

INTRODUCTION TO

FUNCTIONAL PROGRAMMING

OUTLINE

- ▶ More on Monadic Parsing
- ▶ Property-Based Testing

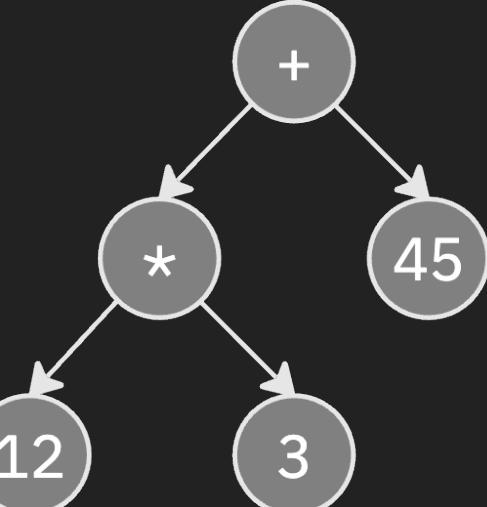
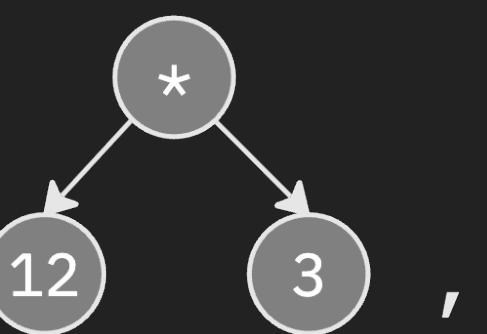
MORE ON MONADIC PARSING

MONADIC PARSERS

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

parseMult "12 * 3 + 45" = (12 , 3 , "+ 45")

parseExpr "12 * 3 + 45" = (12 , 3 , "",)



```
hw07 — vim src/Parser.hs — 65x37

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)

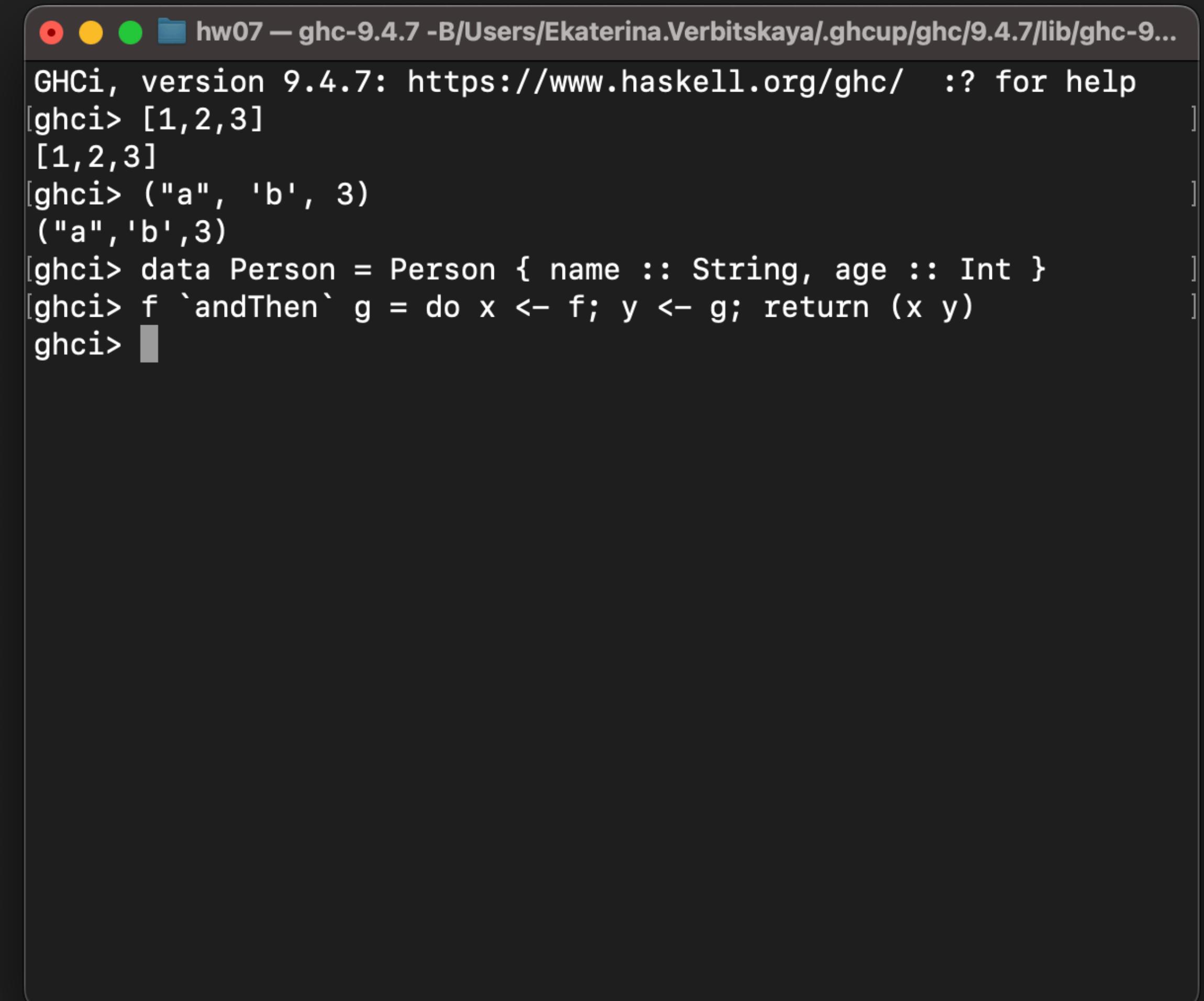
instance Monad Parser where
  (=>) :: Parser a -> (a -> Parser b) -> Parser b
  (=>) f p = Parser $ \str -> do
    (str, x) <- runParser f str
    runParser (p x) str

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

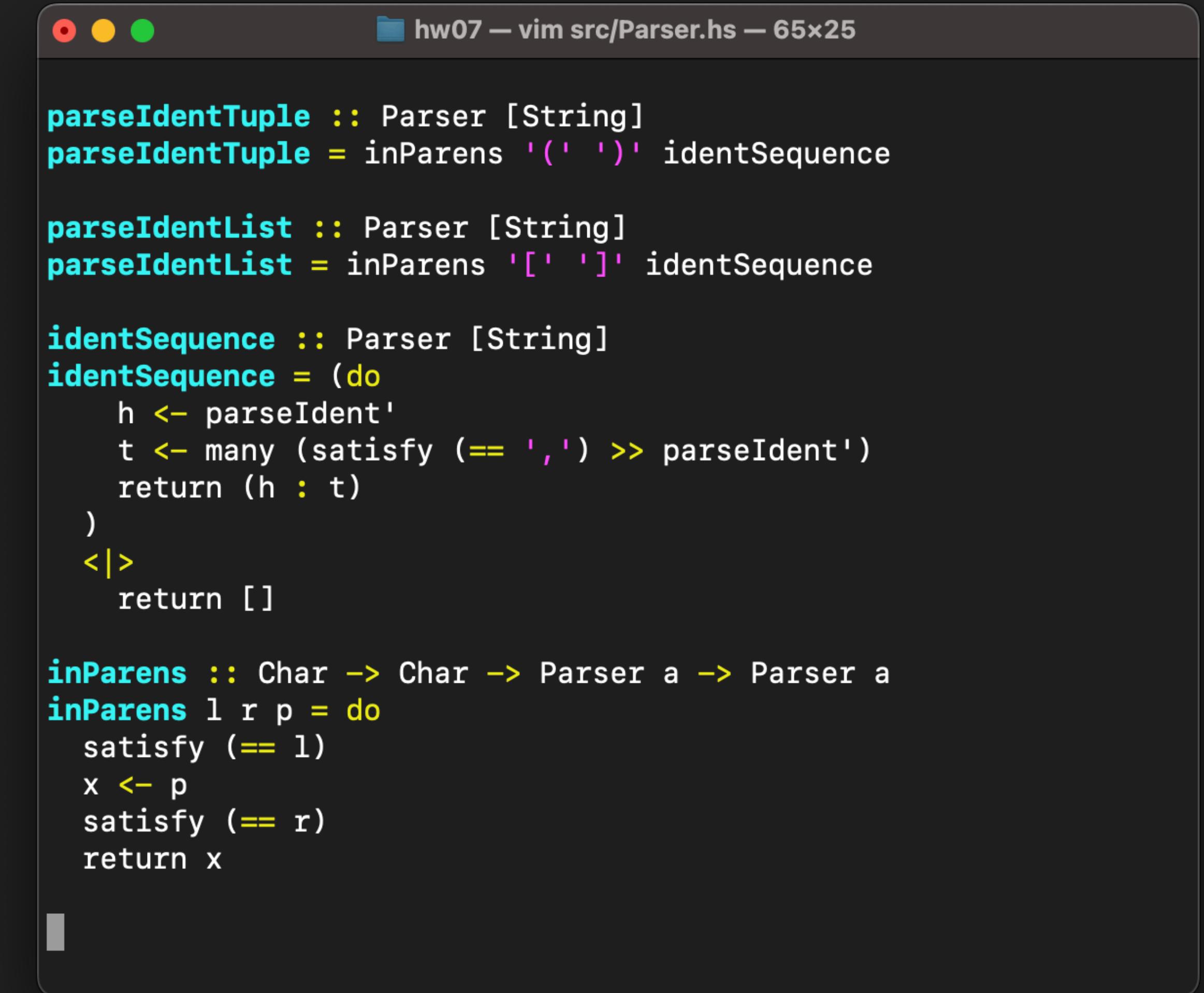


A screenshot of a terminal window titled "hw07 – ghc-9.4.7 -B/Users/Ekaterina.Verbitskaya.ghcup/ghc/9.4.7/lib/ghc-9...". The window shows a GHCi session with the following input and output:

```
GHCI, version 9.4.7: https://www.haskell.org/ghc/  ?: for help
[ghci> [1,2,3]
[1,2,3]
[ghci> ("a", 'b', 3)
("a",'b',3)
[ghci> data Person = Person { name :: String, age :: Int }
[ghci> f `andThen` g = do x <- f; y <- g; return (x y)
ghci> ]
```

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



A screenshot of a terminal window titled "hw07 – vim src/Parser.hs – 65x25". The window displays Haskell code for monadic parsing. The code includes definitions for `parseIdentTuple`, `parseIdentList`, and `identSequence` parsers, and an `inParens` parser combinator.

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

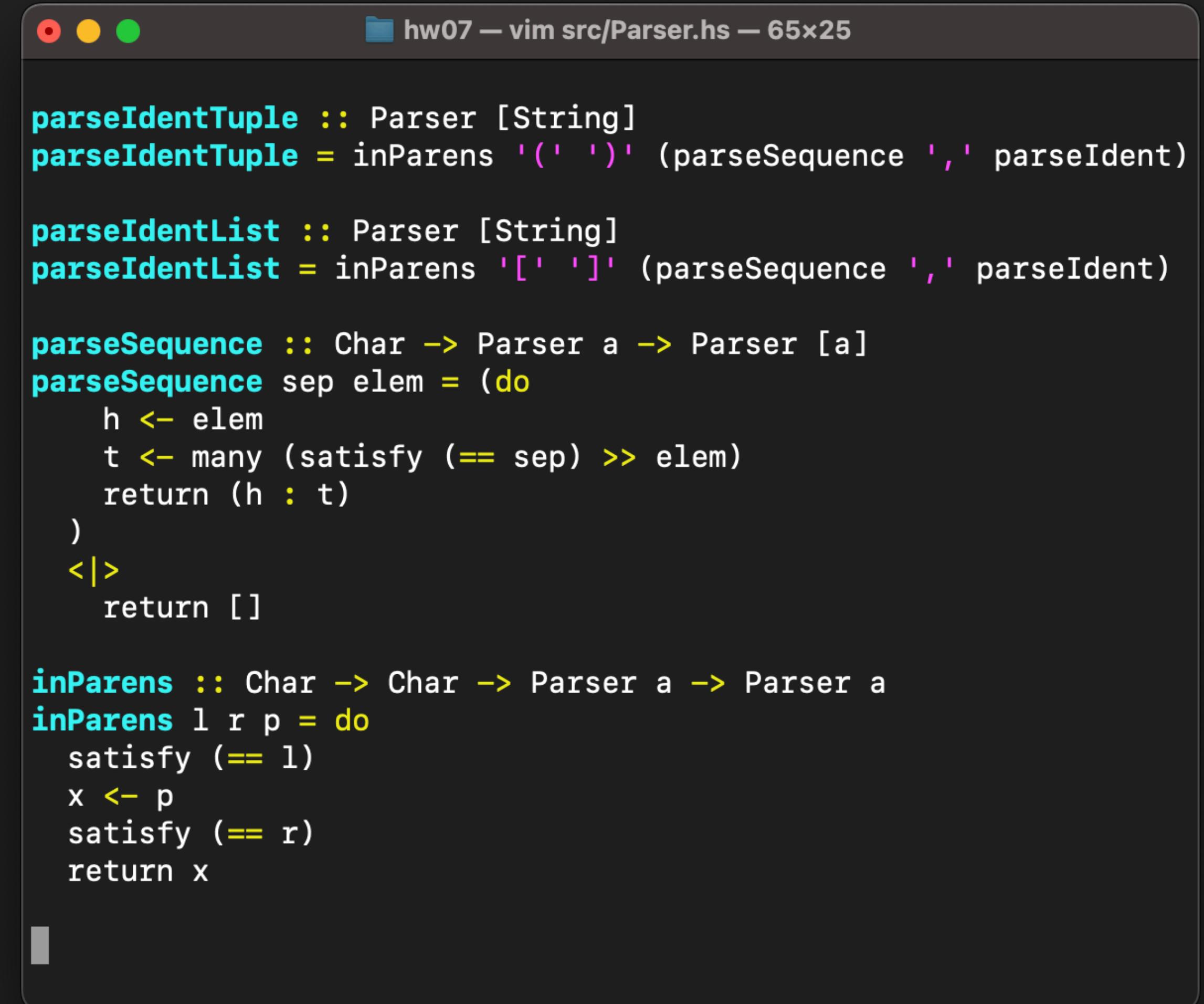
parseIdentList :: Parser [String]
parseIdentList = inParens '[' ']' identSequence

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



A screenshot of a terminal window titled "hw07 – vim src/Parser.hs – 65x25". The window displays Haskell code for monadic parsing. The code defines four functions: `parseIdentTuple`, `parseIdentList`, `parseSequence`, and `inParens`. `parseIdentTuple` and `parseIdentList` are defined using `inParens` and `parseSequence`. `parseSequence` is a higher-order function that takes a separator character and an item parser, then returns a sequence parser. `inParens` is a parser combinator that takes a left parenthesis character, a right parenthesis character, and a parser, then returns a parser that matches the entire sequence between the parentheses.

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

parseIdentList :: Parser [String]
parseIdentList = inParens '["\n"]' (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 ()`

```
hw07 — ghc-9.4.7 -B/Users/Ekaterina.Verbitskaya/.ghcup/ghc/9.4.7/lib/ghc-9...
GHCi, version 9.4.7: https://www.haskell.org/ghc/ :? for help
[1 of 1] Compiling Parser          ( src/Parser.hs, interpreted
)
Ok, one module loaded.
ghci> parse = runParser parseTuple
ghci> parse "()"
Right ("",())
ghci> parse "(((),12,x))"
Right ("",())
ghci> parse "((12),(x,y,()),345)"
Right ("",())
ghci>
ghci> parse ")()"
Left "satisfy failed"
ghci> parse "(12,x"
Left "satisfy failed"
ghci> parse "(12,,x)"
Left "satisfy failed"
ghci> parse "('x\\')"
Left "satisfy failed"
ghci> parse "())"
Right ("()",())
ghci>
```

LET'S PARSE EXPRESSIONS

- ▶ Expression is either:

- ▶ Integer number

- ▶ Sum of two expressions

- ▶ Left recursion is a no-no

```
hw07 — ghc-9.4.7 -B/Users/Ekaterina.Verbitskaya/.ghcup/ghc/9.4.7/lib/ghc-9....  
GHCi, version 9.4.7: https://www.haskell.org/ghc/  :? for help  
[1 of 1] Compiling Parser          ( src/Parser.hs, interpreted )  
Ok, one module loaded.  
[ghci]> :{  
ghci| data Expr  
ghci|   = Int Int  
ghci|   | Sum Expr Expr deriving (Eq, Show)  
ghci|  
ghci| parseExpr :: Parser Expr  
ghci| parseExpr = do  
ghci|   x <- parseExpr  
ghci|   satisfy (== '+')  
ghci|   y <- parseExpr  
ghci|   return (Sum x y)  
ghci|   <|>  
[ghci]>     Int <$> parseInt  
[ghci| :}  
[ghci]> runParser parseExpr "1+2"  
*** Exception: stack overflow  
ghci>  
[ghci]> runParser parseExpr "1"  
*** Exception: stack overflow  
ghci> █
```

LET'S PARSE EXPRESSIONS

- ▶ Expression is either:

- ▶ Integer number

- ▶ Sum of two expressions

- ▶ Left recursion is a no-no

- ▶ Let's reframe our mind

```
hw07 — ghc-9.4.7 -B/Users/Ekaterina.Verbitskaya/.ghcup/ghc/9.4.7/lib/ghc-9....  
GHCi, version 9.4.7: https://www.haskell.org/ghc/  ?: for help  
[1 of 1] Compiling Parser          ( src/Parser.hs, interpreted )  
Ok, one module loaded.  
[ghci]> :{  
ghci| data Expr  
ghci|   = Int Int  
ghci|   | Sum Expr Expr deriving (Eq, Show)  
ghci|  
ghci| parseExpr :: Parser Expr  
ghci| parseExpr = do  
ghci|   x <- parseExpr  
ghci|   satisfy (== '+')  
ghci|   y <- parseExpr  
ghci|   return (Sum x y)  
ghci|   <|>  
[ghci|   Int <$> parseInt  
[ghci| :}  
[ghci]> runParser parseExpr "1+2"  
*** Exception: stack overflow  
ghci>  
[ghci]> runParser parseExpr "1"  
*** Exception: stack overflow  
ghci> █
```

EXPRESSION AS A SEQUENCE

- ▶ Expression is a sequence with

- ▶ + as a separator

- ▶ an integer number as an element

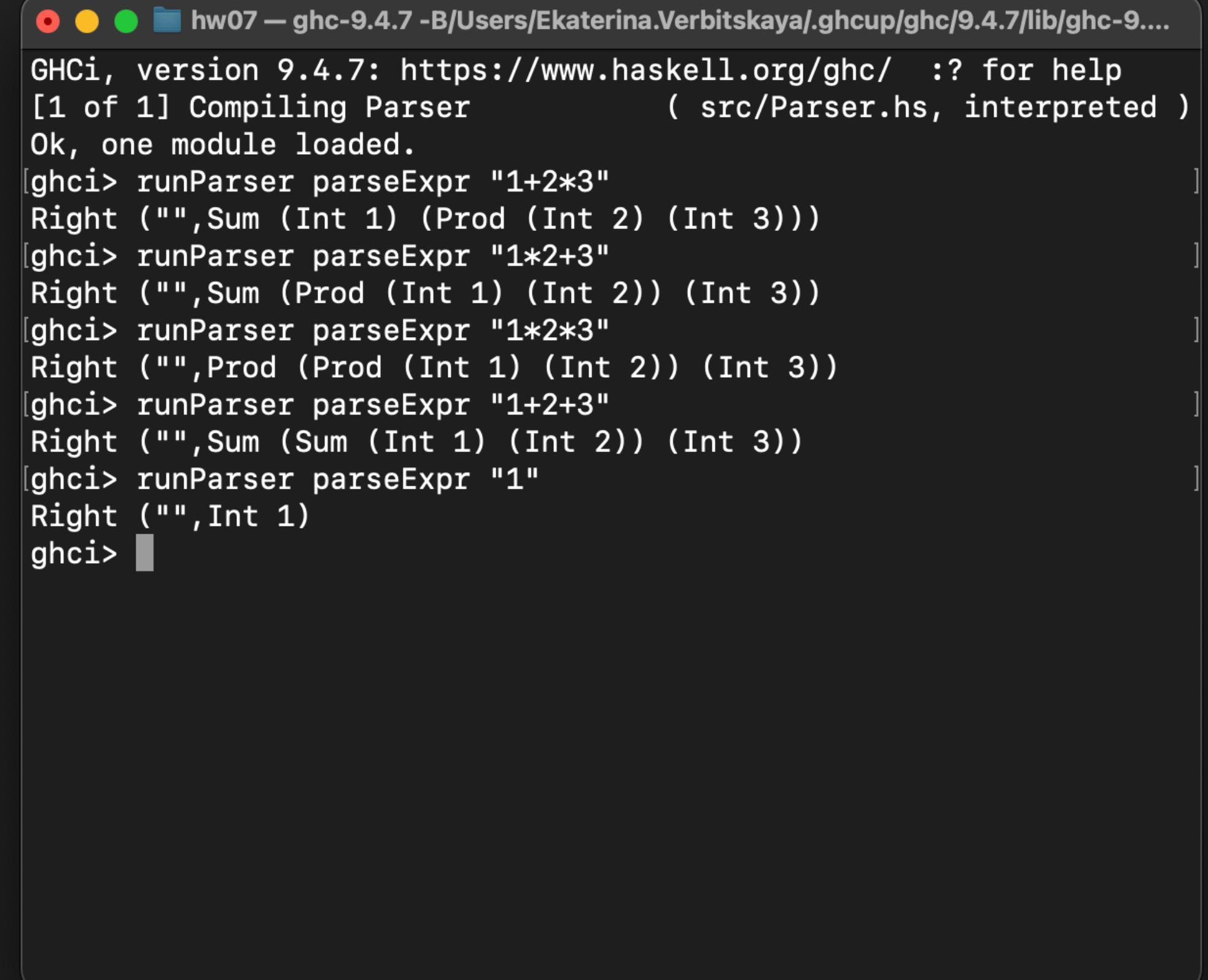
- ▶ No left recursion

- ▶ [gist](#)

```
hw07 — ghc-9.4.7 -B/Users/Ekaterina.Verbitskaya/.ghcup/ghc/9.4.7/lib/ghc-9....  
GHCi, version 9.4.7: https://www.haskell.org/ghc/  :? for help  
[1 of 1] Compiling Parser          ( src/Parser.hs, interpreted )  
Ok, one module loaded.  
[ghci]> :{  
ghci| data Expr  
ghci|   = Int Int  
ghci|   | Sum Expr Expr deriving (Eq, Show)  
ghci|  
ghci| parseExpr = do  
ghci|   args <- parseSequence '+' parseInt  
ghci|   case map Int args of  
ghci|     [x] -> return x  
ghci|     xs -> return $ foldl1 Sum xs  
ghci| :}  
ghci> runParser parseExpr "1+2+3"  
Right ("",Sum (Sum (Int 1) (Int 2)) (Int 3))  
ghci> runParser parseExpr "1"  
Right ("",Int 1)  
ghci>
```

EXERCISE: LET'S ADD MULTIPLICATION!

- ▶ Expression is either:
 - ▶ Integer
 - ▶ Sum of two expressions
 - ▶ Multiplication of two expressions
- ▶ Multiplication binds tighter
 - ▶ $1+2*3 == 1+(2*3)$
 - ▶ $1*2+3 == (1*2)+3$

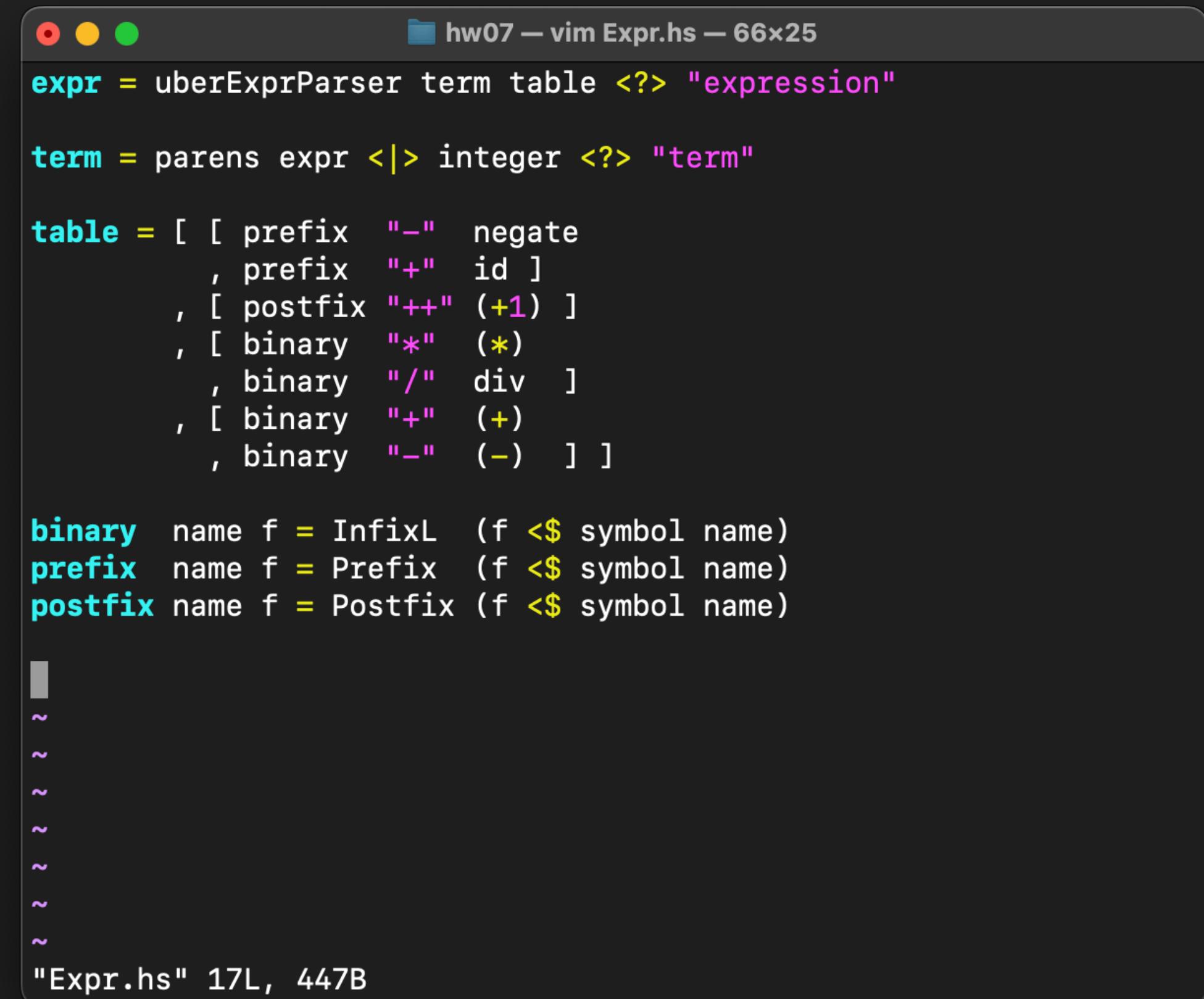


A screenshot of a terminal window titled "hw07 — ghc-9.4.7 -B/Users/Ekaterina.Verbitskaya/.ghcup/ghc/9.4.7/lib/ghc-9....". The window shows a GHCi session with the following output:

```
GHCI, version 9.4.7: https://www.haskell.org/ghc/  :? for help
[1 of 1] Compiling Parser          ( src/Parser.hs, interpreted )
Ok, one module loaded.
ghci> runParser parseExpr "1+2*3"
Right ("",Sum (Int 1) (Prod (Int 2) (Int 3)))
ghci> runParser parseExpr "1*2+3"
Right ("",Sum (Prod (Int 1) (Int 2)) (Int 3))
ghci> runParser parseExpr "1*2*3"
Right ("",Prod (Prod (Int 1) (Int 2)) (Int 3))
ghci> runParser parseExpr "1+2+3"
Right ("",Sum (Sum (Int 1) (Int 2)) (Int 3))
ghci> runParser parseExpr "1"
Right ("",Int 1)
ghci>
```

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



A screenshot of a vim window titled "hw07 — vim Expr.hs — 66x25". The code defines an Uber Parser for expressions. It starts with the type declarations:

```
expr = uberExprParser term table <?> "expression"
term = parens expr <|> integer <?> "term"
```

It then defines a table of operators:

```
table = [ [ prefix  "-"  negate
           , prefix  "+"  id ]
         , [ postfix "+"  (+1) ]
         , [ binary   "*"  (*)
             , binary   "/"  div
             , binary   "+"  (+)
             , binary   "-"  (-) ] ]
```

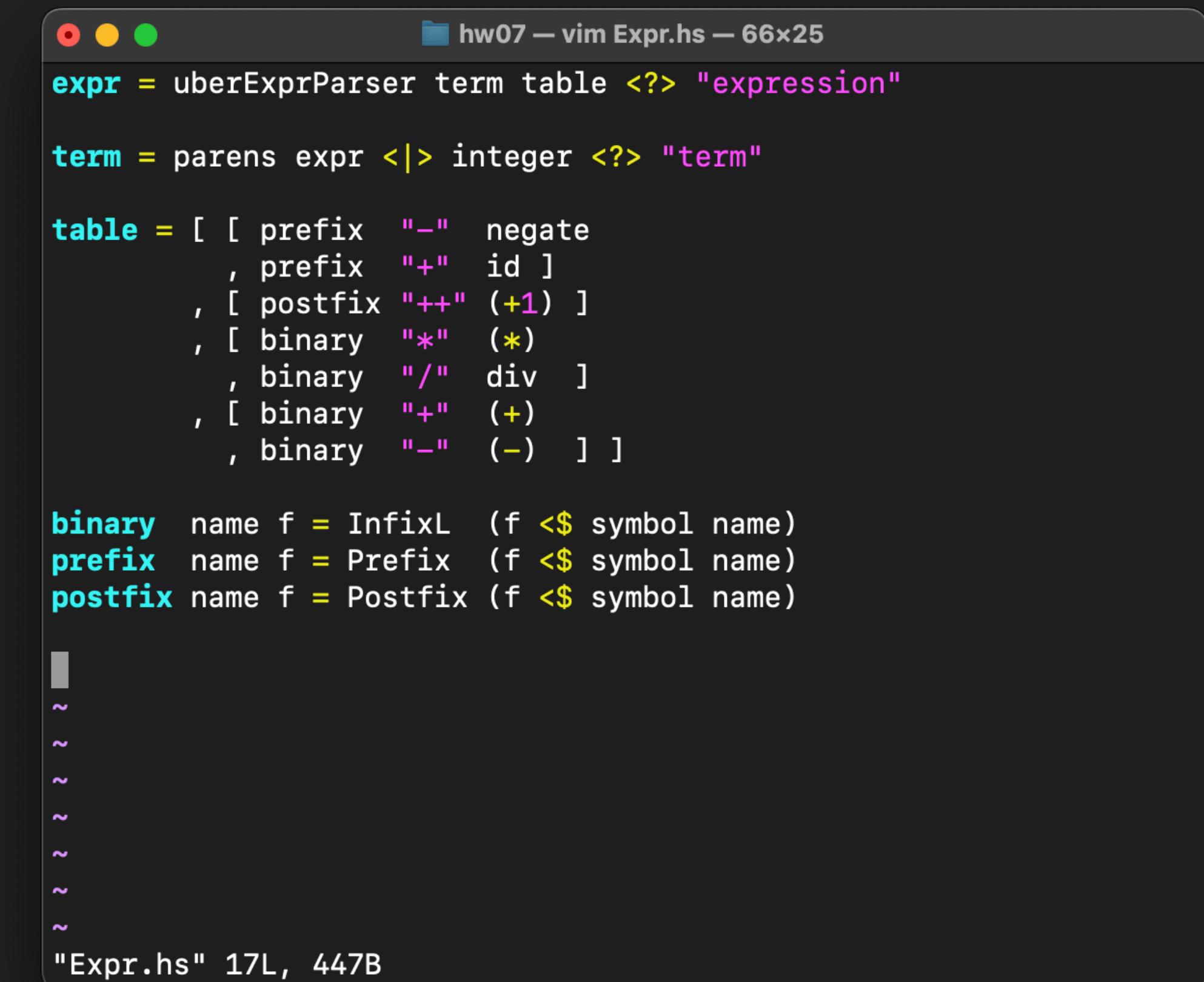
Finally, it defines three helper functions:

```
binary name f = InfixL (f <$> symbol name)
prefix name f = Prefix (f <$> symbol name)
postfix name f = Postfix (f <$> symbol name)
```

The vim window shows several blank lines at the bottom.

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



A screenshot of a vim window titled "hw07 — vim Expr.hs — 66x25". The code defines an expression parser using combinatory parsing:

```
expr = uberExprParser term table <?> "expression"
term = parens expr <|> integer <?> "term"

table = [ [ prefix  "-"  negate
           , prefix  "+"  id ]
         , [ postfix "++" (+1) ]
         , [ binary   "*"  (*)
             , binary   "/"  div ]
         , [ binary   "+"  (+)
             , binary   "-"  (-) ] ]

binary name f = InfixL (f <$ symbol name)
prefix name f = Prefix (f <$ symbol name)
postfix name f = Postfix (f <$ symbol name)
```

The file has 17 lines and 447 bytes.

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?



A screenshot of a vim window titled "hw07 — vim Sort.hs — 66x25". The window displays Haskell code for a quicksort algorithm and a test harness.

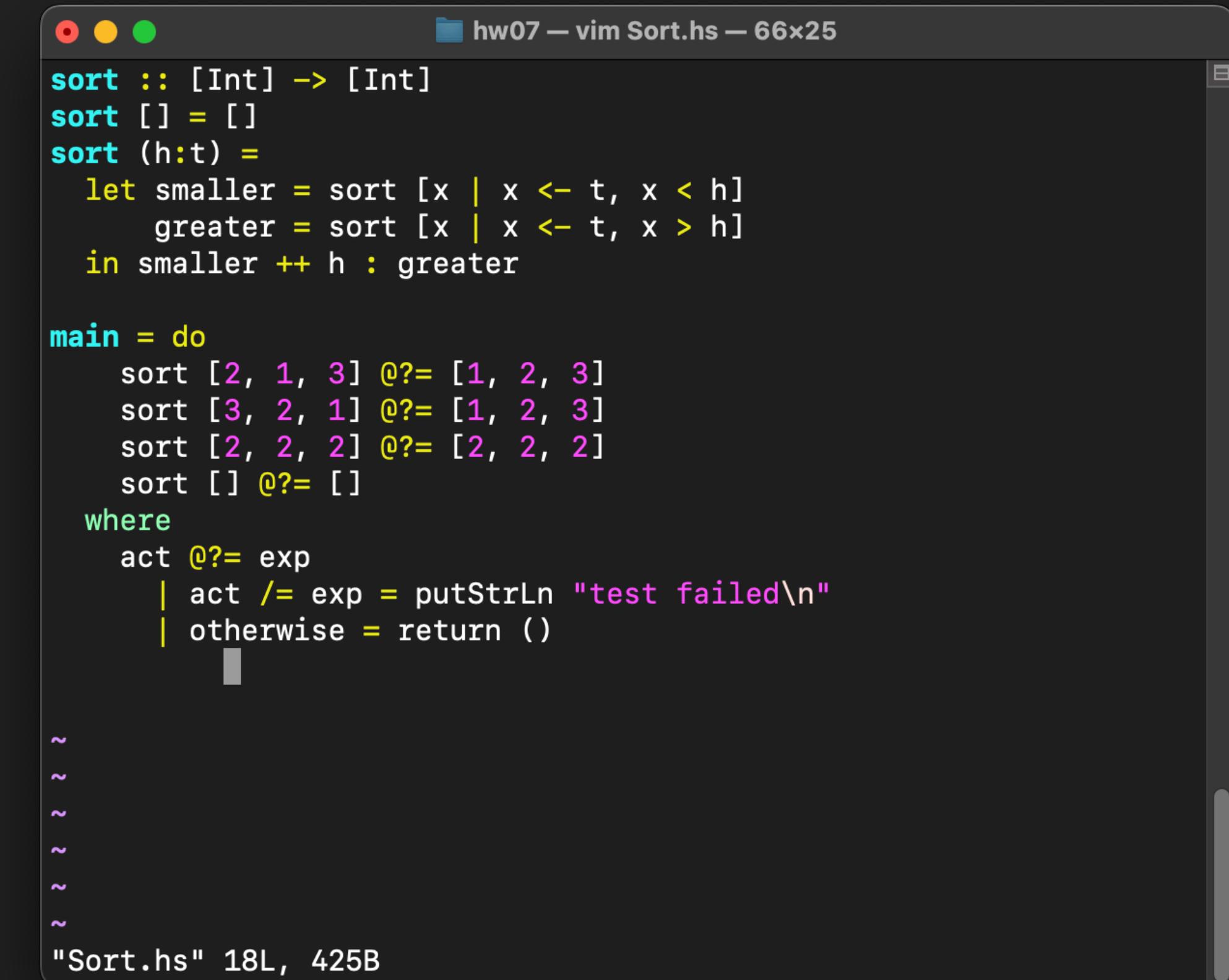
```
sort :: [Int] -> [Int]
sort [] = []
sort (h:t) =
    let smaller = sort [x | x <- t, x < h]
        greater = sort [x | x <- t, x > h]
    in smaller ++ h : greater

main = do
    sort [2, 1, 3] @?= [1, 2, 3]
    sort [3, 2, 1] @?= [1, 2, 3]
    sort [2, 2, 2] @?= [2, 2, 2]
    sort [] @?= []
where
    act @?= exp
        | act /= exp = putStrLn "test failed\n"
        | otherwise = return ()
```

The vim window shows several blank lines at the bottom, indicated by the character "~". The status bar at the bottom right of the vim window shows the file name "Sort.hs", line count "18L", and byte count "425B".

UNIT TESTS: WHEN TO STOP?

- ▶ Sunny day scenario
- ▶ Rainy day scenario
- ▶ Corner cases
- ▶ Good coverage
- ▶ ...



A screenshot of a vim editor window titled "hw07 — vim Sort.hs — 66x25". The window displays Haskell code for a quicksort algorithm and a test function.

```
sort :: [Int] -> [Int]
sort [] = []
sort (h:t) =
    let smaller = sort [x | x <- t, x < h]
        greater = sort [x | x <- t, x > h]
    in smaller ++ h : greater

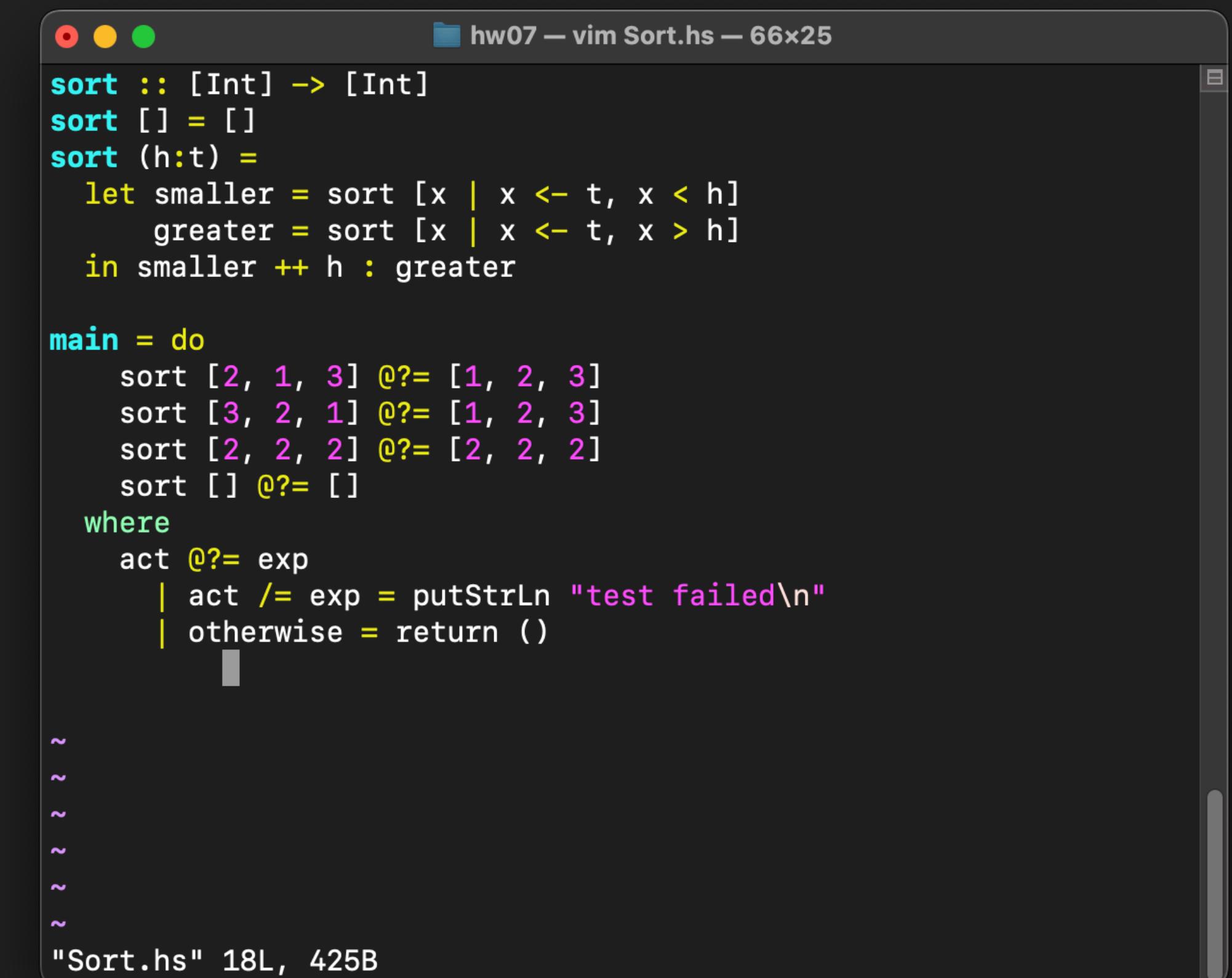
main = do
    sort [2, 1, 3] @?= [1, 2, 3]
    sort [3, 2, 1] @?= [1, 2, 3]
    sort [2, 2, 2] @?= [2, 2, 2]
    sort [] @?= []

where
    act @?= exp
        | act /= exp = putStrLn "test failed\n"
        | otherwise = return ()
```

The code defines a `sort` function that takes a list of integers and returns a sorted list. It uses a recursive partitioning strategy. The `main` function contains four test cases using the `@?=` operator to check if the `sort` function produces the expected result. A `where` clause provides a default action for the `act` function when the actual result does not match the expected result.

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



A screenshot of a vim window titled "hw07 — vim Sort.hs — 66x25". The code defines a function `sort :: [Int] -> [Int]` which handles three cases: an empty list, a list with one element, and a list with multiple elements. It uses recursive calls to sort the smaller and greater halves and concatenates them with the pivot element `h`. A `main` function runs four test cases using the `@?=` operator to check if the result of `sort` matches the expected result. The tests pass for all cases. Below the code, there are several blank lines followed by the file name and line count.

```
sort :: [Int] -> [Int]
sort [] = []
sort (h:t) =
    let smaller = sort [x | x <- t, x < h]
        greater = sort [x | x <- t, x > h]
    in smaller ++ h : greater

main = do
    sort [2, 1, 3] @?= [1, 2, 3]
    sort [3, 2, 1] @?= [1, 2, 3]
    sort [2, 2, 2] @?= [2, 2, 2]
    sort [] @?= []

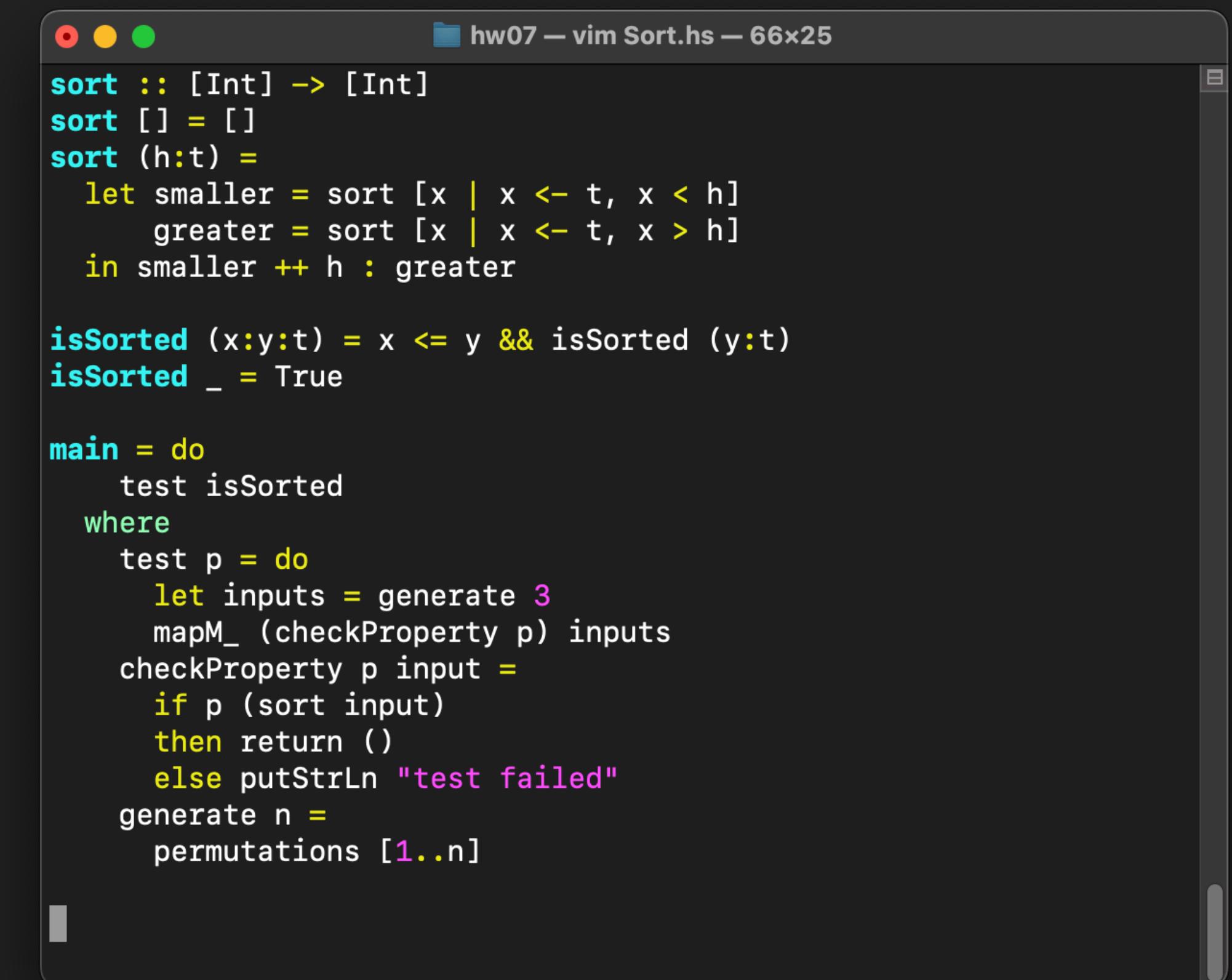
where
    act @?= exp
        | act /= exp = putStrLn "test failed\n"
        | otherwise = return ()
```

"Sort.hs" 18L, 425B

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



A screenshot of a vim window titled "hw07 — vim Sort.hs — 66x25". The window displays Haskell code. The code defines a function `sort` that takes a list of integers and returns a sorted list. It uses a recursive divide-and-conquer approach. It also defines a helper function `isSorted` that checks if a list is sorted. Finally, it defines a `main` function that tests the `isSorted` function using a property-based testing loop.

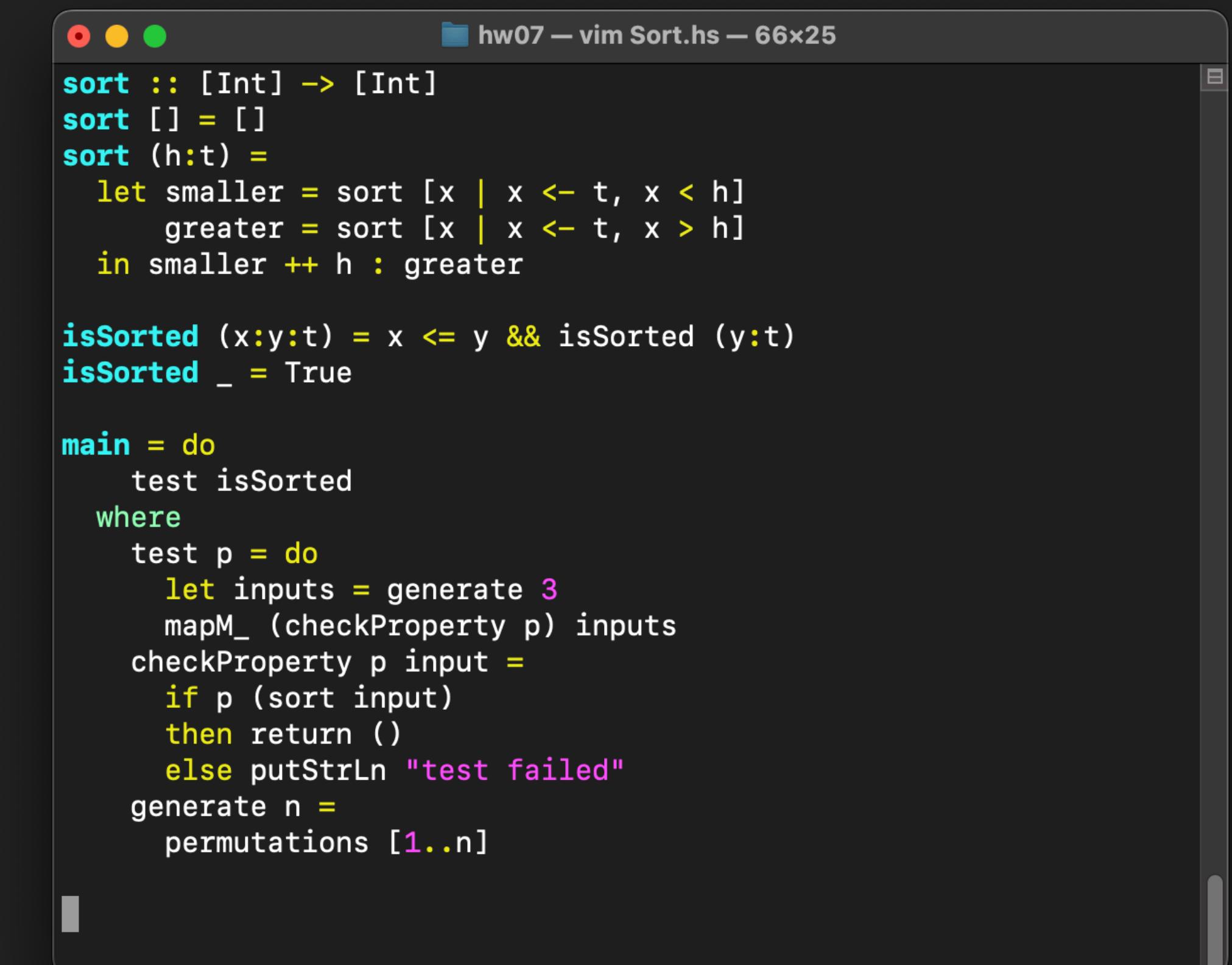
```
sort :: [Int] -> [Int]
sort [] = []
sort (h:t) =
    let smaller = sort [x | x <- t, x < h]
        greater = sort [x | x <- t, x > h]
    in smaller ++ h : greater

isSorted (x:y:t) = x <= y && isSorted (y:t)
isSorted _ = True

main = do
    test isSorted
    where
        test p = do
            let inputs = generate 3
                mapM_ (checkProperty p) inputs
            checkProperty p input =
                if p (sort input)
                then return ()
                else putStrLn "test failed"
        generate n =
            permutations [1..n]
```

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



A screenshot of a vim window titled "hw07 — vim Sort.hs — 66x25". The window displays Haskell code for a sorting function and property-based testing.

```
sort :: [Int] -> [Int]
sort [] = []
sort (h:t) =
    let smaller = sort [x | x <- t, x < h]
        greater = sort [x | x <- t, x > h]
    in smaller ++ h : greater

isSorted (x:y:t) = x <= y && isSorted (y:t)
isSorted _ = True

main = do
    test isSorted
    where
        test p = do
            let inputs = generate 3
            mapM_ (checkProperty p) inputs
            checkProperty p input =
                if p (sort input)
                then return ()
                else putStrLn "test failed"
        generate n =
            permutations [1..n]
```

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

PROPERTY-BASED TESTING

DEMO

```
Test — vim Sort.hs — 84x28

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)

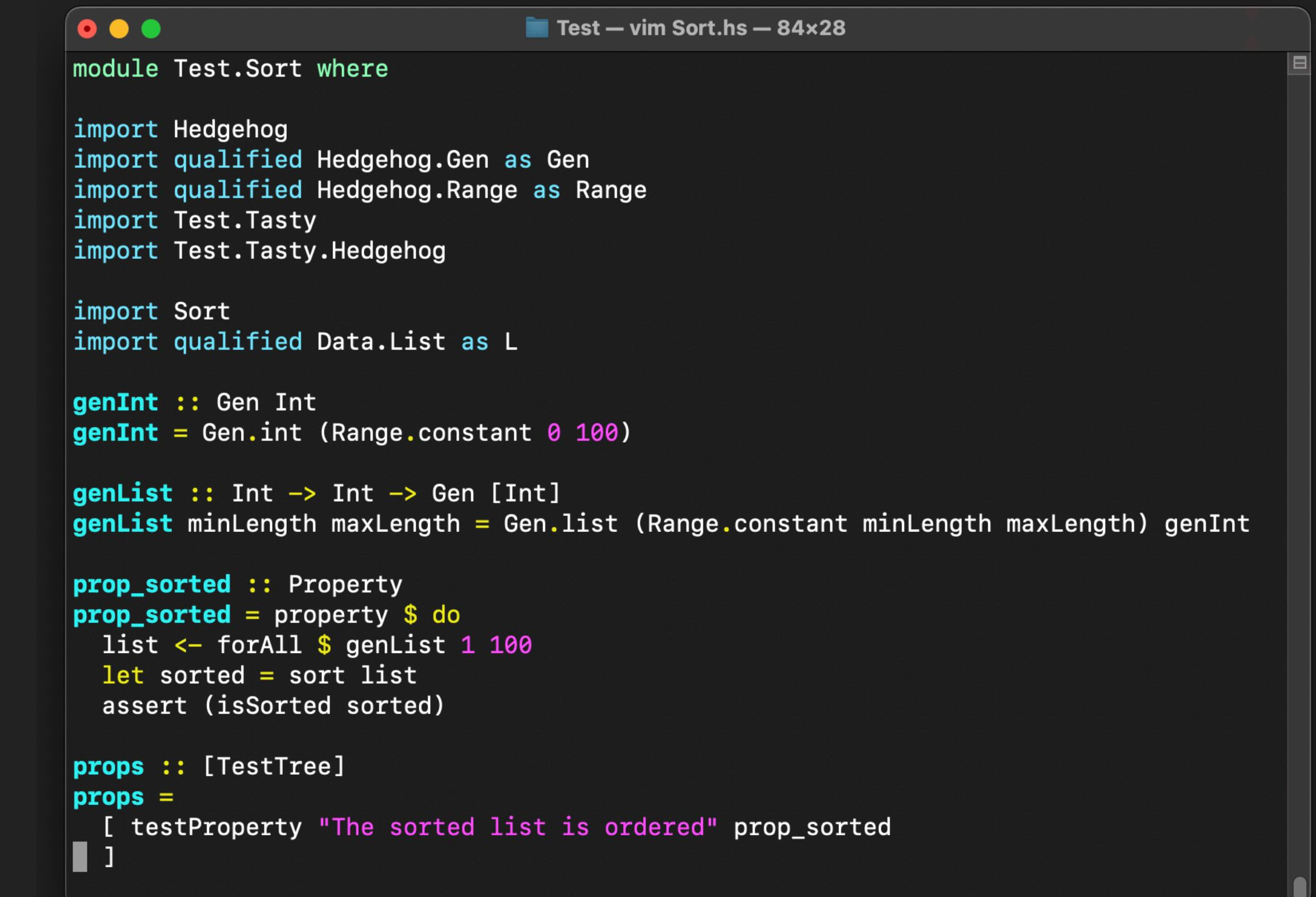
genList :: Int -> Int -> Gen [Int]
genList minLength maxLength = Gen.list (Range.constant minLength maxLength) genInt

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



A screenshot of a terminal window titled "Test – vim Sort.hs – 84x28". The window contains Haskell code for a property-based test. The code imports Hedgehog, qualified Hedgehog.Gen as Gen, qualified Hedgehog.Range as Range, Test.Tasty, and Test.Tasty.Hedgehog. It also imports Sort and qualified Data.List as L. A generator genInt is defined as Gen.int (Range.constant 0 100). A function genList :: Int -> Int -> Gen [Int] is defined as Gen.list (Range.constant minLength maxLength) genInt. A property prop_sorted :: Property is defined using the forAll combinator, asserting that a sorted list is equal to the original list. Finally, a list of properties props :: [TestTree] is defined, containing a single testProperty "The sorted list is ordered" prop_sorted.

```
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

