Lecture 1

Typing: use `::` operator, I think

- `Integer`: arbitrarily large integers
- `Int`: machine dependent int size.

```
42 :: Integer      -- 42
42 :: Double       -- 42.0

let x = 42 :: Int
:t x               -- x :: Int
3 + 24 :: Integer  -- 27
```

- Can run an empty file to use the haskell repl

Can also have conditional expressions (expressions have a value, statements are an action

```
:t (if 3 > 2 then (42 :: Int) else 84)  -- Int

-- Unit
()

-- "Punning". Left () is the value (unit value), the one on the right is a type.
-- The type of unit is the same as the value of unit
() :: ()
```

- abstraction: worrying about the details that matter, and not about the ones that dont (according to jose)
  - abstractions that reveal details that you don't care about are leaky abstractions
  - function is one of the most fundamental abstractions. you hide the details that the user cares about, and allow them to provide the only thing they care about (parameters)
- Jose enforced rule for Haskell: we need to give a type to any top level declarations.

```
-- Guess there's no keyword for functions?
funk x y = x + (y * 21)

-- Idiomatic way to declare types. (All professionals do this)
x :: Integer
x = 42

funk :: Int -> Int -> Int
funk x y = x + (y * 21)

-- Sections
let x = 42 :: Integer
:t (x +)
-- (x +) :: Int -> Int
-- This is called a section, partial application.

-- Another example of partially applying operators.
myPlus :: Int -> Int -> Int
myPlus = (+)
-- The following does not work, needs parentheses
myPlus = +

-- Printing
putStr "Hello"
putStrLn "Hello"

:t putStrLn
-- putStrLn :: String -> IO ()
let x = putStrLn "hello" in (42 :: Int)
-- Results in 42? I think
{-
This did not print to the screen because of 1) laziness and 2)
putStrLn returns an IO Action.
IO Actions must be run to actually do something.
-}
```

```
-- Weird stuff with IO Actions
x :: Integer
x = 42

fun: Int -> Int -> Int
fun x y = let res = fun2 "test" in x + (y * 21)

fun2 :: String -> IO ()
fun2 str = putStrLn str


-- can run IO actions with any function of the type IO ()
main :: IO ()
main = fun2 "test"
```

```
{-
if a function does not perform any IO actions, it is pure.
If the function does not say anything about doing IO actions,
then it should not do any IO actions.
-}

-- to run something like main, you would do :main (there's also :e)
-- to run it like an executable
```

```
let x = 42 :: Int
[x, 3, 4]  -- [42, 3, 4]
let xs = [putStr "Hello"]  -- prints nothing
:t xs   -- xs :: [IO ()]  (means that it's a list of IO Actions)
-- BTW, lists are homogeneous
```

```
-- Sequencing Actions/Operations
main :: IO ()
main = do   -- when we use do expressions for ACTIONS, it acts like sequencing
  putStr "Enter your name"
  res <- getLine  -- put result of getLine in res
  putStr ("Hello " ++ res)
```