INTRODUCTION TO

FUNCTIONAL PROGRAMMING

OUTLINE

- Midterm
- Property-Based Testing

TEST

- First section: use ghci
- Second section: use :t in ghci
- Third section: use ghci and hoogle
- Fourth section: think

TEST: LAWFUL INSTANCE OF A FUNCTOR?

```haskell
instance Functor (Either a) where
  fmap _ (Left x) = Left x
  fmap f (Right y) = Right (f y)

-- Identity
fmap id == id

-- Composition
fmap (f . g) == fmap f . fmap g
```

```
-- Identity
forall x:
  (fmap id) (Left x) == Left x == id (Left x)
forall y:
  (fmap id) (Right y) == Right (id y) ==
     == Right y == id (Right y)

-- Composition
forall x, f, g:
  (fmap (f . g)) (Left x) == Left x
  (fmap f . fmap g) (Left x) ==
     == fmap f (fmap g (Left x)) ==
     == fmap f (Left x) == Left x

forall y, f, g:
  (fmap (f . g)) (Right y) ==
     == Right ((f . g) y) == Right (f (g y))
  (fmap f . fmap g) (Right y) ==
     == fmap f (fmap g (Right y)) ==
     == fmap f (Right (g y)) == Right (f (g y))
```

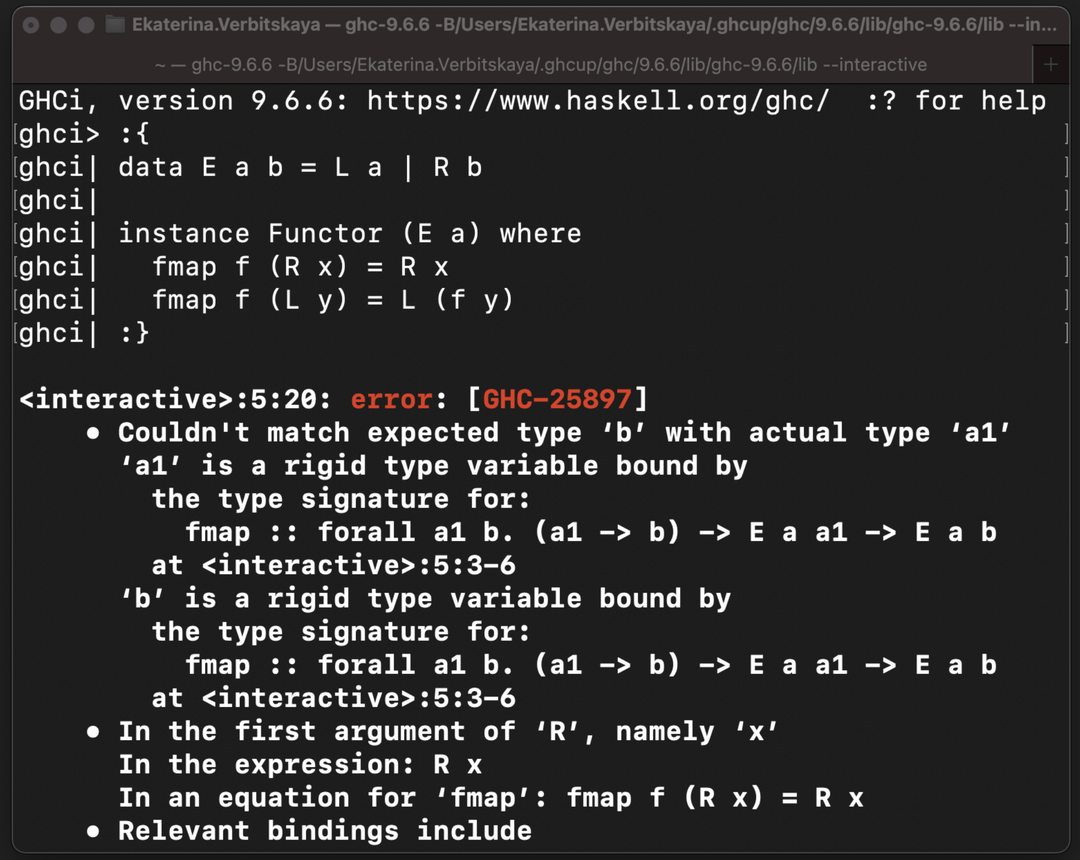
TEST: LAWFUL INSTANCE OF A FUNCTOR?

```haskell
instance Functor (Either a) where
  fmap _ (Right x) = Right x
  fmap f (Left y) = Left (f y)

fmap :: (b → c) → Either a b → Either a c
```

It doesn't typecheck, therefore

it is not a lawful instance of a functor



TEST: LAWFUL INSTANCE OF A FUNCTOR?

Is it possible to provide a lawful instance of Functor for the following data type implementing a tree? Explain your answer.

```
data SetTree a
  = Leaf a
  | Node a (Set (SetTree a))
```

2 possible answers:

1. No, `Set` imposes the `Ord` constraint on `a` which is incompatible with `Functor`.
2. No, `Functor's fmap` is supposed to preserve the shape of the container, but `fmap const` over the set of subtrees would change the shape of it by collapsing them into 1 subtree.

OUTLINE

- Midterm
- Property-Based Testing

UNIT TESTS

- Testing a sort function
  - Is the result ordered?
  - Do we lose any elements?
  - Does the function crash?

UNIT TESTS: WHEN TO STOP?

- Sunny day scenario
- Rainy day scenario
- Corner cases
- Good coverage
- …

UNIT TESTS: DOES THE PROGRAM WORK?

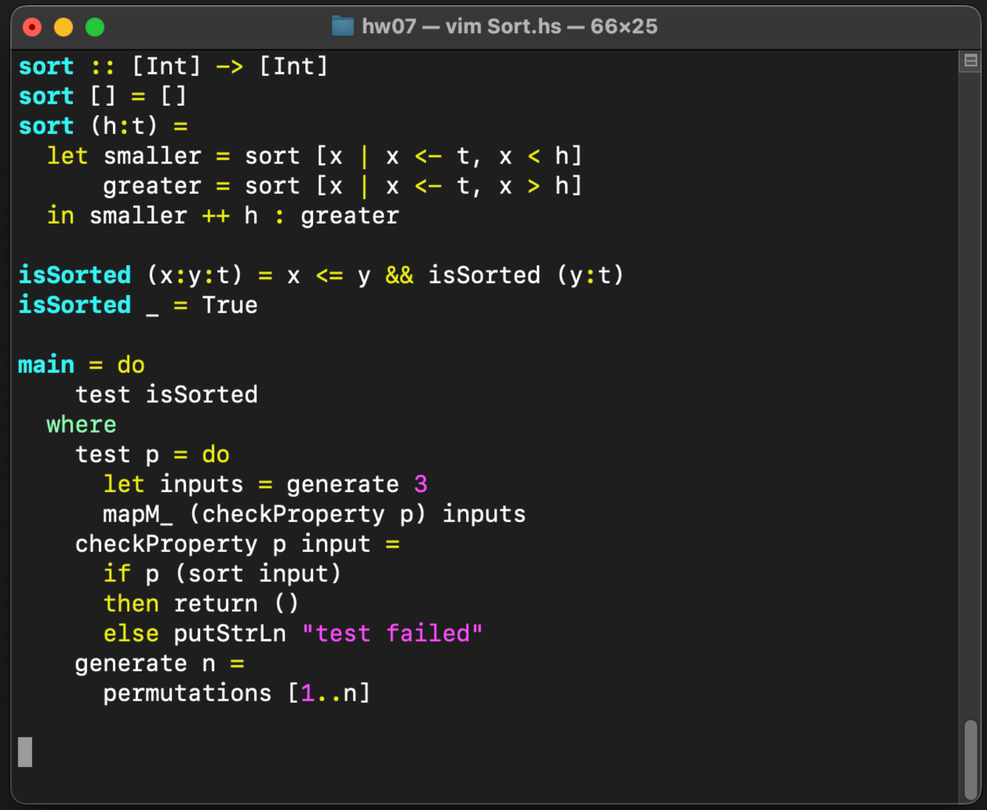
- We only know that it works on our tests
  - And on our machine
    - And at the moment the tests are run…

- Anyone gets bored writing tests
- It’s easy to intentionally skip some trivial cases
- The tests may be convoluted

SOLUTION: DON'T WRITE TESTS

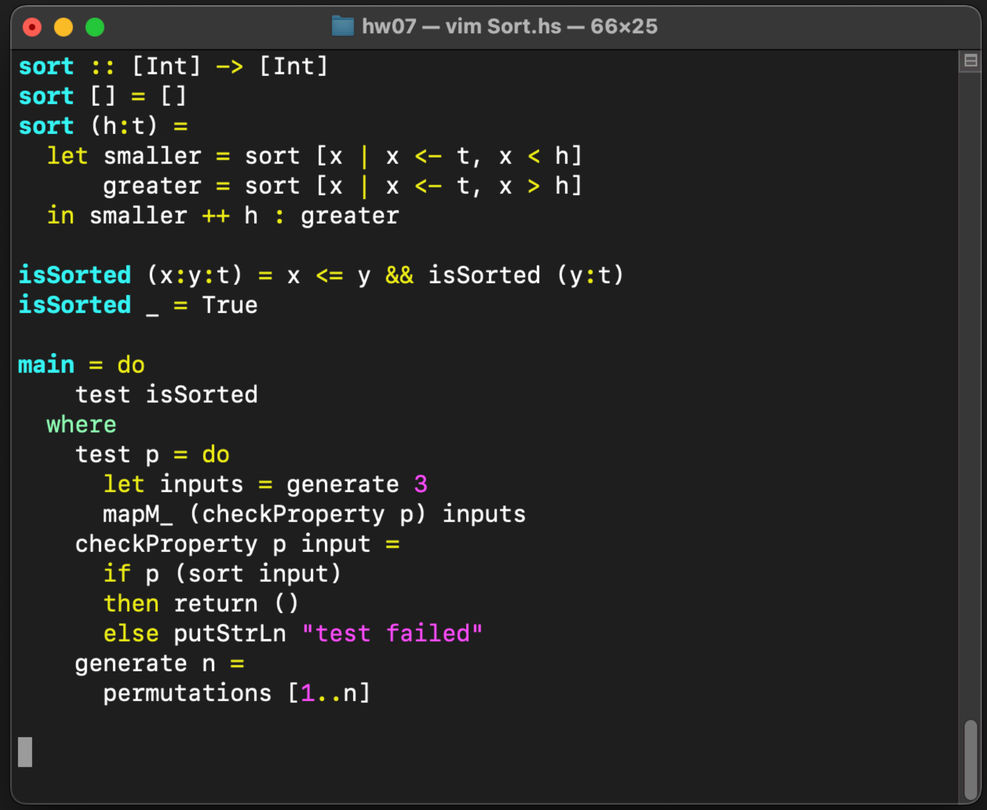
SOLUTION: DON'T WRITE TESTS

- Don't only check that the output is the one you expect
- Check properties of your function
  - Generate inputs
  - Run your function on them
  - Check that a property holds



WE NEED BETTER GENERATORS

- It'll be nice to generate:
  - Not only lists of the given length
  - Not only permutations
  - Really big lists with big numbers



HEDGEHOG

- The OG property-based library is QuickCheck
- We're going to use Hedgehog
  - A little more user-friendly

THREE PARTS OF A PB TEST

- Generator
  - Creates random inputs
- Property
  - What is checked
- Shrinking
  - Makes your tests as small as possible
  - We'll use the default shrinker

DEMO

Test — vim Sort.hs — 84×28

```haskell
module Test.Sort where

import Hedgehog
import qualified Hedgehog.Gen as Gen
import qualified Hedgehog.Range as Range
import Test.Tasty
import Test.Tasty.Hedgehog

import Sort
import qualified Data.List as L

genInt :: Gen Int
genInt = Gen.int (Range.constant 0 100)

genList :: Int -> Int -> Gen [Int]
genList minLength maxLength = Gen.list (Range.constant minLength maxLength) genInt

prop_sorted :: Property
prop_sorted = property $ do
  list <- forAll $ genList 1 100
  let sorted = sort list
  assert (isSorted sorted)

props :: [TestTree]
props =
  [ testProperty "The sorted list is ordered" prop_sorted
  ]
```

EXERCISE: TEST SORT

- Write a PBT that checks that a sorted list is a permutation of the original list

Test — vim Sort.hs — 84×28

```haskell
module Test.Sort where

import Hedgehog
import qualified Hedgehog.Gen as Gen
import qualified Hedgehog.Range as Range
import Test.Tasty
import Test.Tasty.Hedgehog

import Sort
import qualified Data.List as L

genInt :: Gen Int
genInt = Gen.int (Range.constant 0 100)

genList :: Int -> Int -> Gen [Int]
genList minLength maxLength = Gen.list (Range.constant minLength maxLength) genInt

prop_sorted :: Property
prop_sorted = property $ do
  list <- forAll $ genList 1 100
  let sorted = sort list
  assert (isSorted sorted)

props :: [TestTree]
props =
  [ testProperty "The sorted list is ordered" prop_sorted
  ]
```

A WANT TO LEARN MORE!

- Go watch How to specify it! by John Hughes
  - Great introduction
  - Useful techniques
  - Common pitfalls