INTRODUCTION TO

FUNCTIONAL PROGRAMMING

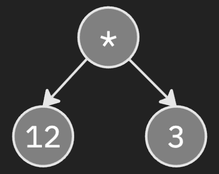
OUTLINE

- More on Monadic Parsing
- Property-Based Testing

MONADIC PARSERS

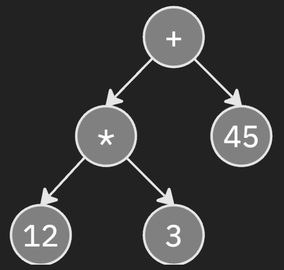
```
parseInt "12 * 3 + 45" = (12, " * 3 + 45")
```

```
parseMult "12 * 3 + 45" = (
```




```
, "+ 45")
```

```
parseExpr "12 * 3 + 45" = (
```



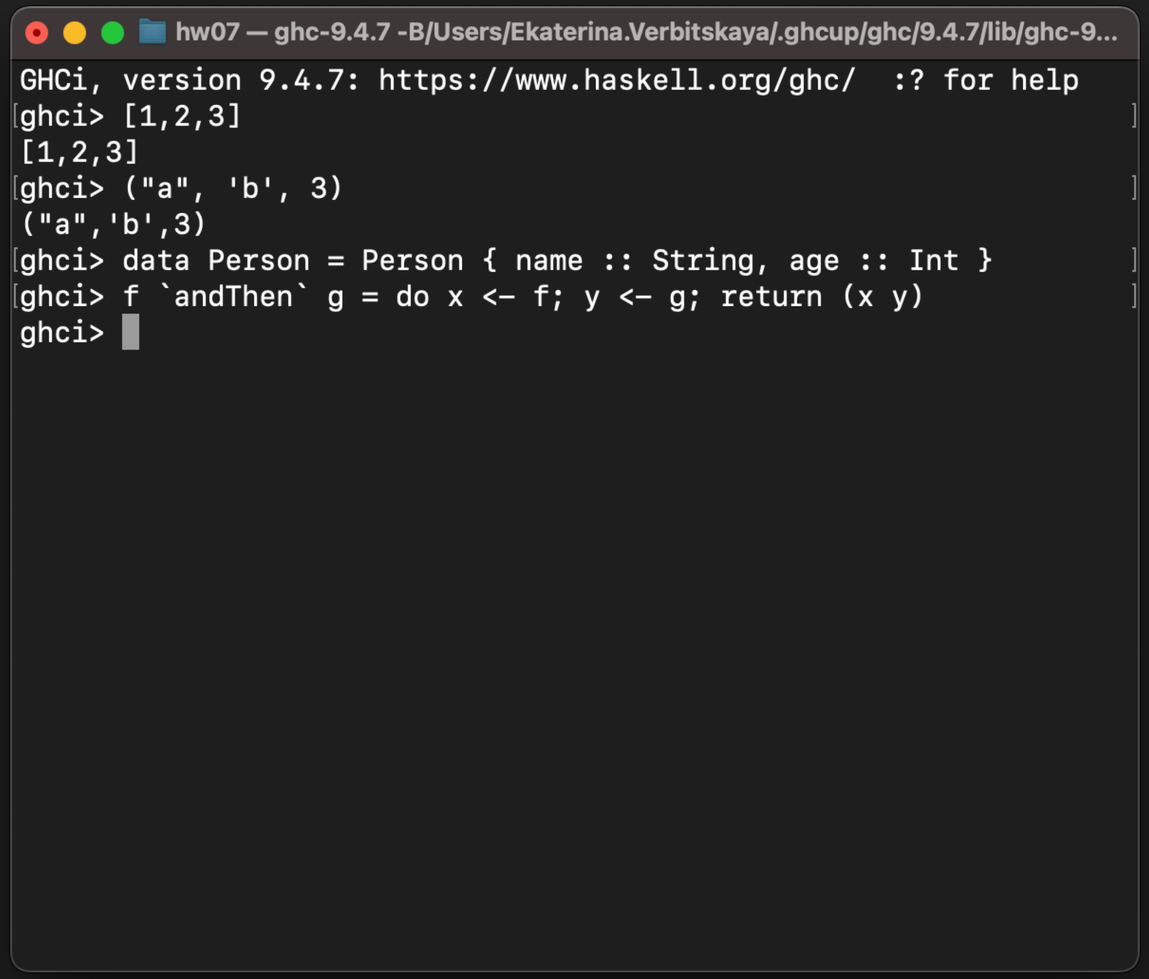

```
, "")
```

hw07 — vim src/Parser.hs — 65×37

```haskell
module Parser where

import Control.Applicative

newtype Parser a
  = Parser { runParser :: String -> Either String (String, a)}

instance Functor Parser where
  fmap :: (a -> b) -> Parser a -> Parser b
  fmap f p = Parser (((f <$>) <$>) . runParser p)

instance Applicative Parser where
  pure :: a -> Parser a
  pure = Parser . (return .) . flip (,)

  (<*>) :: Parser (a -> b) -> Parser a -> Parser b
  (<*>) f p = Parser $ \str -> do
    (str, f) <- runParser f str
    (str, a) <- runParser p str
    return (str, f a)

instance Monad Parser where
  (>>=) :: Parser a -> (a -> Parser b) -> Parser b
  (>>=) f p = Parser $ \str -> do
    (str, x) <- runParser f str
    runParser (p x) str

instance Alternative Parser where
  empty :: Parser a
  empty = Parser $ const $ Left "Empty"

  (<|>) :: Parser a -> Parser a -> Parser a
  (<|>) l r = Parser $ \str ->
    case runParser l str of
      Left _ -> runParser r str
      x -> x
```

SEQUENCES

- Many aspects of syntax are repeated over and over
- Sequence with a separator:
  - List
  - Tuple
  - Record
  - Statements



PARSER FOR SEQUENCES

- We've written a parser for a sequence of identifiers and implemented a parser for
  - List
  - Tuple
- Only supports ',' as a separator
- Only identifiers can be elements

hw07 — vim src/Parser.hs — 65×25

```
parseIdentTuple :: Parser [String]
parseIdentTuple = inParens '(' ')' identSequence

parseIdentList :: Parser [String]
parseIdentList = inParens '[' ']' identSequence

identSequence :: Parser [String]
identSequence = (do
    h <- parseIdent'
    t <- many (satisfy (== ',') >> parseIdent')
    return (h : t)
  )
  <|>
    return []

inParens :: Char -> Char -> Parser a -> Parser a
inParens l r p = do
  satisfy (== l)
  x <- p
  satisfy (== r)
  return x
```

LET'S ABSTRACT SEQUENCE PARSER

- Parsers for a separator and item can passed as parameters
- inParens and parseSequence are *parser combinators*
- A parser combinator is a higher-order function which takes a parser as its argument and creates a parser as a result

hw07 — vim src/Parser.hs — 65×25

```haskell
parseIdentTuple :: Parser [String]
parseIdentTuple = inParens '(' ')' (parseSequence ',' parseIdent)

parseIdentList :: Parser [String]
parseIdentList = inParens '[' ']' (parseSequence ',' parseIdent)

parseSequence :: Char -> Parser a -> Parser [a]
parseSequence sep elem = (do
    h <- elem
    t <- many (satisfy (== sep) >> elem)
    return (h : t)
  )
  <|>
    return []

inParens :: Char -> Char -> Parser a -> Parser a
inParens l r p = do
  satisfy (== l)
  x <- p
  satisfy (== r)
  return x
```

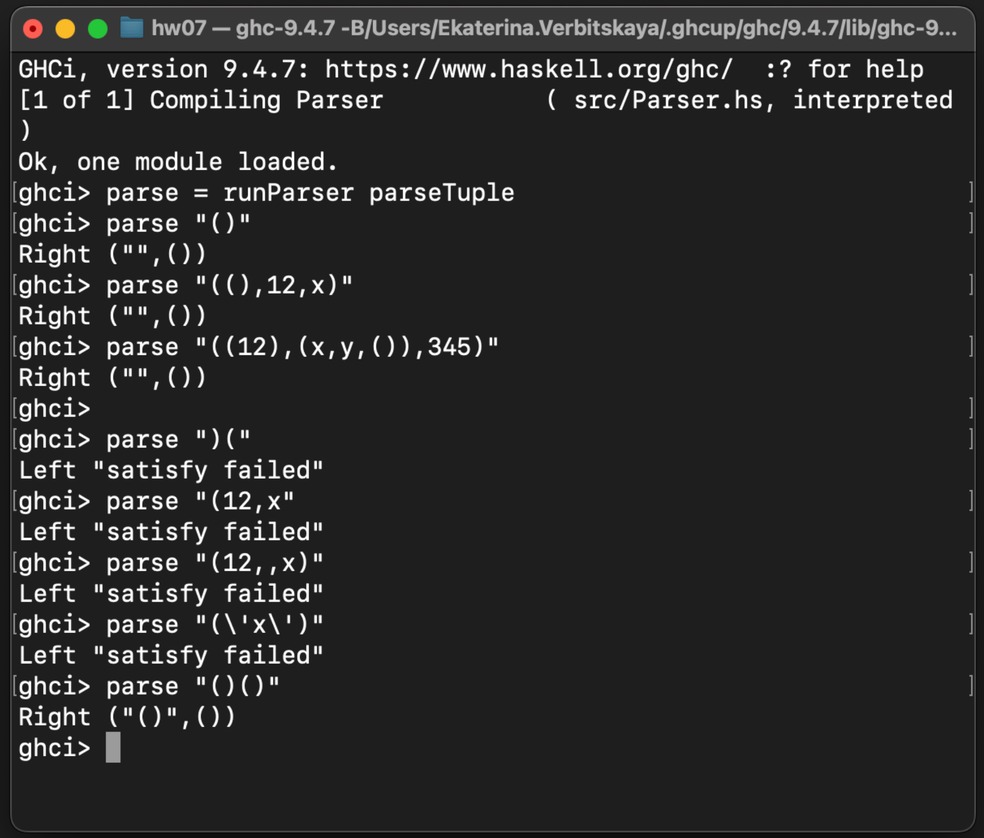
PARSER COMBINATORS: WHY?

- Modularity
- Composability
- Simplicity
- Customisation

- The less a function does
  the easier it is to get it right

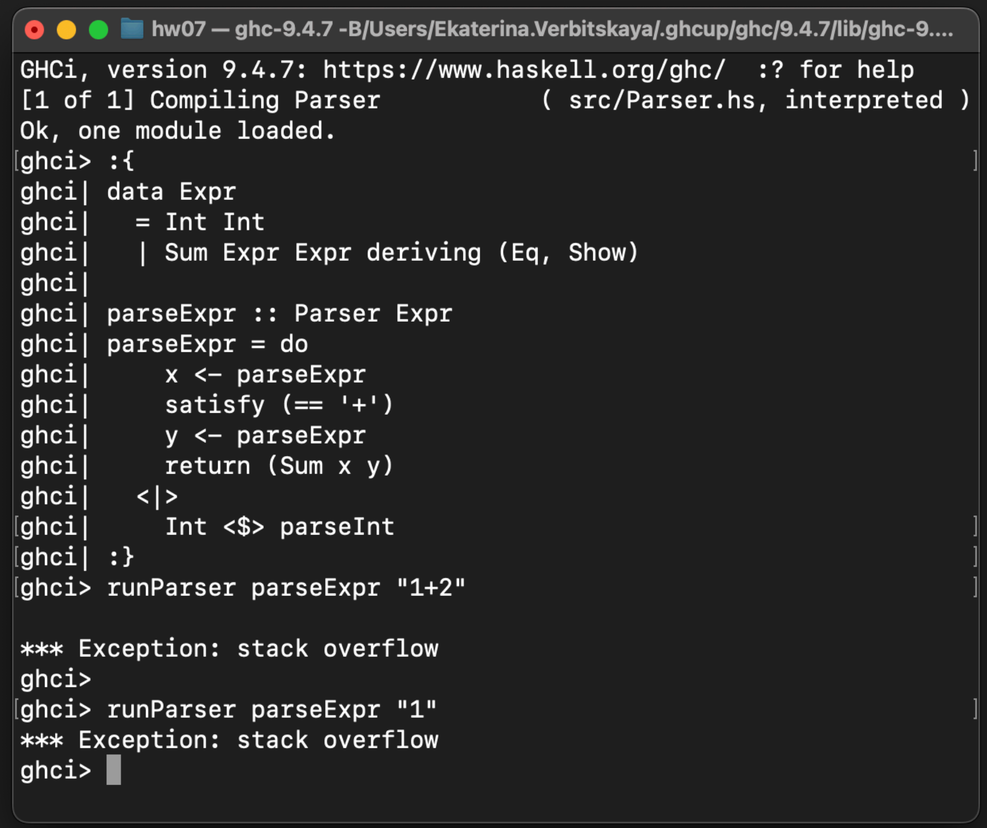
EXERCISE

- Implement a matcher for a language of tuples:
  - An element of a tuple is either:
    - Integer
    - Identifier
    - Another tuple
  - A tuple can have any depth and size
  - A matcher is a parser of type `Parser ()`



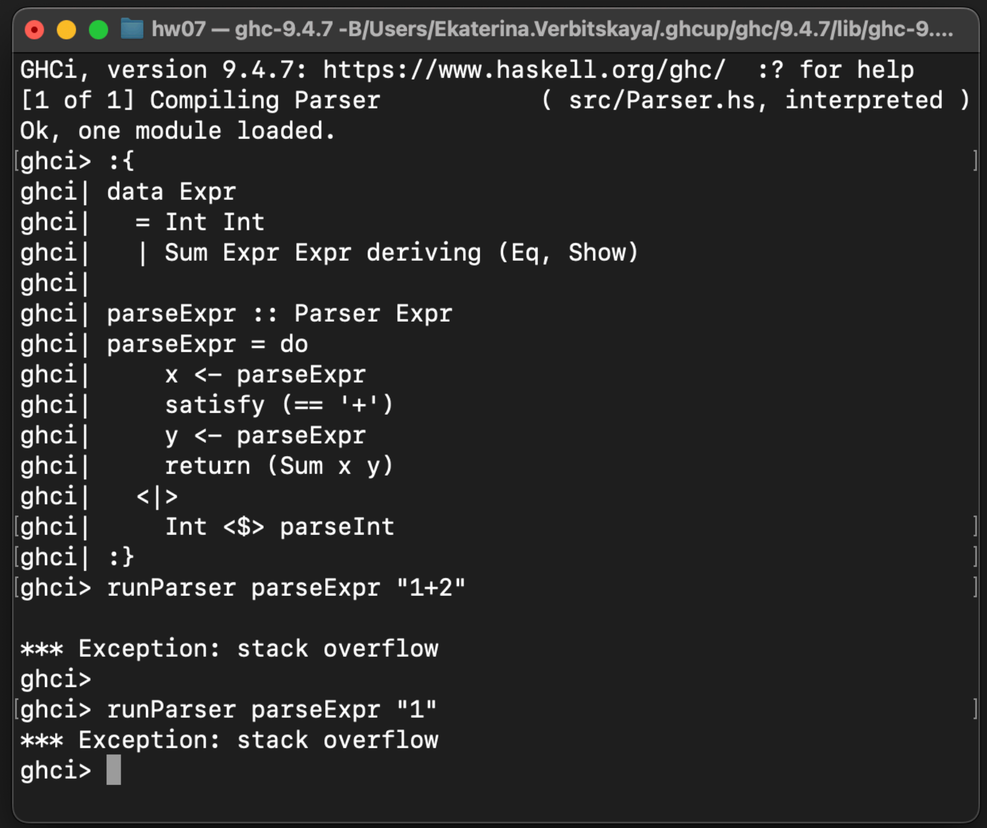
LET'S PARSE EXPRESSIONS

- Expression is either:
  - Integer number
  - Sum of two expressions
- Left recursion is a no-no



LET'S PARSE EXPRESSIONS

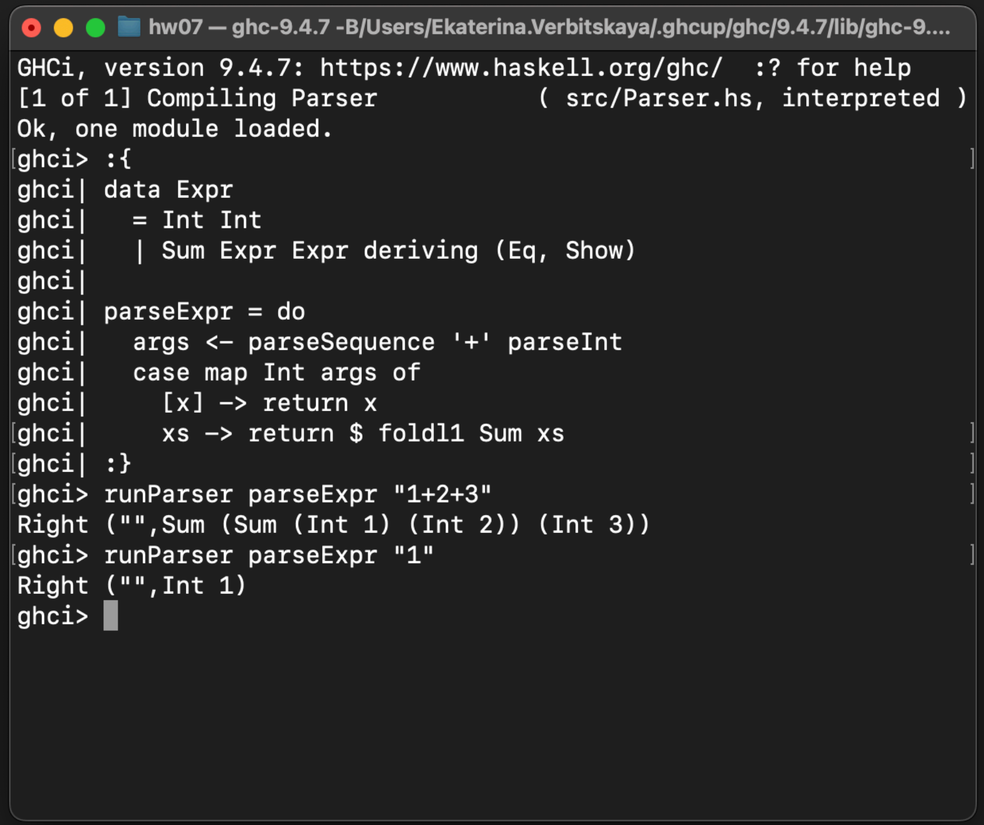
- Expression is either:
  - Integer number
  - Sum of two expressions
- Left recursion is a no-no
- Let's reframe our mind



EXPRESSION AS A SEQUENCE

- Expression is a sequence with
  - + as a separator
  - an integer number as an element
- No left recursion

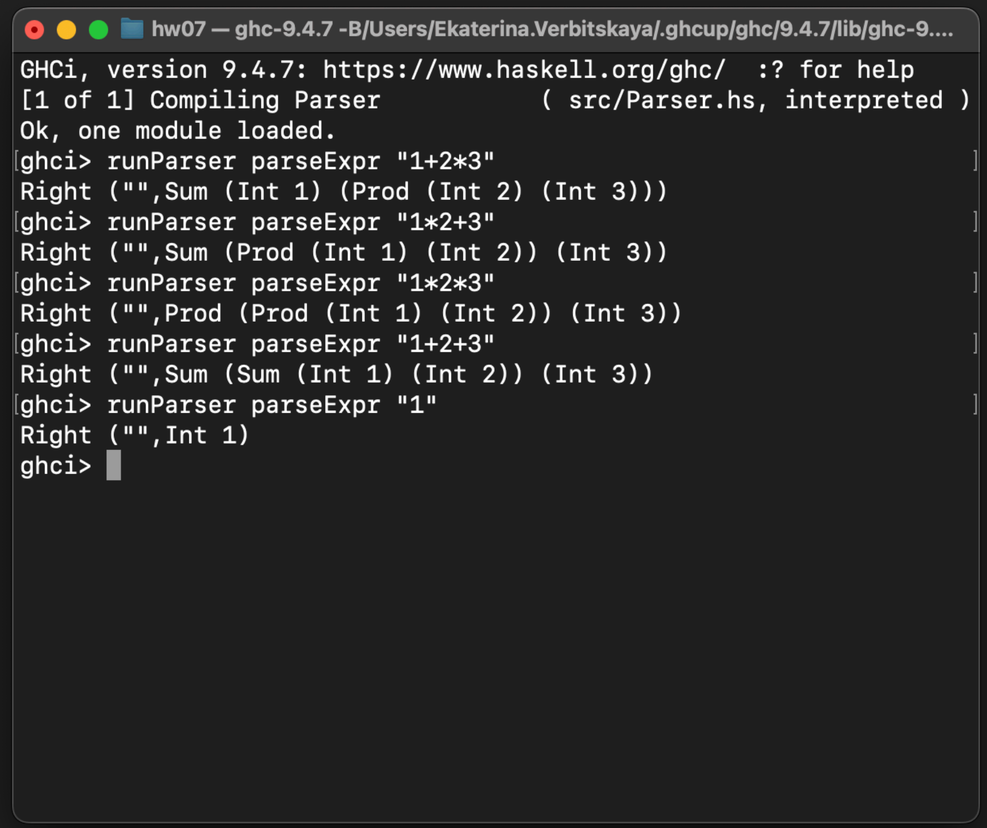
- gist



EXERCISE: LET'S ADD MULTIPLICATION!

- Expression is either:
  - Integer
  - Sum of two expressions
  - Multiplication of two expressions

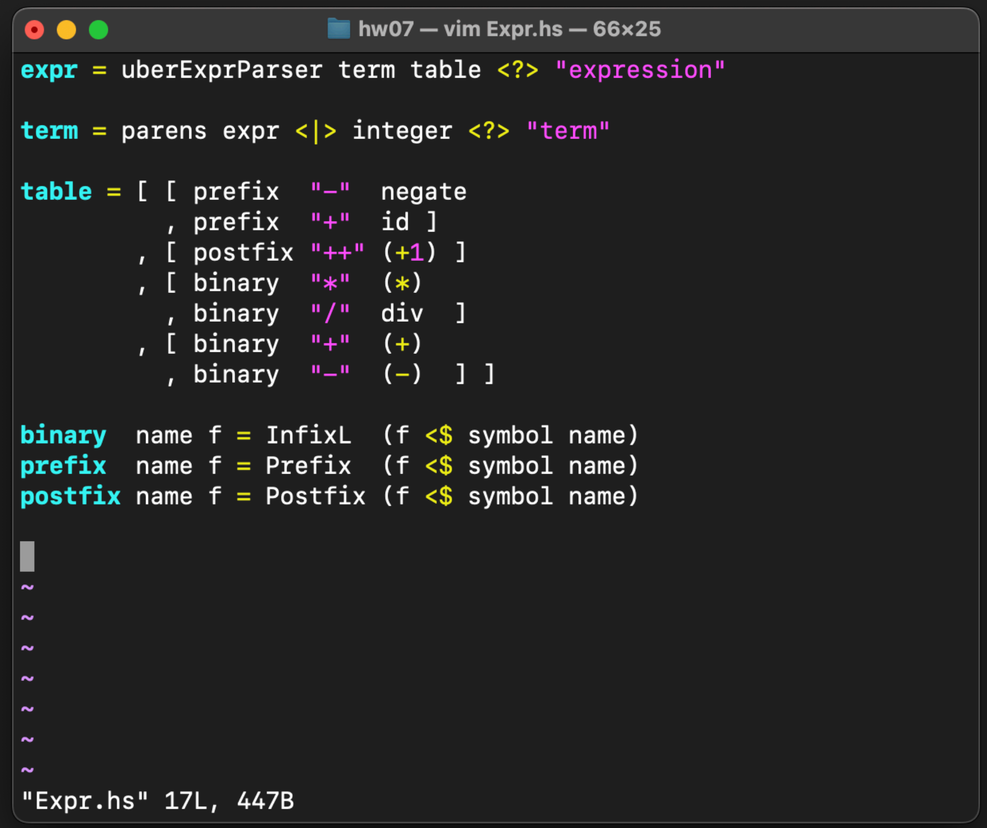
- Multiplication binds tighter
  - 1+2*3 == 1+(2*3)
  - 1*2+3 == (1*2)+3



EXERCISE: LET'S ADD EVERYTHING ELSE

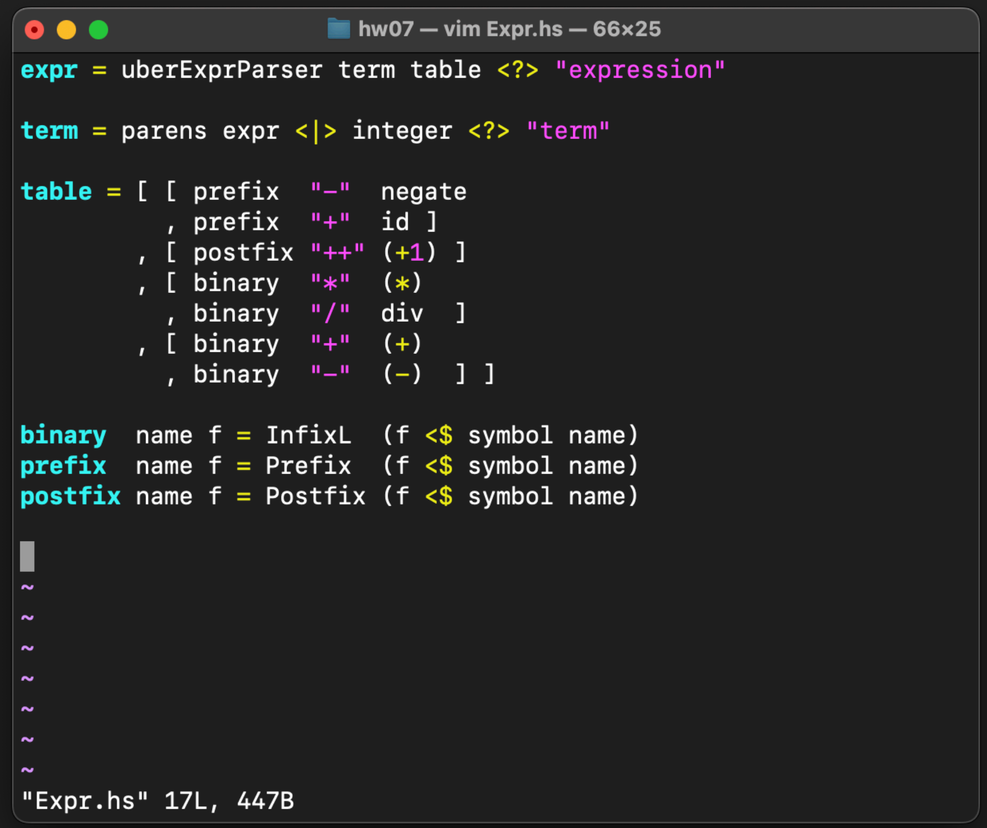
- Expression is either:
  - Integer or identifier
  - Unary operator applied to an expression
  - Binary operator between two expressions
  - Expression in parentheses

- Normal precedence and associativity
- Make an uberExprParser if you dare



I WANNA LEARN MORE

- Tutorial from Serokell
- There are multiple libraries for combinatory parsing
- When in doubt, pick megaparsec
  - Great tutorial
  - Efficient lexing
  - Additional combinators
  - Fast



OUTLINE

- More on Monadic Parsing
- 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?

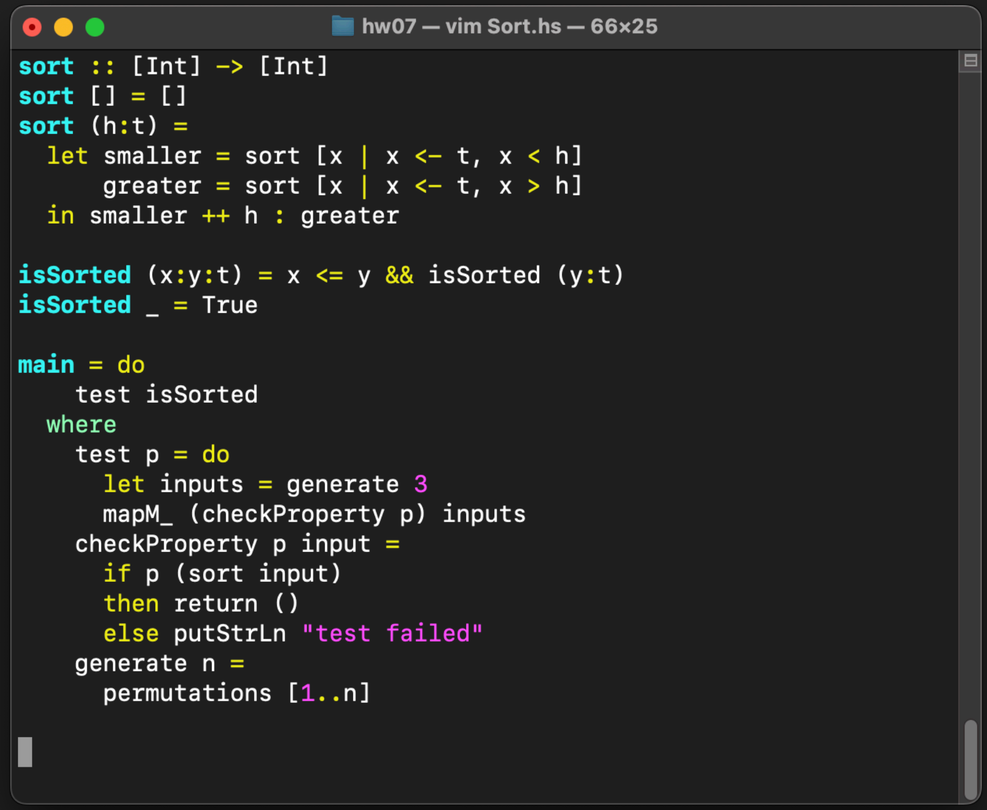
- We only know that it works on our tests
  - And on our machine
    - And at the moment the tests are run…

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

SOLUTION: DON'T WRITE TESTS

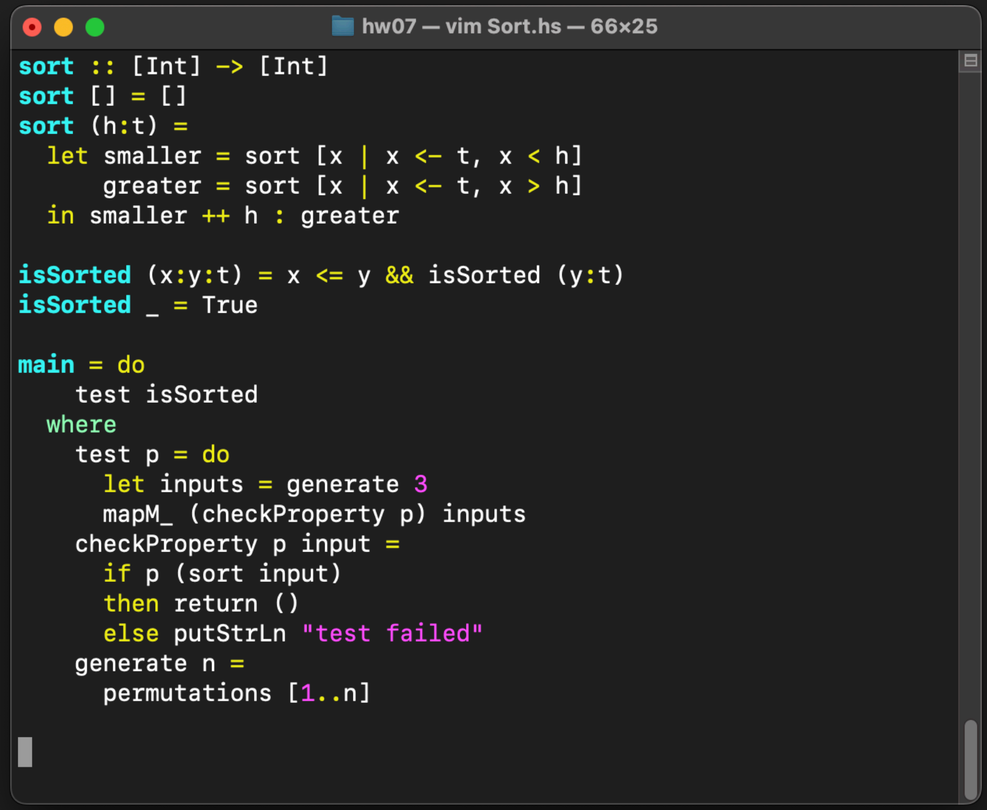
SOLUTION: DON'T WRITE TESTS

- 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