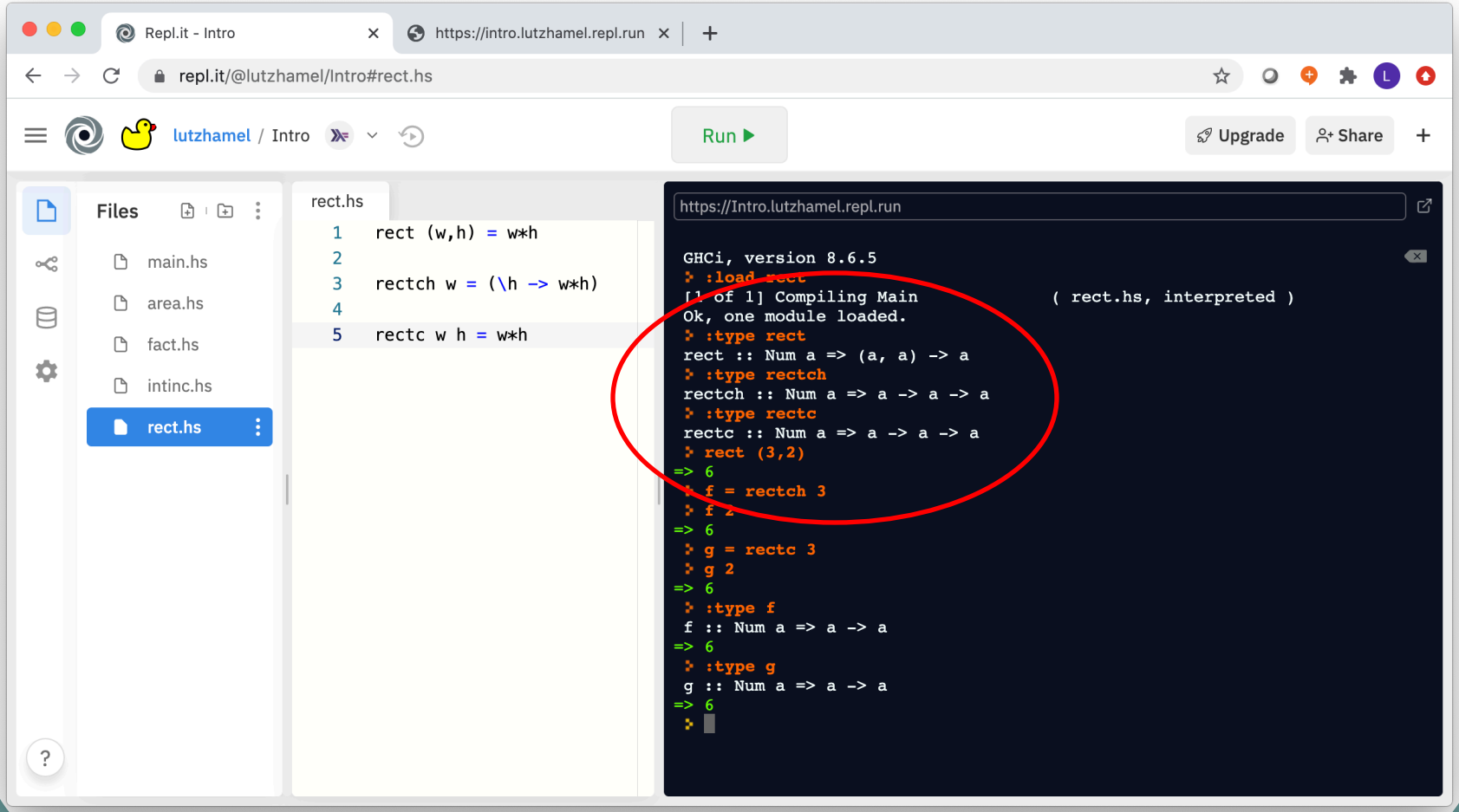
- Now, the trick is that instead of handing the function a pair of values we hand it one value at a time.
- This means we have to rewrite the function like so:

```
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
```

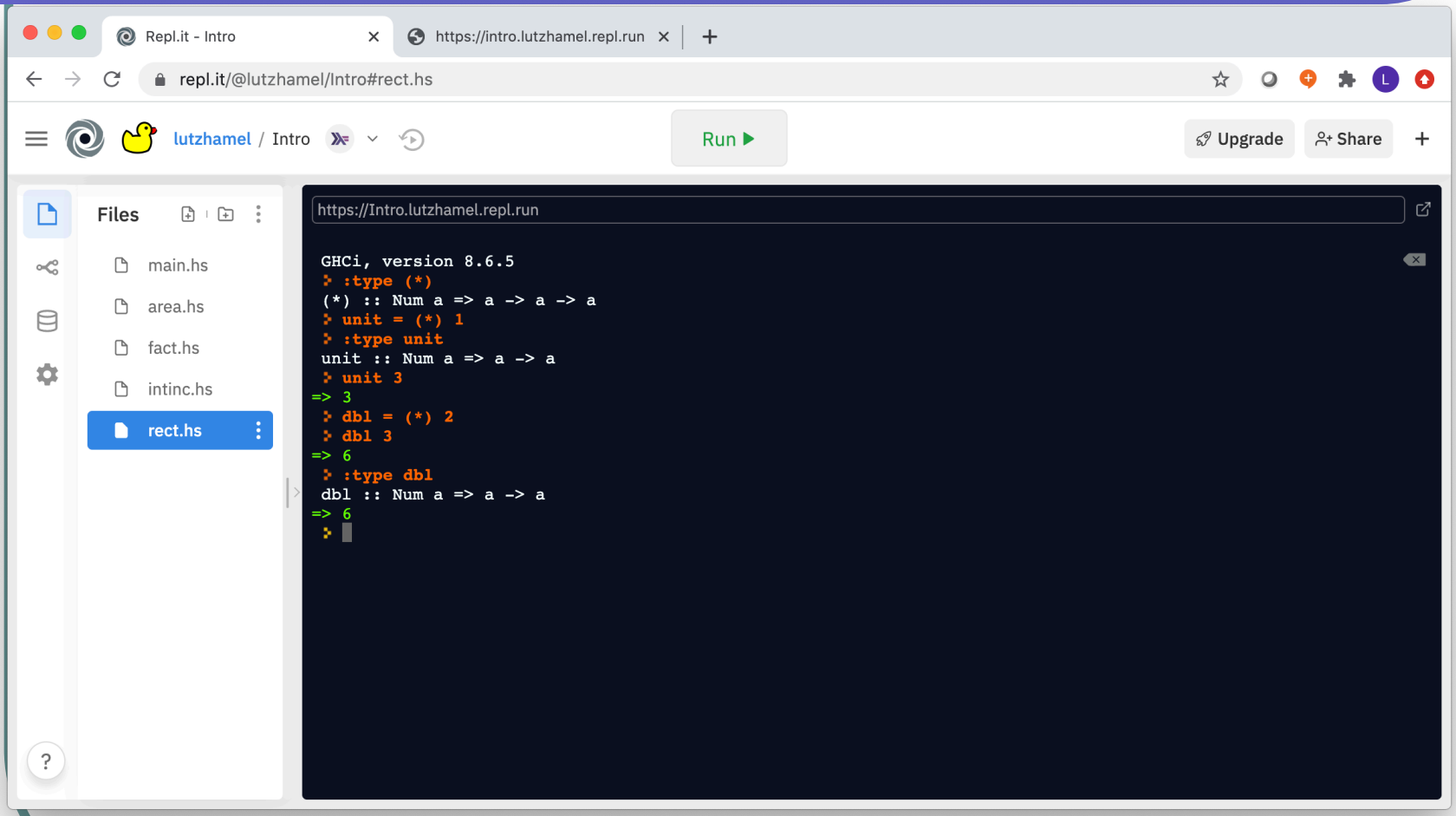
Curried Functions



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



Curried Functions