Lecture 33:
• Quiz 8
• Haskell intro (as time)

Announcements:
• HW-6 out, due Mon
**Using ghci interpreter (basics)**

GHC can be downloaded from [https://www.haskell.org/](https://www.haskell.org/)

- I'm using version 8.8.4 in examples ...
- GHC has both a REPL (`ghci`) and a Compiler (`ghc`)

Running the interpreter from the command line:

```
$ ghci
GHCi, version 8.8.4: ...
Prelude>
```

Simple arithmetic

```
Prelude> 2 + 2
4

Prelude> 31337 * 100
3133700

Prelude> 7 / 2
3.5
```

Can call operators using *infix* (above) notation and as functions

```
Prelude> (+) 2 2
4
```

- In fact "+" is just a function!
- Note that you shouldn’t write arithmetic expressions this way though!
Creating “local” bindings in the interpreter:

Prelude> let x = 4

Prelude> let y = 3

Prelude> x
4
Prelude> y
3
Prelude> x + y
7

Prelude> let z = x + y
Prelude> z
7

Prelude> let x = z -- x shadows previous x
Prelude> x

Prelude> let y = y + x -- watch out! (rhs y is lhs y)
Prelude> y
... infinite expansion ...

Intro to Haskell Types

Primitive data types:

- **Bool** ... True and False
- **Int** (bounded) and **Integer** (unbounded)
- **Double** ... also Float, but Double is default
- **Char** ... e.g., 'a', 'b', etc
- **String** ... just lists of characters

Lists and Tuples:

- Lists in Haskell are "homogeneous": All values of the same type
- **Examples:** [], [True, False], [3, 4, 5], ['a', 'b'] (same as "ab")
- Tuples are fixed size and "heterogeneous"
- **Example:** (True, 1), ("ab", 3.14), (1, "foo", False)

Type classes:

- Similar to an "interface", define a set of functions signatures
- Members (primitive and user-defined types) "implement" the typeclass
- Can think of typeclasses as grouping "sets" of types (based on behavior)
- **Note:** this is a simplification ...
Some examples of type classes:

- **Eq** ... types with equality testing
- **Ord** ... types with ordering (e.g., <, >, etc.)
- **Show** ... types with the `show` function (think `toString()`)
- **Read** ... types with the `read` function (think `parseInt()`)
- **Enum** ... types whose values are enumerable (like 0, 1, 2, …)
- **Num** ... basic numeric types (supporting +, *, `negate`, etc)
- **Integral** ... whole number types (e.g., `Int`, `Integer`)
- **Bounded**, **Floating**, **Fractional**, **Real**, etc

Typeclasses and “class constraints”

- For this type …

```
Prelude> :type 4
4 :: Num p => p
```

- Everything before the `=>` is a *class constraint*
- Here `p` stands for “any type”
- Constrains type `p` (the value’s type) to be a member of the `Num` typeclass
- To see the “default” type (of the value):

```
Prelude> :type +d 4
4 :: Integer
```

- For info on the typeclass

```
Prelude> :info Num
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
negate :: a -> a
abs :: a -> a
signum :: a -> a
fromInteger :: Integer -> a
...
```